Feature-driven API Usage-based Code Example Recommendation for Opportunistic Reuse

PhD Thesis

Shamsa Abid

2015-03-0049

Advisor: Dr. Shafay Shamail

Department of Computer Science

School of Science and Engineering

Lahore University of Management Sciences

October 28, 2021

DEDICATION

To my parents, teachers and students.

Lahore University of Management Sciences

School of Science and Engineering

CERTIFICATE

We hereby recommend that the thesis prepared under our supervision by ***Shamsa Abid*** titled ***Feature-driven API Usage-based Code Example Recommendation for Opportunistic Reuse*** be accepted in partial fulfillment of the requirements for the degree of PhD.

Dr. Shafay Shamail (Advisor)

Acknowledgements

First, I thank the Almighty, the most gracious and the most merciful, who granted me all the abilities to carry out this work. Then I would like to express my heartiest gratitude to my supervisor Dr. Shafay Shamail for his constant guidance and advice during this thesis work. Thank you for the freedom to explore the topics that interested me. Your keen interest in the research made me reach my goal. For all your time and energy, my sincere thanks! I would also like to thank Dr. Hamid Abdul Basit for his critical insights and advice. Always a call away to get things sorted, Dr. Hamid was a constant source of support in various aspects of my research. Thanks for facilitating our collaboration with Dr. Yoshiki Higo and thanks to Dr. Higo for showing interest in our work.

Perhaps words will fall short to describe the gratitude that I have for Dr. Sarah Nadi. I thank her for guiding my research, teaching me the art of writing, and helping me discover the joy of a well-written paper. Her comments helped shape up and improve my thesis significantly. I have immensely enjoyed working with Dr. Sarah and she is a source of inspiration in how she analyzes problems and conducts paper reviews. I also thank all the anonymous reviewers and editors for their valuable comments and suggestions in improving the research papers produced from this work.

I am grateful to LUMS and its Department of Computer Science for their financial support. I would also like to thank Dr. Basit Shafiq and Dr. Junaid Haroon Siddiqui for serving on my committee, and for reading through drafts of my proposal. I am thankful to Dr. Suleman Shahid for allowing me to conduct research on my project as part of the HCI course which also resulted in a publication. It was a lot of fun working on the project with you. Thank you for your energy and support. Thanks to all my teachers in LUMS for grooming me into a research scholar.

This work would not have been possible without the people who provided the technical support and feedback. I would like to thank my lab-mate Ahmad Akhtar for the technical support in setting up our system on the server and for various brainstorming discussions. I would also like to thank my colleagues from Techlogix, Taiba Sarfraz, Annie Iqbal, Ahsan Khushi, and other anonymous

evaluators for their feedback during our evaluation phase. Also thanks to Salman Javed and Momna Anam for facilitating the evaluation of the very first prototype of my thesis work.

Several people, although not directly involved in the work, contributed in various ways through my PhD journey, and I would like to express my gratitude to them. I would like to express my deepest gratitude to my mother and father whose unconditional support and constant well wishes have made me reach this stage in life. I thank my brother, Muhammad Farhan, for his support and for being a great source of inspiration and motivation. I also thank my beloved sister Tayyaba Niaz Ali, and my lovely niece and nephew, Shahnoor and Shahaan for brightening up dull moments. They always helped me recharge and relax when I most needed it. Thanks to Shahnoor for introducing BTS who were a great source of inspiration and energy throughout my thesis work. I also thank my extended family including grandparents, aunts, uncles, and cousins whose moral support and well wishes kept me going.

I would like to thank all of my teachers, co-workers and staff members from the Department of Computer Science who have helped me reach this stage. I thank my seniors Dr. Zeeshan Ali Rana, Dr. Ayesha Afzal, Saima Mushtaq for their support and advice, and all my PhD batch mates Dr. Haroon Shakeel, Dr. Nauman Khurshid, Dr. Nasir Mehmood, Sadaf Jabeen, Hafsa Zafar, and Waleed Riaz for their support. I specially thank all the past members of Software Engineering Research Lab with whom I have had the opportunity to work. In particular, I would like to thank Maham Anwar who was a constant source of motivation and support during all the highs and lows of my PhD journey. Thank you Maham for your wonderful company during lunch hours and for long refreshing phone calls. I am grateful to Habiba Saim for always being there for emotional and spiritual support. Thanks to Talal Shoukat, Abdul Maroof, Saud Tahir, Tehrim Nawaz, Yashfa Iftikhar, Fahad Akhtar, Adil Inam, Ubaid Ullah, Zohaa Qamar and Natasha Khan for being great co-workers. I will always cherish our team lab parties. Thanks to my lab mates Maria, Zarmina and Alina for maintaining a good friendly environment and for nice refreshing discussions.

I am thankful to CS department administrative staff Zulfiqar Sahab, Ishtiaq Sahab, Noor e Sehar, Afaq Sahab, and Samreen for their timely services. I am thankful to the LUMS library staff for always making research materials available when needed. I am also thankful to the office boys

Shahid and Aslam, to the janitorial staff, and to the security guards for facilitating me and for their well wishes. I am thankful to the LUMS gym instructors Shabana and Kanwal for helping me keep physically fit and for the challenging yet transformational workout plans. I would also thank the LUMS swimming instructors Saima, Ayesha, and Kausar for teaching me how to swim and enabling me to experience a wonderful sport. I am thankful to the LUMS Photographic Society and in particular Zuraiz Niazi and his friends for arranging wonderful and unforgettable camping and hiking trips to the beautiful North-side of Pakistan.

I am also lucky to have a great many friends whose support and encouragement has brought me loads of strength: Nimra Khan, Dr. Jaweria Kanwal, Dr. Atiqa Kayani, Sameera Saleem, Shehnaz Afridi, Shereen Abidi, Narmin Khalid, Afsheen Rizwan, and Dr. Aniqa Dilawari, a heartfelt thank you to all of you for all the fun discussions, laughter, wisdom, and comfort that your friendship brought in this journey. Thanks Aniqa for being the source of regaining perspective whenever the going got tough and the destination seemed out of sight. Lastly, I would like to thank my batch mates from my Bachelors program at Lahore College for Women University and from my Masters program at LUMS for staying in touch and for their tremendous moral support; a big thanks to all LC girls and to Dr. Usman Ali and Nabeel Nasar.

I am also thankful to all the wonderful academic people I met during conferences and who are a source of inspiration. Thanks to Dr. Alexander Serebrenik and Dr. Gabriele Bavota for mentoring me for the Shadow PC reviews of MSR 2021. Also thanks to Drs. Federica Sarro, Timofey Bryksin, Foutse Khomh, Lionel Briand, and Massimilliano Di Penta for interesting conversations. Thanks for inspiring me to keep doing great research and sharing it with the world.

List of Publications

Publications*

Journal

1. S. Abid, S. Shamail, H. A. Basit, and S. Nadi, "FACER: An API usage-based code-example recommender for opportunistic reuse," Empirical Software Engineering, vol. 26, no. 6, pp. 1-58, Aug 2021.

Conferences and Workshops

1. S. Abid, "Recommending related functions from API usage-based function clone structures," Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 1193–1195, 2019.

2. S. Abid, S. Javed, M. Naseem, S. Shahid, H. A. Basit, and Y. Higo, "CodeEase: Harnessing Method Clone Structures for Reuse," in 2017 IEEE 11th International Workshop on Software Clones (IWSC), pp. 1–7, IEEE, 2017.

3. S. Abid and H. A. Basit, "Towards a structural clone based recommender system," in 2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER), vol. 3, pp. 51–52, IEEE, 2016.

4. M. Kamal, A. Abaid, S. Abid, and S. Shamail, "FACER-AS: An API Usage-based Code Recommendation Tool for Android Studio," in 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2021.

*as main and corresponding author

Abstract

Software reuse is a common practice in the development and maintenance of a modern software system. Software developers need to search for reusable code that would assist them in implementing a given feature or development task. Often, the timely discovery of a critical piece of information can have a dramatic impact on productivity. Current code search systems focus on providing code against a specific user query and repeated searches need to be performed until the code for the desired feature or set of features is found. The problem of repeated code searches needs to be addressed and a solution is desired for helping developers to get the code for related features, thus enabling opportunistic reuse for increased developer productivity.

We propose FACER (Feature-driven API usage-based Code Examples Recommender), a recommendation system that provides developers with method recommendations having functionality relevant to their feature or development task. The main idea behind FACER is to provide code recommendations against a feature query based on patterns of frequently co-occurring API usage-based Method Clone Classes. Such patterns are called Method Clone Structures (MCS). The heuristic behind Method Clone Classes is that methods with similar uses of APIs are semantically related because they do similar things and are identified as members of a clone class. FACER generates related method recommendations in two stages. In the first stage, the developer provides a feature query (expressed as a comment) to get a set of methods that implement the feature. In the second stage, upon selection of one of these methods by the developer, a subsequent recommendation provides related methods for opportunistic reuse. Our experimental results show that, on average, FACER's recommendations are 80% precise and that developers find the idea of related method recommendations useful. We also propose a novel Context-Aware Feature-driven API usage-based Code Examples Recommender (CA-FACER) that leverages a developer's development context to recommend related code snippets. We consider the methods having API usages in a developer's active project as part of the development context and demonstrate how context-awareness based on API usages can improve the quality of recommendations for Java Android

code. From our experimental evaluation on 120 Java Android projects from GitHub, we observe a 46% improvement of precision using our proposed context-aware approach over our baseline FACER. CA-FACER recommends related code examples with an average precision (P@5) of 94% and 83% and a success rate of 90% and 95% for initial and evolved development stages, respectively.

Contents

List of Figures

List of Tables

List of Terms

This dissertation uses the following definitions of the terms found within the dissertation's text.

Active code The code that a developer is actively working on within their IDE at a given time.

API call An API call is made up of an API class name, API method name, and a set of API method parameters. An API call occurs within a code snippet and performs an atomic functionality.

API call density The fraction of statements in a method containing API calls over the total number of statements in a method.

API Usage A software system implemented in Java can interact with third-party libraries through various API classes. APIs enable a programmer to implement desired features of the software system. For example, building the connection to a Bluetooth device requires the use of a Bluetooth API and different methods of the API may be called to setup the connection. Such a combination of API calls is referred to as API usage. A user-written method may or may not have one or more API usages. FACER considers methods with at least three API usages as methods implementing features.

API usage pattern An API usage pattern documents a set of method calls from multiple API classes to achieve a reusable functionality.

API Usage-based Method Clones If a group of Java methods share similar API usages, we consider them as API usage-based method clones. These methods implement the same feature or functionality and are instances of that particular feature.

API Usage-based Method Clone Group A set of methods in which API usage similarity exists between any pair of methods. These methods may implement the same feature or functionality.

API Usage-based Method Clone Class *See API Usage-based Method Clone Group*

API Usage-based Method Clone Structures A set of methods containing API usages that are frequently cloned together across different projects. In other words, a recurring pattern of API usage-based method clones is an API usage-based Method Clone Structure.

CA-FACER Context-Aware Feature Driven API usage-based Code Example Recommender

Call graph A call graph is a control-flow graph, which represents calling relationships between methods in a software project.

Clone class *See Clone group*

Clone group A set of code snippets in which similarity exists between any pair of methods.

Clone type Code clones can be classified into four categories; *Type-1*: This category includes code fragments which are textually similar and may have differences in white-space, comments, and layout. *Type-2*: This category includes code fragments having lexical similarity i.e. code fragments are identical with the only differences in identifier names and literal values. *Type-3*: These are code fragments that are syntactically similar. Such code fragments can differ at the statement level with statements being added, modified or removed. *Type-4*: This category includes code fragments that are semantically similar in terms of functionality, but possibly different in how the functionality is implemented. These types of clones may have little or no lexical or syntactic similarity and hence are relatively difficult to detect.

Clustering The task of grouping a set of objects in such a way that objects in the same group (called a cluster) are more similar (in some sense) to each other than to those in other groups (clusters).

Code clone Code clones are similar code fragments that may be completely identical or may have some lexical, syntactic, or structural differences.

Code completion A software system predicts the rest of the code a user is typing. Code completion is designed to save time while writing code. As you start to type the first part of a function, it can suggest or complete the function and any arguments or variables. Code completion may occur at different code granularity such as statement-level, method body-level, or class-level.

Code fact repository A database of source code metadata extracted using program analysis on source code files.

Code snippet A continuous segment of code in a code file

Context-aware recommendation Recommending the most relevant items to users taking into account any additional contextual information, such as time, location etc.

Domain analysis Domain analysis is the process of identifying, capturing domain knowledge about the problem domain with the purpose of making it reusable when creating new systems.

FACER Feature Driven API usage-based Code Example Recommender

Feature A feature is defined as a desired functionality of a software product. For example, a Bluetooth chat application can have the features like setting up Bluetooth, scanning for other Bluetooth devices, connecting to a remote device and transferring data over Bluetooth. A feature may be implemented as a single function or a group of interacting functions.

Feature query See *Search Query*

Frequent pattern Frequent patterns are itemsets, subsequences, or substructures that appear in a data set with frequency no less than a user-specified threshold.

Friend methods Friend methods are related to each other because they are each part of a method clone group and members of these clone groups are often found *hanging out* with each other across different software projects.

Function See *Method*

Function Clone Class *See Method Clone Class*

Function Clone Structure *See Method Clone Structure*

Hybrid A thing made by combining two different elements.

Information retrieval The process of obtaining information system resources that are relevant to an information need from a collection of those resources.

Market basket analysis Market basket analysis attempts to identify associations, or patterns, between the various items that have been chosen by a particular shopper and placed in their basket.

Method A method in Java is a block of statements that has a name and can be executed by calling it from some other place in your program.

Method Clone Class *See Method Clone Group*

Method Clone Group A set of methods in which similarity exists between any pair of methods.

Method Clone Structure A Method Clone Structure (MCS) consists of a set of methods that are frequently cloned together across different projects. In other words, a recurring pattern of method clones is a Method Clone Structure. The participating methods in a Method Clone Structure all relate to each other.

Opportunistic Code Reuse When programmers add features to their code in an adhoc manner by reusing existing source code examples, they perform *opportunistic code reuse*.

Package A package in Java is used to group related classes. A package is a namespace that organizes a set of related classes and interfaces. Packages are divided into two categories; built-in Packages (packages from the Java API) and user-defined Packages (create your own packages).

Recommendation system A recommendation system for software engineering (RSSE) is a software application that provides information items estimated to be valuable for a software engineering task in a given context.

Related features A software application is implemented as a collection of features which are related to each other in some way. Similar applications share similar features and such features are related to each other because they commonly occur together based on the market basket analysis principle (See *Market Basket Analysis*).

Representative method A method clone group may consist of multiple methods. One method needs to be chosen as the representative method of the clone group for recommending to a developer.

Search facets Search facets are search fields which are used to establish the criteria for retrieving relevant code.

Search index A search index is used by a search engine to store data to facilitate fast and accurate information retrieval.

Search query A search query is a phrase or a keyword combination users enter in search engines to find things of interest.

Semantic clone Syntactically dissimilar code snippets that implement the same functionality. They are also known as Type 4 or functional clones.

Semantic similarity Semantic similarity over a set of methods is the idea of methods being functionally similar despite being lexically, syntactically or structurally dissimilar.

Sensitivity analysis Sensitivity analysis determines how different values of an independent variable affect a particular dependent variable under a given set of assumptions.

Similarity threshold The similarity threshold is the desired lower limit for the similarity of two data records that belong to the same cluster.

Static program analysis The analysis of computer software that is performed without actually

executing programs, in contrast with dynamic analysis, which is analysis performed on programs while they are executing.

Chapter 1

Introduction

1.1 Software Reuse and Code Recommendation

Software reuse is a common practice in the development and maintenance of a modern software system [1]. Software developers need to search for reusable code that would assist them in implementing a given feature or development task. Often, the timely discovery of a critical piece of information can have a dramatic impact on developers' productivity [2]. Given a software requirements document, developers typically search for code to implement the features listed in the document. They need to find reusable code for the desired features in a way that supports opportunistic programming [3], which is an iterative process of adding features to code under development. Current code search systems [4–7] focus on providing code against a specific user query and repeated searches need to be performed until the code for the desired feature or set of features is found. The problem of repeated code searches needs to be addressed to improve developer's productivity and facilitate rapid application development. As opposed to code recommendation systems, existing feature recommendation systems [8, 9] enable the exploration of related features for rapid application development or domain analysis. Features exist at various levels of granularity in source code, from the broad package-level to the more fine-grained method-level [10]. The existing feature recommendation systems do not provide code at a fine-grained method level for reuse. Two gaps are identified in existing systems; one is the lack of support for opportunistic reuse in existing code

recommendation systems and the other is inability of existing feature recommendation systems to provide functionality at a fine-grained level. A solution is desired for the recommendation of related methods for reuse in an opportunistic manner that enhances productivity. Figure 1.1 shows an application developer making a query to play a media file. The developer not only gets code to play a media file, they also get related method recommendations they might need to implement later for their application.

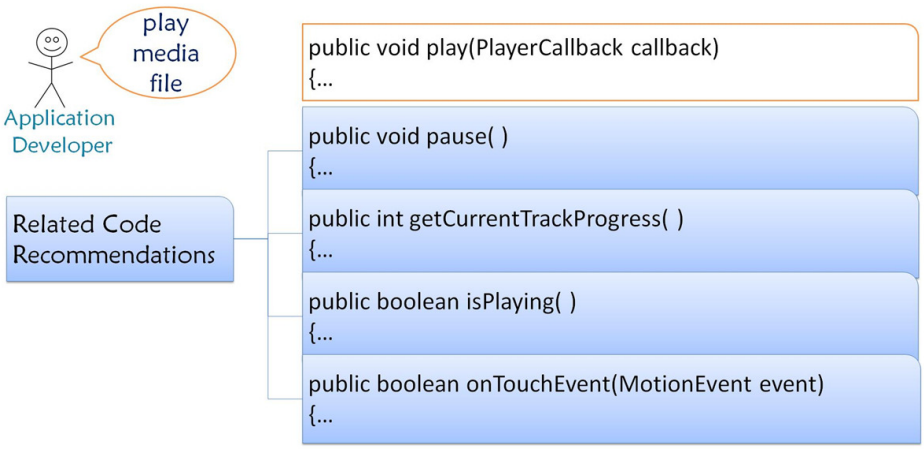


Figure 1.1: A recommendation scenario for recommending related code against a user query

In this research, our **objective** is to develop a recommendation system that provides developers with method recommendations having functionality relevant to their feature or development task, thus enabling opportunistic reuse for increased developer productivity. The problem of repeated code searches needs to be addressed to improve developer's productivity and facilitate rapid application development (RAD). To the best of our knowledge, no existing system recommends related features at the method-level granularity.

We employ a combination of information retrieval, static program analysis and data mining techniques to build the proposed recommendation system called FACER (**F**eature-driven **A**PI usage based **C**ode **E**xamples **R**ecommender). The idea behind FACER is to provide code recommendations against a feature query based on patterns of frequently co-occurring API usage-based Method Clone Classes. Such patterns are called Method Clone Structures. The heuristic behind API usage-based Method Clone Classes is that methods with similar uses of APIs are semanti-

cally related because they do similar things [7], and are identified as members of a clone class. The heuristic behind patterns of frequently co-occurring Method Clone Classes is based on market basket analysis [11]. Market basket analysis attempts to identify associations, or patterns, between the various items that have been chosen by a particular shopper and placed in their market basket. Items that frequently co-occur are related to each other. In the context of software projects, a particular project may use a group of methods which are seen cloned across other projects. Such a pattern of co-occurring method clones forms a Method Clone Structure, which identifies related functionality and forms the basis of suggesting relevant methods for opportunistic reuse.

FACER generates related method recommendations in two stages. In the first stage, the developer provides a feature query (expressed as a comment) to get a set of methods that implement the feature. In the second stage, upon selection of one of these methods by the developer, a subsequent recommendation provides related methods for opportunistic reuse. Thus, the second stage provides recommendations based on feedback from the developer.

FACER has a repository component and a recommender component. The repository consists of data mined from applications' source code and includes methods' search index, method call sequences, API usages, API usage-based Method Clone Classes, and API usage-based Method Clone Structures. A user-selected method is checked by the recommender to see which clone class it belongs to and whether that clone class belongs to an API usage-based Method Clone Structure. Then, the representative methods of the Method Clone Classes which are members of the Method Clone Structure are recommended as related methods. When MCS membership does not exist for some selected method, a call graph traversal retrieves related functionality from caller and callee methods.

Figure 1.2 shows methods nested in files which are in turn nested in a project. Methods which are cloned within and across three media player applications are shown in color. Using this figure, we can estimate the presence of method clones and Method Clone Structures in application source code and thus understand the potential of generating code recommendations from related code identified as Method Clone Structures.

Recommender systems in other fields have used the idea of feedback from a user for improved

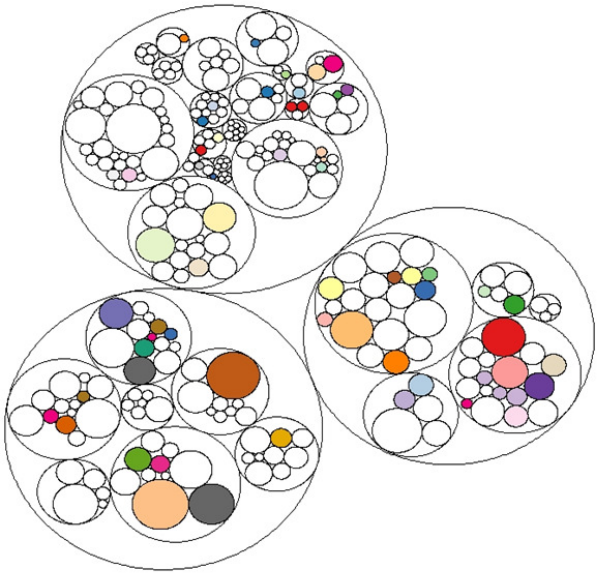


Figure 1.2: Methods which are cloned within and across three media player applications are shown in color

recommendation [12–15]. For code recommendation, the context of a developer can be leveraged to provide better personalized code recommendations. Existing context-aware code recommendation systems have been shown to support developers in code completion [16–29] and code reuse [26, 30–37]. However, there are currently no existing context-aware systems that can provide code recommendations of multiple related features for opportunistic reuse on-the-go. With the passage of time, the activity of a developer on their project can increase the amount of code written by the developer, which results in an evolving development context. While FACER is shown to perform well against user queries, the capabilities of FACER in evolving development contexts need to be investigated. For this dissertation, we focus on evaluating the need for context-awareness in FACER and designing a context-aware approach for opportunistic code reuse.

