HCPC: Human centric program comprehension by grouping static execution scenarios

A thesis submitted to the
College of Graduate and Postdoctoral Studies
in partial fulfillment of the requirements
for the degree of Master of Science
in the Department of Computer Science
University of Saskatchewan
Saskatoon

By
Avijit Bhattacharjee

©Avijit Bhattacharjee, July/2021. All rights reserved.
Unless otherwise noted, copyright of the material in this thesis belongs to the author.

Permission to Use

In presenting this thesis in partial fulfillment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Disclaimer

Reference in this thesis to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement, recommendation, or favoring by the University of Saskatchewan. The views and opinions of the author expressed herein do not state or reflect those of the University of Saskatchewan, and shall not be used for advertising or product endorsement purposes.

Requests for permission to copy or to make other uses of materials in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science
176 Thorvaldson Building, 110 Science Place
University of Saskatchewan
Saskatoon, Saskatchewan S7N 5C9 Canada

OR

Dean
College of Graduate and Postdoctoral Studies
University of Saskatchewan
116 Thorvaldson Building, 110 Science Place
Saskatoon, Saskatchewan S7N 5C9 Canada

Abstract

New members of a software team can struggle to locate user requirements if proper software engineering principles are not practiced. Reading through code, finding relevant methods, classes and files take a significant portion of software development time. Many times developers have to fix issues in code written by others. Having a good tool support for this code browsing activity can reduce human effort and increase overall developers' productivity. To help program comprehension activities, building an abstract code summary of a software system from the call graph is an active research area. A call graph is a visual representation of caller-callee relationships between different methods of a software project. Call graphs can be difficult to comprehend for a larger code-base. The motivation is to extract the essence from the call graph by finding execution scenarios from a call graph and then cluster them together by concentrating the information in the code-base. Later, different techniques are applied to label nodes in the abstract code summary tree. In this thesis, we focus on static call graphs for creating an abstract code summary tree as it clusters all possible program scenarios and groups similar scenarios together. Previous work on static call graph clusters execution paths and uses only one information retrieval technique without any feedback from developers. First, to advance existing work, we introduced new information retrieval techniques alongside human-involved evaluation. We found that developers prefer node labels generated by terms in method names with TFIDF (term frequency-inverse document frequency). Second, from our observation, we introduced two new types of information (text description using comments and execution patterns) for abstraction nodes to provide better overview. Finally, we introduced an interactive software tool which can be used to browse the code-base in a guided way by targeting specific units of the source code. In the user study, we found developers can use our tool to overview a project alongside finding help for doing particular jobs such as locating relevant files and understanding relevant domain knowledge.

Acknowledgements

First, I would like to express my hearfelt gratitude to my respected supervisor Dr. Banani Roy for her constant guidance, suggestions, motivation and patience during my thesis work. I am grateful to Dr. Kevin A. Schneider for his support and feedback temporarily at the end of my program when Dr. Banani was on leave for family emergency.

I would like to thank Dr. Gordon McCalla, Dr. Shahedul Khan, and Dr. Madison Klarkowski for their willingness to take part in the evaluation and advisement of my thesis. In addition, I am grateful to them for their valuable feedback, and suggestions.

I would like to thank anonymous reviewers of different conferences for their valuable comments and feedback which helped to improve this thesis work.

Special thanks goes to the Software Research Lab (SRLab) and Interactive Software Engineering (iSE) lab members for the good time we have together. Specially, I would miss lively discussions during coffee breaks at Tim's, and playtime of soccer and cricket games at various playgrounds in summer. In particular, I would like to thank Dr. Chanchal Roy, Dr. Manishankar Mondal, Dr. Masudur Rahman, Amit Kumar Mondal, Daniel Abediny, Muhammad Mainul Hossain, CM Khaled Saifullah, Shamse Tasnim Cynthia, Zonayed Ahmed, Naz Zarren Oishie, Shamima Yeasmin, Farouq Al-omari, Sristy Sumana Nath, Golam Mostaeen, Kawser Wazed Nafi, Debasish Chakroborti, Saikat Mondal, Md Nadim, Md Shamimur Rahman, Judith Islam, Tonny Kar, Md. Abdul Awal, Hamid Khodabandehloo.

I would like to thank the Computer Science department of the University of Saskatchewan for their financial assistance through scholarships, awards, bursaries which helped me to focus on this thesis work. Moreover, I would like to thank all the staffs of the Department for their constant support. In particular, I would like to thank Sophie Findlay, Heather Webb, Greg Oster, Shakiba Jalal, James Ko, Jeff Long, Maurine Powell, and Cary Bernath.

I would like to thank my friends and family who instead of being thousand miles apart was available for me when I needed them. In addition, I would like to thank Afsana Sultana Ruma, Kinsuk Kalyan Sarker, Tanushree Das and Ushasi Srija Chakroborti for their time, love and mental support during my stay in Saskatoon.

I dedicate this thesis to my mother (Bina Bhattacharjee), my father (Arun Kumar Bhattacharjee) and my younger brother (Bishwajit Bhattacharjee), who always believed in me and inspired me to become the best version of myself.

Contents

Permission to Use i

Abstract ii

Acknowledgements iii

Contents v

List of Tables viii

List of Figures ix

List of Abbreviations x

1 Introduction 1
1.1 Motivation 1
1.2 Problem 3
1.2.1 Sub-Problem #1: Lack of Human Evaluation and Comparison Between IR Techniques 3
1.2.2 Sub-Problem #2: Abstraction Nodes are too Short for Helpful Comprehension 3
1.2.3 Sub-Problem #3: Making the Abstraction Tree Usable for Software Engineering Tasks 3
1.3 Research Questions 4
1.4 Solution 4
1.4.1 Labeling Abstraction Nodes and Human Evaluation 4
1.4.2 Providing Summary and Significant Patterns for Abstraction Nodes 5
1.4.3 Finding Effectiveness of an Abstract Code Summary Tree 5
1.5 Publications 5
1.6 Outline of the Thesis 6

2 Background and Related Work 7
2.1 Call graph 7
2.2 Abstract Code Summary Tree 8
2.3 Motivational Example 9
2.4 Techniques and Algorithms 11
2.4.1 TFIDF 11
2.4.2 LDA 12
2.4.3 LSI 12
2.4.4 Jaccard Distance 13
2.4.5 Agglomerative Hierarchical Clustering 13
2.4.6 Text Rank 14
2.5 Related work 14
2.5.1 Program Comprehension in General 14
2.5.2 IR Techniques to Name Source Code Artifacts 15
2.5.3 Reverse Engineering 15
2.5.4 How Developers Locate Features in Source Code 16
2.5.5 Program Comprehension with Static and Dynamic Call Graph 17

3 Labeling Abstraction Nodes and Human Evaluation 18
3.1 Introduction 18
3.2 Motivational Example 20
3.3 Approach 20

3.3.1 Abstract Code Summary (ACS) Tree . . . 20
3.3.2 Naming Nodes in an Abstract Code Summary Tree . . . 24
3.4 Experimental Design . . . 25
3.4.1 Research questions . . . 25
3.4.2 Dataset Collection . . . 25
3.4.3 User-study . . . 27
3.5 Results and Discussion . . . 28
3.5.1 User Naming vs. Automatic Naming . . . 28
3.5.2 User Rating on Function Name Variant . . . 30
3.5.3 User Rating on Words in Function Name Variant . . . 30
3.5.4 Function Name vs. Words in Function Name . . . 30
3.6 Threats to Validity . . . 31
3.7 Summary and Discussion . . . 31

4 Providing Summary and Significant Patterns for Abstraction Nodes 33
4.1 Introduction . . . 33
4.2 Approach . . . 34
4.2.1 Abstract Code Summary Tree of a Software System . . . 36
4.2.2 Source Code to Abstract Code Summary (ACS) Tree . . . 36
4.2.3 Generating Information for Abstraction Nodes . . . 37
4.3 Experimental Design . . . 38
4.3.1 Research Questions . . . 38
4.3.2 Dataset Collection . . . 39
4.3.3 Case Study Design . . . 39
4.4 An Exploratory Case-study . . . 40
4.4.1 RQ1: Effectiveness of Word Variation Labeling . . . 40
4.4.2 RQ2: Natural Text Summary for Abstraction Nodes . . . 41
4.4.3 RQ3: Effectiveness of Mined Patterns from Execution Paths . . . 43
4.4.4 RQ4: Effectiveness of Using Label, Summary and Patterns Together . . . 44
4.5 Threats to Validity . . . 45
4.6 Summary and Discussion . . . 45

5 Finding Effectiveness of the Abstract Code Summary Tree 46
5.1 Motivation . . . 46
5.2 Approach . . . 47
5.2.1 Strike_A_Match Algorithm . . . 47
5.2.2 Node Summary . . . 47
5.2.3 Execution Patterns . . . 49
5.2.4 Cluster Flatten Technique . . . 49
5.3 Implementation . . . 49
5.4 Interface . . . 51
5.5 Guide to Use the HCPC Tool . . . 51
5.6 Exploring the HCPC tool for *jupyter_client* Project . . . 53
5.7 Human-subject Study . . . 56
5.7.1 Research Questions . . . 56
5.7.2 Study Design . . . 56
5.7.3 Participants and Subject System Selection . . . 57
5.7.4 Results . . . 57
5.7.5 Threats to Validity . . . 59
5.8 Summary and Discussion . . . 59

6 Conclusion and Future Work 60

References 61

Appendix A Simple Calculator program to demonstrate the clustering approach 66

List of Tables

2.1 Abstraction Nodes with summary and execution patterns 10
2.2 Sample Term-Document matrix 12

3.1 Pyramid score computation 27

4.1 3 Subject Systems with their No. Entry, Exit Nodes, LOC, Paths, And Date Retrieved 39

List of Figures

2.1 Call graph with entry node, exit node and execution paths . . . 7
2.2 An abstract code summary tree with its different components . . . 8
2.3 An abstract code summary of the calculator program (EP means Execution path or leaf node and AN means Abstraction Node) . . . 9
2.4 Agglomerative and Divisive clustering algorithm with a sample cluster forest . . . 13

3.1 A portion of the Call graph of Real-Time-Voice-Cloning project by Pyan . . . 20
3.2 Structure of an abstract code summary tree . . . 22
3.3 Overview of the overall approach . . . 23
3.4 Tool UI presented to the study participants . . . 26
3.5 Pyramid score of the 12 clusters . . . 28
3.6 User preference among three implemented naming techniques (considering methods as terms) 29
3.7 User preference among three implemented naming techniques (considering words in methods as terms) . . . 30
3.8 Comparison between three techniques considering function names and words in function names 31

4.1 Structure of a abstract code summary tree . . . 36
4.2 Overview of the overall approach . . . 37
4.3 Snippet from subject system 1 (Our code) . . . 40
4.4 Snippet from subject system 2 (pyan) . . . 41
4.5 Snippet from subject system 3 (Real-Time-Voice-Cloning) . . . 41

5.1 Architecture of HCPC tool . . . 50
5.2 HCPC tool interface . . . 52
5.3 HCPC tool overview for jupyter_client project . . . 53
5.4 HCPC tool when focusing on write_connection_file method . . . 55

List of Abbreviations

ACS	Abstract code summary
EP	Execution Path
IR	Information Retrieval
HCPC	Human-centric program comprehension
AN	Abstraction Node

1 Introduction

In this chapter, we provide a brief description of the thesis. In Section 1.1, we discuss motivation of the thesis. Then we addressed three problems in Section 1.2. In Section 1.3 and 1.4, we have introduced the research questions and provided a brief summary of our solutions. In Section 1.6, we outline the whole thesis chapters.

1.1 Motivation

The growing demands of new requirements for software applications make the codebase large. As the life-cycle of software increases, more resources are devoted to the maintenance of the software. If developers want to add a new feature or fix bugs in the existing features, they need to understand related domain knowledge alongside relevant code structure. The ratio of reading code versus writing code in a software developer's role is over 10 to 1 [33]. In addition, if a new developer joins the team, they need to understand how the high-level feature maps with existing low-level source code. When a software developer has to implement a new feature or enhance an existing feature, they need to look for the relevant methods, classes and files to understand how different parts of the relevant code interacts. After getting a good grasp of the relevant codebase, the developer can start working on the new feature. The process of understanding source code is called program comprehension. However, depending on the knowledge a developer possesses for a specific codebase, the steps for comprehending the program can be different.

Program comprehension techniques mainly consist of two models [58, 59, 55] called top-down and bottom-up models. In the top-down model, where developers have the system's domain knowledge and try to map bottom-level source code to the high-level domain knowledge (features in a system). In many cases, the developers lack domain knowledge, forcing them to go through a low-level codebase and gradually build high-level knowledge. The process of cognitive mapping from source code to domain knowledge is called the bottom-up model. When the codebase is new or unknown to the developers, and they lack domain knowledge, generally, the bottom-up model is followed by the developers [64, 55]. The top-down model is more flexible and efficient than the bottom-up model for developers to have some idea about what to expect in the codebase or where to start from [10].

As program comprehension is an integral part of software maintenance, effective tool support for program comprehension will help developers do their day-to-day job properly and with minimal cognitive load. The tool support for program comprehension can save valuable human resources, which cuts the overall cost for

software maintenance [30]. Developers prefer to have high-level domain knowledge and then map the source code to the domain knowledge [10]. However, in real-world scenarios, developers in industry and open source projects have to resolve issues with no option except to follow the cognitive heavy bottom-up model. For example, GitHub, home to many open source projects, has 56 million developers who have completed 1.9 billion contributions[1] in the range of October 2019 - September 2020. The tech giants companies like Google, Apple, Facebook, Microsoft have dedicated developer times for contributing to open source projects. Visual Studio Code, currently the most popular code editor from Microsoft, is developed by more than thousands of developers across the globe[2]. The developers except the core team mainly fixes bugs or implements new features without being familiar with the whole codebase. In the first step of their contribution, they must acquaint themselves with relevant parts of the codebase, which is the bottom-up model. Therefore, it is essential to have sophisticated tools to help the developers with the bottom-up model. Researchers work on abstracting source code based on call graphs to reduce the cognitive load when developers follow the bottom-up model.

Method names are the lowest level of abstraction in the source code. Method names represent a unit task of the overall system [16, 56]. The interaction between different methods is the building block to understand the high-level concept in source code. Call graphs are visual representations of interactions among methods in the system. Call graphs construction techniques are of two types. The static call graphs are built by analyzing source code to find the caller-callee relationships among methods. Later, building a graph using the relationships where edges represent which method calls which method and nodes represent the method names. The dynamic call graphs are constructed by logging function invocations during run-time. To generate a dynamic call graph, the software system needs to be run for different scenarios. During the scenario execution, function invocations are recorded, which can be converted to a graph similar to the static call graph. The main difference between dynamic call graphs and static call graphs is that the dynamic call graph contains only methods invoked during the execution where the static call graph includes all the methods in the codebase [19]. The advantage of a dynamic call graph is that the call graph can be generated for targeted execution scenarios [18]. One disadvantage of the dynamic call graph is that it generates a massive amount of redundant data (logged information of repeated function executions), which is difficult to process. We have decided to use static call graphs to create a tool for supporting program comprehension models.

As the static call graph properties align more to build an abstract code summary, recently, few studies have been utilizing the static call graph to generate abstract code summary of a software system [19, 60]. In this thesis, we focus on enhancing the capability of an abstract code summary from the existing research by addressing limitations. We also focus on the usability of the abstract code summary tree by building an interactive program comprehension tool in a guided way according to their specific tasks.

[1] https://octoverse.github.com/
[2] https://github.com/microsoft/vscode

1.2 Problem

1.2.1 Sub-Problem #1: Lack of Human Evaluation and Comparison Between IR Techniques

In the literature, a great many studies have focused on generating an abstract code summary of a software system using both static and dynamic call graphs [18, 19, 68]. The abstract code summary is a tree-like structure where execution scenarios are clustered, and each node is labelled using different information retrieval (IR) techniques on source code entities. The success of constructing the abstraction tree depends on how well the labelling techniques perform. Other information retrieval techniques show promising performance in naming source code artifacts [11, 43, 57]. Although a lot of work exists on hierarchical abstraction, they lack comprehensive study on the effectiveness of different information retrieval techniques in labelling nodes of an abstraction tree with humans in the loop. No empirical research exists to find which IR technique works well in which situation. Moreover, methods are treated as a unit [19, 18] while using different information retrieval techniques for labelling nodes. Previous research [16] shows that IR techniques perform better when more information like comments are used instead of method names. Therefore, using method names as unit provides less opportunity to retrieve the overall context.

1.2.2 Sub-Problem #2: Abstraction Nodes are too Short for Helpful Comprehension

In the previous studies [18, 19], each node has five method names as their label in the abstract code summary tree. During our first study to evaluate IR techniques on labelling nodes, we observe that using 5-10 method names or words serves as a title for the node. The title can provide context, although it is difficult to comprehend what is happening inside a node without further detail. Each abstraction node is a collection of execution paths that may have variable lengths. Providing all the execution paths of a node to developers hinders the purpose of abstraction. Therefore, the challenge is to develop a solution that can briefly provide the context of a node without providing everything.

1.2.3 Sub-Problem #3: Making the Abstraction Tree Usable for Software Engineering Tasks

Newcomers to open source software struggle with a lack of domain knowledge. Usually, developers (contributors) look for trending projects in their choice of language and popularity to contribute in social coding platforms (GitHub, GitLab). As most of the time, the problem being solved is unknown to the developer, developers struggle to map low-level source code to high-level concepts. As it is stated in previous studies [10], developers prefer the top-down model to browser source code for program comprehension. In the

top-down model, developers have some domain knowledge, which they later try to map with source code. The hierarchical abstraction tree has the potential to bridge the gap between the top-down and bottom-up cognition models. However, the challenge is to tailor the abstraction tree for the developers to use for a specific task in hand or target a particular unit of the source code (method).

1.3 Research Questions

While considering the above problems discussed in Section 1.2, we came up with five research questions:

- RQ1: How well the automatic techniques generated node titles match with the developers generated node titles?
- RQ2: What are the developers' preferences over full method names and terms in method names as node title?
- RQ3: How can we provide a natural text summary to abstraction nodes?
- RQ4: How can we mine significant patterns from execution paths for each abstraction node?
- RQ5: How can we make the abstraction tree useful for daily day-to-day software engineering jobs?

Research question one and two correspond to *sub-problem 1*, research question three and four correspond to *sub-problem 2* and research question five correspond to *sub-problem 3*. This research aims to help software development activities by generating abstract code summaries using call graphs.

1.4 Solution

Considering the three problems mentioned above statements (Section 1.2) in the domain of program comprehension, we contributed three studies. Below we have briefly discussed the three studies.

1.4.1 Labeling Abstraction Nodes and Human Evaluation

In this study, by mining concepts from source code entities (names of functions/methods), we generate an abstract code summary tree with improved naming of the cluster nodes. Our motivation is to complement existing studies to facilitate more effective program comprehension for developers to address *problem statement #1*. We apply three different information retrieval techniques such as TFIDF (Term frequency-inverse document frequency) [47], LDA (Latent Dirichlet Allocation) [9], and LSI (Latent Semantic Indexing) [17] (i.e., each technique with function names and words in function names variation) to label nodes of an abstract code summary tree generated by clustering execution paths. Our experiment found that among the techniques on average, TFIDF performs better with around 64% matching with developers generated node label than the other two methods (LDA and LSI) that show 37% and 23% matching respectively for 12

cases. Besides, the words in a function name variant perform at least 5% better in the user rating for all the three techniques on average for the use cases. Our study draws on the existing research but considers more techniques and human responses for comprehending outputs using the three techniques.

1.4.2 Providing Summary and Significant Patterns for Abstraction Nodes

In this study, we develop two new techniques to supplement nodes' information in a hierarchical abstraction tree for better comprehension to address *sub-problem #2*. Generally, methods are expected to come with documentation at the start with a single line describing what the function does unless the method is concise and obvious [3]. First, we tried to exploit this standard practice for generating a brief text summary for each node. To complement existing techniques of labeling nodes, we add a text description to the node by summarizing all the method comments under that node.

Second, execution paths in the call graph represent execution scenarios [52, 46]. Therefore, inspired by previous studies [52, 46] we add significant patterns for each node by analyzing all execution paths under the node. We conducted an empirical study with three subject systems to evaluate the potential of the two proposed techniques. We found that the proposed techniques complement the existing abstraction tree, although there are some challenges. By addressing those challenges, the proposed techniques will be more effective for program comprehension.

1.4.3 Finding Effectiveness of an Abstract Code Summary Tree

As discussed in the *sub-problem 3*, making the abstraction tree browse-able with a specific target is helpful for new contributors in open source software systems. Having a system that helps to make top-down cognition possible without domain knowledge can be a game-changer for new contributors. In this study, we have built a system where the tree can be browsed by selecting a specific method. When a particular method is selected, relevant nodes in the tree are highlighted. Moreover, developers can see information like files involved, number of execution paths, summary and frequent patterns of an abstraction node. To evaluate the effectiveness, we have conducted a user study with the developers from the Scidatamanager[4] team. The participants evaluated our approach on the abstract code summary tree generated from the Scidatamanager project. From the participants' feedback, it is viable that the HCPC (Human-centric program comprehension) tool can help developers get an overview of a codebase. In addition, the HCPC tool is helpful to know the relevant method, files to be looked for doing a particular task.

1.5 Publications

Below we have listed published and submitted works (with collaborators) from this thesis.

[3] https://google.github.io/styleguide/pyguide.html
[4] http://scidatamanager.usask.ca

- *Avijit Bhattacharjee*, Banani Roy and Kevin Schneider. Clustering execution scenarios to aid top-down model of program comprehension. 37th International Conference on Software Maintenance and Evolution. (Submitted)

- *Avijit Bhattacharjee*, Banani Roy and Kevin Schneider. Supporting program comprehension by generating abstract code summary tree. 36th IEEE/ACM International Conference on Automated Software Engineering (ASE). (Submitted)

- *A. Bhattacharjee*, S. Nath, S. Zhou, D. Chakroborti, B. Roy, C. Roy, and K. Schneider. An Exploratory Study to Find Motives behind Cross-platform Forks from Software Heritage Dataset. In Proceedings of the 17th International Conference on Mining Software Repositories (MSR) - Mining Challenge Track, 2020.

- Saikat Mondal, C M Khaled Saifullah, *Avijit Bhattacharjee*, Mohammad Masudur Rahman, and Chanchal K. Roy. 2021. Early Detection and Guidelines to Improve Unanswered Questions on Stack Overflow. In Proceedings of 14th Innovations in Software Engineering Conference (formerly known as India Software Engineering Conference), Bhubaneswar, Odisha, India, February25–27, 2021 (ISEC 2021),11 pages

1.6 Outline of the Thesis

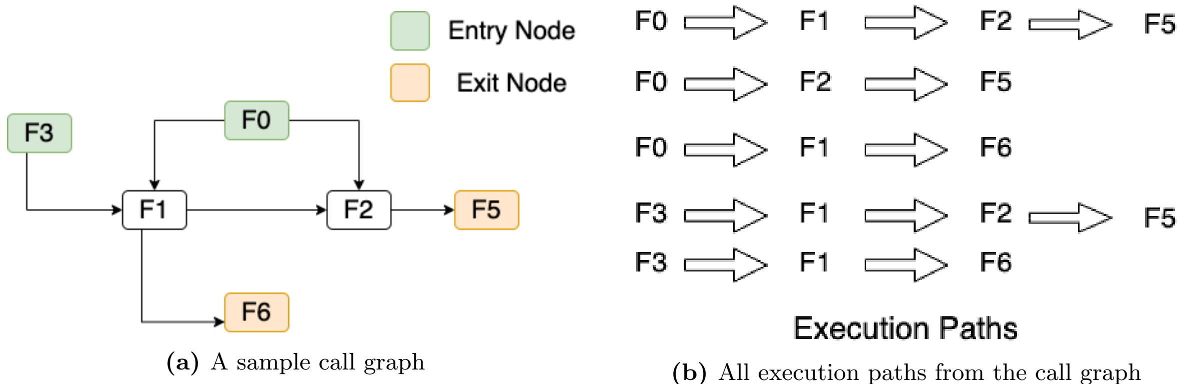
In Chapter 2, we discuss some background on the call graph-related terminologies, clustering techniques, different information retrieval techniques alongside a text summary technique and related works. Chapter 3 focuses on different information retrieval techniques with human evaluation. In Chapter 4, we proposed two techniques for adding node summary and execution patterns in the abstraction tree to aid developers program comprehension. In Chapter 5, we evaluated abstract code summary tree with expert opinion on their system. Finally, in Chapter 6 we conclude the overall summary of the thesis and discuss some future plan.

2 Background and Related Work

In this chapter, we briefly discuss relevant terms, topics and techniques helpful to this thesis. In Section 2.1, we elaborate terms relevant to a call graph. We then present an abstract code summary tree in Section 2.2. In Section 2.3, we provide an abstract code summary tree for a sample calculator program using our system. In Section 2.4, we have elaborated different techniques and algorithms used in the thesis. In Section 2.5, we discuss related work for the studies done in the thesis.

2.1 Call graph

A *call graph* is a control flow graph of a program showing calling relationships between functions. Each node of the graph represents a function and each edge (a, b) represent calling relationship where function a calls function b. Figure 2.1a shows a simple call graph with six nodes indicating functions and six edges indicating calling relationships. Call graphs can be of two types. One type is a static call graph. A static call graph contains all the possible program execution scenarios. To generate a static call graph, source code of the program is analyzed to find the relationships. A dynamic call graph represents one program run scenario. Therefore, a dynamic call graph is exact and limited to the scenarios used to generate the graph. To generate a dynamic call graph, logger or profiler is applied which generates call graph during run-time of the program.



(a) A sample call graph

(b) All execution paths from the call graph

Figure 2.1: Call graph with entry node, exit node and execution paths

An *entry node* for a call graph is the node in which the number of incoming degrees is zero. In Figure 2.1a, the call graph has two entry nodes *F0, F3*. No other nodes call the functions or nodes *F0, F3*. That means program execution can start from these nodes.

An *exit node* for a call graph is the node in which number of outgoing degrees is zero. In Figure 2.1a, the

call graph has two exit nodes *F0, F3*. The exit nodes *F0, F3* do not call any other functions or nodes. That means program execution will end when we come to these nodes.

The *execution paths* of a call graph are the all possible program execution scenarios. A program execution scenario consist of a function call sequence starting from a *entry node* and ending to a *exit node* of the call graph. In Figure 2.1b, all the execution paths from the call graph of Figure 2.1a are listed. The first node of the execution paths are the Entry nodes which is defined above. Similarly, the last node of the execution paths are the Exit nodes.

2.2 Abstract Code Summary Tree

In this thesis, we introduce a term called abstract code summary (ACS) tree. In an ACS tree each leaf node is attached to an execution path extracted from the call graph of a software system. The parent nodes of the leaf nodes are grouping of similar execution paths (leaf nodes). We call this intermediate nodes an abstraction node as it abstracts similar execution scenarios. Each abstraction node has three properties which are title, text summary and execution patterns.

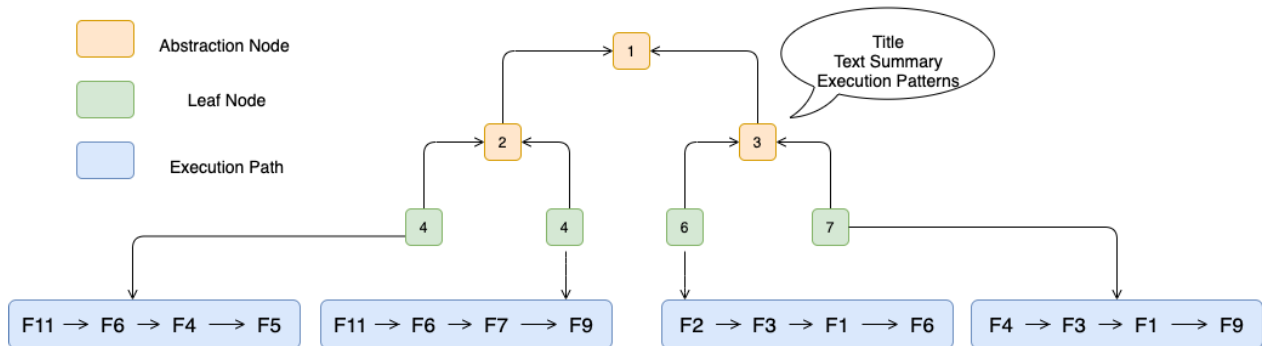


Figure 2.2: An abstract code summary tree with its different components

In Figure 2.2, we present a ACS tree where *4, 5, 6, 7* nodes are leaf node which are attached to execution paths. Nodes *1, 2, 3* are abstraction nodes which are grouping of the leaf nodes. Each abstraction nodes has number of execution paths and we use different information from those execution paths to generate concepts for them. Node 3 has two execution paths which belong to node 6 and 7. Like all other abstraction nodes Node 3 will have title, text summary and execution patterns. The title of Node 3 will be generated using different information retrieval techniques utilizing method signatures. Next, the text summary of Node 3 will be generated by summarizing comments of the methods which belong to the execution paths of Node 3. The execution patterns for Node 3 will be generated by finding frequent patterns from the execution paths of Node 3.

2.3 Motivational Example

To demonstrate how a software system's hierarchical abstraction will work, we have created a sample Calculator program. The program takes two numbers as inputs, validates the inputs, and prompts the user to input which operations they want to perform. Later, according to the input, addition, subtraction, multiplication, division can be performed. This is a brief functionality of the calculator program. We have provided the source code of the Calculator program in appendix A.

In Figure 2.3, we have presented the hierarchical abstraction of the Calculator program. From the figure, we can see our Calculator program has six execution paths. Their node numbers are from 0-5.

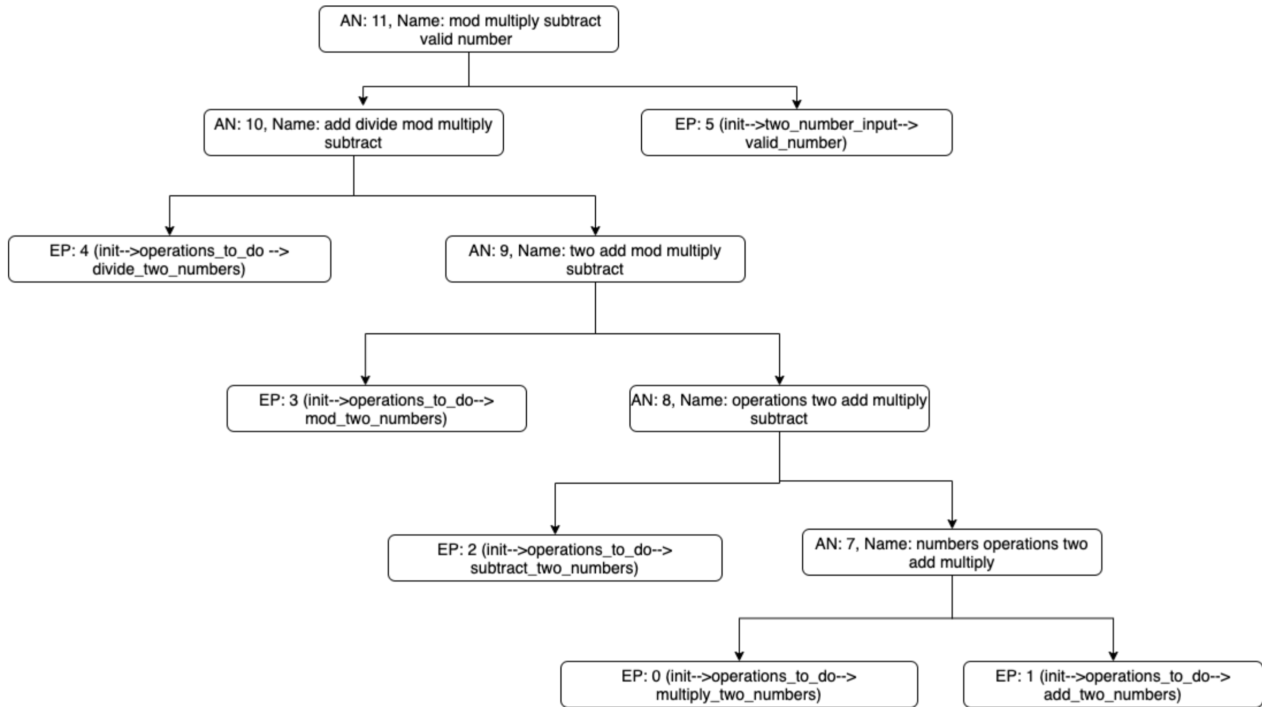


Figure 2.3: An abstract code summary of the calculator program (EP means Execution path or leaf node and AN means Abstraction Node)

Constructing the abstract code summary. To generate the tree shown in Figure 2.3, the following steps are followed.

1. To get the caller-callee relationships from the source code of Calculator program, we use a static source code analyzer.

2. We construct a static call graph from the extracted relationships of *Calculator.py* program.

3. From the call graph, possible execution scenarios are generated which are the execution paths shown in Figure 2.3 (EP 0 - 5).

4. Similarity scores for each pair of execution paths are calculated which is used by the clustering algorithm to group the execution paths. As EP 0, 1, ..., 4 all have three common functions, the similarity measure between them will be same.

5. A clustering algorithm starts grouping the execution paths by taking the most similar two first. In Figure 2.3, we see that EP 0, 1 are grouped together as abstraction node (AN) 7.

6. As AN 7 have EP 0, 1, we use information retrieval techniques on all the terms in functions names of EP 0, 1 to label the node AN 7.

7. Although keywords are helpful for providing hints to features, having a text description and frequent execution patterns for each abstraction node increases comprehension. In Table 2.1, we presented node summary and execution patterns for AN 10, 11.

Table 2.1: Abstraction Nodes with summary and execution patterns

AN	Node Summary	Execution Patterns
11	This function multiplies two numbers. This function mod two numbers. This function subtract two numbers.	•*init* → *two_number_input* → *valid_number*. • *init* → *operations_to_do* → *add_two_numbers*. • *init* → *operations_to_do* → *divide_two_numbers*. • *init* → *operations_to_do* → *mod_two_numbers*. • *init* → *operations_to_do* → *multiply_two_numbers*. • *init* → *operations_to_do* → *subtract_two_numbers*.
10	This function mod two numbers. This function divide two numbers. This function subtract two numbers.	•*init* → *operations_to_do* → *add_two_numbers*. • *init* → *operations_to_do* → *divide_two_numbers*. • *init* → *operations_to_do* → *mod_two_numbers*. • *init* → *operations_to_do* → *multiply_two_numbers*. • *init* → *operations_to_do* → *subtract_two_numbers*.

Exploring the abstract code summary.

- Execution path 0 and 1 represent the functionality of multiplying two numbers and adding two numbers, respectively. For these two clusters, add and multiply are the two different jobs they are doing. Other functions of the two paths are similar. So, the abstraction of these two execution paths is abstraction node 7. Five keywords are picked as the abstraction of execution paths 0 and 1. From the keywords of node 7, it is clear that descendent nodes do addition and multiply on two numbers.

- Next, for node 10, we can see the keywords are add, divide, mod, multiply, and subtract. These five keywords indicate that the descendant nodes of 10 do these numerical operations. If we observe the five execution paths (EP 0 - 4), we find that they perform add, delete, mod, multiply operation on

two input numbers. We can see that the five keywords of node 10 summarize the functionality of its descendants.

- Similarly, for node 11, the keywords are mod, multiply, subtract, valid, and number. We can see the right child node (node 5) of node 11 input two numbers and then validates it. Left descendants of node 11 perform numerical operations. So, the summary of node 11 contains three words relevant to operation and two for input validation.

From our understanding, we can see that this is an almost human level summary for node 10. The summary presented in Figure 2.3 is generated using TFIDF scores on words in method names.

2.4 Techniques and Algorithms

In this Section, we discuss important techniques and algorithms used to construct ACS tree. In Subsection 2.4.1, 2.4.2 and 2.4.3, we discuss TFIDF, LDA and LSI technique which are used for generating node title from method names. In Subsection 2.4.4, we discuss Jaccard Distance which is used to calculate similarity between execution paths. In Subsection 2.4.5, we discuss AHC algorithm which is used to cluster execution paths. Finally, we discuss Text Rank algorithm which generate node summary from method comments in Subsection 2.4.6.

2.4.1 TFIDF

TFIDF [47] is a weight based statistical information retrieval technique. It tries to find important terms to a specific document by analyzing a collection of documents. TFIDF is popular for document classification, search engine ranking and text mining[1]. TFIDF ranks terms by term frequency-inverse document frequency score. Term frequency is the count of a term in a document. Term frequency is biased towards frequent terms which mostly stop words and other fairly meaningless words irrelevant to the document.

$$tf(W_x, D_x) = f_{W_x,D_x} \tag{2.1}$$

$$idf(W_x) = \log(\frac{n}{df(W_x)}) + 1 \tag{2.2}$$

$$tf - idf(W_x, D_x) = tf(W_x, D_x) * idf(W_x) \tag{2.3}$$

Jones [28] introduced inverse document frequency which penalties common terms by counting their occurrence across the corpus. Let, $D = \{D_1, D_2, ..., D_n\}$ is a collection of documents and $W = \{W_1, W_2, ..., W_n\}$ is unique terms in the collection of documents. Now, to calculate term frequency for term W_x in document

[1] https://en.wikipedia.org/wiki/Tf–idf

D_x, we have to count frequency of term W_x in document D_x which is required to calculate term frequency according to equation 2.1. In addition, we have to count the number of documents has term W_x which is used to calculate inverse document frequency using equation 2.2. In equation 2.2, n is the number of documents in the corpus and $df(W_x)$ is the number of documents which contain term W_x. Equation 2.3, multiplies term frequency and inverse document frequency to reward significant terms and penalize common terms. We have adopted `TFIDFVectorizer` class of scikit-learn [44] library for implementing $TFIDF$ technique.

2.4.2 LDA

Latent Dirichlet Allocation (LDA) [9] is a statistical model that tries to describe a set of documents by assuming they are created from some topics. LDA is a very popular topic modeling technique. LDA assumes every term in a document belongs to some topic. So, it assumes each term belongs to some topic and then performs analysis to find which assumptions are supported by statistics of the corpus. We have used Gensim [48] library for implementing LDA for our approach.

2.4.3 LSI

Latent Semantic Indexing (LSI) [17] focuses on information retrieval based on semantic similarity between words where the previous techniques focus on matching words in query with words of documents. The semantic concept used in LSI assumes semantically similar words appear together. Information retrieval techniques which matches words suffer two limitations. They are *synonymy* and *polysemy*. *synonymy* is the issue where the same object is described by different words depending on needs, knowledge and linguistic habits. On the other hand, *polysemy* refers to the fact that words have multiple distinct meanings in different contexts. LSI, first, starts with a Term-Document matrix where all terms are presented in the rows and documents in the columns. Table 2.2 shows an example of a Term-Document matrix.

Table 2.2: Sample Term-Document matrix

	ship	boat	ocean	voyage	trip
Document 1	1	0	1	0	0
Document 2	0	1	0	1	0
Document 3	1	0	0	1	1

Single Value Decomposition (SVD) method is used to project the term-document matrix to reduced numnber of dimensions. The reduced matrix by SVD is an approximation of the term-document matrix which is a representation of the semantic similarity between words in documents. If we need to find similarity between a query, the query is converted to similar representation and compared to find relevant documents. By using this technique, LSI can detect semantic similarity even when the terms are different. Similar to LDA, we used Gensim [48] library for implementing LSI.

2.4.4 Jaccard Distance

Jaccard Distance can measure similarity between two sequences according to equation 3.1. For example, we have two execution path E_i and E_j and they have set of function names F_i and F_j respectively. Therefore, similarity between E_i and E_j can be measured by equation 3.1.

$$JD_similar(E_i, E_j) = \frac{F_i \bigcap F_j}{F_i \bigcup F_j} \tag{2.4}$$

$$JD_dissimilar(E_i, E_j) = 1 - \frac{F_i \bigcap F_j}{F_i \bigcup F_j} \tag{2.5}$$

If E_i and E_j are very similar, according to equation 2.4 similarity score will be near 1 and vice-versa. Clustering algorithm merges those two clusters which distance measures are minimum. Equation 2.5 subtract Jaccard Distance by 1 to get desire dissimilarity measure for clustering algorithms.

2.4.5 Agglomerative Hierarchical Clustering

Clustering algorithms are popular in many data mining, unsupervised machine learning and pattern recognition applications. Clustering algorithms try to group similar observations together to find significant patterns in the observations. Hierarchical clustering can be done in two ways. One is bottom-up (agglomerative) and another is top-down (divisive). For divisive clustering, all observations starts in a single cluster and divided into different clusters using heuristics. Agglomerative clustering starts by considering observations as individual clusters and then group them until all observations end-up in the same cluster.

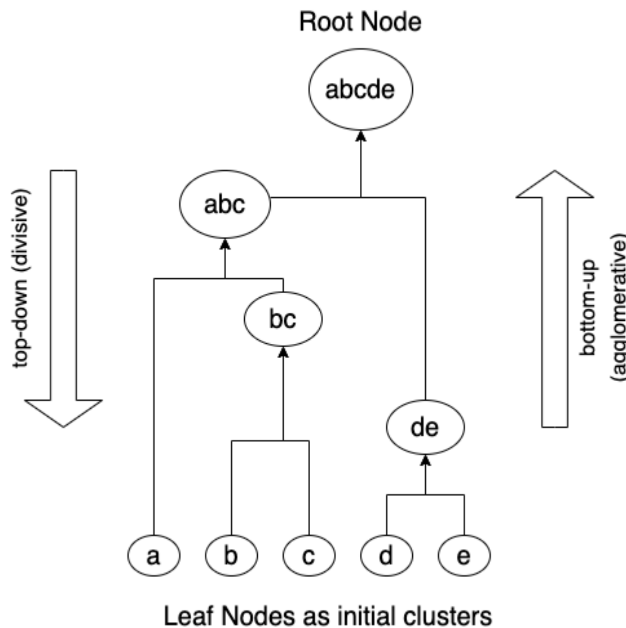


Figure 2.4: Agglomerative and Divisive clustering algorithm with a sample cluster forest

In Figure 2.4, a visualization of how agglomerative and divisive clustering algorithm works are presented. Lets assume there are five observations *a, b, c, d, e* and we have similarity score between all the pairs of the observations. First, we can see five observations are treated as five clusters. From the similarity score we found that clusters *d and e* are most similar. Therefore, we group cluster *d and e* together as a new cluster *de*. Now, in the cluster forest we have four clusters. In the next step, cluster b and c are the most similar. So, agglomerative clustering algorithm will group cluster b and c as a new cluster *bc*. The agglomerative clustering will continue to merge clusters together until there is only one cluster in the cluster forest. For this example, the final cluster *(abcde)* consists of all the initial clusters.