1.2 Motivating Examples

In this section we discuss two examples that demonstrate the related code recommendations against a user's feature request. First is a music player application and second is a Bluetooth chat application.

1.2.1 Music Player Application

Consider the example of an Android music player application. Suppose the developer wants to implement a feature that allows her to play a media file. Using an online code search engine, code that contains the functionality for playing a media file can be obtained. Current search engines stop at this point. They do not provide further suggestions of relevant functionality that may be useful for the developer. Our system allows the user to select a particular function from an initial retrieved list and based on her selection, related functions are suggested.

For example, if the developer queries for the feature "play media file", then our system recommends a list of top relevant functions. Assume that the user-selected function is as shown in Listing 1.1.

```
public void play(PlayerCallback callback)
{
if(mediaPlayer == null)
{
if(callback != null)
callback.onFailure(player);
return;
}
mediaPlayer.start();
if(callback != null)
callback.onSuccess(player);
}
```

Listing 1.1: Function implementing the *play media file* feature

Our system will output functions that are related to the developer's initial request of playing a media file as shown in Listing 1.2,1.3 and 1.4. The user-selected function called the MediaPlayer APIs start method and the related functions recommended also call various methods of the MediaPlayer API and implement functionality related to playing a media file such as pausing, getting

current track progress and checking to see if a file is playing or not. An interesting suggestion is that of the functionality of handling a touch event as shown in Listing 1.5, which is useful for interactivity of the user with the application's UI. Although this functionality is not directly manipulating a media file, it is related to "play media file" feature as it allows user interaction while issuing the command to play a file. By receiving such related function recommendations, the developer can obtain information about related features to enhance her application. Furthermore, the need to perform repeated searches is reduced.

```
public void pause(){
try{
mediaPlayer.pause();
}catch (IllegalStateException ignored){}
setStatus(PAUSED);
setSessionState();
PostNotification();
updateWidget(false);
}
```

Listing 1.2: Function implementing the *pause* feature

```
public int getCurrentTrackProgress(){
if (status > STOPPED){
try{
return mediaPlayer.getCurrentPosition();
}catch (IllegalStateException e) {
return 0;
}
}
else{
return 0;
}
}
```

Listing 1.3: Function implementing the *get current track progress* feature

```
public boolean isPlaying(){
if(mediaplayer != null){
if( mediaplayer.isPlaying() ){
return true;
}
}
return false;
```

```
}
```

Listing 1.4: Function implementing the *is playing* feature

```
public boolean onTouchEvent(MotionEvent event){
switch(event.getAction()){
case MotionEvent.ACTION_DOWN:
requestFocus();
// We use raw x because this window itself is going to
// move, which will screw up the "local" coordinates
mListener.markerTouchStart(this, event.getRawX());
break;
case MotionEvent.ACTION_MOVE:
// We use raw x because this window itself is going to
// move, which will screw up the "local" coordinates
mListener.markerTouchMove(this, event.getRawX());
break;
case MotionEvent.ACTION_UP:
mListener.markerTouchEnd(this);
break;
}
return true;
}
```

Listing 1.5: Function implementing the *handle touch event* feature

1.2.2 Bluetooth Chat Application

Consider the example of an Android Bluetooth Chat application. Suppose the developer wants to implement a feature that allows him to connect to a Bluetooth device. For example, if the developer queries for a feature “connect to a Bluetooth device”, then our system returns a list of top relevant functions. Assume that the user selected function is as shown in Listing 1.6.

```
/**
* Start the ConnectThread to initiate a connection to a remote device.
*
* @param device The BluetoothDevice to connect
* @param secure Socket Security type - Secure (true) , Insecure (false)
*/
public synchronized void connect(BluetoothDevice device, boolean secure) {
Log.d(TAG, "connect to: " + device);

// Cancel any thread attempting to make a connection
if (mState == STATE_CONNECTING) {
```

```
if (mConnectThread != null) {
mConnectThread.cancel();
mConnectThread = null;
}
}

// Cancel any thread currently running a connection
if (mConnectedThread != null) {
mConnectedThread.cancel();
mConnectedThread = null;
}

// Start the thread to connect with the given device
mConnectThread = new ConnectThread(device, secure);
mConnectThread.start();
// Update UI title
updateUserInterfaceTitle();
}
```

Listing 1.6: Function implementing the *connect to bluetooth device* feature

Our system will output functions which are related to the developer's initial request of connecting to a Bluetooth device as shown in Listing 1.7, 1.8 and 1.9. The systems suggests functionality related to enabling Bluetooth on the device, setting up input and output communication channels and closing the connection.

```
protected void enableBluetooth() {
bluetoothAdapter = BluetoothAdapter.getDefaultAdapter();
Intent enableBluetoothIntent = new
    Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
startActivityForResult(enableBluetoothIntent, ENABLE_BT_REQUEST_CODE);
}

public int onStartCommand(Intent intent, int flags, int startId) {

mBleEventBus = BleEventBus.getInstance();
mBleEventBus.register(this);

final BluetoothManager bluetoothManager = (BluetoothManager)
    getSystemService(this.BLUETOOTH_SERVICE);
mBluetoothAdapter = bluetoothManager.getAdapter();

if (mBluetoothAdapter == null || !mBluetoothAdapter.isEnabled()) {
stopSelf();
} else {
scanLeDevice(true);
}
```

```
return START_NOT_STICKY;
}
```

Listing 1.7: Functions implementing the *enable bluetooth* feature

```
IOThread(int threadId, BluetoothSocket socket, BTSocket btSocket) {
mThreadId = threadId;
mBluetoothSocket = socket;
mBTSocket = btSocket;

InputStream tmpIn = null;
OutputStream tmpOut = null;
try {
tmpIn = socket.getInputStream();
tmpOut = socket.getOutputStream();
} catch (IOException e) {
e.printStackTrace();
}
mInStream = tmpIn;
mOutStream = tmpOut;
}
```

Listing 1.8: Function implementing the *establishing input output channels* feature

```
public void cancel() {
try {
mmSocket.close();
} catch (IOException e) { }
}
```

Listing 1.9: Function implementing the *close bluetooth connection* feature

1.3 Problem Statement

There is a large amount of work in the literature [38] covering various code search and recommendation techniques for a particular code granularity (snippet, API call, API method, API class, method, class, component, file, library, etc.) and particular search intent (code reuse, code completion, reference example, bug-fixing, etc.). However, existing code search systems do not support the proactive recommendation of related code that may be relevant to the features of an application under development. On the other hand, existing feature recommendation systems enable domain

analysts in discovering related and relevant features for their application. However, these feature recommendation systems do not provide associated code against recommended features. In this dissertation, we aim to close this gap in existing code search systems by proposing a system that facilitates rapid application development and allows a developer to receive recommendations of multiple related features to implement various functionalities of their application through opportunistic reuse.

Here we summarize the current limitations of code search and recommendation systems that we have observed in the literature and which we are focusing on in this dissertation.

1. Existing code search engines do not support opportunistic reuse. They are effective for locating code against a single feature, but they are not designed to provide code for additional relevant features [9]. As a result, developers might have to conduct a new search for every next feature that they need to implement and later integrate the obtained code.

2. Existing feature recommender systems do not provide associated code against recommended features. Only one system allows the location of Java packages relevant to related features [9]. However, packages have a lot of unrelated code and may be grouping together classes implementing unrelated responsibilities with low cohesion [39, 40]. On the other hand, code suggestions at the method-level granularity provide more concrete reusable code [41], which the existing feature recommender systems do not provide.

3. Although some existing code completion and recommendation systems are context-sensitive, the purpose for context-awareness is mostly limited to automatic query formulation [16, 18, 21, 30, 32, 33, 42–48]. The benefits of context-awareness can be extended to improve the accuracy of recommendations [49] by being able to filter and re-rank recommendations.

We use an example from a real Stack Overflow user question [50] to demonstrate the problem of developers spending a lot of time searching for related functionality. The title of the question is "android:select image from gallery then crop that and show in an imageview" and the description of the question includes the following:

"I really need this code and I searched for 3 hours on internet but i couldn't find a complete and simple code and I tested many codes but some didn't work and others [weren't] good, please help me with a full and simple code ... edit:I have this code for select image but please give me a full code for all [the] things that i said in title ..."

This question is viewed 50K times on Stack Overflow. The user is looking for code that allows her to perform three functionalities: first, selecting an image from gallery, second, cropping the image and third, showing the image in an image view. She has spent a lot of time searching for these related functionalities. StackOverflow lists questions that are linked to this user question in a "Linked" sidebar. These linked questions include "How to pick an image from gallery and save within the app after cropping?" and "crop image by taking photo from camera". We see that a number of functionalities are frequently desired together; users who select an image from a gallery need to crop it and then show it in an image view or save it after cropping. A user may need to crop an image after selecting it from a gallery or capturing it from the camera. There are numerous Android applications from a diverse range of categories including photo sharing applications, photo editing applications, virtual try-on applications for eye glasses, etc. in which all of these functionalities related to manipulating images are present. The current gap in existing code recommendation systems is that they do not cater to the user need for finding related functionality. McMillan et al. highlight programmers' need of accomplishing a whole task quickly, rather than obtaining multiple examples for different components of the task [6].

We now explain how our approach could have helped in this situation. To illustrate this, we collect a set of 30 photo sharing applications from GitHub using the search string "android photo sharing app", sort the results by GitHub Stars and choose the top 30 most relevant apps to populate a sample FACER repository. We then enter a search query to our system "select image from gallery" and from the recommended methods, we select one that contains the desired functionality (shown in Figure 1.3a). Next, we use FACER to retrieve related method recommendations against the selected method. The related methods FACER recommended included methods that implement the above Stack Overflow user's desired features of cropping an image and also showing an image in an *ImageView* as shown in Figures 1.3b and 1.3c respectively. FACER also recommended addi-

```
public static Intent getPickImageChooserIntent(
   @NonNull Context context, CharSequence title, boolean includeDocuments) {
       List<Intent> allIntents = new ArrayList<>();
       PackageManager packageManager = context.getPackageManager();
       if (!isExplicitCameraPermissionRequired(context)) {
           allIntents.addAll(getCameraIntents(context, packageManager));
       }
       List<Intent> galleryIntents = getGalleryIntents(
      packageManager, Intent.ACTION_GET_CONTENT, includeDocuments);
       if (galleryIntents.size() == 0) {
           galleryIntents = getGalleryIntents(packageManager,
           Intent.ACTION_PICK, includeDocuments);
       }
       allIntents.addAll(galleryIntents);
       Intent target;
       if (allIntents.isEmpty()) {
           target = new Intent();
       } else {
           target = allIntents.get(allIntents.size() - 1);
           allIntents.remove(allIntents.size() - 1);
       }
       Intent chooserIntent = Intent.createChooser(target, title);
       chooserIntent.putExtra(Intent.EXTRA_INITIAL_INTENTS,
        allIntents.toArray(new Parcelable[allIntents.size()]));
       return chooserIntent;
}
```

(a) Selected code snippet

```
private static Bitmap cropBitmapObjectWithScale(Bitmap bitmap, float[] points,
   int degreesRotated, boolean fixAspectRatio, int aspectRatioX, int aspectRatioY,
   float scale) {
   Rect rect = getRectFromPoints(points, bitmap.getWidth(), bitmap.getHeight(),
            fixAspectRatio, aspectRatioX, aspectRatioY);
   Matrix matrix = new Matrix();
   matrix.setScale(scale, scale);
   matrix.postRotate(degreesRotated, bitmap.getWidth() / 2, bitmap.getHeight() / 2);
   Bitmap result = Bitmap.createBitmap(bitmap, rect.left, rect.top,
           rect.width(), rect.height(), matrix, true);
   if (result == bitmap) {
      result = bitmap.copy(bitmap.getConfig(), false);
   }
   if (degreesRotated % 90 != 0) {
      result = cropForRotatedImage(result, points, rect, degreesRotated,
            fixAspectRatio, aspectRatioX, aspectRatioY);
   }
   return result;
}
```

(b) Crop image

```
@Override
public View getView(int i, View view, ViewGroup viewGroup)
{
   ImageView imageView;
   if (view == null) {
   int gridWidth = fragment.getScreenWidth();
   imageView = new ImageView(mContext);
   imageView.setLayoutParams(
   new GridView.LayoutParams(gridWidth/5 - 30, gridWidth/5 - 30));
   imageView.setScaleType(ImageView.ScaleType.FIT_CENTER);
   imageView.setPadding(5, 5, 5, 5);
   }else {imageView = (ImageView) view;}
   Bitmap bmp = getResizedBitmap(loadImage(imageFileNames.get(i)), 200);
   imageView.setImageBitmap(bmp);
   return imageView;
}
```

(c) Show image in ImageView

Figure 1.3: Motivating example for code recommendations related to “select image from gallery”. Figure 1.3a shows the selected code snippet based on the initial search query and Figures 1.3b and 1.3c show code snippets corresponding to two related features, as recommended by FACER.

tional related methods, which include functionality for resizing a bitmap, getting a URI to save the cropped image, getting the URI to an image received from a capture by camera, and decoding an image from a URI. By receiving such related method recommendations, the developer can obtain information about related features, in the form of concrete methods, to enhance their application. Furthermore, this reduces the need to perform repeated searches.

1.4 Our Contributions

In this dissertation, we propose a context-aware code recommendation system for providing related code for opportunistic reuse. Our main contributions include identifying the need for related code recommendations, detecting semantically similar functionality as API usage-based Method Clone Groups, recommending related code using Method Clone Structures, and making context-aware code recommendations using a hybrid context-aware paradigm.

Our work makes the following main contributions:

1.4.1 A Code Recommendation Approach for Related Features

We present a recommendation approach named FACER that recommends methods to implement additional features related to the developers' currently searched feature. These recommended additional methods are based on API usage-based Method Clone Groups and Method Clone Structures. Our evaluation results show that FACER achieves 80% precision, on average. Our survey results show that 90% of the professional developers perceive that FACER is effective for their development activities and 95% of the developers find the related method recommendations useful.

1.4.2 A Study of Context-awareness for Code Recommender Systems

We study existing context-aware code recommender systems that facilitate code completion and code reuse. We highlight each system's recommendation category, their triggers to obtain contextual data, the scope of context, the elements that constitute the context, and the contextual modeling paradigm.

1.4.3 A Context-aware Code Recommendation Approach for Related Features

We introduce the idea of capturing context as multiple sets of API usages representing a variety of functionality or software features, where each set of API usages comes from a single method body. Furthermore, we leverage multiple sources to obtain contextual data which include a developer's active development profile, code reuse history, and organizational development activity. We propose a novel hybrid context-aware approach (CA-FACER) to recommend related methods for opportunistic reuse. CA-FACER achieves a 46% improvement over baseline FACER. Furthermore, CA-FACER recommends related code examples with an average precision (P@5) of 94% and 83% and a success rate of 90% and 95% for initial and evolved development stages respectively.

1.4.4 IDE-Integrated Tools for Eclipse and Android

We develop an Eclipse plugin for Java developers and an Android Studio plugin for Android developers that allows developers to use FACER directly from their IDEs.

1.5 Outline of the Dissertation

This dissertation contains nine chapters in total. We conduct three independent but interrelated studies for the recommendation of related code examples for opportunistic reuse. This section outlines different chapters of the dissertation as follows.

1. **Chapter 2** provides a background overview on opportunistic code reuse, code clones, code search and recommendation systems, and context-awareness.

2. **Chapter 3** covers the related literature in the areas of code search systems, code recommendation systems, feature recommendation systems, and context-aware code recommendation systems. Finally, some limitations of code search and feature recommendation systems are discussed.

3. **Chapter 4** presents our first study namely **CodeEase** that uses code completion and Type-2 and Type-3 Method Clone Structures to recommend code examples for reuse.

4. **Chapter 5** presents our second study namely **FACER** that uses code search and API-Usage Based Method Clone Structures to recommend code examples for reuse.

5. **Chapter 6** presents the research questions, evaluation details, and results of our study on using FACER for code recommendation and opportunistic code reuse.

6. **Chapter 7** presents our third study namely **CA-FACER** that incorporates a context-aware mechanism with FACER to recommend more relevant code examples for opportunistic reuse.

7. **Chapter 8** presents the research questions, evaluation details, and results of comparing various context-aware approaches of CA-FACER and evaluating the effect of context size.

8. **Chapter 9** concludes the dissertation with a summarized list of achievements and future research directions inspired by this PhD dissertation.

Part of the contents of Chapter 1 Section 1.3, Chapter 3 Sections 3.1, 3.2, 3.3, Chapter 5, and Chapter 6 are from our journal publication [51] and are reproduced with permission from Springer Nature.

Chapter 2

Background

2.1 Opportunistic Code Reuse

Opportunistic reuse involves extending software with functionality from a third-party software supplier that wasn't originally intended to be integrated and reused [52]. Thus *opportunistic code reuse* is the ad hoc reuse of code in the development of software systems. The availability of open source assets for almost all imaginable domains has led the software industry to *opportunistic design*; an approach in which people develop new software systems in an ad hoc fashion by reusing and combining components that were not designed to be used together [53]. Compared to systematic reuse methodologies [54, 55] proposed more than twenty years ago, today people routinely trawl for ready-made solutions for specific problems online and try to discover libraries and code snippets to be included in their applications [53].

Studies of rapid prototype development have shown that programmers iteratively add features by reusing source code examples [3, 56]. This iterative process is known as *opportunistic programming* [3] which enables rapid application development, enhances developer productivity and saves time [3, 52, 57, 58].

We discuss a simple recommendation scenario that demonstrates the idea of opportunistic reuse. Consider the example of an Android media player application. Suppose the developer wants to implement a feature that allows her to play a media file. Using any code search engine,

she can obtain code that contains the functionality for playing a media file. Assume that the developer queries for the feature *"play media file"*, and receives a list of top relevant methods from the code search engine. Now, let us assume that the developer selects the method shown in Figure 2.1 from that list.

```
public void play(PlayerCallback callback){
    if(mediaPlayer == null){
        if(callback != null)
            callback.onFailure(player);
        return;
    }
    mediaPlayer.start();
    if(callback != null)
        callback.onSuccess(player);
}
```

Figure 2.1: Method implementing the 'play media file' feature

Current search engines stop at this point. They do not provide further suggestions of relevant functionality that may be useful for the developer. Our goal is that once the developer selects a particular method for reuse from the initially retrieved list, we can recommend code from related features that they might want to also implement.

Thus, given the selected method in Figure 2.1, FACER outputs additional methods that implement related functionality. Figures 2.2-2.4 show three such recommended methods, all of which correspond to features that are related to the developer's initial task of playing a media file. Note

```
public void pause(){
    try{
        mediaPlayer.pause();
    }
    catch (IllegalStateException ignored){}
    setStatus(PAUSED);
    setSessionState();
    PostNotification();
    updateWidget(false);
}
```

Figure 2.2: Method implementing the 'pause media file' feature

how the user-selected method in Figure 2.1 calls the *MediaPlayer* API's *start* method while the re-

```
public boolean isPlaying(){
    if(mediaplayer != null){
        if(mediaplayer.isPlaying()){
            return true;
        }
    }
    return false;
}
```

Figure 2.3: Method implementing the 'is playing' feature

```
public int getCurrentTrackProgress(){
    if (status > STOPPED){
        try{
            return mediaPlayer.getCurrentPosition();
        }catch (IllegalStateException e) {
            return 0;
        }
    }
    else{
        return 0;
    }
}
```

Figure 2.4: Method implementing the 'get current track progress' feature

lated recommended methods in Figures 2.2-2.4 also call various methods of the *MediaPlayer* API and implement functionality related to playing a media file such as pausing, getting current track progress, and checking to see if a file is playing or not. By receiving such related method recommendations, the developer can obtain information about related features to enhance her application. Furthermore, this can reduce the need to perform repeated searches.

2.2 Code Clones

Code clones are similar code fragments that may be completely identical or may have some lexical, syntactic, or structural differences [59].

Clone Types: There are two main kinds of similarity between code fragments. Fragments can be similar based on the similarity of their program text, or they can be similar based on their functionality (independent of their text). The first kind of clone is often the result of copying a

code fragment and pasting into another location. Following are four types of clones based on both the textual (Types 1 to 3) [60] and functional (Type 4) [61, 62] similarities [63]:

Type-1: This category includes code fragments which are textually similar and may have differences in white-space, comments, and layout.

Type-2: This category includes code fragments having lexical similarity i.e. code fragments are identical with the only differences in identifier names and literal values.

Type-3: These are code fragments that are syntactically similar. Such code fragments can differ at the statement level with statements being added, modified or removed. They are also called *near-miss clones* or *gapped clones*.

Type-4: This category includes code fragments that are semantically similar in terms of functionality, but possibly different in how the functionality is implemented. These types of clones may have little or no lexical or syntactic similarity and hence are relatively difficult to detect. They are also called *semantic clones* or *functional clones*.

2.3 Code Search

Code search refers to retrieval of relevant code snippets from a code base, according to the intent of a developer that they have expressed as a search query [64]. With public online code repositories, code search is now a big data science problem [65, 66]. Various code search and code recommendation systems have been proposed in the literature. Sadowski [67] found that developers search for examples more than anything else which includes searching for an example to build off, discovering the library for some task and discovering the usage of some API. To understand why programmers search for code, Umarji et al. [68] conducted a web-based survey and found that they could be categorized along two orthogonal dimensions: motivation (reuse vs. reference example) and size of search target.The targets of these searches could range in size from a block (a few lines of code) to a subsystem (e.g. library or API), to an entire system.

Code search also supports many other software engineering tasks. For example, code search tools are helpful for bug/defect localization [4, 61, 94, 119, 125, 126, 129], program repair [2, 80], and code synthesis [98, 99], among others.