2.4.6 Text Rank

Mihalcea [35] proposed a graph based ranking algorithm called TextRank inspired by the PageRank algorithm to rank entities in natural language. Two of the significant application of TextRank are keyword extraction and sentence extraction. Sentence extraction can be formulated to generate summary of natural language text. To generate a summary of a paragraph, first, sentences are split as they are the unit for TextRank algorithm. Next, sentences are converted to word embedding vectors. In the next step, similarity matrix is computed from embedding vectors. Then, a graph is created where vertices are sentences and edges represent similarity scores between sentences[2]. Similarity scores are used to extract top ranked sentences according to equation 2.6.

$$WS(V_i) = (1-d) + d * \sum_{V_j \epsilon IN(V_i)} \frac{w_{ji}}{\sum_{V_k \epsilon Out(V_j)} w_{jk}} WS(V_j) \tag{2.6}$$

Let, $G = (V, E)$ is a directed graph where V is the collection of vertices and E is the collection of edges. $In(V_i)$ is the set of vertices which points to vertex V_i. Similarly, $Out(V_j)$ is the set of vertices which vertex V_j points to. The similarity score between vertex V_i and V_j is represented by w_{ji}.

2.5 Related work

2.5.1 Program Comprehension in General

Program comprehension is a cognitive way of understanding software systems to perform different software maintenance tasks [64, 55]. Three different types of cognitive models [58, 59, 55] can be found in the literature which is followed consciously or unconsciously by developers. The comprehension models are Top-down, Bottom-up, and Integrated. When developers have prior domain knowledge about a software system, the top-down model is preferred as they can map domain knowledge to low-level source code hierarchically [10]. On the other hand, when developers lack domain knowledge, they start with low-level source code and then

[2] https://www.analyticsvidhya.com/blog/2018/11/introduction-text-summarization-textrank-python/

group the functionality together to have a hierarchical abstraction of the system features [54, 45]. Integrated model [53, 59] is a mix of top-down and bottom-up approaches. The problem in hand and the target system have different properties in the real world, which demand switching between top-down and bottom-up models. Generally, a developer can have prior domain knowledge of a few portion and point-blank for the rest of the system. This situation deserves the adapted use of both top-down and bottom-up approaches.

2.5.2 IR Techniques to Name Source Code Artifacts

As software repositories contain unstructured data, topic model techniques are widely applied for different software engineering tasks to retrieve information [11, 43, 57]. Most common tasks where topic models showed promising results are source code comprehension, feature location, refactoring, bug localization, and others [57]. Lucia et al. [16] conducted a study to see how information retrieval techniques perform compared to manual naming Java class files. Developers were asked to pick ten keywords for each class file, and top-10 words are picked using different topic model technique and custom heuristics. Their experiment shows that in 40%-80% cases, automatic and human labels overlap.

2.5.3 Reverse Engineering

Subsystem Identification

Muller et al. [37] proposed subsystem detection algorithm using different clustering components like variable, procedure, and modules. According to Bass et al. [8], two types of software architecture are useful for understanding a complex software system. They are Conceptual and Concrete architecture. A conceptual architecture provides high-level abstraction skipping the code level details. On the other hand, concrete architecture shows the implementation level information. Roy et al. [51] propose and evaluate a framework for the incremental and iterative application of automated architecture recovery (using SWAG Kit) and architecture analysis (using SAAM.). They showed that the reverse engineering tool cannot recover a deeply understood conceptual architecture without SAAM's application but can create a reasonable basis towards that direction. Murphy et al.[39] show that by generating reflexion models from high-level model and source model (i.e., static call graphs), it is possible to facilitate program understanding to the novice developers. In this thesis, we try to automatically recover conceptual architecture from concrete architecture, reducing manual effort.

Call Graphs to Abstract a Software System Behaviors

Static and dynamic call graphs are used in literature to help developers comprehend a software system to aid different maintenance tasks [18, 19, 68]. Feng et al. [18] proposed an approach to use dynamic call graphs for understanding a system's behavior. They instrumented the subject systems to generate execution traces of method entry and exit events. Later, they followed the duplication removal process and constructed a call

graph from the execution traces. Execution phases from this dynamic call graph are clustered to get system behaviors. Similarly, Gharib et al. [19], and Vijay et al. [60] also adopted clustering of execution paths from call graphs of the static variant. Using a static call graph brings the benefit of capturing all possible scenarios and less redundant data to handle than dynamic call graph [19].

IR Techniques on the Hierarchical Abstraction of a Software System

Feng et al. [18] proposed an approach to identify behaviors of a system by hierarchically abstracting dynamic call graph from execution traces. Sequential pattern mining is applied to mine significant portions from the execution phases. Hierarchical clustering is performed to group execution phases. Later, the clusters are labeled using the TFIDF score, where method signatures serve as terms and the phases as document. Paul et al. [34] used static call graph to hierarchically abstract a system. In their hierarchical view, each node represents a method. To mine the topics, keywords from methods are considered. Hierarchical Document Topic Model (HDTM) by [65] Weninger et al. is adopted, which works on graph documents to mine topic. Gharib et al. [19] took a different approach. They went further with the static call graph by extracting execution paths and then clustering the execution paths. Each cluster in the cluster tree is labeled using top-5 method names from Tfidf. Levy et al. [32] found interviewing developers that two kinds of comprehension go for large scale hierarchical view. They are system comprehension and code comprehension. In this thesis, we tried to adopt static call graph analysis from Gharib et al. and then improve their labeling technique. Nodes of the cluster tree is considered as a behavioral abstraction unit of a system. Method comments are used to generate a description of the unit and sequential pattern mining to create sample examples.

2.5.4 How Developers Locate Features in Source Code

Kruger et al. [30] studied two data sets (67 developers IDE activity, 600 developers IR-based tool usage). They suggested that there is room for improvement in the existing code navigation, code search tools. The manual processes followed by developers to locate features are of mostly three types [15, 62, 50]. First, developers use information retrieval based tools to query for feature related keywords. In this thesis, we have used IR based techniques to label nodes. Developers can use our tool to find keywords of their interest. Second, there is an execution-based process where developers try to find execution scenarios where the feature is active. After finding relevant execution scenarios, developers debug the execution scenarios by setting breakpoints. In our second study, we have attached execution patterns to nodes which can be utilized by developers to know where to set the breakpoints for understanding a feature. Third, there is an exploration-based process where developers explore source code to understand method calls to find a feature. In the HCPC tool, we showed method execution patterns for each node. Our tool can also help developers in browsing code using an exploration-based process.

2.5.5 Program Comprehension with Static and Dynamic Call Graph

Feng *et al.* [18] proposed an approach to abstract execution traces for program comprehension. To get execution traces, they used BLINKY to instrument source code for getting method-invocation calls. Different test cases are used to generate execution traces for different scenarios. From dynamic logs, they have built phase trees that are created from caller-callee relationships of invoked methods. After deleting duplicate phases, they clustered unique phases using the Agglomerative hierarchical clustering algorithm. Next, they applied a mining technique to get frequent pattern phases of each level of clustered phase tree. For comprehension purposes, they used TFIDF to rank method names of frequent phases and then used the top 20 method names for the final label. Depending on dynamic call graphs comes with some limitations as it depends on the test cases heavily, and the size of log file generated is difficult to handle. Therefore, we choose static call graphs to remove the test dependency and capture a call graph's overall execution scenario. Gharib *et al.* [19] proposed an approach using static call graphs for hierarchical abstraction. First, they generated a static call graph for a subject system that captures overall function relationships. Second, execution paths from the call graph are extracted, which become the building blocks for their approach. Next, execution paths are clustered together to create abstract code summary of the target subject systems. Feng *et al.* [18] also named the clusters by extracting keywords from the function names present in execution paths. In their study, only the TFIDF technique is applied to extract and name intermediate clusters.

For this study, our motivation is to take forward this approach and enrich it with existing techniques from the literature. Two limitations of the study from Gharib *et al.* are using only TFIDF method for information retrieval and no presence of user study to validate how developers prefer the output abstractions. We adopt two more topic modeling techniques for information retrieval, which show promising results for naming source code artifacts in the literature [16]. Andrea *et al.* [16] tried to apply IR techniques like VSM, LDA, and LSI on source code artifacts. To evaluate IR techniques' effectiveness, they also produced suggestions from 17 users on the same classes. Then, they assessed the performance of automatic naming by comparing overlap with manual naming of users. In their study, authors also find that heuristic based approaches focusing on function signatures perform well for code artifacts summarization. Inspired from their study, we use LDA and LSI on function signatures to extract concepts in code in this study. Another improvement from Gharib *et al.* is to adopt a user study for validating automatic abstraction. Sonia *et al.* [22] used Pyramid score to evaluate the output of automatic code summary with developers' summary. We also adopt this Pyramid score, which is widely used for the evaluation of natural language summaries.

3 Labeling Abstraction Nodes and Human Evaluation

In this chapter, we discuss our approach compared to existing studies for labeling abstraction nodes. In Section 3.1 and 3.2, we introduce important concepts, related works and what we did to advance them. Section 3.3 presents our approaches for cluster naming, Section 3.4 describes our experimental design, Section 3.5 presents the technique evaluations, and finally, Section 3.7 summarizes the chapter by mentioning our future direction.

3.1 Introduction

Understanding the source code of a software system is a prevalent and vital task for the developers because many software engineering tasks depend on program comprehension [14, 20, 67, 18]. It is difficult for an individual developer to develop an enterprise software system on their own. Therefore, when someone is assigned to a task or join a development team, they need to understand the existing system to get used to the system. This program comprehension involves a lot of browsing back-and-forth between different granularity levels of the codebase. To reduce developers' effort to comprehend program artifacts, a lot of research is going on in the field of program comprehension [18, 19, 31, 24]. An abstract representation of the target software system can easily guide the exploration of low-level source code depending on developers' maintenance tasks. One of the approaches is to generate dynamic logs of function executions while running an existing system on different test cases. The logs can then be used with other methods to produce a suitable output for developers to comprehend the software system [18]. Moreover, most of the dynamic approaches generate dynamic call graphs from the generated dynamic logs of various system scenarios. However, the problem with dynamic call logs is that they only consider the function executions invoked during the dynamic log generation of a target system based on the test cases. As a result, not all the functionalities of the target system are considered during the codebase investigation. Another problem with dynamic logs is that they generate billions of data points, which are mostly redundant. If someone wants to abstract the whole system for comprehension purposes, then using dynamic logs does not help much to cover the entire system. On the other hand, a static call-graph can be generated by extracting caller-callee relationships from source files. The benefit of a static call graph is that it is possible to have a target system's overall functionalities. The static call graph also resolves the problem of redundant data of the log generations.

A large portion of a developers' development time is devoted to understanding existing source codes [12, 36, 29]. Because without knowing the cognitive relation between source code with higher-level system

functionalities, it is difficult to perform different software maintenance tasks (e.g., debugging, feature addition, refactoring, and testing). So, browsing back-and-forth between different source files of a system is widespread among developers to comprehend an existing system. What developers usually do is that they first look for the name of a source file's functions to understand the intention of the functions [16, 56]. Therefore, the function names can be utilized for abstracting a system's higher-level functionalities. Moreover, existing studies suggested that [52, 46] sequence of function invocations can help extract usage scenarios or higher-level functionalities of a target system. Hence, having a tool that visualizes the cognitive mapping between source code and high-level functionalities and allows browsing through source code in a more informed way would help developers.

Manually browsing source code for locating concepts is a laborious task. As a consequence, a lot of existing studies have been done to map concepts with source code using dynamic execution logs. However, very few studies considered static call graphs and emphasized function names. Gharib *et al.* [19] proposed a technique based on static call graphs where concepts are mapped with source codes. The authors have presented a whole subject system as a tree where nodes represent concepts of the system. However, they have only applied the TFIDF technique to extract the concepts of a particular codebase. During concept location, they have just considered the name of the function as term. Another drawback of their study is that they have not conducted any use case analysis from users' perspectives. So how developers will be comprehending the source code of a software system is absent in the study for real-world cases.

These limitations of the existing work motivated us to investigate more details on the potential of this approach. We have applied one information retrieval technique, TFIDF, and two topic modeling techniques (LDA and LSI). In the previous study [19], the full function name is treated as a term for the TFIDF technique. Here, we introduce words in function names as another variation. In total, we have six techniques to evaluate, as each technique mentioned above has two variations (function name and words in function name) result. We have also performed a small scale user-study with five developers. We have used 12 clusters from three subject systems as use cases to evaluate our approaches. Developers have rated the summaries generated by each technique and provided their summary of each use case, which we used to assess our automatic techniques using the Pyramid metric. From our investigation, we have found that automatic labeling using TFIDF for words in method names as term variation has an average of 64% overlap with manual labeling of participants. LDA and LSI received 37%, and 23% overlap accordingly. We have also found that words in function name variants got a minimum of 5% more preference rating compared to function name variants from developers.

In summary, our contributions are:

- We adopted two topic modeling techniques to name nodes of the abstract code summary tree.
- We introduce using words in the function name as a term for information retrieval techniques.
- We have conducted small scale user study to evaluate the proposed techniques.

3.2 Motivational Example

This section is presented with a motivational example of real-world scenarios. Suppose Bob joined a new company X as a Junior Software Developer. He needs to work on a software project which is being developed for more than six years. He must have a cognitive mapping between source code artifacts and high-level concepts of the software project, which will boost his integration to the project. To get an understanding of the project, he can use the Call graph of the project, which visualizes functional dependency.

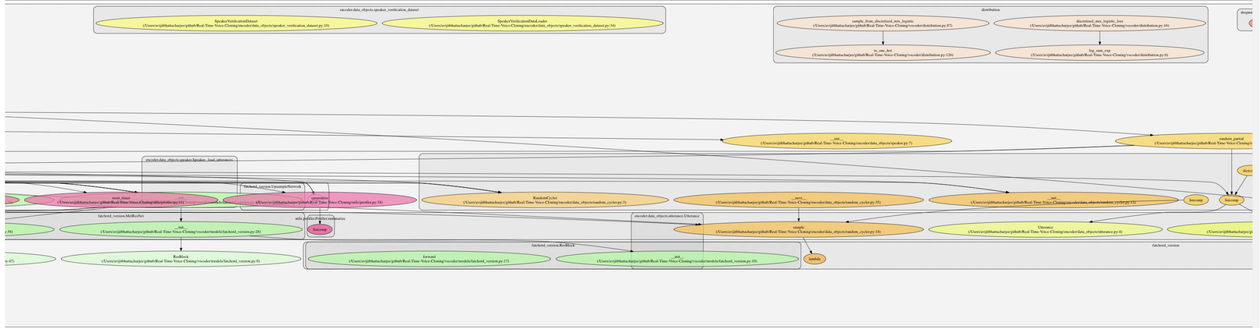


Figure 3.1: A portion of the Call graph of Real-Time-Voice-Cloning project by Pyan

However, in Figure 3.1 we can see a portion of the large call graph generated using Pyan [3] for Real-Time-Voice-Cloning [26] project. This presentation is very complex and hard to comprehend. Furthermore, if Bob has any particular Software Engineering task to do, first, he needs to locate the concept in source code. Locating source code artifacts relevant to the specific task will help Bob do his task faster. Therefore, our approach starts from this complex call graph and extracts concepts from execution paths in various hierarchical levels. Using the proposed approach, Bob can explore concepts from top-to-bottom, which at the end map to execution paths and the name of functions for smooth inquiries.

3.3 Approach

In this section, we discuss two significant steps in our approach with a brief discussion. First, in Section 3.3.1., we described six steps to get the cluster tree of a subject system. Second, in Section 3.3.2, we explain how we used different information retrieval techniques to label nodes of the abstract code summary tree. Data collection for evaluating the approach is depicted in algorithm 1.

3.3.1 Abstract Code Summary (ACS) Tree

The call graph is a visual representation of the relationships between the functions of a project. We adopt static call graphs, which are generated by analyzing source code. As the static call graphs capture all function calls of a target system, we choose to abstract the target system. Previous studies suggested that function

Algorithm 1: Constructing Python source code to an abstract code summary tree

```
1  Call Graph to abstract code summary tree (callgraph);
   Input  : Call graph
   Output: Abstract code summary tree
2  for Iterate each node in the call graph do
3      if Number_of_Incoming_Degree(node) == 0 then
4          entryNodes.append(node);
5      end
6      if Number_of_Outgoing_Degree(node) == 0 then
7          exitNodes.append(node);
8      end
9  end
10 for i ← 1 to entryNodes.length 1 do
11     for j ← 1 to exitNodes.length 1 do
12         execution_paths.append(simple_DFS_path(i, j));
13     end
14 end
15 for i ← 1 to execution_paths.length 1 do
16     for j ← 1 to execution_paths.length 1 do
17         distance_matrix[i][j] = consine_similarity(i, j);
18     end
19 end
20 cluster_tree = create_cluster_tree(distance_matrix);
21 abstract_code_summary_tree = generate_label_for_each_node(cluster_tree);
22 return abstract_code_summary_tree;
```

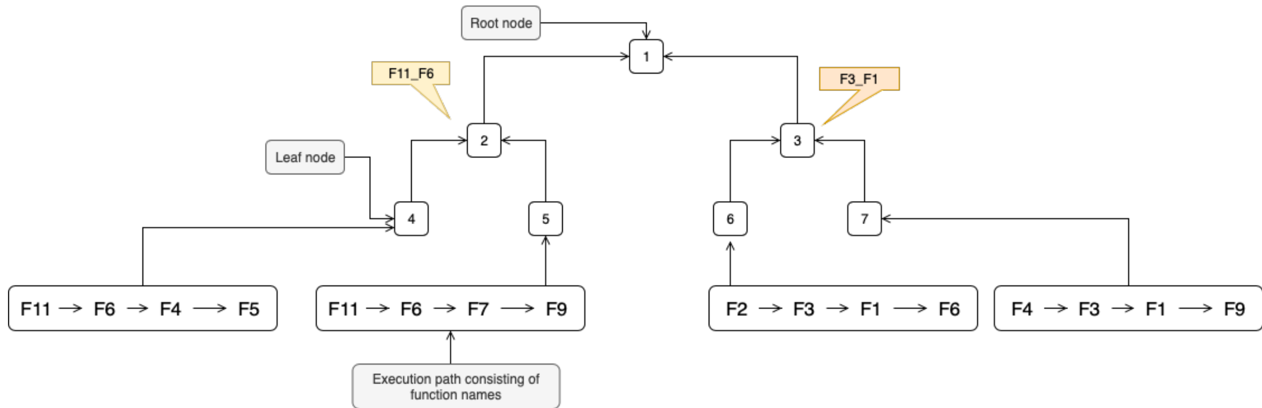


Figure 3.2: Structure of an abstract code summary tree

names contain significant abstraction of source code. Thus, we emphasize mining concepts by analyzing function names in the static call graph. As we want to capture and abstract the overall system's high-level concepts, therefore, the decision for adopting a static call graph as a building-block of our approach and using function names for concept location is well-justified.

In Figure 3.2, we present the structure of our proposed abstract code summary tree. The leaf nodes of this tree are directly mapped to the execution paths. The execution paths are a list of function names executed sequentially during the execution of a software system. For instance, node 5 is mapped to the execution path where `F11, F6, F7, and F9` are called sequentially. Similarly, in this scenario, all the four-leaf nodes 4, 5, 6, and 7 are mapped to four execution paths or function call sequences. Node 1, 2, and 3 are intermediate nodes of the tree. Naming these intermediate nodes analyzing the execution paths that resides under them might reduce the need to go through in detail about their functionalities. In the figure, node 2 has been named `F11_F6`, and node 3 has been named as `F3_F1` by analyzing the function names in the execution paths under those nodes. If we find a proper naming technique that can map concepts in source code with different granularity levels, this approach can make developers program comprehension tasks more flexible. In Figure 3.3, all the steps are visualized to generate ACS tree from source code.

Analyzing source code using modified Pyan module

For extracting function relationships from a python system, we used a modified version of Python module Pyan [3]. Pyan works only for a single directory. We adapted Pyan so that it can consider multiple directories while extracting the relationships. Pyan uses the abstract syntax tree (AST) for extracting function relationships. After analyzing the source code, we generated a graph in TGF (Trivial Graph Format). In TGF, all modules and functions' physical addresses in the source code are printed first. Then, relationships between all functions are presented as the caller and callee pair.

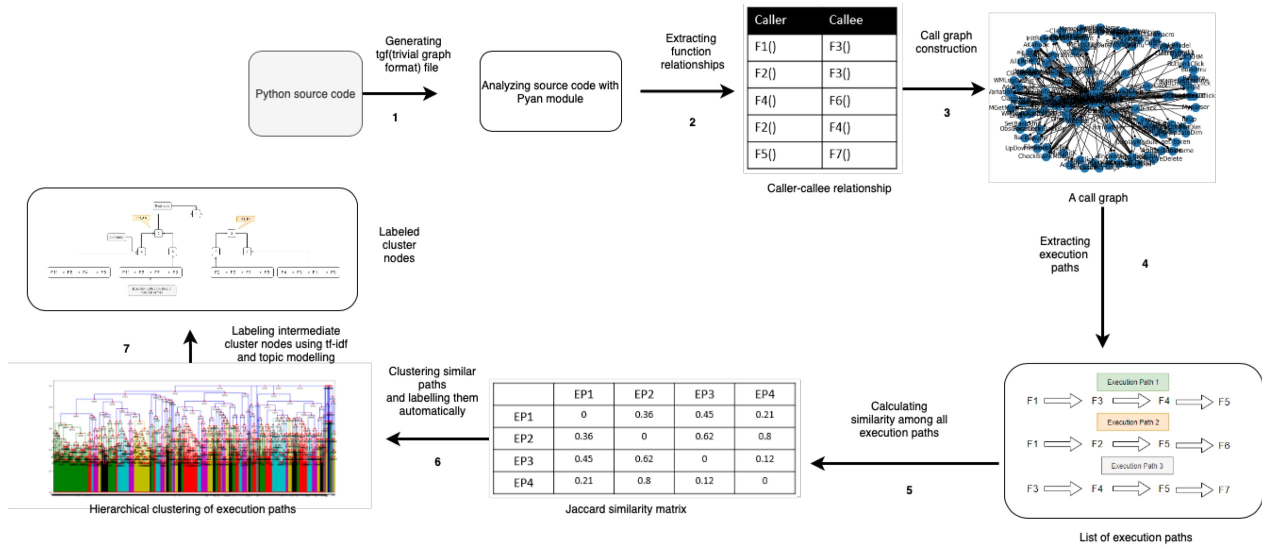


Figure 3.3: Overview of the overall approach

Extracting function relationships from TGF

Function relationships from the TGF file are used as inputs in our technique. Encoded unique identifiers are used to replace function names for ease of processing during the hierarchical clustering step.

Static call graph creation based on the extracted relationships

To perform different graph operations, we have created graph objects of the NetworkX [1] module using the extracted function relationships.

Extracting execution paths

The execution path is a simple path between an entry node and an exit node. An entry node is a node in the call graph which incoming edge degree is zero. Hence, no function is dependant on an entry node. An exit node is a node that has a zero degree of outgoing function calls. We have generated a list of entry and exit nodes to generate execution paths from a call graph. A simple path means no repeated node visit while visiting from the source node to the destination node. We have collected all possible simple paths for all possible combinations of entry node and exit node pairs. We have implemented a simple path finding algorithm from the NetworkX library, which uses a modified DFS algorithm for finding simple paths between a pair of nodes [1]. For our task, a source node is an entry node, and a destination node is an exit node.

Distance matrix for execution paths

For clustering execution paths (sequence of function names), we need to measure the similarity between all pairs of execution paths. For this purpose, we implemented the Jaccard similarity measure [41]. The linkage

algorithm uses this similarity in the next step. If we have two sets A and B, then their Jaccard similarity will be the ratio of their intersection's cardinality by the union. The clustering algorithms work on the distance, which is, in our case, the dissimilarity between two execution paths/clusters. We have subtracted the similarity score with one to get the dissimilarity value according to equation 3.1. After calculating dissimilarity between all pairs of execution paths, we converted the 2d matrix to 1d condensed matrix to make our program memory efficient.

$$Dis(A, B) = 1 - \frac{A \cap B}{A \cup B} \tag{3.1}$$

Clustering execution paths using linkage algorithms

To group similar execution paths as clusters, we have implemented a linkage algorithm using popular python package Scipy [27]. Scipy has different types of linkage algorithms already implemented in its core. To update the distance between two clusters, we have picked Ward the minimum variance method [38]. Equation 3.2 shows how distance using the Ward method is updated when two clusters from cluster forest are merged into a new one [27].

$$d(u, z) = \sqrt{\frac{(n_x + n_z)d(x, z)^2 + (n_y + n_z)d(y, z)^2 - n_z d(x, y)^2}{n_x + n_y + n_z}} \tag{3.2}$$

In equation 3.2, u is a newly formed cluster, and z is an unused cluster which will be used as reference to calculate distance. n_x, n_y and n_z are respectively the number of execution paths (as we are clustering the execution paths) in cluster x, y and z. When a new cluster u is created, the distance between u and all the other clusters are updated in the distance matrix. Additionally, cluster x and y are removed from the distance matrix as they have been merged as a new cluster u. This step is followed iteratively until only a single cluster remains in the cluster forest.

For example, in Figure 2.3, initially, at the start of the clustering process, there are four clusters 4, 5, 6, 7. Next, the hierarchical clustering algorithm selects the two most similar clusters (4, 5) to merge them as a new cluster 2. Now, in the clustering process, we have three clusters 2, 6, 7. Similar to the previous step, the most two similar clusters are merged into one. This process continues until there is only one cluster left. Ward method is used to calculate distance between the newly merged cluster with others.

3.3.2 Naming Nodes in an Abstract Code Summary Tree

After getting a cluster tree from the previous step, our next step is to name the clusters to represent the high-level functionality of source code in a readable way. In this step, we will be able to locate high-level concepts in the ACS tree. However, each cluster has a list of function call sequences, and the function call sequences are called execution paths. Our challenge is to extract essential keywords from this collection so that developers can get an overview of the underlying high-level functionalities under the cluster. Naming the

source artifacts correctly, in our case, which is nodes in the abstract code summary tree, is the fundamental contribution of this work. Proper naming can help developers to comprehend a program promptly. Toward the naming, we have applied three popular techniques used widely in natural language summarization tasks. These methods try to find meaningful and significant topics from a set of documents. In our approach, a document is an execution path that contains a list of function names. All the execution paths under a cluster are considered as documents. A previous study used function names as terms in a document [19]. However, we want to see what happens if we parse the function names and use the words in function names and use them as a term in documents. We used both words in a function name, and method names approach for the three techniques. In Section 2.4, we have discussed TFIDF, LDA and LSI techniques to generate node label.

3.4 Experimental Design

This section will discuss the research questions that need to be answered regarding the abstract code summary tree, how we collected our subject systems for the experiment, and details about users who participated in this study.

3.4.1 Research questions

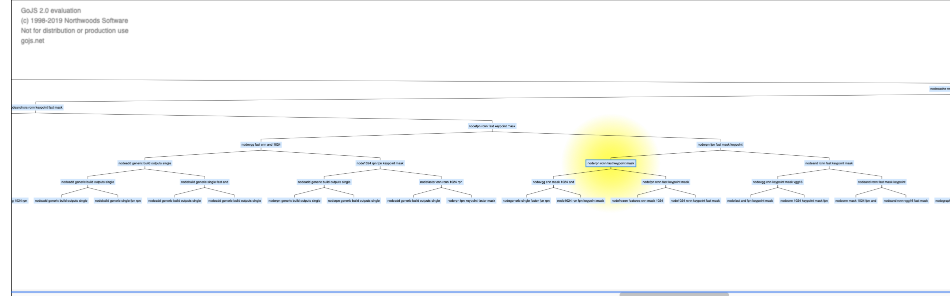
We want to explore how manual naming supports automatic naming techniques. To investigate this, we set `RQ1` described below. Besides, we have compared developer preferences for three different techniques using function names as terms by `RQ2`. Similarly, for `RQ3`, we changed the input for information retrieval techniques by words in function names instead of function names and compared developers' ratings among the three approaches. Finally, we want to see the performances of our two variations of choosing terms by a systematic comparison by `RQ4`. The four research questions correspond to the overarching research questions 1, 2 described in Section 1.3.

- `RQ1` How well does the automatic labeling perform using the candidate approaches compared to manual labeling?
- `RQ2` How do developers evaluate different labeling approaches based on function names?
- `RQ3` How do developers evaluate different labeling approaches based on words in method names?
- `RQ4` How can we compare the preferences of developers between the two approaches addressed in `RQ2` and `RQ3`?

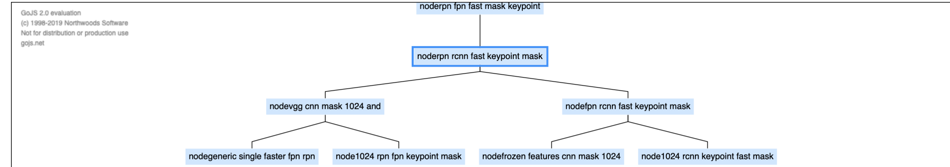
3.4.2 Dataset Collection

In order to conduct the user-study, we have collected source code of three popular Python projects *Detectron* [21], *Real-Time-Voice-Cloning* [26] and *requests* [4]. The reason behind choosing these subject systems

Full diagram



Local diagram



(a) Full abstract code summary tree with local view

Cluster ID: 4929 ×

Description of the cluster functionality

Task of this cluster is running different variation of rcnn model like keypoint, mask, fast, generalized and also building generic detection model and building data parallel model. Also the models are run using GPU and Cuda library.

Technique 1

keypoint fpn fast and build

◯1 ◯2 ◯3 ◯4 ◯5

Technique 2

fpn_rpn keypoint_rcnn mask_and_keypoint_rcnn mask_rcnn rpn

◯1 ◯2 ◯3 ◯4 ◯5

Technique 3

get,detect,test,net,eval,

◯1 ◯2 ◯3 ◯4 ◯5

Technique 4

build_generic_detection_model,single_gpu_build_func,add_generic_rpn_outputs,genera

◯1 ◯2 ◯3 ◯4 ◯5

Technique 5

detect,test,get,net,eval,

◯1 ◯2 ◯3 ◯4 ◯5

Technique 6

build,add,rpn,model,generic,

◯1 ◯2 ◯3 ◯4 ◯5

Your Summary from the description:(pick five significant words from the description and use comma to seperate them)

Any Comments:

(b) Form presented to the participants for answering

Figure 3.4: Tool UI presented to the study participants

for our study is that they are popular among Python developer communities. These projects follow the standard conventions of software developments so participants will be able to relate keywords from their day-to-day knowledge. Additionally, open-source projects tend to follow proper function naming conventions, which is important for our approach as it completely depends on function names. We extracted source code and applied the steps described in Section 2. We have printed clusters with their corresponding execution paths and names suggested by the candidate techniques in a file for doing the user-study. We have chosen 12 clusters semi-randomly, i.e., four from each of the subject systems, which ensures the coverage of different levels' clusters.

Table 3.1: Pyramid score computation

	response	request	dict	send	from	build	cookiejar	create	get	cookie	prepare	merge
D1	x	x	x	x	x							
D2	x		x			x	x			x		
D3	x					x		x	x	x		
D4				x					x	x	x	
D5	x		x					x		x		
TFIDF_word	x(4)			x(2)								x(1)
LDA_word		x(1)							x(2)		x(1)	
LSI_word		x(1)		x(2)			x(1)		x(2)			

3.4.3 User-study

For the 12 clusters, we manually analyzed each cluster's execution paths and come up with a 2-3 line description of what happens inside the clusters. Before the study, we told users to rate the automatic summaries of our three techniques, each with two variations. Additionally, we also provided a text box for the participants to select five keywords from the description produced by manual analysis of execution paths. We have used this summary to compute the Pyramid score for the three techniques of the words in function name variation. A total of five persons participated in the study with a software development background. Among them, three are female, and two are male. Each of the participants has at least a Bachelor's degree in Computer Science. Two of them are graduate students, and the other three are working as developers in three different software firms. All of them have at least three years of experience in programming experience with an average of 3.8 year.

In Figure 3.4a, the abstract code summary tree from our tool is presented. The upper box contains the full abstract code summary tree. Below the concept tree, a local view can be used to look closer to the concept cluster. Developers can click on any node in the concept diagram to get a zoomed view of its child and parents.

In Figure 3.4b, a screenshot of the form provided to the participants is presented. When participants right-click on the target clusters, a form with cluster id and a brief description of the execution paths' manual analysis is popped up. In the form, we asked the participants about their preference (1 means least preferred, 5 means most preferred) for names suggested by the six techniques and selected five keywords from the

descriptions to complete the study.

3.5 Results and Discussion

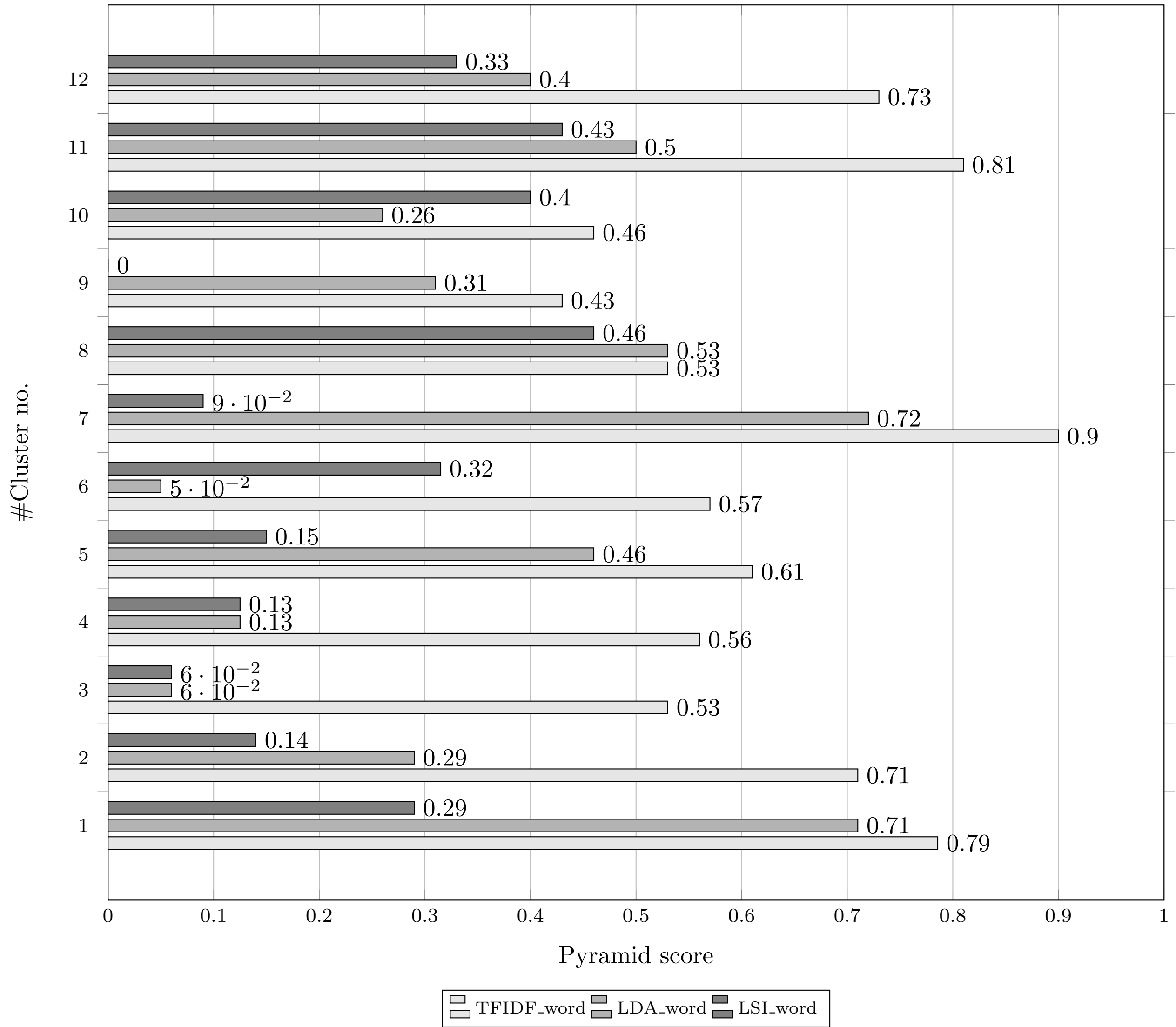


Figure 3.5: Pyramid score of the 12 clusters

3.5.1 User Naming vs. Automatic Naming

To investigate how automatic naming accords with manual naming, we have used Pyramid score [40]. Pyramid score is used in natural text summarization tasks to compare an automatic summary with a manual summary. Haiduc et al. [22] used Pyramid score to support source code summary with developers' summary, which motivated us to adopt Pyramid score to find out how our automatic approaches of abstraction harmonize

with developers' selections. In Table 3.1, we have shown the Pyramid score calculation process for a cluster (i.e., cluster number 10 of the 12 clusters). The preferences of five developers who participated in this study are represented by D1,. . . , D5. X_word represents the corresponding outputs of $X \in TFIDF, LDA, LSI$ by considering words in function names. Each column presents unique keywords from the selections of five participants. Furthermore, we have marked which words are matched with the automatic summary from a developers' summary in the corresponding cells. In each row for the automatic techniques, we have put the support from five developers for keywords being present in the summary.

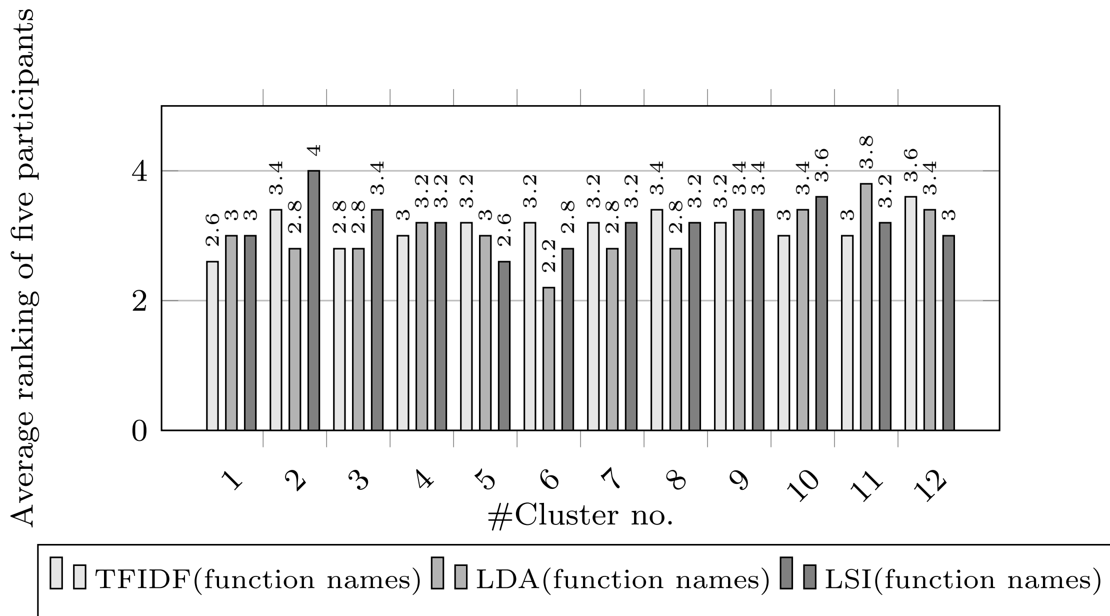


Figure 3.6: User preference among three implemented naming techniques (considering methods as terms)

For example, we can see that keyword *response* is present in TFIDF with words in the function name variation, and four of the developers picked *response* in their summary. So, support for keyword *response* is given 4. To get the Pyramid score for cluster number 10, we have summed each keyword's support in automatic naming by developers. In this case, values are $4(response), 2(send), 1(merge)$. We divide the sum of these support values by the top five most frequent keywords of five developers' summary. So, the score is now $(4 + 2 + 1)/(4 + 4 + 3 + 2 + 2) = 0.466$ for cluster 10. Greater Pyramid score means that the automatic naming is becoming more human in our case. In Figure 3.5, we have plotted Pyramid score for 12 clusters with the three techniques of word variant and support of the five participants for them. In the figure, we can see that for most of the clusters, the `TFIDF_word` based automatic naming technique's summary agrees more compared to other techniques with the developers' provided summaries.

3.5.2 User Rating on Function Name Variant

To answer the RQ2, we use the techniques with function names as unit variation. We asked our participants to rank each technique's summary with a score ranging from 1 to 5 to reflect how well they support the manual description. In Figure 3.6, we have plotted the average ranking of the participants for 12 clusters with the techniques. In the figure, we can see the users preferred LSI naming technique over the LDA. LSI is preferred in almost 50% of the clusters. For clusters 1, 4, 9, participants' preference for LDA and LSI are the same. The reason is that both techniques provided a similar kind of summary for the cluster in the automatic naming process.