Code search tools typically involve the following six key components [64]:

1. **Codebase** In the code search task, the codebase defines the target search space. Code for the codebase may be private and present within an organization or collected from online public code repositories such as GitHub and F-Droid, or online developer forums such as StackOverflow.

2. **Query** Search queries are input to code search tools to express the search requirement during a specific software development task. Existing code search tools can support queries in different forms. For example, free-form text written in natural language is the most common query [5–7, 40, 69–74]. Other code search tools support a more structured code-based query to find relevant code [32, 44, 45, 75–78].

3. **Model** Code search tools may be built using three types of modeling techniques. A traditional model performs code search according to a relevancy algorithm (e.g., TF-IDF) between query and candidate code. Heuristic models leverage code analysis techniques to capture the syntactic and semantic features in code and then rank code with customized matching approaches [7, 79, 80]. Finally, learning-based models learn the relationship between code and query using a large-scale dataset. Machine learning techniques have recently been used for building code search tools and shown high accuracy [73, 74, 81–84].

4. **Auxiliary Technique** To improve performance, code search models are usually associated with auxiliary techniques, such as query reformulation [85–87], code clustering [88], and learning from user feedback [89].

5. **Evaluation Method** To evaluate the validity of a code search technique, the relevancy of retrieved code against a query needs to be assessed. Since the ground-truth is difficult to measure, manual identification is the most prevalent method, however, it cannot scale to large numbers of queries. Thus, other automated evaluation methods are also used based on collected query-code pairs or algorithms that judge query code relevancy.

6. **Performance** Code search tool performance is based on relevancy of the returned code and

rank of the best match in a list of results. Performance can be measured using classification metrics, such as Precision, Recall, and F1-score. When the position or rank of the correctly searched code in the result list is to be measured, MRR (Mean Reciprocal Rank) and NDCG (normalized discounted cumulative gain)[90] are used.

2.4 Code Recommendation

Generally, recommendation systems in software engineering are software applications that provide information items estimated to be valuable for a software engineering task in a given context [38]. Software engineering tasks may include development tasks, maintenance tasks, code exploration, bug fixing, clone detection, testing, debugging, and refactoring among others. While various recommendations systems support such software engineering tasks, our focus is on Reuse-Oriented Code Recommendation (ROCR) systems. A ROCR system includes any tool that recommends code artifacts of any kind and size for the sake of supporting reuse tasks.

The role of code recommendation tools is to non-intrusively and reliably find and recommend high quality code artifacts leveraging software reuse and to help developers integrate them into their systems with minimal effort [38]. A ROCR system consists of two main parts; a search engine or repository hosting the code base used to search for recommendations and the actual recommendation engine that is responsible for the user and IDE interaction [38].

Typically, a ROCR system process consists of five steps; first, a developer need to consciously decide to reuse an artifact, second, they need to describe what they are looking for, third, a search delivers a number of candidate results, fourth is the selection and ranking of results by relevance, and fifth is the reuse and integration of a selected artifact into a developer's IDE. Different code recommendation systems work on different inputs from the user to provide code. The input may be a free form textual query or it may include components of a user's currently active code like method signature, keywords or its structural information. The output of a code recommendation systems may also vary and ranges from finer-grained statement-level code suggestions to large-grained method body-level, class-level, or package-level code suggestions.

2.5 Context-aware Paradigms

A code recommendation system can leverage information about the development context to efficiently find potential recommendations. Depending on the kind of recommendation system, context awareness may range from the immediate environment of the cursor to the source code of the whole project [38]. Here we explain three different algorithmic paradigms – pre-filtering, post-filtering, and modeling – for incorporating contextual information into the recommendation process [91]:

- In contextual pre-filtering methods, the contextual information is used to filter the dataset before applying traditional recommendation algorithms.
- In contextual post filtering, the recommendation algorithm ignores the context information. The output of the algorithm is filtered/adjusted to include only the recommendations that are relevant in the target context.
- In contextual modeling, context data is explicitly used in the prediction model as an explicit predictor of a recommended item's rating. The contextual modeling approach may use either predictive models or heuristic calculations.

While reviewing literature pertaining to context-aware code recommender systems, we identify the context-aware paradigms implemented by these systems. We also use these three basic context-aware paradigms to design and develop context-aware architectures of FACER.

2.6 Chapter Summary

This chapter covered the basic concepts of opportunistic code reuse, code clones, code search, code recommendation, and context-aware paradigms. These concepts will be helpful in understanding the contents of the remaining sections of the dissertation.

Chapter 3

Related Work

There are various types of techniques and support systems proposed in the literature to enable code search, code reuse, and feature exploration. In this section, we discuss these systems under three major categories; namely, code search, code recommendation, and feature recommendation systems. We first define each category and describe the purpose of systems belonging to that category. We then discuss existing systems from each category in relation to our approach. Finally, we highlight the limitations of code search and feature recommendation systems that lead to the inception of our approach.

3.1 Code Search Systems

Code search systems are mainly used to retrieve code samples and reusable open source code from the web [65, 66]. To understand why programmers search for code, Umarji et al. [68] conducted a web-based survey and categorized code search purposes along two orthogonal dimensions: motivation (reuse vs. reference example) and size of search target. The targets of these searches ranged in size from a block (a few lines of code) to a subsystem (e.g. library or API), to an entire system. A study on developers' code search behavior finds that most searches are related to searching for code examples, discovering a library for some task, or discovering the usage of some API [67]. In this section, we discuss code search systems that retrieve code against a user query, because they

are related to the first stage of FACER [5–7, 40, 72–74].

Sniff [72] is one such system which helps users discover code snippets involving library usage. It uses the documentation of the library methods to annotate code with plain English for the purpose of free-form query search. It then takes an intersection of the candidate code snippets obtained from a query search to generate a set of relevant code snippets. The drawback of this technique is the dependency on the availability of library documentation. Our approach is free from this dependency.

The Structural Semantic Indexing (SSI) technique [7] finds API usage examples corresponding to standard keyword-based queries. The authors create a baseline retrieval system that uses a Lucene-based [92] search index of code entities based on their simple name, Fully Qualified Name (FQN), and full method bodies. We implement a similar technique for the first search stage of FACER, but we additionally support free-form queries and tokenize API usages for better matching of queries to source code. We opt for using this technique because of its simplicity and good performance [69, 93].

Keivanloo at al. [5] propose a system that retrieves code examples from a corpus of code snippets based on free-form querying (composed of keywords). They create a *p-string* [94] for each line of a code snippet in the corpus and encode matching *p-strings* as a pattern. By applying identifier splitting techniques on all strings that belong to an encoded pattern, they extract a set of associated keywords. The retrieval involves matching keywords of a user query with keywords associated with patterns and getting the most popular code examples containing the matched patterns.

Portfolio [6] retrieves and ranks relevant functions against query terms that are also connected on a call-graph. They output results as a list of function names and a visualization showing dependencies between retrieved functions. While Portfolio can provide related functions for opportunistic reuse, the functions are limited to call-graph dependencies and therefore, do not cover the scope of an entire project. As such, it might not be able to retrieve related functions that are not necessarily found on call chains.

Ishihara et al. [40] use source-code clone detection to find instances of copy-paste reuse scenarios. Keywords are extracted from the clones and saved in a database. Code is then retrieved

against a user query using keyword matching. This search technique is effective in organizations where similar projects are frequently developed and a local source code repository is maintained.

Several of the latest code search techniques that find code given a natural language query rely on machine learning techniques (e.g., *NCS* [74], *DeepCS* [73], *UNIF* [81], *MMAN* [82], *TBCAA* [83], and *CoaCor* [84]). *NCS* proposes an enhanced word embedding for a natural language query [74]. The NCS model captures the co-occurrence frequency of word pairs from Stack Overflow questions and their respective code solutions. Using its model, NCS finds synonyms of query words to enhance the query and improve the quality of recommendations. *DeepCS* [73] introduces the use of a unified vector representation of code and natural language descriptions. This unified representation bridges the lexical gap between queries and source code resulting in relevant code fragments that do not necessarily contain query words. *UNIF* [81] is an extension of NCS that adds supervision to modify embeddings during training with the overall effect of improving the performance for code search. *MMAN* [82] is a Multi-Modal Attention Network for semantic source code retrieval. It generates a code representation that covers both unstructured and structured features of source code including code tokens, abstract syntax trees, and control flow graphs to form a single hybrid representation. This has been shown to outperform *DeepCS*. *TBCAA* [83] employs tree-based convolution over API-enhanced ASTs for semantics-based code search. This technique aims to capture semantics by incorporating API call information into ASTs which is otherwise abstracted as the same AST node type. *CoaCor* [84] uses reinforcement learning to build a code annotation framework for effective code retrieval. By generating detailed code annotations using multiple keywords, *CoaCor* improves the performance of existing code retrieval models.

For our purposes of locating code against a user query in the first stage of FACER, any of the above code search methods would work. We currently use Lucene [93] to build the code search engine behind FACER. Lucene is a popular search library for the development of various information retrieval solutions because of its scalability, high-performance and efficient search algorithms [95]. It is shown to answer the highest number of queries as compared to other code search approaches [96].

3.2 Code Recommendation Systems

In general, *recommendation systems* aid people to find relevant information and to make decisions when performing particular tasks. Different code recommendation systems use different user inputs to provide the output code. The input may be a free form textual query or it may include components of a user's currently active code environment like method signatures, keywords, or structural information. We focus on *source code-based recommendation systems* (SCoReS) [97], that is, recommendation systems that produce their recommendations by essentially analyzing the source code of a software system. Given our scope, we review only a subset of the research that provides code-snippet based recommendations as output.

Some of the earliest code recommendation systems for methods are *CodeBroker* [30] and *Strathcona* [31, 47, 98]. CodeBroker uses comments and method signatures of a yet-to-be-written method to retrieve relevant code, whereas in Strathcona, the search query is either the structural information of some class or method (signature, object instantiations) that the developer needs help for. Others include: *A-Score* [35], which recommends a list of classes against user code based on cosine similarity of code characteristics; Selene [46], which forms a search query from the code around the user's cursor in an IDE and provides code examples from files containing those lines; and ROSF [4], which recommends code snippets against a free-form query by first generating a candidate set of snippets using information retrieval followed by re-ranking the code snippets using a learned prediction model that is trained on a set of user queries and code-snippet features such as text, topic, and structure.

Among existing code recommendation techniques, Ichii et al. [34] allow opportunistic reuse by using collaborative filtering to help developers find components suitable for their needs. This system extracts a developer's browsing history when the developer starts navigating through the search results provided by a SPARS-J [99] search engine. It recommends components to the developer using browsing session similarities based on the assumption that two developers having similar browsing history require similar components. However, it is effective only if developers' browsing profiles are available. *Rascal* [26] is a collaborative filtering-based recommendation system that predicts the next method that a developer could use by analyzing classes similar to the one

currently being developed. It tracks usage histories of developers for recommending components to an individual developer. Here, a component refers to a method call made on a class instance. We rely on a similar notion of collaborative filtering but instead of relying on method usage profiles of classes or browsing session profiles, we rely on feature co-occurrence profiles for projects, where a feature represents a collection of API usages. In the context of FACER, recommended components are complete methods pertaining to a feature.

In previous work, we developed CodeEase [57], which provides method completion recommendations against a partial method as well as related method recommendations for the completed method. CodeEase mines association patterns over a source code collection of Java projects by first detecting type-2 clones and then finding frequently co-occurring inter-project clones [100]. First, CodeEase uses a type-2 clone search to suggest method completions. Then, for a selected method completion, CodeEase looks up methods that occur alongside the selected method in its collection of association patterns. Our internal experiments on mining Method Clone Structures based on type-2 clones proved that patterns detected using traditional clone detection were very rare. This led us to move beyond the notion of detecting similar methods on the basis of type-2 or type-3 clones and to experiment with the notion of functional similarity based on common API usages among methods [7]. Shifting our focus from syntactic matching in conventional clone detection to API calls matching allows us to identify clones as a set of methods having similar behavior irrespective of syntactic differences. As a result, new co-occurrence patterns emerge, offering more possibilities of opportunistic reuse.

3.2.1 API Recommendation Systems

API recommendation systems are a type of code recommendation systems focusing particularly on helping developers use library APIs. Some of these systems recommend code on the basis of mining API usage patterns [25, 28, 101, 102]; however, none of these use the notion of opportunistic reuse of related API usage patterns. There are systems that suggest complete code snippets or usage sequences that demonstrate how to use a given API [103–111]. API class recommendation systems [29, 87, 112, 113] output only the name of a relevant API class against a query.

Thung et al. recommend additional libraries based on the ones currently used by an application or project [114]. Similarly, FACER recommends additional methods based on the one currently selected. Thung et al. find libraries that are commonly used together with the currently used libraries. They also find libraries that are used by the *n* most similar projects, then rate a library based on how many of the top-n projects use it. Their technique is a combination of association rule mining and collaborative filtering to find the top-n libraries. The notion of recommending additional items based on the market-basket principle of frequent co-occurrence is seen in their systems' $LibRec_{RULE}$ component. FACER bases its recommendation of additional methods on the same principle; however, the goal (library vs method) and code analysis techniques used in both cases are different.

3.2.2 Code Completion Systems

Code completion systems [16, 18, 22, 27, 42, 43] suggest completions based on the context of the code being currently edited. Completions may simply be method calls for a given object [16, 18, 22, 27, 42] or can be complete method code for a given partial code snippet [43]. Code completion is an integral feature of modern IDEs [22]. Most of the proposed techniques enable the integration of the code completion recommendations into the active user context, typically within their Integrated Development Environment (IDE). A code recommendation technique is typically at the back-end of code completion systems and so we consider code completion systems to be a sub-type of code recommender systems.

Code completion helps to avoid remembering every detail of the available API methods, write error-free code, speed up typing, and enables the completion of partial method bodies [22]. Hill and Rideout [43] propose automatic method completion based on the idea of atomic clones. Atomic clones are usually small units of implementation of 5-10 lines each, such as implementing a listener interface, or handling a keyboard event. By looking at these atomic clones and comparing them with the current code, a programmer is able to identify any critical points that she should remember to address. *Lancer* [37], a context-aware tool, also assists method completion by analyzing partial method code to recommend relevant code samples. *Lancer* predicts and appends tokens to the

current tokens within the context of a partially written method in order to produce a more complete token sequence for code retrieval. *Lancer* trains a Library-Sensitive Language Model (LSLM) on source code files to capture code patterns for each library separately. Using tokens from the original context, *Lancer* finds relevant libraries and predicts more tokens from these libraries. The final set of tokens is used to retrieve code samples which are further filtered and ranked based on similarity of the original context's tokens with tokens of the retrieved code samples. *Aroma* [76] is another tool that takes partial code as input and recommends code snippets containing the partial code in order to help developers write additional code and complete programming tasks effectively.

Bruch et al. [16] make context sensitive method call recommendations against object instances of a particular framework. Their technique is based on a variant of the K-nearest neighbors algorithm, called Best Matching Neighbors (BMN). The context of the variable is extracted and variables used in similar situations are searched in an example codebase, then method recommendations are synthesized out of these nearest snippets. *CSCC* [22] also performs API method call completion. To recommend completion proposals, *CSCC* ranks candidate methods by the similarities between their contexts and the context of the target call. *SLANG* [27] is a code completion tool for Java that synthesizes complete method invocation sequences, including the arguments for each invocation. It inputs partial code snippet with holes, specified using a special construct and outputs API method calls with parameters as completions for these holes.

GraPacc [18] is a graph-based, pattern-oriented, context-sensitive code completion approach that is based on a database of API usage patterns. *GraPacc* extracts the context-sensitive features from the code being edited and uses these features to search and rank the patterns that best match the current code. When a pattern is selected, the current code is completed via a graph-based code completion algorithm.

MACs [115] is another system aimed at providing code completions for reuse and rapid application development. It recommends code against an input statement for completing an API usage sequence inside a method declaration or for completing code within a class declaration. While *MACs* facilitates code completions by recommending statements at the class or method scope, FACER facilitates feature completions at the project scope. Whereas *MACs* mines co-occurring

associations between individual statements found across code files, FACER mines co-occurring associations between features found across projects.

Our proposed approach goes beyond the completion of a developer's current statement or partial method. FACER's scope of providing completions consists of the current project being developed and the completion proposals are methods containing code for the features relevant to the application.

3.3 Feature Recommendation Systems

Feature recommendation systems are meant to help software requirements engineers or developers with the discovery of new software features for their product by providing a list of relevant software feature descriptions [8, 116–119]. Due to the popularity of mobile applications, recent work proposes solutions for recommending software features for mobile applications [116–118]. For these recommender systems, a feature is recommended as a textual description. Jiang et al. [116] recommend features from applications that are similar to the developer's application. Recommended features are those that frequently co-occur with a developer's feature across highly similar projects. Chen et al. recommend features against a given User Interface (UI) based on user interface comparison of mobile applications [117]. Their idea is based on the intuition that mobile applications with similar UIs may have both shared and unique features. They leverage the similarity of a given UI's components to other similar UIs in order to recommend unique features from the text of similar UIs. Yu et al. propose a hybrid feature recommendation approach that processes both textual descriptions and code information of mobile applications [118]. They detect the most relevant applications against the query and recommend the main features of the relevant applications.

There are feature recommendation approaches not specific to mobile applications. One is proposed by Yu et al. in which a list of related textual feature descriptions are offered to users against an input textual query [119]. The features are taken from marketing-like summaries, release notes and feature descriptions on the online profile pages of products. They perform feature pattern mining from a co-occurrence matrix of software projects and features. Their approach works for

applications hosted on web-based repositories with rich profiles for effective topic modeling. Another approach facilitates domain analysis by recommending features derived from mining product descriptions [8]. In both of these approaches, the recommended features are textual descriptions and do not map to actual code, whereas FACER's feature recommendations are methods with API usages.

The closest work to ours in terms of goals is that by McMillan et al. [9]. Given a natural language query representing a description of their desired product, their system first uses cosine similarity with existing descriptions in software documentation to find related features. After the user confirms the desired features, the system does a feature to module mapping to recommend associated source code modules, specifically Java packages. While their goals are similar in terms of allowing a developer to quickly locate code for multiple related features, there are fundamental differences in terms of code granularity level, techniques used, and user workflow. First, they recommend code at the level of a Java package while we recommend a single atomic method that encapsulates the desired functionality. This allows users to narrow down to relevant code without having to look at an entire package which may have irrelevant code. Second, while they rely on textual cosine similarity, FACER identifies features at the code level based on API usages and looks for co-occurring features in the code. Finally, their system first finds related features and then performs the feature to module mapping, whereas our system performs the query to method mapping first through code search and then finds the related methods based on co-occurrence across different projects.

3.4 Context Aware Code Recommendation Systems

Context-aware systems in software engineering support developers in change tasks, API usage, refactoring, debugging, component recommendation, and code exploration [120] (Table 3.1). The context-aware architectures for code recommendation systems remain undocumented in current research literature. Therefore, we aim to develop a better understanding of the current nature, purpose and use of context for code recommendation.

In this dissertation, we focus only on code recommendation systems that use context in the

Table 3.1: Context-aware approaches for various software development tasks

Change tasks	[121–125]
API usage	[19, 79, 98, 107, 126]
Refactoring	[39, 127]
Problem-solving/debugging/testing	[128–131]
Component recommendation	[26, 34, 132]
Code exploration	[133–135]

recommendation process for code completion and reuse tasks. We study existing context-aware code recommendation systems that facilitate code completion and reuse. We highlight each system's recommendation category, their triggers to obtain contextual data, the scope of context, the elements that constitute the context, and the stage of contextual data usage. Thus, we note the following aspects of current context-aware code recommender systems:

- Category: We categorize existing context-aware code recommender systems based on the item a system recommends. For example, systems which offer code completions may recommend items such as API method names (AM), complete API method calls (AMC), API usage patterns (AUP), method names (M), or library class names (LC). Systems which support reuse may recommend items such as reusable class components (CC), code snippets (CS) or complete method declarations (MD). Table 3.2 shows the categories of recommender systems along with the abbreviations we assign to each category in alphabetical order.

- Trigger: The context extraction process of a recommender system can be triggered either reactively as an explicit user request, or proactively as a result of some event in an IDE [120]. For example, clicking on a button to get recommendations is a reactive context extraction trigger, whereas proactive extraction may be triggered due to various IDE events such as browsing through search results, editing code, writing a code comment, scrolling with a mouse etc.

- Scope: Scope presents space or time aspects [120] of a development environment for obtaining contextual data. The scope of context may range from the immediate environment of the cursor to the source code and/or artifacts of the whole project. The spatial range of

scope can include various code, workspace and project artifacts. The temporal range of context can be defined using the notion of a session (e.g., typically from the launch of the IDE) or a time interval (e.g., month, year, etc.). After reviewing the existing context-aware code recommender systems, we observe different scopes of context shown in Table 3.3.

- Code Elements: We identify the parts of code that are captured during context extraction. Table 3.4 shows the code elements that existing code recommender systems consider as part of context. The code elements are listed in alphabetic order.

- Contextual Modeling Paradigm: We identify the method of incorporating contextual data within a system's recommendation process. As discussed earlier in Section 2.5, there can be three possible paradigms of contextual data usage. First, the post-retrieval filtering (PRF), ranking (PRR) or additional retrieval of results. Second, pre-filtering (PF) to select relevant portions of data. Third, contextual modeling (M), which can be either model-based or heuristics-based. A main distinction between contextual modeling (M) techniques is that the model-based techniques first learn a model on the dataset of items to be recommended and recommendations are generated using the model, whereas for heuristics-based techniques the recommendations are calculated directly from the entire dataset of items.

In Table 3.5, we list some context-aware code recommender systems, tools and approaches that make use of context to make context-sensitive code recommendations to support code completion or reuse. In particular, we identify each system's category, context-extraction trigger, scope of context including code elements, and context-aware paradigm.