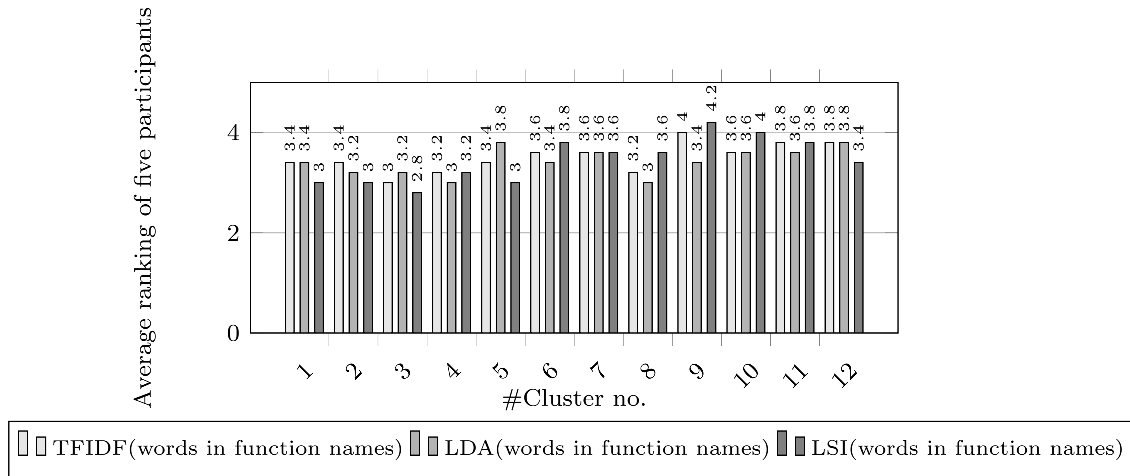


Figure 3.7: User preference among three implemented naming techniques (considering words in methods as terms)

3.5.3 User Rating on Words in Function Name Variant

We have followed a similar approach to answer RQ3 that we used to answer RQ2. We averaged five participants' rankings for 12 clusters for the three techniques (TFIDF, LDA, LSI). In RQ3, we want to know participants' preference when we consider words in the function names as unit for the TFIDF, LDA, LSI-based techniques. In Figure 3.7, we have plotted user rankings of the automatically suggested names for 12 clusters. Among twelve clusters, we can see that in seven of them, developers preferred names suggested by TFIDF and LSI technique in preference to the LDA technique, which covers almost 60% of the clusters. Therefore, `RQ3` can be answered to establish that words in function name variation perform better with TFIDF, LSI than LDA.

3.5.4 Function Name vs. Words in Function Name

For `RQ4`, we want to see users' preference on TFIDF, LDA, and LSI based techniques of the two variations we mentioned in `RQ2` and `RQ3`. So, we averaged the user rankings of 12 clusters of three techniques from Figure

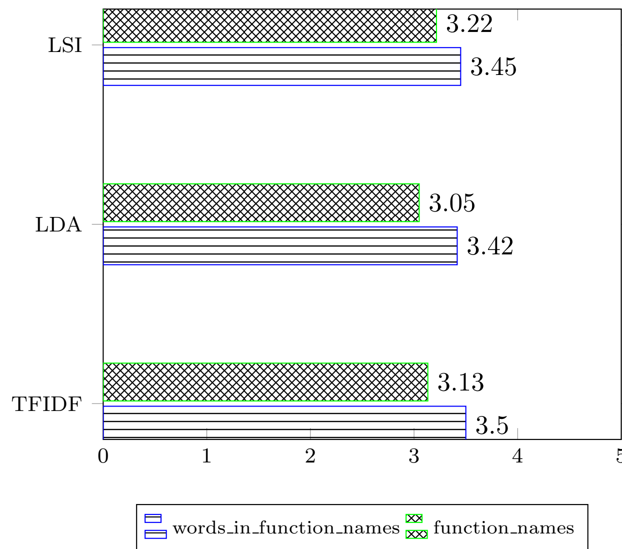


Figure 3.8: Comparison between three techniques considering function names and words in function names

3.6 and Figure 3.7. In Figure 3.8, we plotted the average ranks of the three techniques in two variations (i.e.,function_names and words_in_function_names). From the figure, we can observe that developers preferred TFIDF, LDA, and LSI techniques with word as unit over method name variations. Words in function names variation get at least 5% higher preference than the method names variations for each of the three techniques.

3.6 Threats to Validity

We have used three subject systems for the user study, and all of them are written with the Python language. We acknowledge that our user sample size is small. To mitigate the effect of randomness, we used three different systems, considered four clusters from each of them and invited experienced developers for the study. Our approach depends on function names. Therefore, our approach would be less successful when the naming conventions are not properly followed. We have used open-source projects which generally maintain good naming conventions. We have collected user summary after they evaluated six techniques to understand the limitation of automatic naming and provide feedback accordingly.

3.7 Summary and Discussion

While proposing an approach to find concepts in source code from static call graph analysis, we try to remove the shortcomings of existing approaches in terms of techniques evaluation and use cases. We use two different variations of terms to recommend concepts that leverage developers' program perception effort while understanding a system. As program comprehension is a subjective matter, we collect user data to evaluate

how our automatic labeling approach accords with user choice. The techniques we use are TFIDF, LDA, and LSI, with two variations (i. e., naming by function names, and naming by words in function names), where we found the TFIDF works better in cluster naming, and users prefer words in functions variants.

During our manual analysis to generate a brief description of twelve clusters by observing execution paths, we found patterns in execution paths that might make the naming of concept cluster more human. In the next chapter, we will explore how we can generate more information for abstraction nodes.

4 Providing Summary and Significant Patterns for Abstraction Nodes

In this chapter, we briefly discuss how we generated summary and execution patterns for the abstraction nodes. In Section 4.1, we discuss the importance of providing additional information for each abstraction node. Section 4.2 describes how the proposed approach works. In Section 4.3, an exploratory case study is reported to validate our proposed techniques.

4.1 Introduction

One of the crucial parts of a software engineering job is software maintenance. Usually there are four types of software maintenance tasks, such as perfective, preventive, corrective, and adaptive [66]. To perform all of these tasks, developers first need to understand the target system, how its different components work together, and locate the relevant classes, methods, and files for completing a specific task. To add or change something in the system accurately and adequately, developers need to understand how its different components work together and map the implementation level source code to high-level features. Proper tool support for program comprehension can reduce the manual and economic cost of software maintenance, which will result in cheaper software [6]. In the literature, the studies on program comprehension are divided into two parts [32]. First, how developers understand a code snippet. Second, understanding how large software systems are comprehended. Levy et al. [32] conducted a study to find how comprehending a large system works from an experienced developer's perspective. The comprehension of a system has a conceptual and concrete level [8, 32]. In reverse software engineering, different tools are used to extract implementation level architecture from source code (call graph). Later, through manual analysis, they are mapped to concept level architecture, which helps cognitive mapping [51]. However, as software systems are getting more complex in size, manual analysis of implementation level architecture to high-level concepts requires more human resources. In most cases, they are exhausting.

Studies [13, 18, 49, 63] on processing call graphs to facilitate overall system comprehension are very common in literature. The dynamic call graph is used for most studies, which is appropriate for specific test cases or scenarios. The problem with the dynamic call graph is they have redundancy problems and cannot capture the whole software systems [19]. Recently research on overall system comprehension focused on static call graph took attention [19, 60]. Execution paths from static call graphs [46, 52] can be used to

extract usage scenario or high level functionality. Clustering execution paths from both static and dynamic call graph pave the way for the abstract code summary of the system [18, 19]. We argue that labeling nodes of an abstraction tree can aid developers in using different program comprehension models. For example, the Bottom-up model is used by developers when they do not have any knowledge about the domain of the system. They gradually try to map low-level properties to high-level concepts. Developers can use the cluster tree of execution paths to facilitate Bottom-up cognition. The clustering starts from execution paths (low-level features) to a gradual grouping of similar paths, which are high-level features. Similarly, the abstraction tree can help automate the top-down cognition model.

In the top-down model, when developers have domain knowledge of a system, they try to map the knowledge to low-level implementations. The cluster tree hierarchically abstracts the features so that we have domain knowledge at the top of the tree that we can relate to low-level features by browsing the tree in a top-down manner. From our manual investigation to the proposed approach of Gharib et al. [19], we found that the abstraction tree has the potential to support program comprehension models automatically. However, they only used top-5 function names from the execution paths as the abstraction node label. We found that labeling the abstraction node properly with supporting documentation and example can make the abstract code summary tree more attractive and comprehensive to the developers.

- First, we experimented with labeling the nodes using TFIDF, LDA, and LSI information retrieval techniques. Previous studies only used the TFIDF technique.

- Second, we generated natural text descriptions for each node by summarizing comments from the execution paths' methods.

- Third, inspired by Feng et al. [18], for each node, we attached significant patterns from execution paths by applying Sequential pattern mining. To validate our techniques, we conducted an exploratory case study with three subject systems to find how these techniques can automatically help developers in program comprehension.

Our investigation shows that providing a natural text description and sample execution patterns increase the comprehensibility of abstraction nodes.

4.2 Approach

In this section, we discuss two significant steps in our approach with a brief discussion. First, we described six steps to get the subject system's abstract code summary tree in Section 4.2.2. Second, in Section 4.2.3, we describe how we used different information retrieval techniques to define the tree's hypothetical abstraction nodes. Data collection for evaluating the approach is depicted in algorithm 2.

Algorithm 2: Python source code to abstract code summary tree with node title, summary and execution patterns

```
1  Call Graph to abstract code summary tree (callgraph);
   Input  : Call graph
   Output: Abstract code summary tree
2  for Iterate each node in the call graph do
3  |  if Number_of_Incoming_Degree(node) == 0 then
4  |  |  entryNodes.append(node);
5  |  end
6  |  if Number_of_Outgoing_Degree(node) == 0 then
7  |  |  exitNodes.append(node);
8  |  end
9  end
10 for i ← 1 to entryNodes.length 1 do
11 |  for j ← 1 to exitNodes.length 1 do
12 |  |  execution_paths.append(simple_DFS_path(i, j));
13 |  end
14 end
15 for i ← 1 to execution_paths.length 1 do
16 |  for j ← 1 to execution_paths.length 1 do
17 |  |  distance_matrix[i][j] = consine_similarity(i, j);
18 |  end
19 end
20 cluster_tree = create_cluster_tree(distance_matrix);
21 abstract_code_summary_tree = generate_label_summary_pattern_for_each_node(cluster_tree);
22 return abstract_code_summary_tree;
```

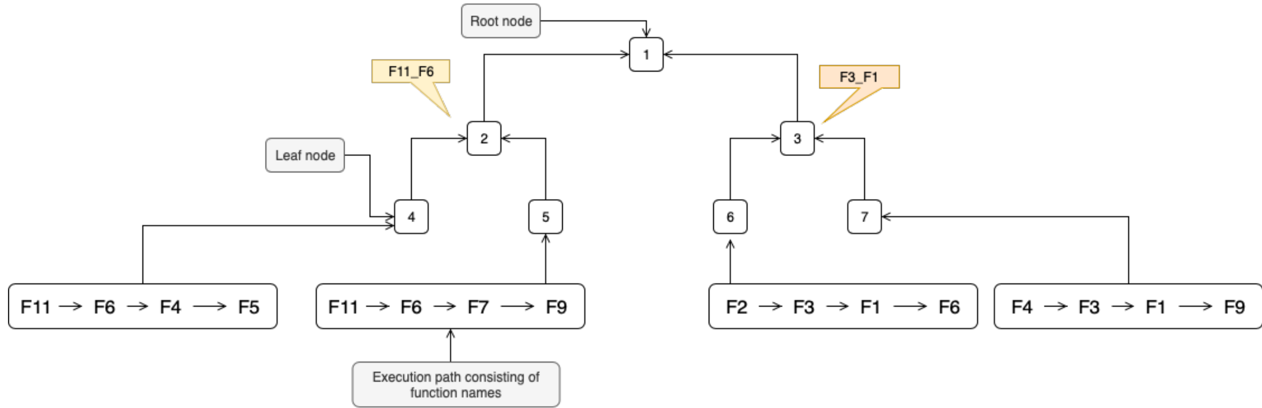


Figure 4.1: Structure of a abstract code summary tree

4.2.1 Abstract Code Summary Tree of a Software System

A call graph is a visual representation of a software system's method invocation relationships between different methods. We adopted a static call graph, which is generated by analyzing source code. As a static call graph captures whole function calls of a target system, we choose to abstract the target system. Previous studies suggested that function names contain significant abstraction of source code. Thus, we emphasized mining keywords by analyzing function names in the static call graph. As we wanted to abstract the whole system's high-level functionality hierarchically, therefore the decision to adopt the static call graph as a building-block of our approach is well-justified.

In Figure 4.1, we presented the abstract code summary tree structure. The leaf nodes of this tree are directly mapped to the execution paths. Execution paths are a list of function names executed sequentially during the execution of a software system. For instance, node 5 is mapped to the execution path where F11, F6, F7, and F9 are called sequentially. In this scenario, all the leaf nodes (4, 5, 6, 7) are mapped to four execution paths or function call sequences. Node 1, 2, and 3 are hypothetical abstractions of the four leaf nodes. Generating meaningful descriptions for these intermediate nodes can make the abstraction tree helpful towards different program comprehension cognition models. In the figure, nodes 2, 3 have been labeled F11_F6, F3_F1 respectively. These labels are generated by analyzing their child nodes' function names. We plan to generate five keywords for each intermediate node, alongside a short natural text description (from the docstring of function names) and few significant patterns from analyzing execution paths under investigation.

4.2.2 Source Code to Abstract Code Summary (ACS) Tree

In Figure 4.2, we visualized the six steps required to get ACS tree. First, we collected all the Python files from the source code of the subject system. Second, we analyzed abstract syntax tree of the Python files for extracting caller-callee relationships. Third, we build a static call graph from the extracted caller-callee relationships where nodes are methods and edges are relationship between methods. Fourth, we extract

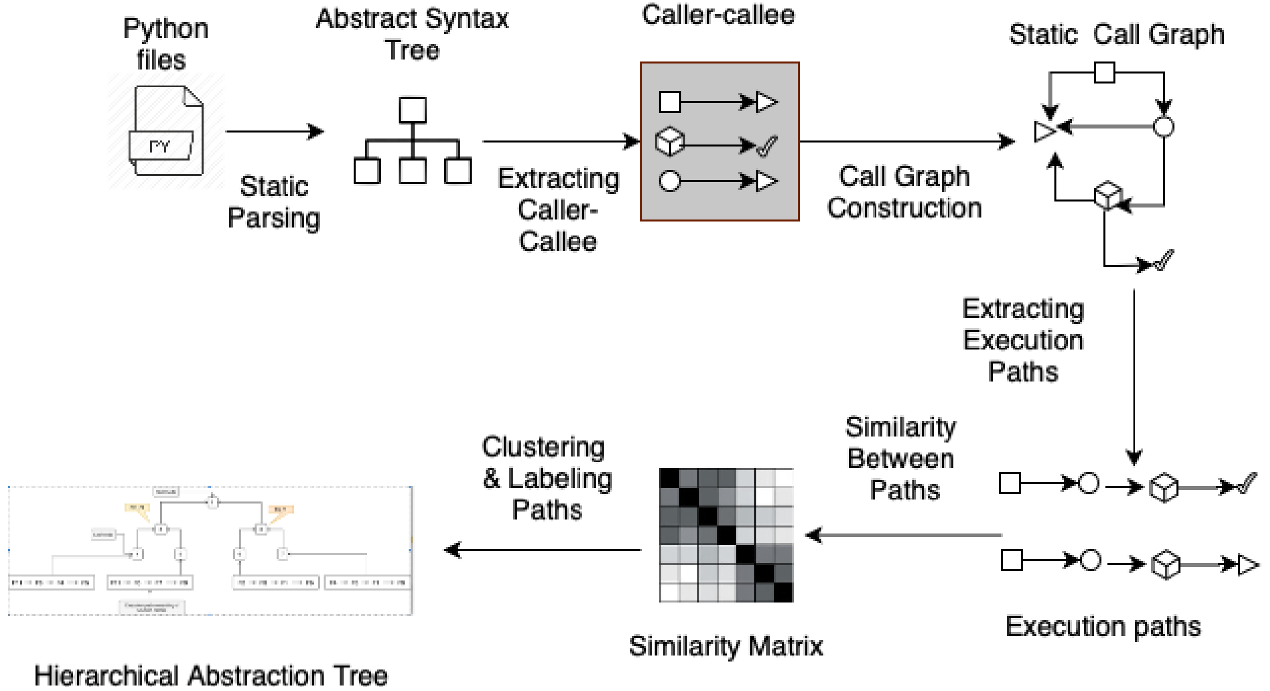


Figure 4.2: Overview of the overall approach

execution paths from the static call graph. Fifth, we computed pair-wise similarity between all pairs of execution paths. Sixth, we generated abstract code summary tree by clustering execution paths. We have discussed the steps in details in Section 3.3.1.

4.2.3 Generating Information for Abstraction Nodes

After getting a tree by clustering execution paths in the previous step, we generate three types of summaries for each intermediate node. First, we used different information retrieval techniques like TFIDF, LDA, and LSI for selecting five keywords or five function names from analyzing execution paths descendant to an intermediate node. This information is the title of the abstraction nodes. Second, this time instead of considering the function names, we considered the function names' comments to provide natural text summary for each intermediate node. Comments from the functions are summarized using TextRank [7] algorithm. Given a collection of sentences as input, this algorithm can summarize the collection to a fixed number of sentences. Third, inspired by Feng et al. [18], to provide a glimpse of the significant patterns among execution paths SPAM (sequential pattern mining) algorithm PrefixSpan [23] is implemented. We find that all the execution paths in an intermediate node share some patterns from our manual investigation of the execution paths. By taking a look at the significant patterns, we can comprehend more elaborately about the intermediate nodes. We present these patterns in support of the Label and Summary generated by the previous two steps. Therefore, to comprehend an abstraction node, we have a label, summary description,

and patterns from the execution paths. In Section 2.4, we have discussed TFIDF, LDA and LSI techniques to generate node title. Below we discussed node summary and execution patterns generation techniques.

Node Summary for Abstraction Nodes

To generate a summary for node 3 of Figure 4.1, we collect the first line of docstring comment for the function F1, F2, F3, F4, F6, F9 as they consist of the execution paths of node 3's descendant nodes. Next, we remove duplicates from the comments and provide these sentences to TextRank [7] algorithm to generate summary. There are many functions in an execution path for real-world software, so using the TextRank algorithm, we get a short five sentence comprehensive summary.

TextRank [7] is a graph-based automatic summarization technique. TextRank is language and domain-independent. To generate a summary, training a corpus is not required, making it suitable for our task. All the sentences of the target document make the nodes of a graph. Edges between the nodes are created using different similarity measures between two nodes or sentences. At last, the PageRank [42] algorithm is used to obtain a summary from the graph.

Execution Patterns for Abstraction Nodes

To get significant patterns for node 3 in Figure 4.1, we have to analyze execution paths of node 6 and node 7. The execution paths of node 6 and 7 have $F3 \rightarrow F1$ sequence common. So, presenting this common sequence as a significant pattern for node 3 make a good abstraction of descendant execution paths of node 3. To mine this sequential patterns, we implement PrefixSpan [23] sequential pattern mining algorithm. If we provide a collection of execution paths to PrefixSpan, it gives a significant pattern analyzing the collections. PrefixSpan creates a prefixed based projection database to find sequential patterns efficiently.

4.3 Experimental Design

This section will discuss research questions that drive this study, how we collected our subject systems, what criteria were considered, and how we designed our exploratory case study.

4.3.1 Research Questions

In this study, we tried to improve the comprehensiveness of the abstraction of nodes. First, we split function names to get words so that TFIDF, LDA, and LSI methods perform naturally. There is also another benefit of using words in method names as they will be fixed length. We investigate how effective node names are using the word variant in our RQ1. Besides, we attach a natural text summary for each node using the docstring of functions, which consists of RQ2. Similarly, we generate significant patterns from execution paths to support node comprehension, and this is our RQ3. Finally, we investigate how merging the results

of RQ1, RQ2, and RQ3 improves the abstraction tree in RQ4. The four research questions correspond to the overarching research questions 3, 4 described in Section 1.3.

- `RQ1` How effective is the word variation of TFIDF compared to method variation?
- `RQ2` How comprehensive is the natural text summary for abstraction nodes?
- `RQ3` How effective are the mined patterns to comprehend abstraction nodes?
- `RQ4` How effective is the comprehension of an abstraction node, if label, summary, and patterns are used together?

4.3.2 Dataset Collection

In this study, we have experimented with three subject systems. We cloned the source code of the subject systems from their Github repository. We used the Pyan library to extract caller-callee relationships in trivial graph format (TGF). Next, we created a networkX graph object to iterate through the call graph and extract execution paths. Finally, the Ward linkage clustering algorithm was used to create an abstract code summary tree. In Table 4.1, we present the entry, exit nodes, line of code, number of execution paths. We chose our subject systems carefully to have three different execution paths as the number of execution paths determines how big the abstraction tree will be. We wanted to keep the size manageable for performing our analysis to find our proposed techniques' effectiveness.

Table 4.1: 3 Subject Systems with their No. Entry, Exit Nodes, LOC, Paths, And Date Retrieved

No	URL (https://github.com)	Name Nodes	Entry Nodes	Exit	LOC	Paths	Date retrieved
1	`Ourcode`	higher_level_abstraction	2	22	999	31	28 May 2020
2	`/davidfraser/pyan`	pyan	36	50	3711	637	28 May 2020
3	`/CorentinJ/Real-Time-Voice-Cloning`	Real-Time-Voice-Cloning	21	93	9117	404	28 May 2020

4.3.3 Case Study Design

To find the effectiveness of the proposed techniques, we carefully chose different abstraction nodes and their neighbourhood. After that, we manually checked whether the label, summary and mined patterns suitably abstract and describe the system's high-level concepts. To verify whether the approaches properly support our claim, we manually browsed the source code of target systems to know the systems' high-level concepts. To generalize our findings to some extent, we have used three different subject systems so that our claim is stronger.