3.4.1 Context-aware Code Completion Techniques

All code completion systems are by design context-sensitive because to complete the current code, the currently active code context needs to be taken into consideration. The active code can be used in two ways; 1) to formulate the query for retrieving code completions, or 2) as input to a statistical model-based technique to get rating estimations. The code elements that form the context differ among these systems and may include a single type, methods invoked on that type,

Table 3.2: Code Recommender System Categories

Category	Abbreviation
API Method	AM
API Method Call	AMC
API Usage Pattern	AUP
Class Component	CC
Code Snippet	CS
Library Class	LC
Method	M
Method Declaration	MD

Table 3.3: Scope of Context

Scope	Abbreviation
Current line	CL
x lines before cursor	LB
x tokens before cursor	TB
Code snippet	CS
Current method	CM
Call graph hierarchy	CG
Current class	CC
Current file	CF
Current editing session	ES
Current project	CP
Workspace	W
Search results	SR
User interaction-based	UI
Test case	TC

Table 3.4: Code elements of context

Code Element	Abbreviation
API call	AC
API call sequence	ACS
Abstract Syntax Tree	AST
Class name	C
Code comment	CC
Control Flow	CF
Data Flow	DF
Field	F
Field Type	FT
Interface name	I
Identifier	ID
Java keyword	JK
Library name	L
Library class name	LC
Lexical token	LT
Method name	M
Method invocation	MI
Method invocation prefix	MIP
Method signature element	MS
Parameter type	PT
Receiver type of method invocation	RCT
Return type	RT
Super Class name	SC
Variable type	VT

Table 3.5: Context-sensitive code recommendation systems: A comparison

System	Ref.	Category[a]	Trigger[b]	Scope[c]	Element[d]	Contextual Modeling Paradigm[e]
BMNCCS	[16]	AM	R	CM	VT, RT, M	Model-based
IBCDAMR	[19]	AM	-	CM, LB	ID	Heuristics-based
PBN	[20]	AM	R	CM	VT, RT, M, SC, MI	Model-based
DroidAssist	[21]	AM	R	CM	ACS	Model-based
CSCC	[22]	AM	R	CM, LB	VT, M, C, I, JK	Model-based
PyReco	[23]	AM	R	CF	VT, AST, CF, DF, ACS	Model-based
APIREC	[24]	AM	R	ES, TB	LT, AST	Model-based
HiRec	[136]	AM	-	CM, CG	AC	Model-based
Rascal	[26]	M, AM	P	CC	MI	Heuristics-based
OCompletion	[17]	M	P	CL, ES, CP, CM	MIP, RT, AST, C , MI	PRR
SLANG	[27]	AMC	-	CS	ACS	Model-based
FOCUS	[28]	AMC, CS	-	CP	AC	PF, Model-based
GraPacc	[18]	AUP, CS	R	CM	LT, AST, MI, RT, CF, DF	Model-based
Javawock	[29]	LC	R	CC	LC	PF
Code Conjurer	[33]	CC	P	TC	-	PRF
CF for SPARS-J	[34]	CC	P	ES, UI	C	PRR
A-Score	[35]	CC	P	CF	CC, ID	Heuristics-based
CodeBroker	[30]	MD	P	CM, UI, ES	CC, MS, MI	Heuristics-based, PRF
Strathcona	[47]	MD	R	CC, CM	C, SC, FT, MS, MI	Heuristics-based
Lancer	[37]	MD	P	CM	L, C, M, PT, RT, LT	Model-based, PRR
XSnippet	[32]	CS	R	CP	SC, I, FT	Heuristics-based, PRR
DOI model for Selene	[36]	CS	P	W, UI	C, M, F	Heuristics-based

[a]see Table 3.2; [b](R=Reactive, P=Proactive, -= not specified); [c]see Table 3.3; [d]see Table 3.4; [e](PF=Pre-Filtering, PRR=Post-Retrieval Ranking, PRF=Post-Retrieval Filtering)

identifiers, lexical tokens, Java keywords, method names, class names, interface names, super class names, and API calls from code surrounding a cursor position. For some code completion systems [16, 18–20, 22–24, 26, 136], the completions occur at the level of API methods. *BMNCCS* [16], *OCompletion* [17], and *PBN* [20] specifically improve upon the default order of the API method completions suggested in the IDE's content assist pop-up menu.

Context-aware API Method Recommender Systems

BMNCCS [16] filters those elements from the list of proposals which are irrelevant for the current context. Given a local variable t in the code a developer is working on, *BMNCCS* extracts the variable's context as the declared type of the variable, the names of the methods already invoked on t, and the method within which t is being used. Similarly, it extracts the context for each variable of a training code base. For making recommendations, *BMNCCS* computes the distances between a current programming context and the code base variables' contexts, and identifies method calls to be recommended based on their frequencies in nearest neighbors.

Heinemann et al. [19] provide context-dependent API method recommendation by considering

a certain number of identifiers in the code before cursor and using those as a query for retrieval. They analyze Java classes from existing software systems to build an association index, where an entry in the index associates an API method call with a set of terms extracted from preceding identifiers. The API methods whose terms sets are most similar to query terms are ranked and recommended.

The *PBN* [20] approach uses Bayesian networks for API method recommendation and uses extended contextual information for object usages inside a method. In addition to capturing the type of receiver object, the set of already performed calls on the receiver, and the enclosing method definition, PBN captures the origin of an object instance, the method calls to which the object instance is passed as a parameter, and the super type of its enclosing class.

DroidAssist [21] provides method call recommendations for a currently active object based on existing surrounding method calls. The context for a completion suggestion is extracted within the scope of a currently active method body, and consists of API call sequences. *DroidAssist* learns and builds probabilistic state diagrams to model API object usages in an existing software code base. For API method recommendation, it places each available method at the active cursor position to create new API sequences. It then uses the trained probabilistic models to compute probabilities for the new sequences and assigns a final score to each available method. The methods are ranked by those scores and recommended to the developer.

CSCC [22] also offers API method call recommendations. It captures API method call contexts from code repositories as method names, class or interface names, and Java keywords that occur within four lines of a call. For any method call completion request, CSCC captures the current active context and matches it to those extracted from the code repositories. It then ranks and recommends top three method calls from the top three most similar contexts.

PyReco [23] recommends API methods in Python. It extracts API object usages from an open source code base and uses a nearest neighbor classifier on extracted usage patterns to order API method recommendations based on relevance to a query object's working context. *PyReco* extracts an object's usage context from code within a Python source file by converting the file to an AST, parsing the AST to form a program dependence graph, and then extracting API method calls in-

voked on the object and all the other methods that are invoked between the creation and death of the object. Method call occurrence frequencies across the nearest neighbor contexts are used to rank and recommend the API methods.

APIREC [24] is another code completion tool which recommends an API method call at an editing location based on the code and change context of a developer. It captures code context as a set of code tokens that precede the current editing location. It captures change context as a set of code changes that occurred before the current change in an editing session. These are *change*, *add*, *delete*, and *move* changes to an AST node in a Java program. The editing location change and code context is input to a model trained on change and code co-occurrences to get scores for candidate API calls. Each API call is scored based on its frequent co-occurrence with contextual code changes and tokens to recommend the API call with the highest score.

HiRec [136] is another API method recommender. It is based on the concept of hierarchical context which consists of all third party API methods in a method's call graph. It analyzes the call graph structure of project source code and inlines the project-specific method calls with any third party API call(s) found inside the called method(s). It trains a hierarchy inference model on co-occurring API methods within a hierarchical context.

Context-aware Method Recommenders

Rascal [26] uses collaborative filtering to predict the next method that a developer could use by analyzing classes similar to the one currently being developed. It extracts method usage histories from all Java classes in a source code repository to recommend methods that are expected to be needed by a developer. Methods can be either user-written Java methods or API methods. Based on the last method invocation l in a developer's active code context, Rascal ranks a method recommendation higher if it occurs after l in some usage history.

OCompletion [17] provides method name and class name completion suggestions against partially-written names for a given project under development. Here, we discuss the method invocation completion only. *OCompletion*'s optimistic recommendation algorithm uses a range of contextual information which includes the prefix of a method name to be completed, the receiver

type on which the method is being invoked, the AST of the project being edited, the parent class name of the method being edited, and all those method calls in all sessions in which the parent class was modified. Together, this contextual data is used by *OCompletion* to prioritize the relevant method calls.

Context-aware API Method Call Recommenders

SLANG [27] is a code completion tool for Java that takes as input a partial code snippet with holes, specified using a special construct, and outputs API method calls with parameters as completions for code snippet holes. SLANG extracts sequences of method calls from a large code base and indexes these into a statistical model. Similarly, it extracts API method call sequences from the input partial code snippet and uses the statistical model to compute a set of candidate completion sequences.

FOCUS [28] uses context-aware collaborative filtering to provide API method call recommendations from projects that are similar to the developer's active project. User context is extracted as a set of API method invocations from an active method declaration and also from other method declarations of the active project. This contextual data is used to find similar projects and then to select, rank and recommend API method invocations. Furthermore, FOCUS also recommends code snippets by using the top N method invocations to search for method declarations.

Context-aware API Usage Pattern Recommender

GraPacc [18] helps complete the code under editing based on a database of API usage patterns. It extracts the context features from the current method being edited to obtain lexical tokens and an AST, and forms a *Groum* (Graph-based Object Usage Model) which models actions (i.e. method calls), data (i.e. objects/variables), and control points (i.e. branching points of control structures such as if, while, for, etc). *GraPacc* uses these context features to search, rank and recommend API usage patterns. Finally, for a user-selected pattern, it fills in the code under editing with proper code elements.

Context-aware Library Class Recommender

Javawock [29] is a Java library class recommender system which also uses collaborative filtering. It uses the Java class libraries used in a given program to find similar Java programs inside a code repository and recommends library classes that are used in those similar programs.

3.4.2 Context-aware Code Reuse Techniques

Most of the code recommendation systems shown in Table 3.5 use context to form a query. However, *CodeBroker* [30] additionally uses the context to filter recommendations whereas *XSnippet* [32], *Lancer* [37], and *CF for SPARS-J* [34] additionally use the context to rank recommendations.

Context-aware Class Component Recommenders

Code Conjurer [33] recommends classes as reusable components from a code repository. A developer can define the interface of a desired class component as a special query, and write a test-case that uses the class. When the developer executes the test case, Code Conjurer proactively initiates a search to find matching components against the class extracted from the test-case, and filters out the components that fail on the test-case.

Ichii et al. [34] use collaborative filtering to rank reusable class components retrieved from the SPARS-J component search engine. Their system tracks a developer's browsing session history while the developer browses and interacts with components provided by SPARS-J. It then ranks the retrieved components based on component ratings from similar browsing histories of other SPARS-J users.

A-Score [35] recommends a list of reusable classes against user code from a component repository containing classes indexed using code elements such as comments, class/method/field/local-variable declarations, and method invocations. A-Score proactively generates a query using terms extracted from comments and identifiers in a user's currently active file.

Context-aware Method Declaration Recommenders

CodeBroker [30] proactively recommends task-relevant and personalized reusable components (methods) to developers from a component repository. It uses a doc comment and method signature written by a developer to retrieve matching components using Latent Semantic Analysis (LSA [137]). It filters out components marked as irrelevant by a developer during their development session. It also filters out components which are invoked by the developer based on a developer's personal history of projects developed in the past.

Strathcona [47] uses structural context to recommend code examples relevant to a developer's code being edited. The structural context is extracted from a developer's code being edited to form a query. The contextual data obtained from a developer's class includes the class type, parent's type and field types, whereas for a developer's method, the context includes the method's signature, method invocations, and object instantiations. The query is compared to projects existing in a code repository and structurally relevant code examples are recommended.

Lancer [37] is a context-aware code-to-code recommendation tool which analyzes partial method code to recommend relevant code samples. It extracts contextual data as tokens from library names, method names, return types, and parameter types of a partial method. Lancer predicts and appends tokens to the contextual data extracted to produce a more complete token sequence for code retrieval. Lancer trains a Library-Sensitive Language Model (LSLM) on source code files to capture code patterns for each library used in the code files. Using tokens from context, Lancer finds relevant libraries and predicts more tokens from these libraries by aggregating the predicted probabilities of a token from all language models. The final set of tokens is used to retrieve code samples which are further filtered and ranked based on similarities of contextual tokens with tokens of the retrieved code samples.

Context-aware Code Snippet Recommenders

XSnippet [32] recommends code snippets to a developer writing object instantiation code inside a method. XSnippet can process three types of object instantiations; simple constructor invocations, static method invocations, and a sequence of method invocations. For an object instantiation query,

XSnippet extracts the method's context by identifying the super class extended by its containing class, as well as interfaces implemented by its containing source class. Furthermore, it identifies the set of types of all inherited and local fields, as well as all lexically visible types in the scope of the developer's current method. The contextual data is then used to detect and form code snippets from a graphical representation of source code files in the repository. The recommended snippets are further ranked by matching the types within snippets with the contextual data.

Muarakami et al. [36] introduce the use of the degree-of-interest (DOI) model [138] in conjunction with *Selene* [46] code recommendation system to get better recommendations. The DOI model keeps track of a developer's interactions with code elements (e.g., method, class, field) and assigns each code element a relevance value with respect to the task at hand. A code repository search incorporates the DOI information with the text from a developer's currently active code file to provide relevant code snippets.

3.4.3 Context-awareness for Opportunistic Reuse

After an examination of various code recommender systems, we observe that model-based techniques are more common for code completion than for code reuse. In this dissertation, we propose a novel model-based context-aware code recommendation technique (CA-FACER) that recommends method declarations that are relevant for reuse in an active project of a developer. CA-FACER uses a hybrid contextual modeling approach which includes a combination of pre-filtering, post-filtering and model-based context-sensitive recommendation. CA-FACER is designed to keep track of a developer's interactions with FACER's search results and previously reused methods in an editing session to predict related methods that a developer may need to implement next. Compared to existing code recommenders, CA-FACER belongs to a unique category of code recommenders which cater to opportunistic reuse. Furthermore, CA-FACER uses a unique hybrid contextual modeling approach which exploits the API usages across multiple methods forming a developer's active context to make precise recommendations.

3.5 Chapter Summary

This chapter provided a brief overview of the systems and techniques found in literature for code search, code recommendation, context-aware code recommendation, and feature recommendation. With a large amount of source code available in public code repositories and online developer forums, there is a keen interest in the research community to leverage such collections of code to assist programmers with relevant code recommendations.

Chapter 4

CodeEase: Harnessing Method Clone Structures for Reuse

In this chapter, we present our prototype tool CodeEase, developed as an Eclipse plugin, which generates method recommendations against the code of the developer. The recommendations are based on Type 2 clone detection for code completion and an analysis of Type 2 Method Clone Structures (MCS) for related methods recommendations from a large repository of code. Early experiments with our CodeEase prototype led to some important realizations and limitations of Type-2 clone detection which changed our direction towards Type-4 semantic clones. The CodeEase prototype system was an intermediate step towards the development of our main contribution - FACER, thus we dedicate this chapter to describing our early attempts at providing related code recommendations for opportunistic reuse [57, 139].

4.1 Introduction

A typical problem scenario occurs when a developer is editing the code of some method and has a few lines written. He is either adjusting the code to make it work, or enhancing its functionality. At this stage, providing the developer with suitable method completions would enable him to choose the method that performs his desired functionality. After completing his method, he might want to

search for additional methods that are associated with the functionality of the completed method. In this case, providing method recommendations based on Method Clone Structures (MCS) would enable the user to choose from a list of methods that frequently appear together across various files/projects. In previous work, Basit et al. introduced the concept of structural clones to indicate design-level, large granularity, similar program structures [140, 141], such as similar methods, classes, source files, directories, or recurring combinations of these similar modules. In this work, we evaluate the usefulness of Method Clone Structures (MCS) – a kind of structural clones - for method-level code recommendations. A Method Clone Structure (MCS) consists of a group of methods, which are cloned across modules (e.g., files, sub-systems) of possibly the same or different projects in a large code repository. To better illustrate the idea of MCS, consider the example shown in Figure 4.1. In this example, three methods from the file PeerAdapter.java (a) have clones in the file BaseRecyclerAdapter.java (b). The two files belong to two different chat application software systems Qmunicate* and BLEMeshChat†. Such a group of repeating method clones is called a Method Clone Structure (MCS). The participating methods in a method clone structure are 'friends' of each other because they are seen hanging out together across different files. MCS may be found within the same project or across different projects.

In this chapter, we investigate the following research questions:

RQ 4.1: Are the method clones based recommendations provided by CodeEase useful for the developer?

Method completion recommendations are provided through clone detection and friend method recommendations are provided through MCS detection. By answering this research question, we determine whether the recommendations based on our clone detection techniques are useful for developers.

RQ 4.2: Do recommendations from CodeEase help reduce development time as compared to other traditional approaches?

A user study was conducted to answer this research question.

Our contributions are summarized as follows:

*https://github.com/QuickBlox/q-municate-android

†https://github.com/chrisballinger/BLEMeshChat

```java
public void notifyPeerAdded(Peer peer) {
    mPeers.add(peer);
    notifyItemInserted(mPeers.size() - 1);
}

public void notifyPeerRemoved(Peer peer) {
    int idx = mPeers.indexOf(peer);
    if (idx != -1) {
        mPeers.remove(idx);
        notifyItemRemoved(idx);
    }
}

public void clearPeers() {
    mPeers.clear();
    notifyDataSetChanged();
}
```

(a) PeerAdapter.java

```java
public void addItem(T item) {
    objectsList.add(item);
    notifyItemInserted(objectsList.size() - 1);
}

public void removeItem(T item) {
    int position = objectsList.indexOf(item);
    if (position != -1) {
        objectsList.remove(item);
        notifyItemRemoved(position);
    }
}

public void clear() {
    objectsList.clear();
    notifyDataSetChanged();
}
```

(b) BaseRecyclerViewAdapter.java

Figure 4.1: Example of a method clone structure across two files (a) and (b) belonging to two different systems

- We introduce a novel code recommendation technique in which we combine method completion based on type-2 clone detection and method recommendation based on MCS mined from a large repository of code.
- We have developed CodeEase, an Eclipse plugin, which allows a developer to efficiently complete his code using method recommendations that our tool provides.
- We conduct a user study to evaluate the effectiveness of our prototype tool and the usefulness of method clone structures.

4.2 Overview of the Approach

4.2.1 System Components

CodeEase consists of four components: a recommendation engine, two subsystems for MCS detection and clone detection respectively, and a code fact repository. The code fact repository is populated from a large source code repository using the clone detection technique introduced by Ishihara et al. [100] and MCS detection introduced in this chapter.

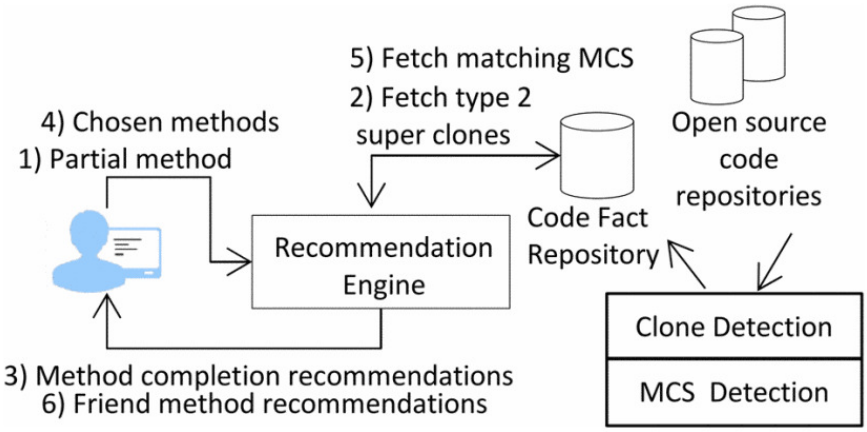


Figure 4.2: CodeEase system architecture: Recommending methods from Method Clone Structures

Clone Detection

In the current implementation, type-1 and type-2 method clones are found. A token size threshold determines the size of method clones to be detected. In the clone detection process, variables and literals are replaced with special tokens. To avoid accidental coincidence, the replacements are conducted with a parameterized matching technique. After the replacements, an MD5 hash value is calculated from the text of each method. Methods whose hash values are the same are detected as a group of clones. More details of the detection process can be found in literature [100].

MCS Detection

After the method clones are detected, the MCS detection algorithm executes by finding frequent item sets of method clones across code files. A frequent item set with a minimum support of 3 and a minimum depth of 3 is currently reported as an MCS. Figure 4.2 shows the system architecture diagram for CodeEase. The numbers in this figure show the sequence of activities.

4.2.2 Recommendation Process

We introduce a two-step method recommendation system based on method clones. The first step involves the developer writing some partial method code, which triggers a recommendation request (1). This results in searching the code repository for complete methods (type-2 clones) that partially match the body of the developer's method and also contain extra code (super clones) (2). These methods are provided as recommendations for method completion (3). If the developer chooses a method among these recommended methods to complete or supplement his code, this triggers the second recommendation step (4). The chosen method is looked up inside a repository of mined MCSs (5). If it is found to be a participant method of a particular MCS, the other methods of that MCS are also recommended to the developer (6). The two-step recommendation idea derives from the traditional market-basket analysis concept: if you buy a certain group of items, you are more likely to buy another group of items that are usually bought together with the first group of items. In other words, if you include a certain method in your code, then you are more likely to also include some other methods from a group of methods that occur together (friend methods). The user can also directly request for friend method recommendations against a method, which is either complete or partially coded. In this case, our system performs the same steps to retrieve friend methods. The only difference is that the method completion recommendations obtained in the first step are not sent to the user but only used to find the friend methods from MCS. At least one of the method completion recommendations should belong to some MCS to retrieve friend methods.

4.3 CodeEase Tool Features

CodeEase is designed as an Eclipse plugin for Java programmers. It currently supports two interfaces for interaction with users, which we call Interface1 and Interface2. Figure 4.3 shows CodeEase Interface 1 with method completion recommendations and friend methods of selected method. Figure 4.4 shows CodeEase Interface 2 with method completion recommendations and method body for selected method.

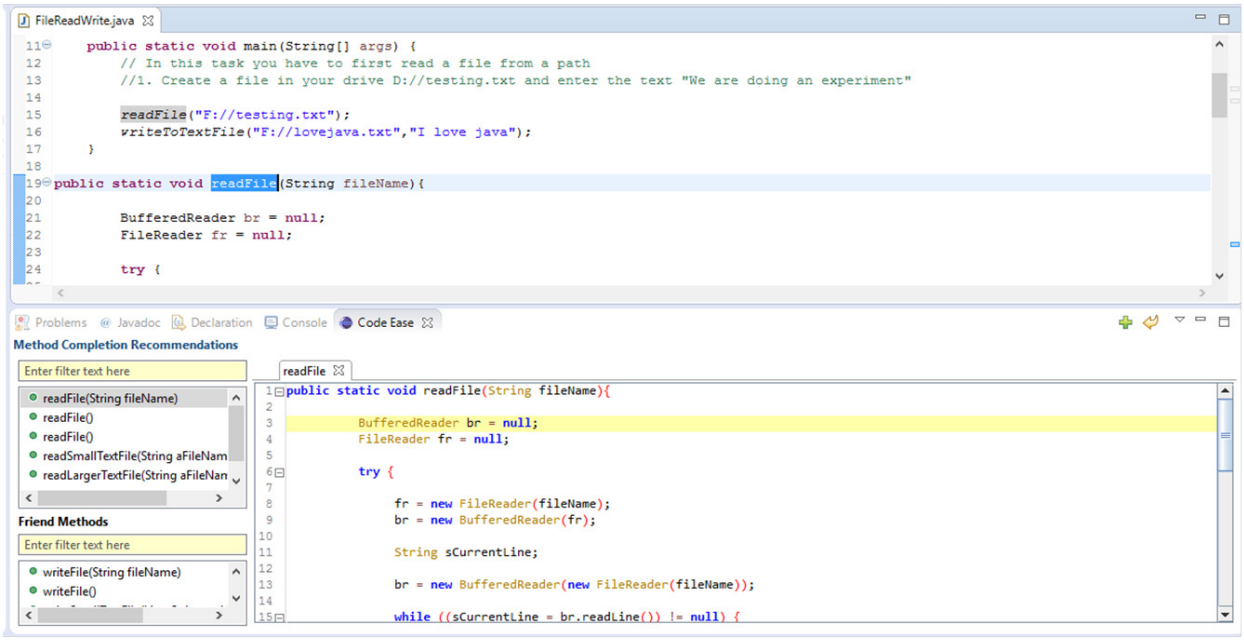


Figure 4.3: CodeEase Interface 1 with method completion recommendations and friend methods of selected method

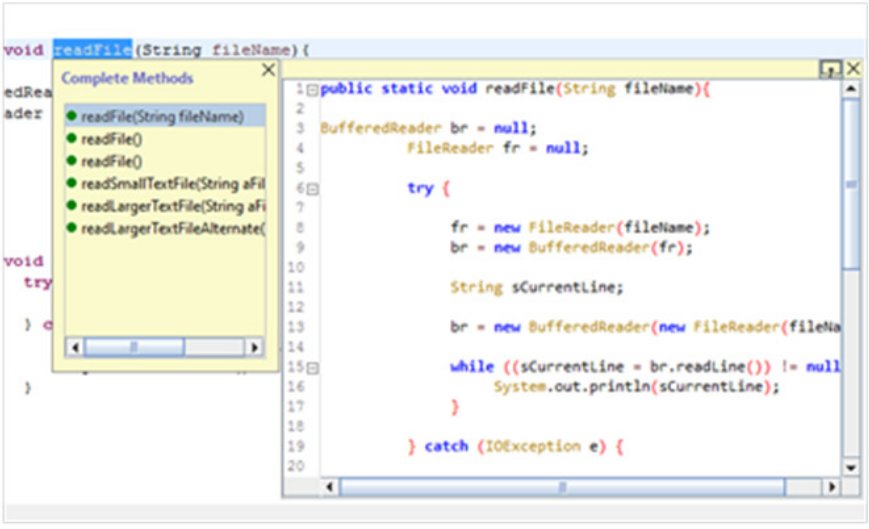


Figure 4.4: CodeEase Interface 2 with method completion recommendations and method body for selected method

Currently, the recommendations are simulated in this prototype and are based on a code repository tailored to meet the needs of the programming tasks discussed in Section 4.5.

4.3.1 CodeEase Interface 1

Triggering CodeEase

As a user writes or edits some lines of code in a method, he invokes CodeEase to provide completion recommendations for the partial method. The user selects the method name and right-clicks. This opens a popup menu showing the option of "Complete Method" which triggers CodeEase method completion recommendations. In case the user requires the friend methods of some method, he can choose the option of "Find Friends" from this popup menu.

Opening CodeEase

When the user clicks the option of "Complete Method" or "Find Friends" depending on the scenario, a new view for Code Ease will appear at the bottom of the screen. This is called the docked view of CodeEase. We have two list view panels on the left side of screen, each of which contains a list of method names that are being offered as recommendations. If only method completions are requested, then only the 'completions list view' is shown. If only friend methods are requested, then only the 'friends list view' is shown. Upon the selection of a method by the user, the 'friends list view' is also shown below the completions list view. The body of a selected method is shown to the right of the list view panel, which we call the 'method body panel'. The typical maximize, restore, and drop down arrow for optional settings buttons are shown on the top right view of CodeEase. We also included undo and refresh buttons to undo some method insertion and to refresh the list view containing method names respectively.

Showing method completion recommendations

The method completion recommendations are shown as method names (signature) in the 'completions list view' panel. A filter box is also included in the panel to enable the user to filter the

fetched methods using some keywords. Horizontal and vertical scrollbars are also available in the list view panel. These recommendations are obtained by finding clones of the partially written code statements of the user, and returning the method containing the statement clones.

Showing multiple method bodies

All selected methods appear in separate tabs in the method body panel so that the user can compare the different method bodies.

Showing friend method recommendations

Whenever a method is selected from the list view panel, another list view panel opens below the active list view panel showing friend method recommendations, if available. If the friend methods are invoked from the IDE, the 'friends list view panel' appears as the active panel in the CodeEase view. The 'friends list view panel' also has a filter box and navigation scroll bars. The friend methods of the selected method are obtained from those MCS in which the selected method participates.

Enabling code integration

For the integration of a chosen method into user written code, the user has the option to manually copy and paste the complete method body or parts of it, or use the "Insert" button. The insert button is shown at the top right corner of the method body panel. It automatically inserts the method body into the user written code at the closest cursor position.

4.3.2 CodeEase Interface 2

Triggering CodeEase

The user writes some code for a method and invokes CodeEase by selecting the method name and pressing Ctrl+1 keys combination. Two separate options of "Complete Method" and "Find Friends" appear in a quick assist popup.

Showing Recommendations

When the user clicks the option of “Complete Method” or “Find Friends”, a small popup window appears at the center besides the method name. The popup contains a list of method recommendations. If method completions are requested, then the methods that are type-2 super clones of the selected method are displayed. If friend methods are requested, then the method recommendations are retrieved from MCS having the selected method as a participant. Horizontal and vertical scrollbars allow easy navigation of the method names list. Upon clicking a method name, its body appears in a new popup window called the ‘method body popup’, horizontally aligned with the first popup. The method body can be enlarged with the help of handles at the corners. The method body popup refreshes itself upon each method selection.

Enabling code integration

The user can either manually copy and paste method code from the method body or use the “Insert” button on the method body popup to insert the method body into user written code.

4.4 CodeEase Evaluation

This section discusses the internal experiment we conducted to validate the correctness of our approach. First, we built a code repository from Java chat application systems downloaded from GitHub. Then we detected type-2 method clones on the projects contained in the repository. This was followed by using the MCS detection engine to mine MCS from the repository. We chose the MCS example in Figure 1 to determine how accurately our recommendation system performs in practice.

4.4.1 Validating Completion

The code for the method notifyPeerRemoved( ) was partially written in a test file and the CodeEase tool was invoked to provide completion. The completion recommendation for notifyPeerRemoved( ) was available and the method was completed.

Table 4.1: User Demographics

User Group	Industrial Experience	Total Users
Senior Developers	2 or more years	10
Junior Developers	Less than 2 years	8
Students	none	8

4.4.2 Validating Friend Method Recommendations

Upon selecting the notifyPeerRemoved( ) method, the notifyPeerAdded( ) and clearPeers( ) methods were recommended as friend methods. This validated the correctness of our recommendations based on MCS.

4.5 User Study

The user study was conducted to evaluate the usefulness of the tool in allowing a developer to quickly and efficiently complete his code. Participants for the user study were divided into three groups based on their experience. Experienced developers were recruited from a large software house in Lahore whereas junior developers were from various small sized software houses. Students were from the undergraduate senior batch in LUMS. Table 4.1 summarizes the user demographics.

The user study had 26 participants and was divided into two segments (Segment 1 and Segment 2), each involving 16 participants. 6 participants appeared in both the segments. The two study segments were conducted 10 weeks apart. In Segment 1, the users were assigned three programming tasks. In order to complete their programming tasks, they could use Google search, Snipt or StackOverflow Eclispe plugin . In Segment 2, the same three programming tasks were to be performed using the CodeEase tool only. We did not use a full fledge code repository for our user study, however, we had a smaller repository of code tailored to provide recommendations for the specified programming tasks.

4.5.1 Programming Tasks

We had three programming tasks. Task 1 involved writing methods to read from a text file, and write text to some text file. Task 2 involved writing code to extract the file name from a string containing the physical path of a file. Task 3 involved writing code to find the maximum from an array of double values, and also to find the minimum from an array of double values. In all, the users were supposed to write a total of five methods - namely readFile( ), writeFile( ), extractFileName( ), findMin( ), and findMax( ). In Segment 1, each task had to be performed using a different search tool. In Segment 2, the users had to use Interface1 and Interface2 at least once, and for the third task they were free to use any of the two. To facilitate the user, a precompiled skeleton project was provided with method definitions.

4.5.2 Measuring Performance

The programming session of every user was recorded using a screen capture tool. This allowed us to observe the task completion times, how the users used the tool, and any issues that occurred during development. We have enumerated the performance measures of the users on the programming tasks for the first segment of the user study in Table 4.2. In this segment, users performed tasks using search tools other than CodeEase. A task was either marked done, abandoned or failed. Tasks marked failed indicate the inability of the search tool to provide useful results. The same tool might fail to retrieve results for one user and succeed for another user depending on his search query. Tasks were abandoned because the programmer got frustrated after testing multiple code results or after attempts to fix a code. There was no fixed threshold for a task to be abandoned and it varied among different users. Table 4.3 contains observations from the second segment of the user study in which the programming tasks were performed using CodeEase. The skeleton project provided to the users in this segment also included partially written methods, so that they would not need to write any code before triggering CodeEase for completion or friend method recommendations. Tasks which were completed successfully were marked as done whereas tasks in which code errors were not fixed by the users were marked as partial. In Table 4.3 we can see that the first six users are the same from Table 4.2.

Table 4.2: Segment 1: User performance on programming tasks using other search tools

User#	Task 1			Task 2			Task 3			Total Time
	Tool	Time	Status	Tool	Time	Status	Tool	Time	Status	
1	P	6	D	G	3	D	S	4	F	13
2	S	20.5	D	P	3	D	G	7	D	30.5
3	S	3	F	P	5	A	G	3	A	11
4	P	5	F	S	1	F	G	6	A	12
5	G	10	D	S	2	F	P	4	D	16
6	P	4	D	S	1	F	G	4	D	9
7	G	6	D	P	3	D	S	1	F	10
8	S	2.5	F	G	3	D	P	6	D	11.5
9	P	6	A	S	3	F	G	5	A	14
10	S	5	F	G	2	D	P	4	D	11
11	S	4	F	G	7	A	P	5.5	A	16.5
12	G	5	D	S	2	F	P	6	D	13
13	G	6	A	P	8	D	S	5	F	19
14	S	7.5	F	P	10	A	G	7	A	24.5
15	S	1.6	A	G	3.6	D	P	5	D	10.2
16	S	1	A	G	2.6	D	P	3	D	6.6

(G=Google, P=Stack Overflow Plugin, S=Snipt, D= Done, F= Failed, A=Abandoned)

Table 4.3: Segment 2: User performance on programming tasks using CodeEase

U#	Task 1		Task 2		Task 3		Total Time
	Time (min)	Status*	Time (min)	Status	Time (min)	Status	
1	5	D	3	D	4.5	D	12.5
2	1.6	D	0.75	D	2	D	4
3	3.5	D	1	P	5	D	9.5
4	4	D	1	D	1	D	6
5	2	D	1	D	1	D	4
6	1	D	1	D	4	D	6
17	3	D	1	D	3	D	7
18	4	D	3.5	D	2	D	9.5
19	2	D	2	D	5	D	9
20	3	D	1	D	3	D	7
21	1	D	3.5	D	1	D	5.5
22	2	P	1.2	D	1.7	D	5
23	0.8	D	0.7	D	1.4	D	4
24	1	D	1	D	4	D	6
25	1	D	0.8	D	1	D	2.8
26	4.5	D	4	D	4	D	12.5

(D= Done, P= Partial)

Table 4.4: A comparison of task completion by users using CodeEase and other tools

Number of Tasks Completed	Number of Users using Other Search Tools	Number of Users using CodeEase
3	1	14
2	9	2
1	1	-
0	5	-

4.5.3 Empirical Analysis of Results

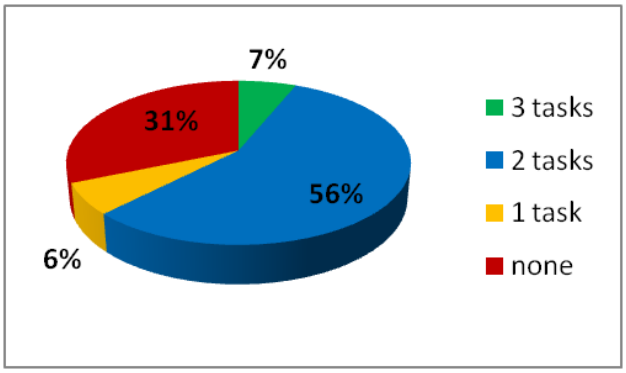
This section discusses findings from the user study. This includes both quantitative and qualitative analysis.

User Performance on Programming Tasks Table 4.4 lists the number of users who were able to complete a certain number of tasks using other search tools. We can see that 5 out of 16 users were unable to complete even a single task. Only one user was able to complete all three tasks using other search tools, whereas CodeEase allowed a majority of users to complete all three tasks.

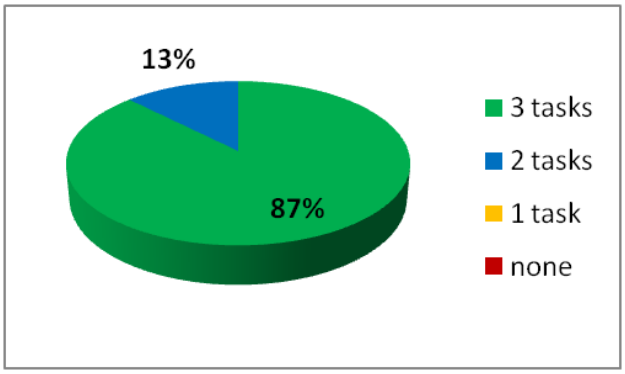
Figure 4.5a shows that using other approaches, only 7% users complete all three tasks, 5% users complete only two task, 6% users complete one tasks only and 31% users are unable to complete any task. On the other hand, Figure 4.5b shows that using CodeEase, 78% of the users were able to complete all three tasks and the remaining 13% users were able to complete two tasks. Figure 4.6 shows how much time it took six users to perform the programming tasks using CodeEase as compared to using other approaches. For all users, the time required by CodeEase is lower than that required by other tools for the same task.

Coding Activity Experience Using Other Search Tools 60% of the developers thought the tasks were moderately difficult, whereas 40% thought they were quite easy. 60% users thought Google provided the best experience, whereas, for 30% Stack Overflow Plugin was the best. 60% voted that Snipt was the worst tool. It was observed that a user having greater experience does not necessarily perform tasks quicker than users with lower experience.

Coding Activity Experience Using Code Ease 100% users claimed that they would prefer the



(a) Task completion using other approaches



(b) Task completion for CodeEase approach

Figure 4.5: Percentage of users completing tasks using existing code search methods versus CodeEase approach

CodeEase tool over other search tools. 70% of the users were of the view that CodeEase reduced their development time.

Qualitative Analysis of User Feedback Through the general user comments, it was revealed that users found it easy and satisfying to browse for code within the IDE. Some users preferred Interface1 whereas others preferred Interface2. Based on this feedback, we are considering keeping both interfaces functional to satisfy all users.

4.6 Summary of Results

This section summarizes answers to the research questions posed earlier.

RQ 4.1: Are the clone-based recommendations provided by CodeEase useful for the developer?

From the results of the user study, we observed that recommendations based on type 2 clones are useful for method completion and subsequent recommendation of friend methods based on MCS. This is demonstrated by the higher task completion rate as shown in Table 4.4 and Figure 4.5. Furthermore, qualitative data showed that users found it useful to get friend method recommendations without writing an explicit query.

RQ 4.2: Do the recommendations from CodeEase tool help reduce development time as compared to other traditional approaches?

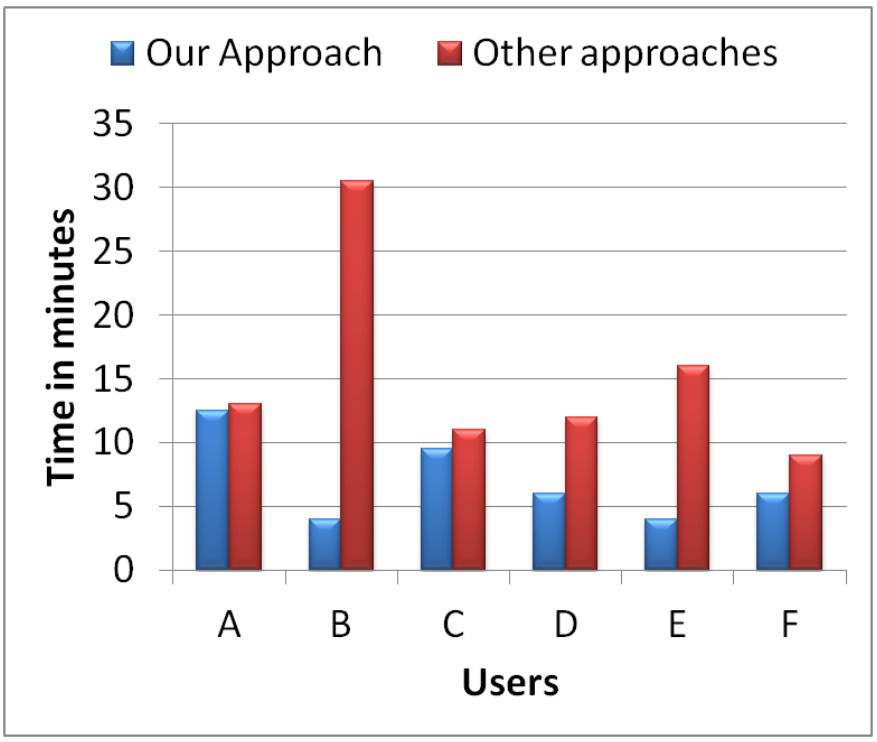


Figure 4.6: Comparing user performance on programming tasks

From the analysis of experimental data, we observed that developers were able to complete their tasks using the CodeEase tool in a shorter time span. In fact, with traditional search techniques, the developers abandoned their tasks and were not able to complete their programming tasks. However, with CodeEase, the developers successfully executed their code without errors. 70% of the users were of the view that CodeEase reduced their development time.

4.7 Threats to Validity

There are certain internal threats to the validity of the user study. Some users were very meticulous about the removal of bugs and execution of the program to test their output, however, some users did not remove errors like missing libraries. In Segment 1, we have considered those tasks complete in which there were missing library errors, the removal of which would result in a successful execution. However, such user behavior has resulted in differences in completion times. In Seg-

ment 2, such tasks are considered as partially completed. In Segment 1, it was observed that some users performed a task but did not execute the code to check for correctness. A manual inspection of the code revealed incorrect implementations which would not provide the desired result. Such tasks were considered as abandoned. It was noticed that the time it took for users to abandon a task varied for the same tool. The varying attention span and diligence of individual users has affected the correctness of the time and status measures from the user study. The reason for the provision of partially written methods in Segment 2 was to enable users to trigger CodeEase directly without going to some search engine to start writing code. Without this provision, the time required would have been longer and would have varied for each user. By eliminating the time required to write partial code, we measure the time spent on the usage of the tool to complete the programming tasks. The repository from which CodeEase provided recommendations was tailored to specifically include recommendations that would allow the completion of the programming tasks. In other programming scenarios, getting a recommendation will depend on the richness of the repository. For this reason, we propose using code from large scale open source code repositories.

4.8 Chapter Summary

In this chapter, we introduced CodeEase, a tool for recommending methods to complete a developer's code. We have shown that Method Clone Structures are useful for generating code recommendations. Being coarser-grained than single methods; they can provide bigger reuse opportunity, and eliminate the need to execute subsequent search queries to fetch related methods. Our approach targets to minimize the time required in searching for such related methods, and the time required in stitching together independent methods. Using CodeEase, we enable the user to perform the same tasks with reduced time and higher success rate.

In further internal experiments on Type-2 clone detection and MCS mining, we found that since Type-2 clones are rare across projects, Type-2 MCS are even more rare. This posed a problem for generating recommendations for related code. Since Type-2 clones fail to capture semantically similar code, we decided to move to Type-4 semantic clones. Our intuition was that by using a more flexible and less restrictive semantic clone detection, we would be able to get more patterns

of co-occurring functionality. This was a major learning point and helped us move in the right direction.

Chapter 5

FACER: Feature Driven API usage-based Code Examples Recommendation

5.1 Introduction

When developers are implementing a given system, they usually have a set of *features* (i.e., units of functionality) they need to implement. For example, a Bluetooth chat application can have features like setting up Bluetooth, scanning for other Bluetooth devices, connecting to a remote device, and transferring data over Bluetooth. A feature may be implemented in a single method or across a group of methods that call each other.