4.4 An Exploratory Case-study

4.4.1 RQ1: Effectiveness of Word Variation Labeling

To see the effectiveness of labelling, we manually picked the root node and its neighbourhood. We explored a similar tree snippet for the three systems. In Figure 4.3, we see root node 60 has the label *lda pair get docstring jaccard*. From this label, one can guess that something related to docstring, Jaccard distance, and topic modelling LDA occurs in the higher_level_abstraction subject system. An interesting thing to notice is that name of node 60 and 59 is the same. Although node 58 is a child node of 60, which has two new keywords py and view that indicate something related to Python file and view occurs inside the nodes' execution paths. On the other hand, if we see the name for node 60 using TFIDF method variant (*pretty_print_leaf_node bfs_with_parent mining_sequential_patterns id_to_sentence cluster_view*), we see that using method as unit for TFIDF is more comprehensible than using word as unit for TFIDF. Another benefit of TFIDF method variant is for node 60 and 59; it provides different names according to their execution paths. On the other side, the word variant of TFIDF gives the same name for nodes 60 and 59 because of overfitting.

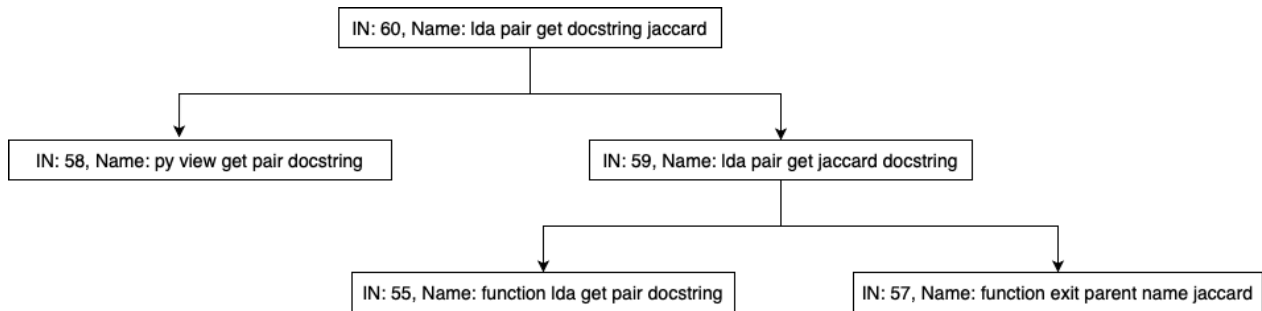


Figure 4.3: Snippet from subject system 1 (Our code)

In Figure 4.4 shows that we have a snippet of the Pyan subject system's abstraction tree. Pyan [3] is an open-source software which can extract call graph from a Python project. From our general knowledge, we can expect the concepts related to source code. If we look at node 1272 at Figure 4.4, the name is *c3 module label use idx*. Except for the module, other keywords are not that much expressive. For node 1268, we see keywords like class, node, namespace indicate that the node is relevant to processing source code. However, we can see a recurrent occurrence of the same name for nodes 1272, 1271, which is an over-fit situation. The names for nodes 1272, 1271 using method variant TFIDF are *write_edge find_filenames DotWriter*, *write_edge TgfWriter DotWriter visit_Assign* which clearly indicates some hint what the nodes do.

In Figure 4.5, we have a snippet from our third subject system (Real-Time-Voice-Cloning [26]). This open-source project can clone someone's voice from a clip of at least five seconds. So, this system's high-level functionalities can be converting wavelength, processing audio, training model. The name of root node 806

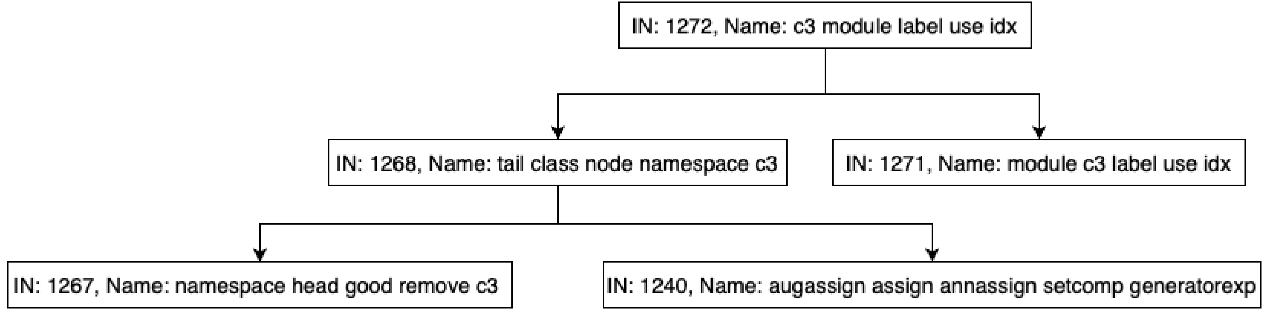


Figure 4.4: Snippet from subject system 2 (pyan)

is *synthesize train synthesizer synthesize toolbox*. Here, train indicates training models, synthesize means processing audio signal, and toolbox indicates the tool system. For node 791, we see keywords like encoder, spec which indicates processing of signals. Using method name variant of TFIDF the name for node 806 and 791 are *save_wav encoder.audio discretized_mix_logistic_loss profile_noise encoder.visualizations*, *current_encoder_fpath make_spectrogram load_preprocess_wav normalize_volume*. TFIDF method variant provides more contextual information from the name of node 806, 791.

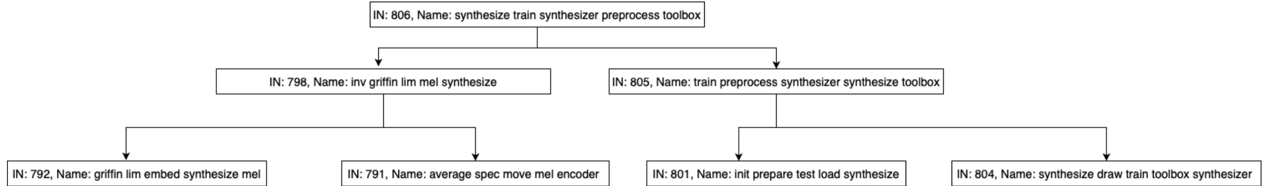


Figure 4.5: Snippet from subject system 3 (Real-Time-Voice-Cloning)

From the above manual investigation of node names using the method and word variant, it is evident that using method name variant provides more semantic abstraction in observed context. However, word variant provides a fixed-length name which is crucial for creating flexible ACS tree. The output for word variant in the top-level nodes are mostly similar. However, to use word variant we need to address ambiguity. Our informed guess is that it is possible to improve the output for word variant by using more appropriate similarity matrix and reducing redundant nodes in the ACS tree.

4.4.2 RQ2: Natural Text Summary for Abstraction Nodes

Natural text is more comprehensive than a few keywords. Therefore, to support abstraction nodes' comprehension in a hierarchical tree, we propose to summarize the methods' docstring in all execution paths of the node. As the number of lines in comment vary for methods, we used only the first line of the docstring. Also, from our manual analysis, it is evident that the first line describes the function's purpose most of the time. However, for many cases, we found that docstring is absent. In those cases, we just omitted the method for generating summary. To answer our RQ2, we investigated the summary for nodes in three subject systems.

The root node 60 of subject system 1 has the text summary *clustering execution paths using scipy Labeling a cluster using six variants This function returns function name with their docstring analyzing Python programs to build cluster tree of execution paths.* Subject system 1 is our program to cluster execution paths from a call graph. Then, we labelled the nodes in the cluster using six different techniques and also analyzed docstring to produce a summary as we discussed when introducing this research question. If we carefully observe the summary for node 60, using the TextRank algorithm, our produced summary represents very well what the first subject system does. For node 57, our approach's summary is *converting tgf file to a networkX graph extracting function names from TGF file analyzing Python programs to build cluster tree of execution paths.* From the summary, we can confidently tell that abstraction node 57 deals with extracting function names from the TGF file, converting TGF format file to networkX graph.

The root node 1272 of subject system 2 (Pyan) has the summary *Resolve those calls to built-in functions whose return values Return a label for this node, suitable for use in graph formats.* As Pyan deals with source code, we can see the summary saying something about resolving built-in functions and labelling nodes for graph format. We can relate this summary to the purpose of Pyan partially. For node 1271, the summary is *Try to determine the full module name of a source file, by figuring out Return the node representing the current class, or None if not inside a class definition.* The summary for node 1271 says that the execution paths it abstracted mostly deal with determining a source file module, getting the class name a node represents. These are some standard utilities for a project which process source code. The summary for node 58 is *Generate cluster figure from a dendrogram. Flattens a nested list. This function returns function name with their docstring.* Node 58 deals with plotting the dendrogram, mapping function name to docstring.

The root node 806 of subject system 3 (Real-Time-Voice-Cloning) has the summary *If this function is not explicitely called, it will be run on the Args: Computes where to split an utterance waveform and its corresponding mel spectrogram to obtain Derives a mel spectrogram ready to be used by the encoder from a preprocessed audio waveform.* As we have described previously, Real-Time-Voice-Cloning software can clone a voice to produce speech from text. If we see the summary generated by TextRank for node 806, we can say it deals with processing audio wave-forms. Furthermore, for node 801, the summary is *Args: Synthesizes mel spectrograms from texts and speaker embeddings.* Summary for node 801 is very small. It indicates that mostly docstring for Real-Time-Voice-Cloning is empty, and the short summary indicates text to speaker embedding, which is essential for voice cloning.

From the observation of the node summary generated by TextRank for the three subject systems, we can conclude that if functions are properly documented with docstring, this approach can complement the comprehensiveness of abstraction nodes. We faced the challenge of different formats of comments, which hampered the extraction of the docstring.

4.4.3 RQ3: Effectiveness of Mined Patterns from Execution Paths

From our manual investigation into the execution paths of an abstracted node, we find that there are recurrent patterns that can help comprehend the abstracted node. Therefore, we develop a technique to use sequential pattern mining for selecting patterns among the execution paths from those findings.

The patterns for root node 60 of subject system 1 are

- ClusteringCallGraph, python_analysis, clustering_using_ scipy
- ClusteringCallGraph, python_analysis, clustering_using _scipy, labeling_cluster
- ClusteringCallGraph, python_analysis, clustering_using_scipy, labeling_cluster, tf_idf_score_for_scipy_cluster

We can tell that node 60 works with Python code, clustering using scipy library, labelling the clusters from observing this pattern. As this is the root node of the subject system 1, we can conclude that the patterns represent the purpose.

The patterns for node 58 are

- ClusteringCallGraph, PlayingWithAST
- ClusteringCallGraph, get_all_method_docstring_pair_of_a_project
- ClusteringCallGraph, get_all_method_docstring_pair_of_a_project, get_all_py_files

From the patterns for node 58 retrieved by sequential pattern mining, we can see it extracts docstring from all Python files, which is one of the essential parts for answering our RQ2.

The patterns for root node 1272 are

- get_attribute
- resolve_builtins, get_attribute
- analyze_binding, resolve_builtins

From the list of patterns, we can see there is very little information. Although these patterns are for the root node, they are most frequent. Limiting the length of the minimum pattern can solve the problem. However, we can understand that getting attributes, analyzing bindings, and resolving built-ins is the most common concept for root node 1272.

The patterns for node 1240 are

- resolve_builtins, resolve_method_resolution_order, C3_linearize, C3_merge
- analyze_binding, resolve_builtins, resolve_method_resolution_ order, C3 _linearize, C3_merge

- resolve_builtins, resolve_method_resolution_order, C3_linearize, C3_merge, C3_find_good_head, LinearizationImpossible

From the patterns of node 1240, we can see that method resolution order, linearize, resolve builtins are the main task.

The patterns for root node 806 of subject system 3 are

- init, setup_events
- wav_to_mel_spectrogram
- embed_utterance
- train

From the patterns for node 806, we see that it is creating different events, converting wave to spectrogram, and training model, which summarizes what RealTimeVoiceCloning does. The patterns for node 804 are

- wav_to_mel_spectrogram
- encoder_preprocess
- embed_utterance
- encoder_preprocess, _preprocess_speaker_dirs, preprocess_speaker

From the patterns of node 804, we can say that node 804 is embedding and encoding audio signals, prepossessing speaker audios.

From observing patterns of different nodes from the three subject systems, we can conclude that providing them with an abstraction node can enhance a node's comprehensibility. However, tuning the minimum length of each pattern and removing frequency-based bias should be considered to improve the patterns.

4.4.4 RQ4: Effectiveness of Using Label, Summary and Patterns Together

In RQ1, we manually analyzed how expressive the label for nodes is using word and method variants. We found that method variation of the TFIDF technique provides a more sophisticated label than its word variant, which seems ambiguous. From our analysis of RQ2, we have seen a good summary for nodes using TextRank. However, this method's success largely depends on how well the method docstring is written, excluding unrelated information is a challenge due to different formations. From RQ3, it is clear that patterns from execution paths are helpful to support nodes, although effectiveness hugely depends on selecting tuning mining pattern algorithms. Therefore, if the challenges for generating a name, summary, and patterns are solved accordingly, they will enrich the comprehension of the abstraction node, in total, the overall abstract code summary tree.

4.5 Threats to Validity

We have picked three different subject systems of varying size so that our approach's effectiveness can be generalized to some extent. We manually analyzed the results of our techniques to reach a saturated decision. Furthermore, two of the authors of a paper submitted based on this experiment individually analyzed the findings to remove subjective biases. We carefully picked the first line skipping lines with special characters to extract the docstring for each method.

4.6 Summary and Discussion

In software engineering, program comprehension is an important research area that involves many other software maintenance tasks. Nowadays, software size and complexity are growing. To perform a maintenance task, developers need to understand how different components of the system interact. Other cognition models are studied in the literature to aid developers. Top-down and bottom-up models are popular program comprehension models. In these models, developers map high-level features with low-level implementations depending on a specific situation. Different hierarchical abstraction techniques which use call graph of dynamic and static variation exists.

This study focused on improving a software system's abstraction hierarchically using execution paths from a static call graph. Execution paths represent low-level implementation. Grouping execution paths in a cluster tree, a software system is hierarchically abstracted. Information presented with the nodes of a cluster tree is helpful for developers to map high-level features to low-level implementations. We proposed different techniques like using word and method variant for TFIDF to label nodes, generated a summary for each node from method docstring, and mined significant patterns to attach all these three types of information with each node to aid comprehension.

To evaluate our approach, we conducted an exploratory case study to determine our proposed techniques' effectiveness. We discussed the generated output for different nodes and challenges to improve. We found that generalizing the techniques with more subject system would improve the techniques.

5 Finding Effectiveness of the Abstract Code Summary Tree

In this chapter, we introduce our motivation to build the HCPC tool in Section 5.1. Next, in Section 5.2 we discuss different steps followed to improve ACS tree. In Section 5.3 and 5.4, we discuss the interface and implementation of the HCPC tool. In Section 5.5 and 5.6, we discuss how to use HCPC tool for two use cases and present an example using *jupyter_client* project. Finally, we present a human-subject study of the developed the HCPC tool in Section 5.7.

5.1 Motivation

Finding relevant methods, classes and files are frequent part of daily activities of a software developer. Most of the software maintenance tasks require to find relevant locations for solving the task. As the size of codebase grows, it becomes difficult to remember everything in detail. Therefore, common practice is to figure out some relevant keywords and search for the files containing the keywords. The problem with this approach is search results are random and it gives no idea of exploring the codebase according to the order different components are called.

In the previous studies, we have advanced the existing works on hierarchical abstraction of static execution paths by finding appropriate techniques to label nodes in the tree and further complement the nodes with natural text summary and execution patterns for better comprehension. In this study, our motivation is to make the abstraction tree usable for developers' daily concept location activities. We have found two areas for improvement on the previous studies. First, we observe the abstraction tree became complex to explore as the number of execution paths grows. Therefore, we implemented a cluster flattening technique to have more flexibility and simple structure with cut-off depth. Second, we have changed the similarity metric for comparing execution paths from Jaccard distance to match_a_strike score. By updating the similarity measure, we ensure more accurate grouping of clusters. After the technique changes, we have developed a tool called HCPC for doing a human-subject study to find the effectiveness of HCPC. In the HCPC tool, we added a node highlight feature where specific function can be selected to highlight relevant nodes. From our study with developers, we have found that the HCPC tool can be helpful for exploring the codebase in a guided way in daily software maintenance activities.

5.2 Approach

In this study, we have followed similar steps as study 1 and 2 except two changes. First, we have changed the similarity score from Jaccard distance to strike_a_match algorithm. The strike_a_match algorithm takes into account the contents of two lists and the sequence they appear. On the contrary, Jaccard distance only considers the content of two lists. To improve the clustering result, we have made the change in similarity metrics. Second, we have added one more step to reach the final abstract code summary tree. Previously we have used the step by step tree returned by a linkage algorithm. However, the linkage tree is not flexible for browsing. Therefore, we have used a cluster flattening technique to get more flexible 5-6 depth tree. We discuss strike_a_match, node summary, execution pattern and cluster flattening techniques in the following subsections.

5.2.1 Strike_A_Match Algorithm

In Algorithm 3, we provided the pseudo code for reproducing the strike_a_match algorithm[1]. The method takes input two lists which are execution paths one and two. The method returns a similarity score between 0 and 1 where 0 means no match and 1 means full match. In line 2-3, all method pairs in consecutive order are generated. In line 4, we calculate union value by summing length of the two generated pair lists. From line 6 to 14, we iterate over the pair lists and see if they match to calculate intersection value. When we find a match, we remove the pair from *ep2_pairs* to avoid considering the same match again. Finally, we return the similarity score using union and intersection values. The method considers the order in addition to the content of two lists.

5.2.2 Node Summary

In Algorithm 4, we provided the pseudo code for generating a node summary for each abstraction node. The two input of the method are execution paths and function_id to comment dictionary. The execution paths are in a 2D list where each row corresponds to an execution path and the cells contain function id. The second argument is a dictionary where function id are mapped to their first line docstring comment. In line 3 - 7, we iterate through all the execution paths and all functions in an execution path. We add all the comments to *all_comments* variable for use in summarize. In line 8, we provoke the summarize method from Gensim [48] library which by default returns one-third of the *all_comments* as summary using the TextRank [35] algorithm. We have tried with different ratios of input to summary and found the default settings sufficient for our purpose.

[1]http://www.catalysoft.com/articles/strikeamatch.html

Algorithm 3: Strike_A_Match algorithm

```
1  Compare_execution_paths
   Input  : ep1, ep2
   Output: similarity_score
   // method_pairs returns all the two length consecutive pairs from execution paths
2  ep1_pairs = method_pairs(ep1);
3  ep2_pairs = method_pairs(ep2);
4  union = len(ep1_pairs) + len(ep2_pairs);
5  intersection = 0;
6  for i ← 0 to len(ep1_pairs) do
7      for j ← 0 to len(ep2_pairs) do
8          if ep1_pairs[i] == ep2_pairs[j] then
9              intersection += 1 ;
10             ep2_pairs.pop(j);
11             break;
12         end
13     end
14 end
15 return ( 2 * intersection) / union
```

Algorithm 4: Generate node summary from execution paths of an abstraction node

```
1  Generate_node_summary
   Input  : execution_paths, function_id_to_comment
   Output: node_summary
2  all_comments = ' ';
3  for execution_path in execution_paths do
4      for function_id in execution_path do
5          all_comments += function_id_to_comment[function_id];
6      end
7  end
8  node_summary = summarize(all_comments);
   // summarize by Gensim
9  return node_summary
```

5.2.3 Execution Patterns

In Algorithm 5, we present pseudo code for generating execution patterns. The method takes execution paths as input and outputs frequent patterns found by analyzing all the execution paths. For our approach, we have mined the top-15 most frequent patterns. Execution paths is a list of lists of function ids which can be called sequentially. We use the PrefixSpan algorithm which mines frequent patterns from a set of lists. We use the topk method to get the top-15 execution patterns. We use default settings for maximum length and minimum length of the patterns.

Algorithm 5: Generate node summary from execution paths of an abstraction node

```
1 Generate_Execution_Patterns
  Input  : execution_paths
  Output: execution_patterns
2 NUMBER_OF_PATTERNS = 15;
3 ps = PrefixSpan(execution_paths);
4 top_patterns = ps.topk(NUMBER_OF_PATTERNS);
5 return top_patterns;
```

5.2.4 Cluster Flatten Technique

In previous studies, we have used a step-by-step clustering tree as the abstract code summary tree. However, for n number of execution paths, the abstraction tree will have $2n+1$ nodes which is not practical for medium to large projects. Therefore, we processed the cluster tree by using cluster flattening. The cluster flattening technique groups all clusters between a given distance as one. Therefore, by giving a larger distance, we can get very few clusters from a linkage matrix by merging all clusters between the distance as one. Similarly, we can get a larger number of clusters by using a small distance as the threshold value for flattening. For generating nodes at different depths, we use increasing threshold value for distance (e.g. [5, 4, 3, 2, 1.5]). We have manually tuned the distance values for individual subject systems.

5.3 Implementation

In this section, we briefly highlight different parts of our HCPC tool[2] implementation as shown in Figure 5.1.