To speed up development, developers often resort to code search to find code that they can reuse for certain features in their application [67, 142]. However, most of the existing code search systems focus on providing code corresponding to a single query related to the current feature the developer needs to implement [5–7, 40, 72–74, 85]. As a result, developers may have to conduct a new search for every next feature they need to implement and later integrate the obtained code. Existing code search and recommendation systems do not support the need to find code for additional features related to a developer's query. On the other hand, existing feature recommendation systems enable the exploration of text-based related features and either support domain analysis for the requirements gathering phase [8, 116–119] or enable rapid prototyping by recommending

related code modules [9]. However, they do not provide code examples at a fine-grained method-level for reuse. Hence, we identify two gaps in existing systems: one is the lack of support for providing related code recommendations in existing code search systems and the other is the inability of existing feature recommendation systems to provide code for features at the method-level granularity.

Based on this perspective, in this dissertation, we fill in the above gaps by proposing a recommendation system that provides developers with related method recommendations that have functionality relevant to their application under development. Studies of rapid prototype development have shown that programmers iteratively add features by reusing source code examples [3, 56]. This iterative process is known as *opportunistic programming* [3]. To this end, we propose a system that provides code recommendations for these related features to support *opportunistic code reuse* [52]; such support enables rapid application development without the need to conduct multiple searches and thus enhances developer productivity and saves time [3, 52, 57, 58].

Our proposed recommendation system is called FACER, **F**eature-driven **A**PI usage-based **C**ode **E**xamples **R**ecommender, and works at the granularity of methods, where the recommended code snippet is a full method itself. We use a combination of static program analysis, information retrieval, and data mining techniques to build FACER. More precisely, we add another layer on top of traditional code search techniques in order to additionally propose code snippets corresponding to features related to the original search query.

FACER generates related method recommendations in two stages.The first stage corresponds to traditional code search where any existing code search technique [5, 7, 72–74] can be used. We use Lucene [93] to implement the code search engine behind FACER. Given a developer's feature query in the form of natural language description, the search stage of FACER recommends a set of methods that implement the desired feature. Upon selection of one of these recommended methods by the developer, the second stage of FACER starts. In this second stage, which is the main contribution of this dissertation, FACER provides subsequent recommendation of related methods for reuse. FACER recommends these related methods based on patterns of frequently co-occurring features which we identify as frequently co-occurring method clones. Since methods

with similar uses of Application Programming Interfaces (APIs) are semantically related [7], we identify Method Clone Groups (MCG) based on API usages. Thus, a *method clone group* contains code examples for a common feature. To find semantically related features, we then identify frequently co-occurring Method Clone Groups (which we refer to as *Method Clone Structures*), leveraging the idea of market basket analysis [11]. *Market basket analysis* attempts to identify associations, or patterns, between the various items that have been chosen by a particular shopper and placed in their basket [143]. Items that frequently co-occur are related to each other. In our context, one or more methods of a particular project may be cloned across other projects. Such a pattern of co-occurring method clones identifies related functionality and forms the basis of suggesting relevant related methods.

While the concepts behind FACER are not tied to a particular programming language or type of application, we focus on Java Android apps for building and evaluating the first version of FACER. According to the latest Stack Overflow developer survey 2020, 57.1% of 65,000 developers surveyed are developing Android apps [144]. A recent exploratory study focusing on code reuse from StackOverflow in the context of mobile apps found that feature additions and enhancements in apps are the main reasons for code reuse from StackOverflow [145]. Furthermore, findings from a large-scale empirical study on software reuse in mobile apps indicate a high percentage of code reuse across applications [146]. Since Android development involves rapid release cycles [147], there is a need to facilitate opportunistic reuse to enable rapid application development. An empirical study involving the manual analysis of 5,000 commit messages from 8,280 Android apps found that application enhancement is the most frequent self-reported activity of Android developers [148]. In the same study, self-reported activities of Android developers have been categorized and each development category is seen to be composed of a related set of activities. For example, activities related to using the device camera include taking a picture when using the app, when and how to show the preview of a taken picture, usage of the flash light, switching between front and rear camera. The challenge is whether code for these related activities can be made available to a developer without the need for the user to explicitly perform a search for each desired activity, which is what we address in this dissertation.

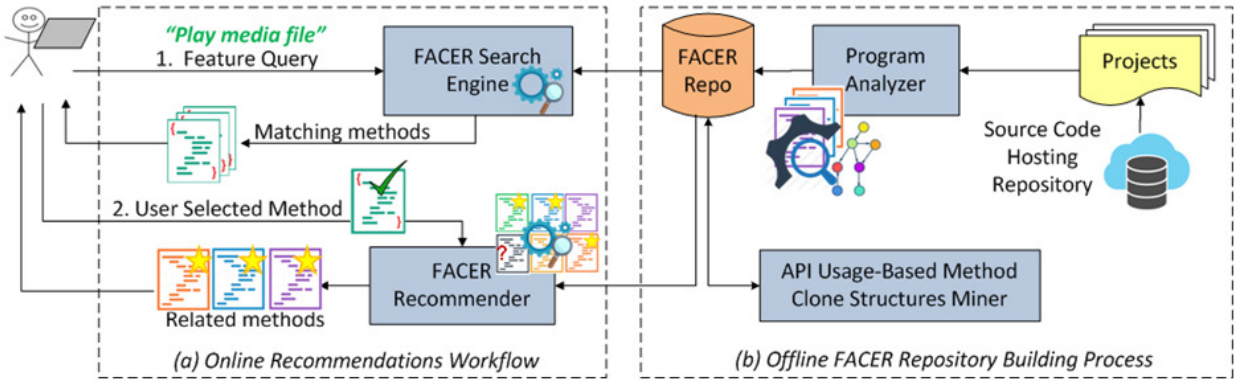


Figure 5.1: FACER system components and workflow

5.2 Overview of Proposed Approach: FACER

We propose a system for opportunistic reuse of related code which allows developers to receive code examples for functionality they may like to implement next. Allowing developers to receive code for related features can enhance productivity and save search time [57]. In this section, we discuss the components of our proposed Feature-driven API usage-based Code Examples Recommendation (FACER) system. Figure 5.1 provides an overview of the various components in FACER and its workflow.

FACER has two main workflows: (1) the *offline FACER repository building workflow* which builds facts through mining information from source code repositories and (2) the *online recommendation workflow* which uses this information to make recommendations. From a user perspective, the user provides a feature query as a natural language description; in other words, this is the task or feature they want to implement. The FACER search engine then returns a list of matching methods. After the user selects a method from that list, the FACER recommender then returns a list of related methods that correspond to additional features the developer may want to implement. We now discuss these two workflows in more detail.

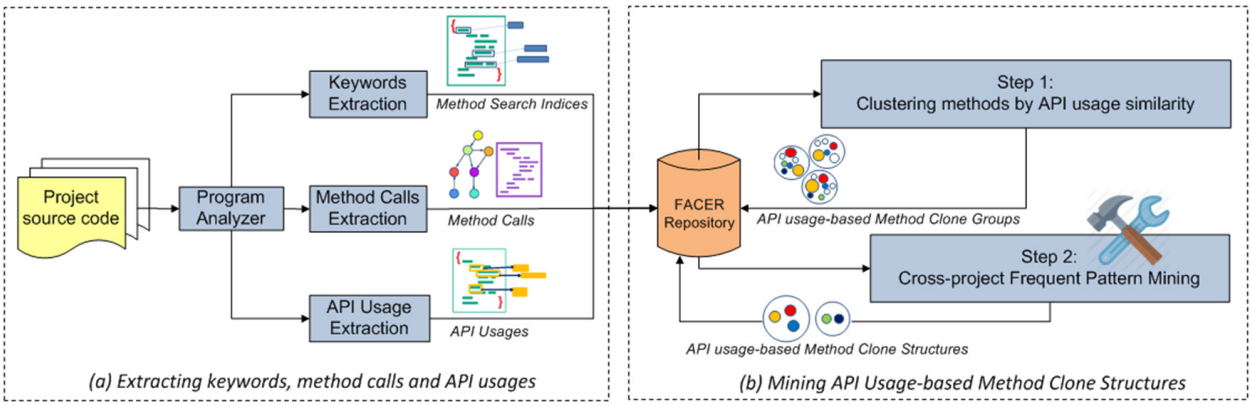


Figure 5.2: Offline FACER repository building components

5.3 Offline FACER Repository Building Workflow

In order to provide its recommendations, FACER first has an "offline" phase where it populates its repository (a MySQL database) with source code information from open-source Java applications hosted on GitHub [149]. We discuss the details of the data we select to populate this repository for our evaluation in Section 8.1.2. Figure 5.2a shows the three types of information we extract from each application's methods using the Eclipse JDT parser [150]: keywords, method calls, and API usages.

5.3.1 Extracting Keywords for Search Index

To implement a simple retrieval scheme [7] for code search, FACER's program analyzer builds a search index. Any code search technique can be used to retrieve code. For the purposes of this work, we implement a simple Lucene-based search index. Lucene is a high-performance, full-featured text search engine library suitable for full-text search over documents [93]. We use Lucene to build a search index over methods and store them as a collection of documents. A document is a set of fields. Each field has a name and a textual value. A field may be stored with the document, in which case it is returned with search hits on the document. Thus each document should typically contain one or more stored fields that uniquely identify it [151]. The program analyzer extracts all the terms from the simple name, Fully Qualified Name (FQN), and full text of

a method, and tokenizes each set of terms using camel-case and special characters. It then creates a separate Lucene field to store the extracted terms from the method name, Fully Qualified Name (FQN), and the full text of the method body respectively. To give more significance to matches with the method name during code search, it assigns the *method name* field a higher boost value [152] than the other fields. Finally, it creates a Lucene document against every method to build the search index.

5.3.2 Extracting Method Calls, API Calls, and API Call Density

An *API usage* is a set of API calls found in a method. The underlying premise of FACER is that these API calls together represent the implementation of a feature. FACER detects repeatedly co-occurring features on the basis of repeatedly co-occurring API usages. Thus, in its repository, FACER needs information about API usages. A software application interacts with external libraries or system libraries/packages through various API classes to implement desired features. For example, building the connection to a Bluetooth device requires the use of the Bluetooth API package and different methods of the API to setup the connection. When analyzing a source code project, we parse the class declarations in Java files and save them as user-defined classes. The Eclipse JDT [150] parser is able to trace the objects to their respective types. While parsing method invocations, if a type identified by the parser does not match any user-defined class, then we consider this type as an API class. Thus, we refer to any method call from an API class as an *API call*. For example, `BluetoothAdapter.getDefaultAdapter()` is an API call of class `BluetoothAdapter` from the `android.bluetooth` API package.

Using the Abstract Syntax Tree (AST) [153] provided by Eclipse JDT [150], the Program Analyzer module visits all method declarations and parses their content to identify calls. For each detected call, we record the call site, which is the location of the call in the method body and is identified by a line number. In the FACER repository, we differentiate between *user-defined method calls* (or *method calls* for short), which are invocations of methods that have been defined in the current project, and *API calls* which are invocations of API methods. To differentiate the types of calls, we check the receiver type of the call. API types may be classes from the Java

```
protected void onCreate(Bundle savedInstanceState){
super.onCreate(savedInstanceState);
setContentView(R.layout.activity_video);

//for play video by net

//id
VideoView videoView = findViewById(R.id.videoView);
//set url
videoView.setVideoURI(Uri.parse(LINK));
// Open lib MediaController for stop and play and set time play
MediaController controller = new MediaController(this);

//set controllerto video view
controller.setAnchorView(videoView);
//set videoView to controller
videoView.setMediaController(controller);

//start this put
videoView.start();
}
```

Listing 5.1: Example of extracted API calls (underlined) from a given method

Table 5.1: Assigning API Call IDs to methods. Example based on code shown in Listing 5.1.

Method ID	API name	API method	API Call ID
6	VideoView	setVideoURI	31
6	Uri	parse	11
6	MediaController	new	32
6	MediaController	setAnchorView	12
6	VideoView	setMediaController	13
6	VideoView	start	33

Class libraries (JCL) [154] in JDK [155] or Android classes [156] in Android SDK [157] or any other third-party library imported by the user. FACER stores API calls for every new API instance

created (i.e., constructor calls) and for every API method called, including static calls. Listing 5.1 shows an example method where the API calls that FACER extracts are underlined.

FACER stores API calls that it mines from all the methods in its repository. We encode the API calls occurring across the entire FACER repository with unique identifiers which we call *API Call IDs*. Table 5.1 shows an example of the information we store.

At this point, the program analyzer also calculates an API call density for each method it analyzes. We define *API call density* as the fraction of statements containing API calls over the total number of statements in the method as shown in Equation 5.1 as follows:

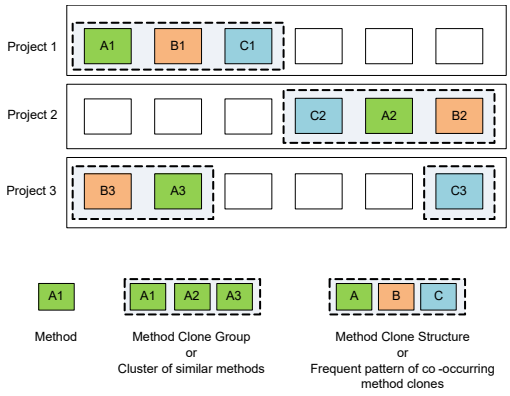
$$APICallDensity(M) = \frac{|StatementsContainingAPIcalls(M)|}{|Statements(M)|} \tag{5.1}$$

The API call density value of a method indicates the concentration of statements containing API calls with respect to other statements in a method. For example, in Listing 5.1, the method contains 8 statements out of which 5 contain API calls. This results in an API call density score of 0.6. If there are no API calls in a method then its API call density score will be 0 and if each statement in the method body contains one or more API calls, then its API call density will be 1.

5.3.3 Mining API Usage-based Method Clone Structures

To find related methods that implement related features, FACER's high-level idea is to find similar methods based on their API usages and cluster them together, where a cluster represents a particular feature. Then, we can find commonly co-occurring method clusters, where commonly co-occurring method clusters represent related features since they frequently appear together.

We use the term *Method Clone Group* to refer to such a method cluster. Traditionally, a clone group is a set of code snippets in which token sequence similarity exists between any pair of code snippets [75]. Given our purposes, we specifically look at similar API usages to identify members of a clone group. Thus, we define a *Method Clone Group* (or *clone group* for short) as a set of methods in which API usage similarity exists between any pair of methods. These methods may implement the same feature or functionality and could be instances of a particular feature. For high-level illustration, Figure 5.3a shows methods found across three projects where methods of



(a) Abstract Method Clone Structure across projects

```
public void startMeasureResult() {
   btAdapter = BluetoothAdapter.getDefaultAdapter();
   BluetoothDevice device =
      btAdapter.getRemoteDevice(address);
   try {
      btSocket = createBluetoothSocket(device);
   } catch (IOException e) {
      errorExit("Fatal Error",
         "In onResume() and socket create failed: " +
         e.getMessage() +
         ".");}
   btAdapter.cancelDiscovery();
   Log.d("bluetooth", "...Connecting...");
   try {
      btSocket.connect();
      Log.d("bluetooth", "....Connection ok...");
   } catch (IOException e) {
      try {
         btSocket.close();
      } catch (IOException e2) {
         errorExit("Fatal Error",
            "In onResume() and unable to close " +
            "socket ..." +
            e2.getMessage() +
            ".");}}
   Log.d("bluetooth", "...Create Socket...");
   mConnectedThread = new ConnectedThread(btSocket);
   mConnectedThread.start();
}
```

(b) Method A1 from Project 1

```
public void run() {
   Log.i(TAG, "BEGIN mConnectThread SocketType:"
      + mSocketType);
   setName("ConnectThread" + mSocketType);
   // Always cancel discovery because
   //it will slow down a connection
   mAdapter.cancelDiscovery();
   // Make a connection to the BluetoothSocket
   try {
      // This is a blocking call and will
      // only return on a successful
      // connection or an exception
      mmSocket.connect();
   } catch (IOException e) {
      // Close the socket
      try {
         mmSocket.close();
      } catch (IOException e2) {
         Log.e(TAG, "unable to close() " +
            mSocketType +
         " socket during connection failure", e2);
      }
   connectionFailed();
   return;
   }
   // Reset the ConnectThread because we're done
   synchronized (BluetoothChatService.this) {
      mConnectThread = null;
   }
   connected(mmSocket, mmDevice, mSocketType);
}
```

(c) Method A2 from Project 2

```
public ConnectedThread(BluetoothSocket socket) {
   InputStream tmpIn = null;
   OutputStream tmpOut = null;
   // Get the BluetoothSocket input and output streams
   try {
      tmpIn = socket.getInputStream();
      tmpOut = socket.getOutputStream();
   } catch (IOException e) { }
   mmInStream = tmpIn;
   mmOutStream = tmpOut;
}
```

(d) Method B1 from Project 1

```
public ConnectedThread(BluetoothSocket socket,
      String socketType) {
   Log.d(TAG, "create ConnectedThread: " +
      socketType);
   mmSocket = socket;
   InputStream tmpIn = null;
   OutputStream tmpOut = null;
   // Get the BluetoothSocket input and
   //output streams
   try {
      tmpIn = socket.getInputStream();
      tmpOut = socket.getOutputStream();
   } catch (IOException e) {
      Log.e(TAG, "temp sockets not created", e);
   }
   mmInStream = tmpIn;
   mmOutStream = tmpOut;
   mState = STATE_CONNECTED;
}
```

(e) Method B2 from Project 2

Figure 5.3: A real example of a API Usage-based Method Clone Structure taken from Bluetooth chat projects. Highlighting shows common API usages

the same color have similar API usages and thus are functionally similar. Thus, methods A_1, A_2, and A_3 belong to the same clone group A.

Since our goal is to find related functionality, we want to find clone groups that frequently occur together. For example, if methods that implement a *connect to Bluetooth* functionality often occur with methods that implement a *send file over Bluetooth* functionality, then we know that these two functionalities are related. Accordingly, we use the term *Method Clone Structure (MCS)* [158] to refer to a set of methods that are frequently cloned together across different projects. In other words, a recurring pattern of method clones is a Method Clone Structure. The participating methods in a method clone structure all relate to each other. Since our method clones are based on API usages, we call these structures *API usage-based Method Clone Structure*. In Figure 5.3a, clone groups A, B and C together form a frequent pattern across the three analyzed projects. Hence, they form a method clone structure. Based on the heuristic of frequent co-occurrence, members of the clone structure are all related to each other. Figures 5.3b-5.3e show some of the corresponding methods taken from a real clone structure, which we mine from projects implementing Bluetooth chat functionality. The green highlighting represents clone group A and the pink highlighting represents the clone group B. The highlighted API usages are the basis of similarity between members of a clone group.

Figure 5.2b shows the two steps we take to mine clone structures. We now explain these steps in detail.

Step 1: Cluster methods by API usage similarity In this step, we group all the methods in the repository into clusters on the basis of similar API usages and API call densities between them. We use Figure 5.4 to explain this process. As explained in Section 5.3.2, we already record unique API call IDs for all API usages that we analyze across all projects. Assume that our repository consists of the nine methods listed in the table in Figure 5.4a. The sequence of numbers shown in the second column represents the IDs of the API calls that appear in each method. For the sake of simplicity, we only demonstrate the effect of clustering on the basis of API calls similarity without considering the effect of API call density. So, we assume that all methods have an API call density equal to 1.

Distance matrix computation We define the similarity of two methods based on the intuition that if two methods share a high percentage of API calls (represented by their corresponding set of API call IDs), then they perform the same functionality and implement the same feature. We also factor in the API call density similarity to favor the methods with similar and high API call density to be clustered together. Otherwise, a method that contains a single statement with API call `foo()` might be clustered together with a method that contains 20 statements, only one of which contains the same API call `foo()`.

Let $M_1 = \{u_1, u_2, ..., u_n\}$ and $M_2 = \{u_1, u_2, ..., u_m\}$ be the sets of API call IDs of two methods M_1 and M_2. We compute the API usage similarity of two methods M_1 and M_2 using the Jaccard index [159] as follows:

$$api\_sim(M_1, M_2) = \frac{|M_1 \cap M_2|}{|M_1 \cup M_2|} \tag{5.2}$$

The similarity score has a value between 0 and 1, where 0 means completely dissimilar and 1 means completely similar. Let d_1 and d_2 be the API call densities (calculated using Equation 5.1) of methods M_1 and M_2 respectively. We define the API call density similarity of the two methods as follows:

$$density\_sim(M_1, M_2) = \frac{d_1 + d_2 + (1 - |d_1 - d_2|)}{3} \tag{5.3}$$

In Equation 5.3, $1-|d_1-d_2|$ indicates the similarity of density values between the two methods. We factor in individual density values of methods together with density similarity, because we want to give a higher score to two methods with similar high density than to two methods with similar low-density values. The final similarity score is calculated as follows:

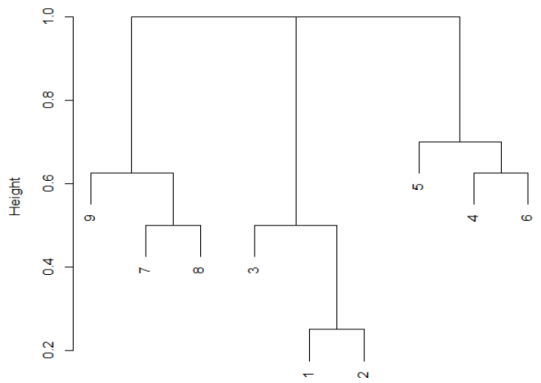
$$Sim(M_1, M_2) = api\_sim(M_1, M_2) \times density\_sim(M_1, M_2) \tag{5.4}$$

In preparation for clustering, we calculate the distance between the two methods as follows and store all pairwise method distances in a distance matrix:

$$Dist(M_1, M_2) = 1 - Sim(M_1, M_2) \tag{5.5}$$

Method ID	API Call IDs
1	1 2 3 4
2	1 2 3
3	7 8 1 2 3
4	11 12 13 24 25
5	26 27 11 28 12 29 13
6	31 11 32 12 13 33
7	8 35 9 10
8	8 9 10 15 16
9	41 42 8 43 9 10

(a) Example methods and API Call IDs



(b) Dendrogram obtained by clustering methods 1 - 9

Method ID	Clone Group ID
1	1
2	1
3	1
4	2
5	2
6	2
7	3
8	3
9	3

(c) Resulting clone group for each method

Figure 5.4: Step 1: Cluster methods by API usage similarity. After this step, each method in our repository has a clone group ID.

Cluster Identification To identify method clusters, we pass the calculated distance matrix as input to the standard average linkage hierarchical clustering algorithm [160]. This algorithm performs a hierarchical cluster analysis using a set of dissimilarities for the *n* methods being clustered. Initially, the algorithm assigns each method to its own cluster and then the algorithm proceeds iteratively, at each stage joining the two most similar clusters, continuing until there is just a single cluster. The result is a tree-based representation of the methods being clustered which is called a dendrogram [161]. Figure 5.4b shows the dendrogram obtained after clustering the nine methods from Figure 5.4a. Each leaf of the dendrogram corresponds to one method. As we move up the tree, methods that are similar to each other get linked into branches, which are themselves fused at a higher height. The height of the fusion, provided on the vertical axis, indicates the dis-similarity between two methods. In order to identify sub-groups (i.e. clusters), we can cut the dendrogram at a certain height. A cut is a demarcation line at a certain height of a dendrogram which results in the intersection of dendrogram branches with the cut. All nodes of the branch that intersects the cut end up in one cluster. The branches and nodes above the cut form independent clusters. Choosing the optimal cut-point on a dendrogram is an NP-complete problem. In our case, we experiment with a number of cut-point values at different heights corresponding to a similarity threshold α. The height of the cut to the dendrogram controls the similarity threshold, and thus the number of clusters obtained. The greater the height of the cut, the looser the similarity threshold and the fewer the number of clusters formed. The smaller the height of the cut, the stricter the similarity threshold and the greater the number of clusters formed. If we specify a height of 0.7, this means that the final clusters at that height would have at least $1 - 0.7 = 0.3$ similarity score between their members. Methods joined at height 0 are exactly similar. We obtain a vector containing the clone group ID (i.e., cluster ID) of each method after cutting at a certain height and store this information in our repository. In our evaluation in Chapter 6 Section 6.4, we evaluate the effect of varying similarity thresholds by obtaining clusters against various values of height =$\{0.1, 0.3, 0.5, 0.7\}$, corresponding to similarity thresholds $\alpha = \{0.9, 0.7, 0.5, 0.3\}$ respectively.

The results of clustering the nine methods from our example are shown in the table of Figure 5.4c. The clone group IDs are obtained using a similarity threshold of 0.3, which implies a

Project ID	Clone Group IDs
1	1 2 3 11 19
2	1 9 2 4 3
3	5 6 15 18 19
4	21 5 22 6
5	26 1 2 3

(a) Example clone group IDs recorded for each project

Clone Structure ID	Clone Group IDs	Support
C1	5 6	2
C2	1 2 3	3

(b) Resulting Method Clone Structures across projects

Figure 5.5: Step2: Mining frequent patterns of method clones across projects

height of 0.7. We can see that the first three methods are assigned to the first cluster, methods 4, 5 and 6 are assigned to the second cluster, and the last three methods are assigned to the third cluster. The output of Step 1 is now a mapping of method IDs against the unique clone group IDs of each cluster.

Step 2: Mining frequent patterns of method clones across projects The idea of frequent association pattern mining is to find recurring sets of items among transactions. The concept of transactions originates from sales transactions where one or more items are purchased in a single sales transaction. In our context, items are clone groups of a project that make up a transaction in the FACER repository R. We are interested in mining recurrent patterns of clone groups. The strength of a frequent pattern is measured by a support count. Support count is the number of transactions in R containing a unique pattern of clone groups. A frequent association pattern describes a set of items that has support greater than a predetermined threshold called a minimum support threshold which we identify as β.

We have so far identified the clone group IDs of all methods in our repository. To mine API usage-based method clone structures across projects, we first create a transaction table where each row of the table contains all the clone group IDs assigned to methods of a project. Assume that we

have a repository of five projects with the transaction table shown in Figure 5.5a.

Given this table, we perform frequent item set mining to get frequently co-occurring clone groups that repeat across projects. Such repeating item sets represent the API usage-based Method Clone Structures. We can say that for a given clone structure and its constituent clone groups, the methods mapped to those clone groups are all related to each other.This is based on the market basket intuition [11]. Market basket analysis attempts to identify associations, or patterns, between the various items that have been chosen by a particular shopper and placed in their basket [143]. Items that frequently co-occur are related to each other. In our context, a particular project may use a group of methods which may be cloned across other projects. Such a pattern of co-occurring method clones identifies related functionality and forms the basis of suggesting relevant methods.

We use the frequent closed itemsets mining algorithm *FPClose* [162, 163] to get frequent items. *FPClose* is an algorithm of the FPGrowth family of algorithms, designed for mining frequent closed itemsets and is claimed to be one of the fastest closed itemset mining algorithm. The input to the algorithm is a transaction table and a support/frequency threshold β. We evaluate the sensitivity of recommendation results against varying thresholds of β=(3, 5, 10, 15) in Chapter 6 Section 6.4. The result of the execution of FP mining on our example transaction table with β = 2 is shown in the right table in Figure 5.5b. The clone structure C1 indicates that clone groups 5 and 6 are frequently found together. Similarly, C2 indicates that the clone groups 1, 2 and 3 are frequently found together. This provides the basis of FACER's recommendation which we explain next in Section 5.4.

To summarize, Algorithm 1 shows all the steps discussed above which are involved in mining Method Clone Structures (MCS) in the FACER repository R given a similarity threshold α and a minimum support threshold β. First, we obtain API calls and API call densities of all methods in the FACER repository R (Lines 4–7). Then, we obtain pairwise similarities for all methods (Lines 8–10). The distance matrix is obtained from the similarity matrix and used by the clustering algorithm to detect and label clusters with respect to α (Lines 11–13). The resulting clusters are saved as clone groups in FACER (Line 14). Next, in order to mine frequently co-occurring features across all projects, we create a transcation table where each row contains clone group IDs for a

Algorithm 1 Mining API usage-based Method Clone Structures

```
1:  procedure MINEMCS(R, α, β)
2:      APICallsMap ← {}
3:      APICallDensityMap ← {}
4:      for all m ∈ methods(R) do
5:          APICallsMap.add(m.ID, m.APICalls)
6:          APICallDensityMap.add(m.ID, m.APICallDensity)
7:      end for
8:      simMatrix1 ← getPairwiseJaccardSim(APICallsMap)
9:      simMatrix2 ← getPairwiseDensityBasedSim(APICallDensityMap)
10:     simMatrix3 ← simMatrix1 × simMatrix2
11:     distMatrix ← 1 − simMatrix3
12:     dendrogram ← cluster(distMatrix)
13:     methodClusterIDMap ← getClusters(dendrogram, α)
14:     saveCloneGroups(methodClusterIDMap, R)
15:     transactionTable ← φ
16:     for p ∈ projects(R) do
17:         transaction ← getCloneGroupIDs(p)
18:         transactionTable.add(transaction)
19:     end for
20:     CloneStructures ← getFreqPatterns(transactionTable, β)
21:     save(CloneStructures, R)
22: end procedure
```

project in the FACER repository R (Lines 16-19). The resulting table is used to perform frequent item set mining with respect to a certain threshold β to obtain frequently co-occurring sets of clone groups which are then saved as Method Clone Structures in the repository R (Lines 20–21).

5.4 Online FACER Recommendation Workflow

We implement FACER as an Eclipse IDE plugin and also as an Android Studio IDE plugin. In this section we discuss the user interface of the Eclipse IDE plugin in detail, whereas the Android Studio IDE plugin interface and details are mentioned in Appendix A.

The "online" workflow is what developers experience when they interact with FACER. This interaction process is comprised of two stages. Stage 1 performs retrieval against a user's feature query to provide a ranked list of top methods that implement the requested feature. Upon selec-

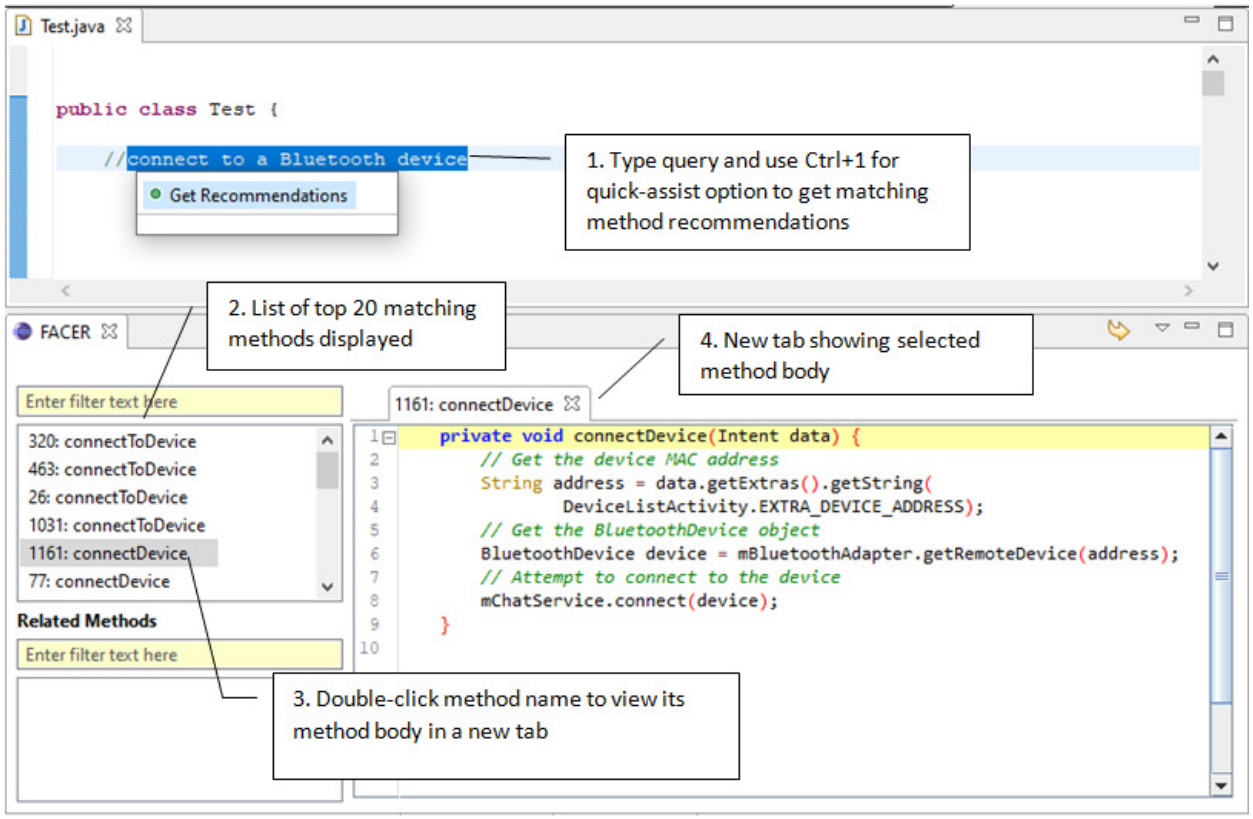


Figure 5.6: Stage 1: Method Search

tion of a method by the user, Stage 2 recommends related methods for opportunistic reuse. The FACER system components for online recommendation workflow are shown in Figure 5.1a and are discussed below.

5.4.1 FACER Stage 1: Method Search

This module gets triggered whenever the user types a comment and presses the *CTRL+1* key combination afterwards. The comment should describe the feature they wish to implement. We show an example in Figure 5.6 where the developer types the feature query *"Connect to a Bluetooth device"*, uses Ctrl+1 and selects *"Get Recommendations"* from a quick-assist popup. FACER then processes the input comment (query string) and initiates a search to retrieve top-N matching methods from the FACER repository. Note that we are not contributing a novel code search engine. Any code search technique [5, 7, 68, 72–74, 164] can be used here. However, to implement the whole

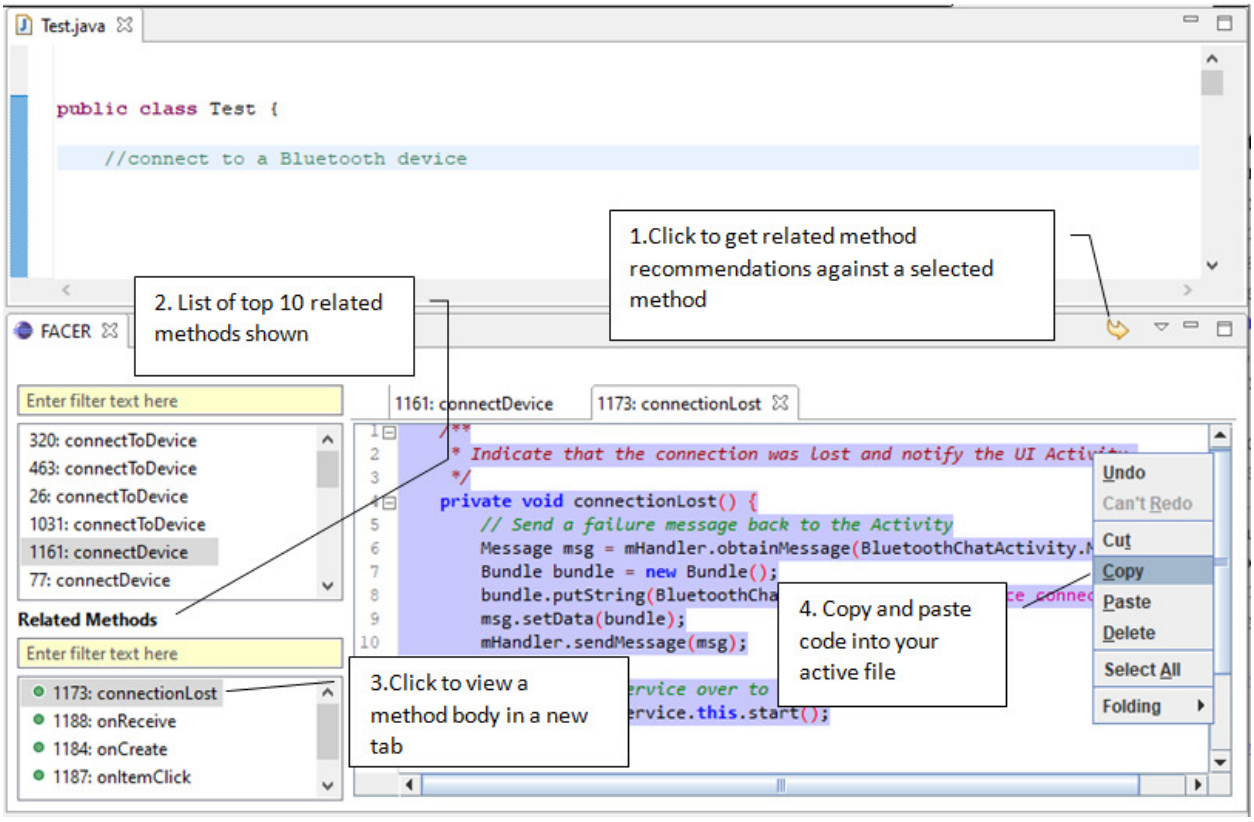


Figure 5.7: Stage 2: Related Method Recommendations

workflow, we develop a simple code search engine (B1 [7]) using Lucene [92]. The performance of such a search engine has been shown to significantly improve software retrieval performance, increasing the area under the curve (AUC) retrieval metric to 0.92 – roughly 10–30% better than previous approaches based on text alone [69]. Lucene uses the BM25 (Best Matching) textual similarity ranking method implemented in Okapi [165]. The output of this FACER search stage is a ranked list of top matching methods against the input comment. We currently show the developer the top 20 matching methods. The bottom left of Figure 5.6 shows the list of methods retrieved against the example query. Developers can click on any of these methods to view their content in the right pane. Once decided, they can get related method recommendations against the currently selected method by clicking the arrow button on the top right corner of FACER's view panel as shown in Figure 5.7.

5.4.2 FACER Stage 2: Related Method Recommendations

To obtain a list of related method recommendations, we use the user-selected method (m_u) from the previous step along with a minimum support threshold (β) as input. The clone structures of co-occurring API usage-based Method Clone Groups mined from Step 2 (Section 5.3.3) are the basis for FACER's recommendations.

Algorithm 2 summarizes the steps for recommending related methods against an input method (m_u). We first identify which clone group (i.e., cluster) m_u belongs to (Line 3), and use it to get related method recommendations (Line 5). The procedure `GETRELATEDMETHODS` for getting recommendations against a clone group ID is shown on Line 14. In this procedure, FACER retrieves only those Method Clone Structures (MCS) that satisfy the threshold β and performs highest-support-first ordering of the MCS (Line 15). After obtaining a list of MCS, we gather all distinct clone groups found in each MCS as co-occurring features against our input feature (Line 17). Next, for each of the clone groups, we get representative methods and add those methods to the list of recommended related methods (Lines 19-23). To select a representative method from each clone group, we follow a simple rule: if a clone group contains a method that belongs to the same project as the user's already selected m_u, then we choose that method as the representative method. Otherwise, we choose the method with the highest API call density within the clone group. This is to ensure that the recommended method has the least amount of noise in the form of statements without API calls.

In case m_u's clone group does not belong to any clone structure (Lines 6–8) or if m_u does not belong to a clone group (Lines 9–11), we scan the neighboring methods of m_u to perform neighborhood-based retrieval. The procedure for getting recommendations based on neighboring methods is shown on Line 25. We first use the call graph of m_u as the source of neighboring methods to obtain recommendations (Lines 26–27). Specifically, we use the caller and callee methods of m_u as its neighborhood and thus input methods. If this returns an empty set of related methods, we then use the host file of m_u as the source of neighboring methods to obtain recommendations (Lines 29–30). In this case, all the methods of the host file containing m_u form its neighborhood and are used as input.

Algorithm 2 FACER Stage 2: Getting Related Method Recommendations

```
1: procedure MAIN(m_u, β)
2:     relatedMethods ← {}
3:     cloneGroupID ← getCloneGroupID(m_u.ID)
4:     if cloneGroupID ≠ ∅ then
5:         relatedMethods ← GETRELATEDMETHODS(cloneGroupID, β)
6:         if relatedMethods = ∅ then
7:             relatedMethods ← GETFROMNEIGHBORHOOD(m_u.ID, cloneGroupID, β)
8:         end if
9:     else
10:        relatedMethods ← GETFROMNEIGHBORHOOD(m_u.ID, cloneGroupID, β)
11:    end if
12:    return relatedMethods
13: end procedure
14: procedure GETRELATEDMETHODS(cloneGroupID, β)
15:    cloneStructsList ← getMCS(cloneGroupID, β)          ▷ highest support first
16:    if cloneStructsList ≠ φ then
17:        cloneGroupsList ← getUniqueCloneGroups(cloneStructsList)
18:        for all CID ∈ cloneGroupsList do
19:            m ← getRepresentativeMethod(CID)
20:            relatedMethods.add(m)
21:        end for
22:    end if
23:    return relatedMethods
24: end procedure
25: procedure GETFROMNEIGHBORHOOD(m.ID, cloneGroupID, β)
26:    neighborSource ← CallGraph(m.ID)
27:    relatedMethods ← NEIGHBORHOODRETRIEVAL(neighborSource, m.ID, cloneGroupID, β)
28:    if relatedMethods = ∅ then
29:        neighborSource ← HostFile(m.ID)
30:        relatedMethods ← NEIGHBORHOODRETRIEVAL(neighborSource, m.ID, cloneGroupID, β)
31:    end if
32:    return relatedMethods
33: end procedure
34: procedure NEIGHBORHOODRETRIEVAL(neighborSource, m.ID, cloneGroupID, β)
35:    NeighborCloneGroupsList ← getCloneGroups(neighborSource, m.ID)
36:    for CID ∈ NeighborCloneGroupsList do
37:        MCS_Support_Map.add(getMCS(CID, β))
38:    end for
39:    if MCS_Support_Map ≠ ∅ then
40:        sortedMCS_Support_Map ← sortMCSByDecreasingSupport(MCS_Support_Map)
41:        for all cloneStructs ∈ sortedMCS_Support_Map do
42:            cloneGroupsList ← getUniqueCloneGroups(cloneStructs)
43:            for all CID ∈ cloneGroupsList do
44:                m ← getRepresentativeMethod(CID)
45:                relatedMethods.add(m)
46:            end for
47:        end for
48:    end if
49:    return relatedMethods
50: end procedure
```

Line 34 shows the procedure to perform neighborhood retrieval, regardless of the neighborhood source used. It involves getting the clone groups of all methods in the neighborhood of m_u. For each of these clone groups, we get all the MCS in which they occur (Lines 36–38). We then sort these MCS in order of highest-support-first and build a list of distinct clone groups that occur in those MCS (Lines 40-42). Finally, we get representative methods against the clone groups as before (Lines 43-45) and return the list of related method recommendations for m_u.

5.5 Chapter Summary

The ultimate goal of our related feature recommendation system is to support opportunistic reuse through recommending relevant related methods. Our approach is designed to minimize the number of irrelevant related features recommended, while maximizing the success rate of obtaining code examples for relevant features.

Current code recommendation and code search systems focus on the immediate requirements of the developer. They retrieve code against a specific query. For a new but related task, the developer has to perform a new search. The need to perform repeated searches for associated functionality can impede the performance and productivity of the developer. In this chapter, we proposed a solution that allows developers to receive recommendations for their potential future requirements. Our main contribution is a recommendation system FACER that provides developers with method recommendations having functionality relevant to their current feature under development.

FACER works in two stages. The first stage is a simple code search engine which given a query returns a code snippet implementing the feature in the query. The second stage, which is the main contribution of this dissertation, is a recommender which given a selected method from Stage 1 recommends related methods that implement related functionality. For example, if a developer is currently implementing the "connect to Bluetooth" feature and found a relevant method to reuse, FACER would recommend a method implementing the "disconnect from Bluetooth" functionality as a related feature the developer may need to implement. To accomplish this, FACER relies on clustering methods according to their API usages, where a cluster represents methods implementing the same or similar functionality. It then finds frequently co-occurring method clusters which

it uses to recommend related functionality.