1. We clone the source code from GitHub in a temporary folder. The source code will be used in the next phase by the Python static code analyzer.

[2]https://hcpc.usask.ca

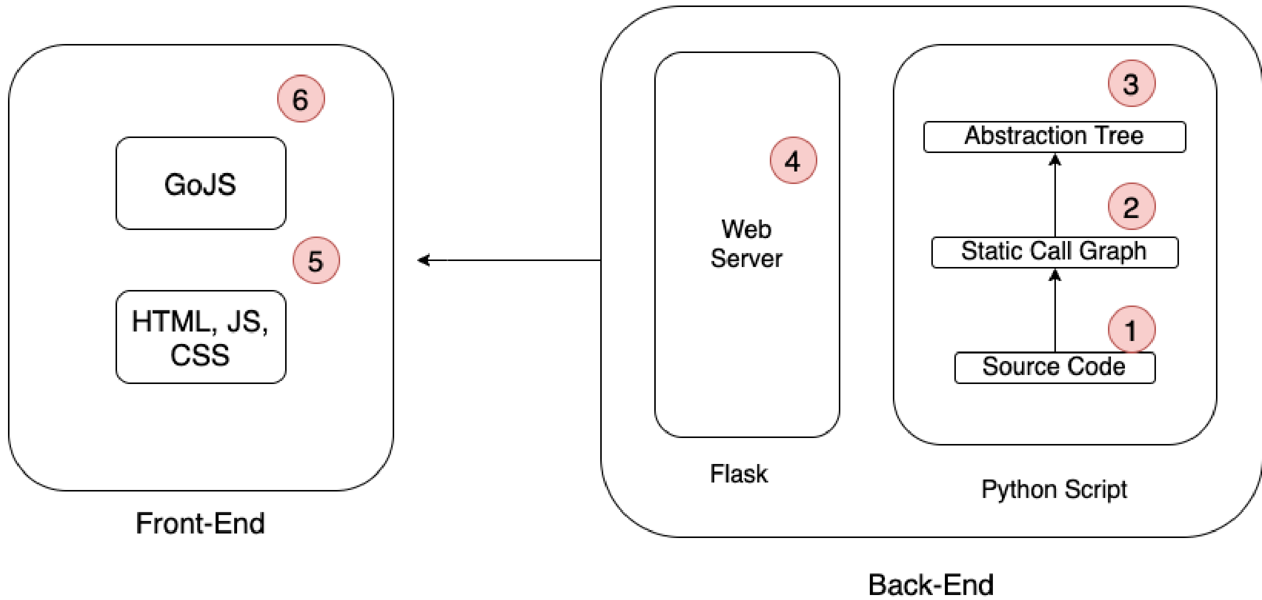


Figure 5.1: Architecture of HCPC tool

2. We use Pyan [3] as static Python code analyzer. Pyan goes through all the **.py* files looking for which method calls which method. Pyan generates a text file which encodes all the methods with numbers and then contains which method calls which method. We generate static call graph using NetworkX [1] with the caller-callee relationships generated by Pyan.

3. We generate execution paths from the call graph created in previous step. Execution paths are grouped using the Agglomerative Hierarchical Clustering (AHC) algorithm provided by the Scipy [27] library with *ward* method as a distance metric. We have a binary tree structure where leaf nodes are execution paths and other nodes are clusters at different levels. We call these cluster nodes abstraction nodes. The abstraction nodes have a collection of execution paths. For each abstraction node, we generate three properties. For each node, we create node title by applying information retrieval techniques (Scikit-learn [44] for TFIDF and Gensim [48] for LDA, LSI) on the method names of all execution paths of a node. Then we produce node summary by summarizing (TextRank by Gensim) method comments of all the execution paths of the node. Last we generate execution patterns by pattern mining among the execution paths of the node (PrefixSpan [2]). We write all the node data in a text file. Data is written in JSON format where each node is keyed with their ID and they have parent_id, node title, node summary, execution patterns and execution paths associated with them.

4. We have Flask server for interacting with front-end. Client requests which subject system they want to explore and the server returns JSON response with the abstraction tree.

5. For the interface of our web application, we have used HTML, CSS, and JQuery. When a specific node

is right-clicked, detail information about the node is filled to the node details panel.

6. We used GoJS for building the abstraction tree diagram. Each abstraction is a GoJS node and different properties of the abstraction nodes are binded to GoJS nodes.

5.4 Interface

In this section, we will discuss the different components of our HCPC tool shown in Figure 5.2.

- **Abstract Tree Panel(A).** In the panel, the main abstraction tree is presented. The root nodes are presented vertically which can be possible to expand with their child nodes. By right clicking the mouse on a node will load different information of the abstraction node in the right side of the interface.
- **Number of execution paths(B).** As each node in the abstraction tree are a collection of execution paths, we show the number of execution paths for a selected node in this element.
- **Files (C).** In the element, we show the unique files of all the methods that the execution paths belong to.
- **Node summary (D).** In the element, we have provided natural text description of a node. When developers select a node, the text description of the node will appear in the element.
- **Execution Patterns (E).** In the element, for a selected abstraction node, frequent function call patterns are presented with the file they are associated with. In the current setting, top-10 frequent execution patterns are shown.
- **Execution paths (F).** In the element, we show five execution paths of a selected abstraction node. The execution paths complement the execution patterns by showing a glimpse of the real execution paths. Moreover, when a specific method is searched, the execution paths with the searched method is presented instead first five methods.
- **Node label technique and search panel (G).** The panel has three drop-down boxes. First, developers can select which subject system they want to explore. Second, they can choose which technique to be used for labeling the nodes in abstraction tree. Third, this drop-down box is search enabled and it helps to highlight the nodes which have the searched method in their execution paths.

5.5 Guide to Use the HCPC Tool

The tool can be used in two ways. First, a developer new to the code-base can load the abstraction tree which starts with top abstraction nodes. In the node details panel, for each node the number of execution paths, a brief natural text summary, and few frequent execution patterns are presented. Therefore, the developer can

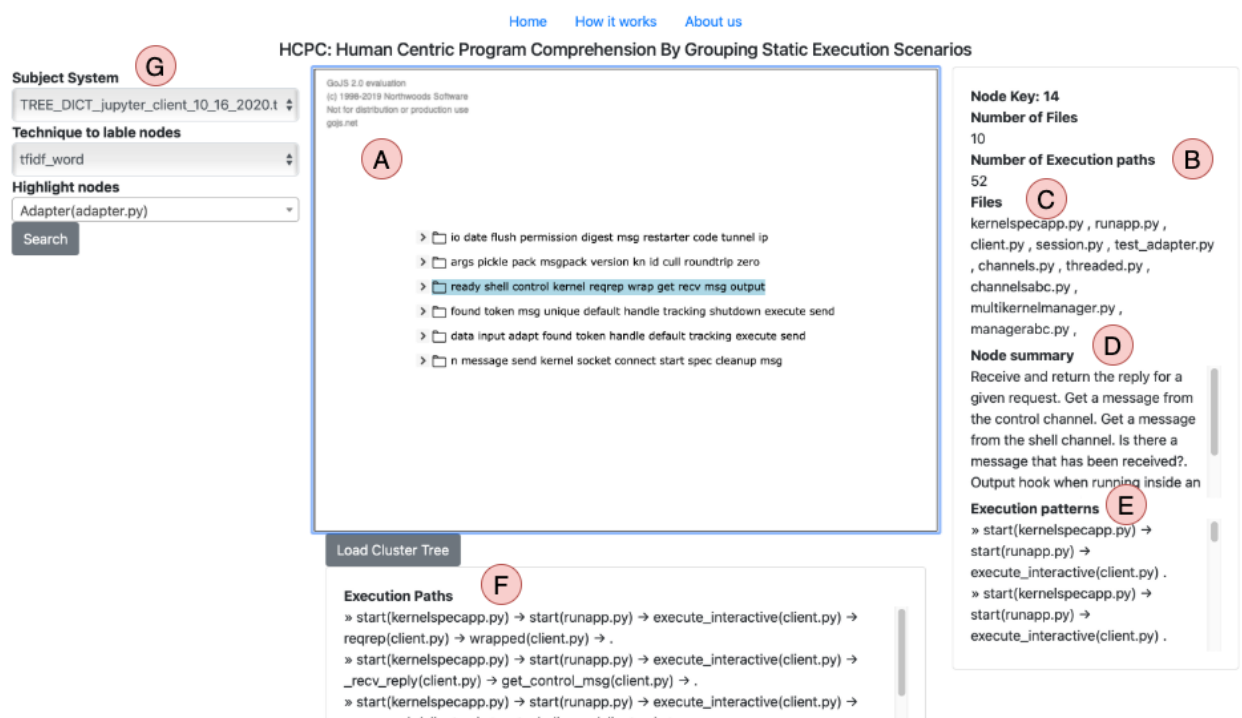


Figure 5.2: HCPC tool interface

start first by observing summary and patterns of the top nodes. Now, the child nodes of the top nodes can be expanded and similarly explored by observing corresponding node summary and patterns. The developer can continue this way according to their need to get acquainted with the coda-base behavior and high-level concepts in the code-base.

Second, a new contributor to a open source project or someone new to a team can utilize the tool to understand high-level concepts related to a specific method. Developers first start from looking to open issues of a repository to find something work on. The issues are natural text description which provides information regarding a bug or a feature enhancement request. Developers can identify a few keywords and use our tool to find matching methods relevant to the keywords. Next, a specific method can be selected to highlight relevant nodes in the tree. The difference between the first approach here is developers will be able to browse the tree with focus to the selected method. The node titles relevant to selected methods will be highlighted so that the developer can expand their child nodes. In this way, the developer can navigate from the high-level concept to low-level source code related concepts for a specific method. By iterating this process, the developer can grasp high-level domain knowledge (with comment summary and IR techniques on function names) alongside insight into program execution scenarios which decreases the overhead due to lack of domain knowledge in the code-base.

5.6 Exploring the HCPC tool for *jupyter_client* Project

Exploring overview. We have picked *jupyter-client*[3] as the subject system to show how the tool can be used following the two above mentioned techniques. To discuss the effectiveness of our tool using *jupyter-client*, first we will discuss high level functionalities of *jupyter-client* from their documentation. Later, we will present the information provided by our tool and discuss whether our tool provides similar or more information to comprehend the *jupyter-client* project. *jupyter-client* has three components. First, *kernelspec* deals with specify different type of kernels from predefined files. Second, kernel manager which is responsible for start, stop and signaling kernels for different scenarios. Third, kernel client which is responsible for communicating with kernels for code execution and other tasks[4]. From the above components we can get an abstract idea of the features of *jupyter-client*. Now, we will discuss the high-level features suggested by the HCPC tool shown in Figure 5.3. Below we have listed few high-level node summary of the *jupyter-client* project and discuss them with respect to the documentation.

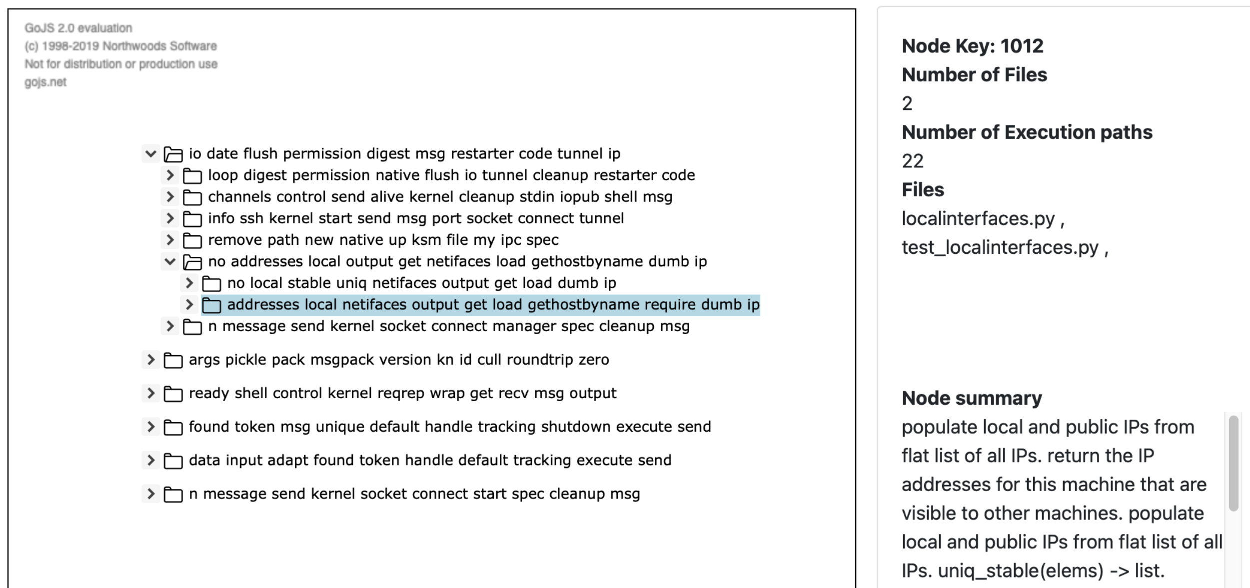


Figure 5.3: HCPC tool overview for jupyter_client project

- *Restarts a kernel with the arguments that were used to launch it. Prepares a kernel for startup in a separate process. Write connection info to JSON dict in self.connection_file. replace templated args (e.g. Verify realpath is used when formatting connection file). Walks env entries in templated_env and applies possible substitutions from current env.*

[3]https://github.com/jupyter/jupyter_client
[4]https://jupyter-client.readthedocs.io/en/stable/index.html

From this node summary, we can understand *jupyter_client* restarting kernels, writing connection information to file and creates different kernel environments.

- *Create a zmq Socket and connect it to the kernel. Start a new kernel, and return its Manager and Client. return zmq Socket connected to the Control channel. Get the stdin channel object for this kernel. Wait for kernel shutdown, then kill process if it doesn't shutdown. Pass a message to the ZMQ socket to send. return zmq Socket connected to the Heartbeat channel. Get the shell channel object for this kernel. Get the iopub channel object for this kernel. Get the control channel object for this kernel. Sends a signal to the process group of the kernel (this. Stops all the running channels for this kernel. return zmq Socket connected to the Shell channel. return zmq Socket connected to the IOPub channel. return zmq Socket connected to the StdIn channel.*

  From this node summary, we observe that the *jupyter_client* project has ZMQ socket which helps with message communication. It has different channels like iopub, stdin, shell and Heartbeat channel.

- *load the IPs that point to this machine. populate local and public IPs from flat list of all IPs. return the IP addresses that point to this machine.*

  From this node summary, we can comprehend that the *jupyter_client* project also deals with public, local IP address of a machine.

From the above text blocks, we can understand that *jupyter-client* is relevant to working with kernels, it uses ZMQ socket to communicate with kernels, and deals with IP addresses of a machine. In addition to the above node summaries when developers see the execution patterns, they can very quickly learn about the domain knowledge of *jupyter_client* project.

Exploring for specific task. Next, it is possible to browse the tree by focusing on a specific method. In Figure 5.4, we can see the nodes in the tree are marked to indicate they are relevant to write_connection_file method. Developers can investigate the nodes marked to understand relevant concepts of write_connection_file method. In Figure 5.4, at the bottom of the tree we can see execution paths which have write_connection_file. At the right side of Figure 5.4, we can observe node summary and execution patterns for the red marked nodes for better understanding of our target concept. Below we have mentioned and discussed few significant node summary relevant to write in connection file.

- *Create a zmq Socket and connect it to the kernel. return the IP addresses that point to this machine. Write connection info to JSON dict in self.connection_file. Restarts a kernel with the arguments that were used to launch it. Restarts a kernel with the arguments that were used to launch it. Pass a message to the ZMQ socket to send. Cleanup connection file *if we wrote it*. Given a message or header, return the header. Forgets randomly assigned port numbers and cleans up the connection file. Sends a signal to the process group of the kernel*

HCPC: Human Centric Program Comprehension By Grouping Static Execution Scenarios

Select Subject System

TREE_DICT_jupyter_client_10_16_2020.t

Select technique to lable nodes

tfidf_word

Highlight nodes with similar function

write_connection_file(connect.py)

Search

GoJS 2.0 evaluation
(c) 1998-2019 Northwoods Software
Not for distribution or production use
gojs.net

io date flush permission digest msg restarter code tunnel ip
loop digest permission native flush io tunnel cleanup restarter code
loop digest permission native flush io tunnel cleanup code restarter
args packer pack msgpack version kn id cull pickle zero
serialize raw naive pickle msgpack ms handle default precision unserialize
find get install specs prefix priority subclass spec kernel remove
manager load no run async such cleanup file connection kernel
default ip manager load such no client file connection kernel
mixin load find run cleanup async app file connection kernel
names localhost ip default kn find app file load connection
url get create connection file start kernel spec connect socket
channels control send alive kernel cleanup stdin iopub shell msg
info ssh kernel start send msg port socket connect tunnel
remove path new native up ksm file my ipc spec
no addresses local output get netifaces load gethostbyname dumb ip
no local stable uniq netifaces output get load dumb ip
addresses local netifaces output get load gethostbyname require dumb ip
n message send kernel socket connect manager spec cleanup msg
args pickle pack msgpack version kn id cull roundtrip zero
ready shell control kernel reqrep wrap get recv msg output
found token msg unique default handle tracking shutdown execute send
data input adapt found token handle default tracking execute send
n message send kernel socket connect start spec cleanup msg

Load Cluster Tree

Execution Paths
» initialize(runapp.py) → initialize(consoleapp.py) →
init_connection_file(consoleapp.py) → load_connection_file(connect.py) →
write_connection_file(connect.py) → _record_random_port_names(connect.py) → .

Node Key: 1017
Number of Files
5
Number of Execution paths
23
Files
manager.py , connect.py ,
localinterfaces.py , kernelspec.py ,
test_manager.py ,

Node summary
{connection_file}) Restarts a kernel with the arguments that were used to launch it. {connection_file}) Prepares a kernel for startup in a separate process. Write connection info to JSON dict in

Execution patterns
»
start_new_async_kernel(manager.py
) → start_kernel(manager.py) →
pre_start_kernel(manager.py) .
» restart_kernel(manager.py) →
start_kernel(manager.py) →

Figure 5.4: HCPC tool when focusing on write_connection_file method

From this node summary, we comprehend that in *jupyter_client* some concepts related to write in connection files are write connection info as JSON dict, cleanup of connection file and forgetting randomly assigned port numbers.

- *Restarts a kernel with the arguments that were used to launch it. Prepares a kernel for startup in a separate process. Write connection info to JSON dict in self.connection_file. replace templated args (e.g. Verify realpath is used when formatting connection file. Walks env entries in templated_env and applies possible substitutions from current env.*

  From this node summary, we comprehend that in *jupyter_client* some concepts related to write in connection files are restart kernel, creating environments and prepare a kernel startup in separate process.

- *Load connection info from JSON dict in self.connection_file. return ip for localhost (almost always 127.0.0.1) set up ssh tunnels, if needed.*

  From this node summary, we comprehend that in *jupyter_client* some concepts related to write in connection files are set up ssh tunnel, loading connection info from file.

From above discussion with regard to write_connection_file method, we can see that HCPC helps to understand relevant concepts for a specific task.

5.7 Human-subject Study

To evaluate the effectiveness of HCPC, we contacted with *SciDataManager*[5] development team. We have collected their source code to analyze using our system. We have conducted the study with three developers of the *SciDataManager* project to find out their opinion about the HCPC tool.

5.7.1 Research Questions

We want to evaluate the effectiveness of the HCPC tool for helping developers comprehend a software project. We address two research questions which correspond to the overarching research question 5 of Section 1.3.

- **RQ1:** To what extent developers do agree with our approach for getting overview of a project?
- **RQ2:** How helpful is our approach to understand relevant high-level concepts targeting a low-level source code (method)?

5.7.2 Study Design

The interview with developers are conducted remotely via Skype. The interview process was divided into four steps:

- *Introduction:* First, we brief each participants about our research. Then, we share our screen to show how to use the HCPC tool. We demonstrate the HCPC tool by exploring *jupyter-client* project. We also discuss different components' role to help program comprehension. Later, we asked the participants to go to a specific URL where our application is hosted and share their screen. We informed participants about two parts of the study.
- *Feedback on getting overview (RQ1):* In this phase, we asked the participants to explore the ACS tree alongside different components like node summary, execution patterns. We requested them to check whether they can get an overview of the *SciDataManager* project. We encouraged the participants to express their thoughts in accordance with think-aloud protocol [25, 69] while they explore different parts of the system. At the same time, we observed the participants' interaction with the system and noted feedback provided by them. When they explored the tree, we asked them whether the keywords and groups provide any reasonable clue about what the system does. Similarly, we asked them about their opinion on node summary and execution Patterns. We also inquired whether they have any suggestions or expectations for the components to be more helpful.
- *Effectiveness of finding help for specific task (RQ2):* After we complete the second step, we move on to the third phase. In this step, we ask the participants to use the search option to find relevant nodes in

[5]http://scidatamanager.usask.ca

the ACS tree and see whether they can find any help to do tasks they recently did. We have encouraged them to remember any recent feature or issue they solved and try to see whether the HCPC tool could help them for completing the tasks. We asked the participants about how helpful Node summary, Execution patterns and the highlight of execution paths can be for someone new to the codebase to accomplish the tasks.

- *Open discussion and closing:* At the end, we asked some open-ended questions like suggestion for new features, feedback for existing features. The meetings lasted between 40 to 60 minutes. We ended the meeting thanking the participants for their valuable feedback and time.

5.7.3 Participants and Subject System Selection

While observing the HCPC tool output for *jupyter-client* project, we can relate the different nodes content to the components in *jupyter-client* documentation. We decided to conduct the study on a subject system where the team members can participate in the study to evaluate the HCPC tool performance on their known codebase. We contacted the *SciDataManager* team whether they could share their source code and participate in the study to evaluate the HCPC. The development team agreed to share the codebase and three of them participated in the study.

5.7.4 Results

Answering RQ1. Participants agreed that the HCPC tool can help in getting an overview of their project. When we asked the participants, they started to explore the abstraction tree by carefully observing the keywords for each node and expanding to child nodes. The participants agreed that high-level nodes provide hints to the features in their project. For example, participant P3 said, *"I can relate to different basic components from high level nodes. If someone new joins the team, they can start from top nodes and see the path patterns for getting most frequent behaviour and then explore the code-base easily."* Participants appreciated the node summary as it states in plain text what are the purposes of the keywords in the project. Participants also find that when they see node summary for deeper nodes, the summary becomes more precise for specific features. According to participant P1, *"This part is helpful as it states in natural texts instead of a few words. Another interesting fact about the summary is when going deeper the summary became more precise."* While exploring the execution patterns, we observed that participants find it helpful to know some frequent call sequences in specific nodes. However, participant P2, P3 suggested that having the frequency with the patterns would be interesting to know for understanding the importance.

In summary, **Participants find the HCPC tool helpful for getting an overview of their software system with node title, summary and execution patterns.** According to their final feedback for comprehending overview, they pointed out that the HCPC tool has the potential to decrease the getting started time for a project. According to participant P1, they believe it can help to decrease getting started

time around 50%-60%.

Answering RQ2. Participants find it useful to be able to search for specific keywords. From the interview, we observed that developers tried to highlight nodes for some recent work they have done or something they are familiar with to check how the HCPC tool is representing the relevant concepts. For example, participant P3 tried to highlight the nodes related to dataset publishing as it is one of the core feature of the project. While browsing the highlighted nodes and its supporting contents (node summary, execution patterns, execution paths), participant P3 identified that it is possible to know similar paths where the function is called. Another interesting observation by participant P2 is, *" I see the nodes can be searched by functions. In addition, I would love to see filters such as class, files."* Participant P3 shared from their previous experience that sometimes they have to fix some issues of another project which are not very well documented and they struggle a lot to figure out the abstraction patterns followed in the codebase. Both participants P3, P1 suggested using the search option to explore execution paths will be helpful to decrease time required for completing tasks in those scenarios. Another interesting observation from the interview is for some searches multiple nodes are highlighted which shows the specific functions being used in different scenarios. We observe participants were enthusiastic to know what are the different directions the function is being used by going deeper in the abstraction tree. In addition, participant P1 shared that many times they try to search the codebase with some keywords using the find option provided by the editor to retrieve relevant files. However, the search result does not show any order or how these classes or methods are being called. They suggested that with the execution patterns and paths the HCPC tool can help to convert the raw find workflow into more execution based search process. In summary, **the feedback from the participants and our observation during the interview indicate that it is viable that the search option of the HCPC tool has the potential to help in day-to-day software maintenance activities.**

During our open-ended questions and suggestions, we found valuable feedback for future development and adaptation of the HCPC tool. One important suggestion is to incorporate automatic comment generation techniques for methods which have no comments. This will be a valuable future work suggestion for our HCPC tool, as it will be helpful for projects which do not follow best practices. Another worth mentioning future work suggested by participant P2 is to generate a report of the abstraction structure where developers can edit the components' names according to their understanding from the HCPC tool. This report can be used as a documentation of the project structure from a static execution perspective. In addition, participants suggested to enable the option to export projects from GitHub which will be useful for quickly exploring a new codebase. From the above discussion, **we can conclude that the HCPC tool can help to get an overview of a software project from a static execution perspective and can be used to help doing a specific task in hand.**

5.7.5 Threats to Validity

To address external validity, we have collected a software project which is developed in industry settings instead of working with a sample project. We have selected a professionally developed project to ensure generalizability to some extent. Although the subject system is written in Python, our approach will work with both static and dynamic typed language as our approach depends on only caller-callee relationship between methods.

To address internal validity, we have tried to minimize any communication issue by repeating the feedback when in doubt. We acknowledge that our sample size for participants in the study is small. However, our participants' are experienced in the subject system and we got repetitive feedback which indicates acceptance to some extent. We asked open-ended questions at the end so that participants are able to provide feedback outside the questions asked.

5.8 Summary and Discussion

In this study, we have proposed two new approaches to enhance the abstract code summary tree for program comprehension. First, we change the similarity metrics for comparing execution paths. In previous studies, Jaccard distance is used which only considers the content not the sequence. Therefore, in this study, we have changed the similarity metrics to strike_a_match algorithm. Second, we changed the clustering approach for more precise grouping of the execution paths. Previous studies suggested to use the hierarchy tree for browsing. However, from previous studies we found that hierarchy tree has abundant abstraction nodes which hinders going deeper levels. We used cluster flattening technique which reduces redundant abstraction nodes from hierarchy tree. We have built an interactive system to explore the new abstraction tree with supporting information alongside searching the tree. To evaluate, the system we have conducted a human subject study. We find that our system can be useful for getting acquainted with a software project as well as accomplishing tasks in hand.

6 Conclusion and Future Work

In this thesis, we have worked on grouping execution paths of a software project for helping developers comprehend the codebase faster and locate related concepts for their tasks in hand. We start with existing works on clustering static execution paths for presenting high-level features in a software project. However, we find some limitations and scope of improvement to make the abstract code summary more usable for the daily activities of software developers. First, we experimented with different information retrieval techniques to find out which techniques provide more helpful labels for abstraction nodes. We also proposed using the terms in method names instead of whole method name as input for IR techniques. We also conducted a human subject study to find out how developers rate different IR techniques and compared automatic naming with manual naming by developers. From the study, we found TFIDF with terms in method are better supported by the manual labeling compared to LDA, LSI techniques. Moreover, developers preferred the words variant than the method variant of the labeling technique. Second, we proposed to add additional information such as node summary, execution patterns for each abstraction node to make the abstract code summary tree more comprehensible. We conducted a case study with three different subject systems to find the potential of attaching the two new information for each abstraction node. We found that attaching node summary, and execution patterns can complement node labels for more detailed understanding in relation to source code. However, we observed that using an agglomerative cluster tree poses some difficulty to browse as it presents all the clustering step by step. In addition, we noticed that it is difficult to explore the tree when targeting some specific keywords. In our third study, we addressed the issue of abstraction tree being overwhelming to browse by simplifying the agglomerative cluster tree using cluster flattening technique. Additionally, we have added an abstraction node highlighting technique for browsing the tree targeting specific keywords or methods. To evaluate the usefulness of our technique, we developed a web application called the HCPC. We performed a human subject study with an industry project and their developers. From the study, we found that the HCPC tool can help developers get started with a project alongside finding relevant execution patterns for specific tasks in hand.

In future, we plan to adopt automatic method summarizing techniques [61, 5, 70] since in industry settings not every method is properly documented. We also plan to incorporate the GitHub project import option for exploring in the HCPC tool. Moreover, we will add a feature to export reports with our analysis result. We have a plan to conduct a wide-scale user study with popular Python open-source projects.

References

[1] Networkx. https://github.com/networkx/networkx, 2019.

[2] Prefixspan-py. https://github.com/chuanconggao/PrefixSpan-py, 2019.

[3] Pyan. https://github.com/davidfraser/pyan, 2019.

[4] requests. https://github.com/psf/requests, 2019.

[5] Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. A transformer-based approach for source code summarization. *arXiv preprint arXiv:2005.00653*, 2020.

[6] Erik Arisholm, Lionel C Briand, Siw Elisabeth Hove, and Yvan Labiche. The impact of uml documentation on software maintenance: An experimental evaluation. *IEEE Transactions on Software Engineering*, 32(6):365–381, 2006.

[7] Federico Barrios, Federico López, Luis Argerich, and Rosa Wachenchauzer. Variations of the similarity function of textrank for automated summarization. *arXiv preprint arXiv:1602.03606*, 2016.

[8] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Professional, 2003.

[9] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003.

[10] Ruven Brooks. Towards a theory of the comprehension of computer programs. *International journal of man-machine studies*, 18(6):543–554, 1983.

[11] Tse-Hsun Chen, Stephen W Thomas, and Ahmed E Hassan. A survey on the use of topic models when mining software repositories. *Empirical Software Engineering*, 21(5):1843–1919, 2016.

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

[13] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J Van Wijk, and Arie Van Deursen. Understanding execution traces using massive sequence and circular bundle views. In *15th IEEE International Conference on Program Comprehension (ICPC'07)*, pages 49–58. IEEE, 2007.

[14] Bas Cornelissen, Andy Zaidman, Arie Van Deursen, Leon Moonen, and Rainer Koschke. A systematic survey of program comprehension through dynamic analysis. *IEEE Transactions on Software Engineering*, 35(5):684–702, 2009.

[15] Kostadin Damevski, David Shepherd, and Lori Pollock. A field study of how developers locate features in source code. *Empirical Software Engineering*, 21(2):724–747, 2016.

[16] Andrea De Lucia, Massimiliano Di Penta, Rocco Oliveto, Annibale Panichella, and Sebastiano Panichella. Using ir methods for labeling source code artifacts: Is it worthwhile? In *2012 20th IEEE International Conference on Program Comprehension (ICPC)*, pages 193–202. IEEE, 2012.

[17] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41(6):391–407, 1990.

[18] Yang Feng, Kaj Dreef, James A Jones, and Arie van Deursen. Hierarchical abstraction of execution traces for program comprehension. In *Proceedings of the 26th Conference on Program Comprehension*, pages 86–96, 2018.

[19] Gharib Gharibi, Rakan Alanazi, and Yugyung Lee. Automatic hierarchical clustering of static call graphs for program comprehension. In *2018 IEEE International conference on big data (Big Data)*, pages 4016–4025. IEEE, 2018.

[20] David J Gilmore. Models of debugging. *Acta psychologica*, 78(1-3):151–172, 1991.

[21] Ross Girshick, Ilija Radosavovic, Georgia Gkioxari, Piotr Dollár, and Kaiming He. Detectron. `https://github.com/facebookresearch/detectron`, 2018.

[22] Sonia Haiduc, Jairo Aponte, and Andrian Marcus. Supporting program comprehension with source code summarization. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 2*, pages 223–226. ACM, 2010.

[23] Jiawei Han, Jian Pei, Behzad Mortazavi-Asl, Helen Pinto, Qiming Chen, Umeshwar Dayal, and Meichun Hsu. Prefixspan: Mining sequential patterns efficiently by prefix-projected pattern growth. In *proceedings of the 17th international conference on data engineering*, pages 215–224. Citeseer, 2001.

[24] Cruz Izu, Carsten Schulte, Ashish Aggarwal, Quintin Cutts, Rodrigo Duran, Mirela Gutica, Birte Heinemann, Eileen Kraemer, Violetta Lonati, Claudio Mirolo, et al. Program comprehension: Identifying learning trajectories for novice programmers. In *Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education*, pages 261–262, 2019.

[25] Riitta Jääskeläinen. Think-aloud protocol. *Handbook of translation studies*, 1:371–374, 2010.

[26] Corentin Jemine. Real-time-voice-cloning, 2019.

[27] Eric Jones, Travis Oliphant, Pearu Peterson, et al. SciPy: Open source scientific tools for Python, 2001–. [Online; accessed 20-09-2019].

[28] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of documentation*, 1972.

[29] Andrew J Ko, Brad A Myers, Michael J Coblenz, and Htet Htet Aung. An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks. *IEEE Transactions on software engineering*, 32(12):971–987, 2006.

[30] Jacob Krüger, Thorsten Berger, and Thomas Leich. Features and how to find them: a survey of manual feature location. *Software Engineering for Variability Intensive Systems*, pages 153–172, 2019.

[31] Naveen Kulkarni and Vasudeva Varma. Supporting comprehension of unfamiliar programs by modeling an expert's perception. In *Proceedings of the 3rd International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, pages 19–24, 2014.

[32] Omer Levy and Dror Feitelson. Understanding large-scale software–a hierarchical view. In *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, pages 283–293. IEEE, 2019.

[33] Robert C Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Pearson Education, 2008.

[34] Paul W McBurney, Cheng Liu, Collin McMillan, and Tim Weninger. Improving topic model source code summarization. In *Proceedings of the 22nd international conference on program comprehension*, pages 291–294, 2014.

[35] Rada Mihalcea and Paul Tarau. Textrank: Bringing order into text. In *Proceedings of the 2004 conference on empirical methods in natural language processing*, pages 404–411, 2004.

[36] Roberto Minelli, Andrea Mocci, and Michele Lanza. I know what you did last summer-an investigation of how developers spend their time. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 25–35. IEEE, 2015.

[37] Hausi A Muller and James Scott Uhl. Composing subsystem structures using (k, 2)-partite graphs. In *Proceedings. Conference on Software Maintenance 1990*, pages 12–19. IEEE, 1990.

[38] Daniel Mullner. Modern hierarchical, agglomerative clustering algorithms, 2019.

[39] G. C. Murphy, D. Notkin, and K. J. Sullivan. Software reflexion models: bridging the gap between design and implementation. *IEEE Transactions on Software Engineering*, 27(4):364–380, 2001.

[40] Ani Nenkova and Rebecca Passonneau. Evaluating content selection in summarization: The pyramid method. In *Proceedings of the human language technology conference of the north american chapter of the association for computational linguistics: Hlt-naacl 2004*, pages 145–152, 2004.

[41] Suphakit Niwattanakul, Jatsada Singthongchai, Ekkachai Naenudorn, and Supachanun Wanapu. Using of jaccard coefficient for keywords similarity. In *Proceedings of the international multiconference of engineers and computer scientists*, volume 1, pages 380–384, 2013.

[42] Lawrence Page, Sergey Brin, Rajeev Motwani, and Terry Winograd. The pagerank citation ranking: Bringing order to the web. Technical report, Stanford InfoLab, 1999.

[43] Annibale Panichella, Bogdan Dit, Rocco Oliveto, Massimilano Di Penta, Denys Poshynanyk, and Andrea De Lucia. How to effectively use topic models for software engineering tasks? an approach based on genetic algorithms. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 522–531. IEEE, 2013.

[44] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[45] Nancy Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19(3):295–341, 1987.

[46] Michael Pradel and Thomas R Gross. Automatic generation of object usage specifications from large method traces. In *2009 IEEE/ACM International Conference on Automated Software Engineering*, pages 371–382. IEEE, 2009.

[47] Juan Ramos et al. Using tf-idf to determine word relevance in document queries. In *Proceedings of the first instructional conference on machine learning*, volume 242, pages 133–142. Piscataway, NJ, 2003.

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

[49] Steven P Reiss. Dynamic detection and visualization of software phases. In *Proceedings of the third international workshop on Dynamic analysis*, pages 1–6, 2005.

[50] Meghan Revelle, Tiffany Broadbent, and David Coppit. Understanding concerns in software: insights gained from two case studies. In *13th International Workshop on Program Comprehension (IWPC'05)*, pages 23–32. IEEE, 2005.

[51] Banani Roy and TC Nicholas Graham. An iterative framework for software architecture recovery: An experience report. In *European Conference on Software Architecture*, pages 210–224. Springer, 2008.

[52] Maher Salah, Trip Denton, Spiros Mancoridis, Ali Shokoufandeh, and Filippos I Vokolos. Scenariographer: A tool for reverse engineering class usage scenarios from method invocation sequences. In *21st IEEE International Conference on Software Maintenance (ICSM'05)*, pages 155–164. IEEE, 2005.

[53] Teresa M Shaft and Iris Vessey. The relevance of application domain knowledge: the case of computer program comprehension. *Information systems research*, 6(3):286–299, 1995.

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

[55] Janet Siegmund. Program comprehension: Past, present, and future. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, volume 5, pages 13–20. IEEE, 2016.

[56] Jamie Starke, Chris Luce, and Jonathan Sillito. Searching and skimming: An exploratory study. In *2009 IEEE International Conference on Software Maintenance*, pages 157–166. IEEE, 2009.

[57] Xiaobing Sun, Xiangyue Liu, Bin Li, Yucong Duan, Hui Yang, and Jiajun Hu. Exploring topic models in software engineering data analysis: A survey. In *2016 17th IEEE/ACIS International Conference on Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing (SNPD)*, pages 357–362. IEEE, 2016.

[58] Scott Tilley. A reverse-engineering environment framework. Technical report, CARNEGIE-MELLON UNIV PITTSBURGH PA SOFTWARE ENGINEERING INST, 1998.

[59] Anneliese Von Mayrhauser and A Marie Vans. From program comprehension to tool requirements for an industrial environment. In *[1993] IEEE Second Workshop on Program Comprehension*, pages 78–86. IEEE, 1993.

[60] Vijay Walunj, Gharib Gharibi, Duy H Ho, and Yugyung Lee. Graphevo: Characterizing and understanding software evolution using call graphs. In *2019 IEEE International Conference on Big Data (Big Data)*, pages 4799–4807. IEEE, 2019.

[61] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Wu, and Philip S Yu. Improving automatic source code summarization via deep reinforcement learning. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 397–407, 2018.

[62] Jinshui Wang, Xin Peng, Zhenchang Xing, and Wenyun Zhao. An exploratory study of feature location process: Distinct phases, recurring patterns, and elementary actions. In *2011 27th IEEE International Conference on Software Maintenance (ICSM)*, pages 213–222. IEEE, 2011.

[63] Yui Watanabe, Takashi Ishio, and Katsuro Inoue. Feature-level phase detection for execution trace using object cache. In *Proceedings of the 2008 international workshop on dynamic analysis: held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2008)*, pages 8–14, 2008.

[64] Song Wei. *A survey and categorization of program comprehension techniques*. PhD thesis, Concordia University, 2002.

[65] Tim Weninger, Yonatan Bisk, and Jiawei Han. Document-topic hierarchies from document graphs. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 635–644, 2012.

[66] Byron J Williams and Jeffrey C Carver. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, 52(1):31–51, 2010.

[67] Xiaoyuan Xie, Zicong Liu, Shuo Song, Zhenyu Chen, Jifeng Xuan, and Baowen Xu. Revisit of automatic debugging via human focus-tracking analysis. In *Proceedings of the 38th International Conference on Software Engineering*, pages 808–819. ACM, 2016.

[68] Qi Xin, Farnaz Behrang, Mattia Fazzini, and Alessandro Orso. Identifying features of android apps from execution traces. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 35–39. IEEE, 2019.

[69] Shurui Zhou, Stefan Stanciulescu, Olaf Leßenich, Yingfei Xiong, Andrzej Wasowski, and Christian Kästner. Identifying features in forks. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 105–116. IEEE, 2018.

[70] Yuxiang Zhu and Minxue Pan. Automatic code summarization: A systematic literature review. *arXiv preprint arXiv:1909.04352*, 2019.

Appendix A

Simple Calculator program to demonstrate the clustering approach

```
class Calculator:
    """ This class calculates sum, sub, div, mul operation on two given numbers. """

    def init(self):
        """ This function welcomes users to the calculator """

        print("Welcome to One Two calculator. ")
        a, b = self.two_number_input()
        self.operations_to_do(a, b)

    def operations_to_do(self, a, b):

        print('Please enter signs for operations to do on this two number.')
        print('Add = +, Sub = -, Div = /, Mul = *, Modular = %')
        print('Enter . to stop doing operations')
        while 1:
            sign = input()
            if sign == '+':
                result = self.add_two_numbers(a, b)
            elif sign == '-':
                result = self.subtract_two_numbers(a, b)
            elif sign == '/':
                result = self.divide_two_numbers(a, b)
            elif sign == '*':
                result = self.multiply_two_numbers(a, b)
            elif sign == '%':
                result = self.mod_two_numbers(a, b)
            elif sign == '.':
                break
            else:
                print("Invalid input")

            print(a, ' ', sign, ' ', b, ' = ', result)

    def add_two_numbers(self, a, b):
        """ This function adds two numbers """

        return a + b

    def subtract_two_numbers(self, a, b):
        """ This function subtract two numbers """

        return a - b

    def divide_two_numbers(self, a, b):
```

```
        """ This function divide two numbers """

        return a / b

    def multiply_two_numbers(self, a, b):
        """ This function multiply two numbers """

        return a * b

    def mod_two_numbers(self, a, b):
        """ This function mod two numbers """

        return a % b

    def valid_number(self, num):
        """ This function verifies a variable of int type """

        try:
            value = int(num)
            return True
        except ValueError:
            return False

    def two_number_input(self):
        """ inputs two number """

        loop_condition = True
        while loop_condition:
            a = input("Please enter valid first number")

            print(self.valid_number(a))
            if self.valid_number(a):
                loop_condition = False

        loop_condition = True
        while loop_condition:
            b = input("Please enter valid second number")
            if self.valid_number(b):
                loop_condition = False

        return int(a), int(b)


c = Calculator()

c.init()
```