Chapter 6

FACER Evaluation

In this chapter, we discuss the evaluation of the Method Clone Groups detected by FACER to determine whether the methods that FACER clusters into the same clone group actually implement the same functionality/feature. Furthermore, we also discuss the evaluation of the main contribution of FACER, recommending related features. This is the functionality for FACER's Stage 2 described in Section 7.3.3. We have two evaluation goals. One is to determine the optimal threshold parameters for similarity (α) and minimum support (β) used in providing recommendations. We determine these thresholds using an automated evaluation setup. The second goal is to determine the precision of the related methods that FACER recommends as judged by a human. Specifically, given a query and a selected method matching this query, we recruit participants to evaluate whether the related methods recommended by FACER indeed represent additional functionality related to the initial query, considering the application being developed.

The ultimate goal of our related feature recommendation system is to support opportunistic reuse through recommending relevant related methods. Our approach is designed to minimize the number of irrelevant related features recommended, while maximizing the success rate of obtaining code examples for relevant features. We now discuss our research questions and the evaluation setup we use.

6.1 Research Questions

We aim to answer the following four research questions (RQs):

- **RQ 6.1:** Are the clone groups detected by FACER valid?
- **RQ 6.2:** How precise is FACER in terms of recommending related features?
- **RQ 6.3:** Do developers need to search for related features?
- **RQ 6.4:** What are developers' perceptions regarding the usefulness and usability of FACER?

The underlying premise of FACER is that methods with similar API usages are semantically related and can represent methods implementing the same feature. If this is not true in practice or if our clone groups are meaningless, then the rest of FACER's workflow will not be useful. Thus, in RQ 6.1 (Section 6.3), we manually validate a sample of the clone groups detected by FACER. The goal is to make sure that methods belonging to the same clone group implement the same functionality and that different clone groups represent different functionality. The aim of RQ 6.2 is to evaluate whether the methods recommended in FACER's Stage 2 actually implement features that relate to the user's selected feature/method. To evaluate this, we perform two types of evaluation. The first is an automated evaluation (Section 6.4) that compares the recommendations against ground truth data to determine the best threshold values and the second is a manual evaluation (Section 6.5) that involves human validation. The aim of RQ 6.3 is to understand the code search and reuse practices of developers and find out whether they need to search for related features. In RQ 6.4, we determine how developers perceive the usability and usefulness of the current FACER tool and its recommendations. To answer RQ 6.3 and RQ 6.4, we conduct a user survey which includes assessing developers' code search and reuse practices (Section 6.6), presenting the developers with recommendation scenarios for reviewing FACER's related method recommendations and then getting their feedback on the FACER tool's interface and recommendations (Section 6.7).

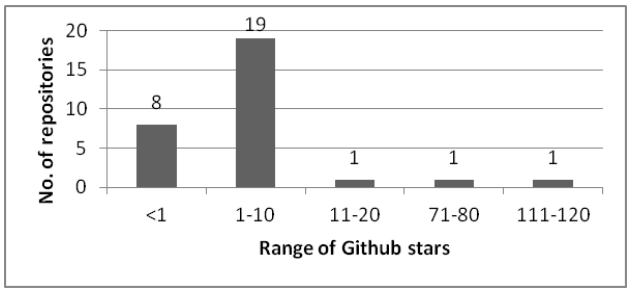
Table 6.1: FACER code fact repository statistics

Metric	Value
No. of applications	120
No. of files	4,369
Lines of comments	175,000
Lines of code (LOC)	498,261
No. of methods	37,303
No. of method calls	150,341
No. of API classes	2,209
No. of unique API calls	7,607
Total no. of API calls	85,386

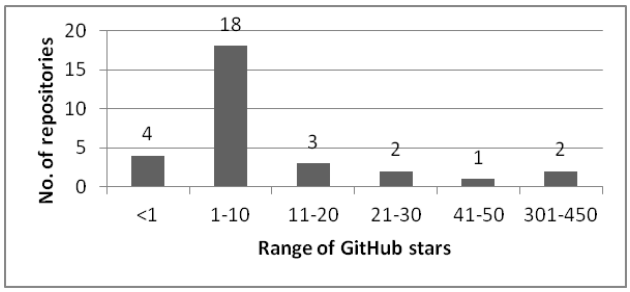
6.2 Dataset

We collect applications from four different categories of Java-based Android applications: (1) music player, (2) Bluetooth chat, (3) weather, and (4) file management [166]. We choose these categories because of their use in previous research on feature recommendations [8] and API usage pattern recommendations [102]. We include 30 applications from each category, resulting in a total of 120 applications. We intentionally choose multiple applications from each category to allow the discovery of frequently co-occurring features across similar category applications.

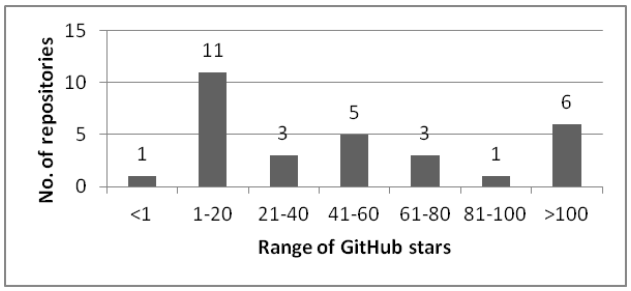
To collect the applications forming the dataset, we use GitHub's search where we use each category name prefixed with *android* and postfixed with *app* as search queries. Then, we filter the search results by choosing Java as the language and sort them using *relevance* option. We then select the top 30 relevant GitHub repositories against each search query. We manually judge the relevance to a category by analyzing the description of each application on GitHub. If an application is not deemed relevant, we skip it. Figure 6.1 shows the distribution of star ratings for the selected applications across the four categories. We can see that the weather category has the highest starred applications, followed by music, file manager and Bluetooth categories.



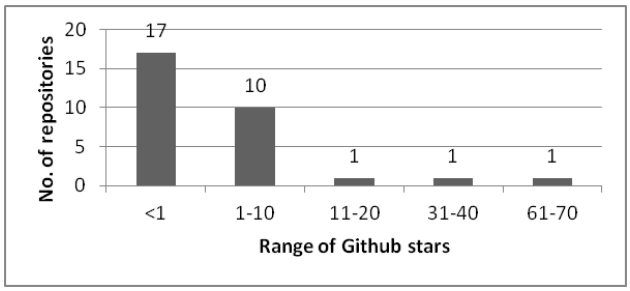
(a) File Manager Category



(b) Music Category



(c) Weather Category



(d) Bluetooth Chat Category

Figure 6.1: The number of GitHub repositories from the four categories across different ranges of the number of stars

Overall, our dataset for the evaluation consists of 120 Java-based Android applications which we analyze in order to populate the FACER repository.

6.2.1 Constructing the FACER Repository

To populate the FACER repository in offline mode, we analyze the source code of the collected 120 applications. The time to execute the program analyzer on this dataset is almost 55 minutes on a Core i7 2.2 GHz machine with 8GB memory running Windows 10. Table 8.1 summarizes some of the key statistics of the FACER repository that we built from the 120 applications, and which we use to answer our research questions.

During the detection of clone groups, we consider only methods having a minimum of three unique API calls to ensure that we have meaningful clusters [167, 168]. We also ignore API calls involving the usage of *Log*, *Intent* and *Toast* API classes, because want to filter out common API

Table 6.2: Method Clone Groups (MCG) and Method Clone Structures (MCS) detected with varying similarity threshold α

α	No. of MCG	No. of MCS
0.3	1445	536
0.5	1397	107
0.7	812	37
0.9	347	11

calls which appear in almost every application and do not contribute towards a particular feature of an application. Thus, out of the 37,303 methods in the repository, we mine clusters from 7,922 methods. Overall, these 7,922 methods have 7,028 unique API calls.

We input a 7922 $\times$ 7028 binary matrix whose rows represent methods and columns represent all unique API calls found across all the methods. A value of 1 in the matrix means that the API call exists in the method. One of the challenges of clustering methods on the basis of API calls is the storage and computation required to process large matrix sizes when the number of methods and the number of API calls increase. To efficiently calculate pair-wise API usage similarity between methods, we make use of a third-party function [169, 170] that performs rapid calculation of the Jaccard distance of a matrix by making use of raw vectors with the binary data packed efficiently. For calculating pair-wise API call density-based distances between methods, we use another third-party library function [171] to perform distance matrix computation in parallel using multiple threads. It supports predefined distance measures and user-defined distance functions. For our purpose, we specify our own distance function based on our similarity formula shown in Equation 5.3. Table 8.2 shows the number of clone groups and Method Clone Structures that we obtain as a result of clustering and frequent pattern mining under various similarity thresholds. Increasing the threshold results in fewer clone groups and Method Clone Structures because of the stricter clustering criteria.

6.3 Method Clone Group Evaluation

In RQ 6.1, we evaluate the Method Clone Groups detected by FACER to determine whether the methods that FACER clusters into the same clone group actually implement the same functionality/feature. This is *intra-clone group similarity validation*. We also evaluate *inter-clone group dissimilarity* to verify that the Method Clone Groups do not share any functionality with each other.

6.3.1 Validation Method

We manually evaluate the clone groups which FACER detects with a minimum similarity score threshold α = 0.5. This relieves us from evaluating clone groups obtained with larger threshold values of α since they will always be better due to a higher similarity between clone group members. We also observe from our automated evaluation in Section 6.4.3 that this alpha value gives us the optimal precision and success rate. We use this same alpha value for all our evaluations in this chapter. Myself, three faculty members having experience in the area of software engineering, as well as one professional senior Android developer conduct the manual validation. We first explain how we select the clone groups that we evaluate and then explain the manual validation process we follow for inter and intra clone group validation.

Clone Group Sampling

Since manually evaluating all 1,397 clone groups where each clone group has several methods is not practically feasible, we perform multi-stage sampling to select the clone group and methods for our evaluation.

We first need to select clone groups to evaluate. We want to make sure we manually validate a diverse set of clone groups. Thus, we take into account the following clone-group characteristics during our sampling:

- *size*: We calculate the size of a clone group as the number of methods in a clone group. The higher the number of methods in a clone group, the more common the feature represented by the clone group is. Sampling by size allows us to choose from a spectrum of less common

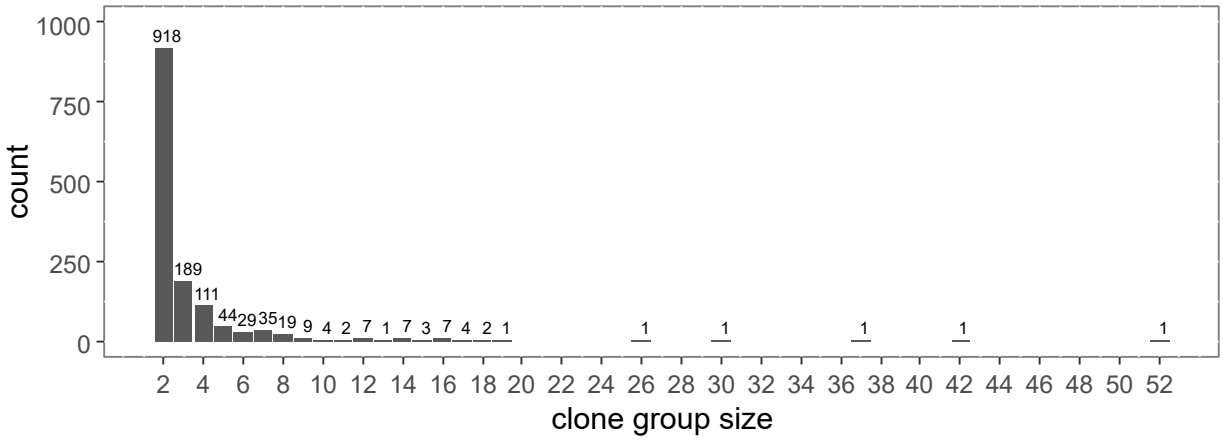


Figure 6.2: Frequencies of clone groups of varying sizes with similarity threshold $\alpha = 0.5$

features as well as widespread features. Figure 6.2 shows that the *size* of clone groups in our data set varies from 2 to 52. We observe that, not surprisingly, there are a larger number of small-sized clone groups when compared to larger clone groups. This observation is also reported in previous code clone detection studies where small clone groups are overwhelmingly the most common of all clone groups [172, 173].

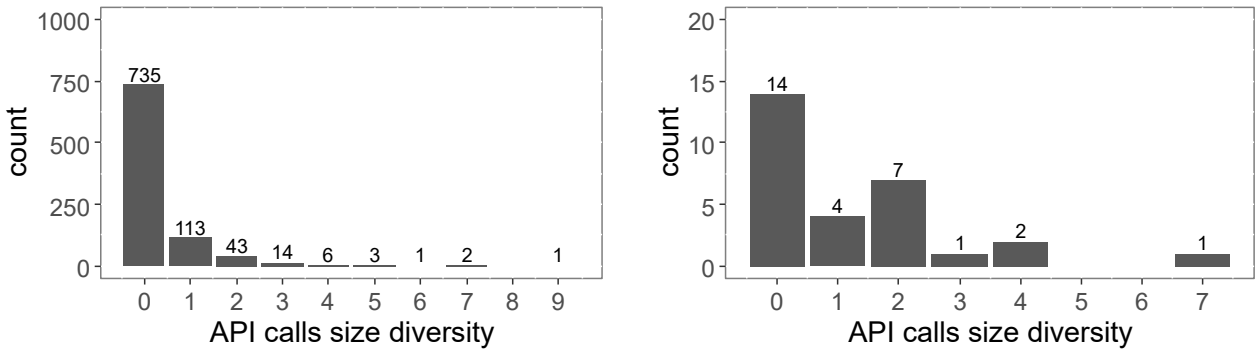
- *API call size diversity*: Each clone group can have methods with a varying number of unique API calls. We calculate the *API call size diversity score* of a clone group as the difference between the minimum and the maximum number of unique API calls found across all methods of a clone group. For example, if a clone group of size 2 has a method with API calls A, A, B, and C and the other method has API calls A, B, C, D, E, and F, then the diversity score of this clone group will be 6-3=3. A higher diversity value is a proxy for more diverse functionality in the clone group. Sampling clone groups by diversity enables us to sample methods having various API call sizes in the next sampling stage.

Given the above two criteria, we select clone groups using a two-stage sampling. In the first stage, we perform systematic cluster sampling to select clone group sizes. For the number of different clone group sizes *n*, we systematically select every alternate clone group size which results in selecting 50% of the available sizes. In the second sampling stage, we select clone groups from low, median and high API call size diversity strata found within each sampled clone group size.

Table 6.3: Two-stage sampling of 126 clone groups from a total of 1,397 available clone groups

Sampled clone group sizes (Stage 1 sampling)	No. of clone groups in each size	No. of sampled clone groups of each size (Stage 2 sampling)
2	918	92
4	111	12
6	29	3
8	19	3
10	4	3
12	7	3
14	7	3
16	7	3
18	2	1
26	1	1
37	1	1
52	1	1

For example, from Figure 6.3a we can see that for clone groups of size 2, the API call size diversity values range from 0 to 9 on the x-axis. The median of these values is 4, and we form strata using the median value as a reference. Values close to the median value (3 and 5) fall in the *median stratum*, whereas values (0, 1, 2) lower than those of the median stratum values fall in the *low stratum* and values (6, 7, 9) higher than median stratum values fall in the *high stratum*. Having determined the strata, we now randomly select a diversity value from each stratum. From Figure 6.3a, we choose the API diversity values 0, 3, and 5. We then continue randomly selecting one clone group from the sampled diversity values until we sample at least 10% of the total number of clone groups of a particular clone group size. Table 6.3 shows the results of our sampling criteria until this step. This sample size of 126 clone groups gives us an 8% margin of error at a 95% level of confidence.



(a) Frequency of API call size diversity for clone groups of size 2

(b) Frequency of API call size diversity for clone groups of size 6

Figure 6.3: Example API call size diversity for clone groups of size 2 and 6

Method Sampling

Note that we have so far identified clone groups to evaluate but not the particular methods that we will manually validate from those clone groups. Thus, we now discuss how we identify the particular methods for validation.

For sampled clone groups of size 2 which only contain two methods, we include both methods in our sample. However, it is a big manual overhead to manually validate each method for large-sized clone groups. Thus, we sample representative methods from the selected clone groups for manual validation. To sample these methods, we take into account the following method characteristics:

- *API-call size*: We select methods with the smallest, median and largest number of unique API calls within a sampled clone group. This allows us to sample methods of various API call sizes. Figure 6.4 shows the distribution of API-call sizes for all methods in our selected sample of 126 clone groups from Table 6.3. We use this distribution to sample methods on the basis of API-call size. The number of unique API calls ranges from a minimum of 3 (due to our clustering criteria explained in Section 6.2.1) to a maximum of 46.

- *API-call density*: In addition to selecting methods by API-call size, we select methods with the highest and lowest API-call densities to add more methods to the sample. Note that the

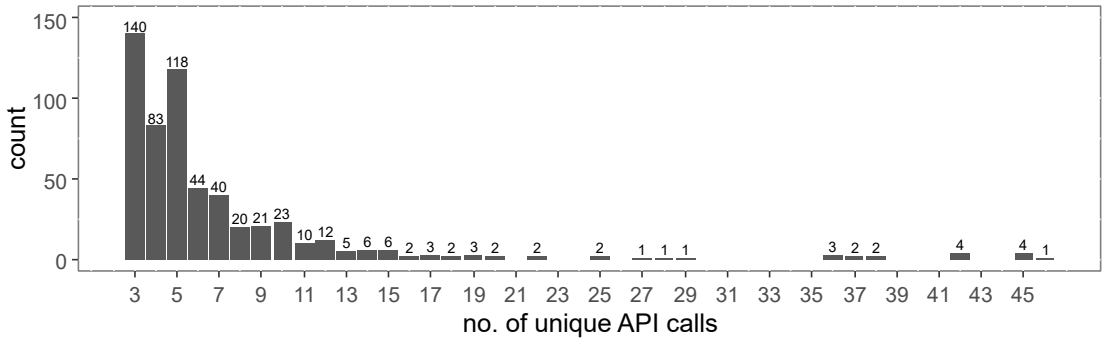


Figure 6.4: Distribution of API call size for all the methods from our sampled clone groups in Table 6.3

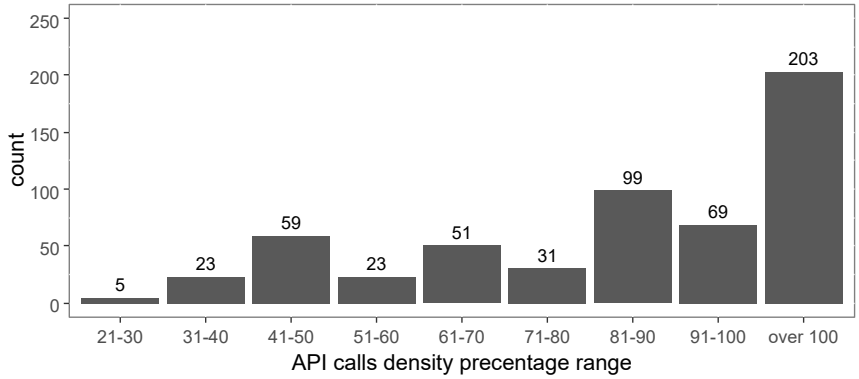


Figure 6.5: Method distribution from sampled clone groups based on API call density

previous sampling using API-call size may already contain a high and low density method from each clone group, in which case we do not need to add more methods from this step. Figure 6.5 shows the distribution of all the methods from our sampled clone groups across different API-call density ranges. We observe that almost 99% of methods in our sampled clone groups have API-call density values greater than 50%. This implies that API calls form a major part of the code of these methods and also supports our technique of clustering methods on the basis of API calls to detect common functionality. We sample methods on the basis of API-call density from this distribution.

Based on the above sampling criteria, our final sample consists of 126 clone groups with a total of 305 methods.

Intra-clone group similarity validation

Setup Myself, three faculty members having experience in the area of software engineering, and one professional Android developer perform the manual intra-clone group validation where our goal is to check whether methods of a clone group are functionally similar. We first explain the evaluation procedure and then explain how the clone groups were distributed among the evaluators.

We follow the following evaluation procedure for all 126 clone groups. For each method of a clone group, the evaluator writes a feature description describing what the method is doing. This description is based on using the code in the method body (including method invocations and API calls), any Javadoc comments, the method name, and any inline comments. Once the evaluator writes feature descriptions for all methods in the clone group, they then write a feature description for the clone group which represents the functionality shared by all the methods in the clone group. For methods having functionality that does not match with the core functionality of the clone group, this functionality is noted by the evaluator as a *divergent* feature. Finally, the evaluator gives a decision regarding the validity of the clone group. A clone group is *valid* if all of its member methods (or the analyzed sub sample) implement similar functionality. A clone group is *invalid* if the evaluator is able to identify one or more of its member methods having several divergent functionalities. Having too many divergent features in a method indicates that the methods do not implement the same functionality, which affects the validity of the clone group. In other words, the decision to assign a valid label to a clone group is based on the following observations:

- The resulting feature description of a clone group describes the common functionality in the clone group.
- There are no major divergent features inside any member method or the number of divergent features is few or minor in comparison to the clone group's feature description.

We first assign the evaluation of each sampled clone group to two authors such that we get two unique author evaluations per clone group. Each author independently evaluates their assigned clone groups and labels each clone group as *valid* or *invalid*.

To ensure external validation and reduce author subjectivity, we also recruit a professional

Android developer to evaluate and label all 126 clone groups. We recruit the professional Android developer using a freelance website Fiverr [174]. The developer has six years of experience developing Android applications such as online food ordering, shopping, personalization, video players, live streaming, VPN, utility apps, and customized apps for several businesses. We provide the professional developer with the evaluation data accompanied by instructions about the evaluation procedure we described above. This means that each of the 126 clone groups are evaluated by two authors and one external professional developer. For all agreements between authors, we compare the professional developers' evaluation with the author evaluations to see whether their rating matches ours. We obtain the final labels of all clone groups using a majority vote of the three ratings.

Intra-clone group similarity results

We evaluate 126 clone groups containing a total of 305 methods. For the author ratings, we obtain an 84% agreement rate and a Cohen's kappa score [175, 176] of 0.38, which indicates a fair agreement. There were a total of 20 disagreements between the author ratings, which we resolve using the majority vote of all three ratings.

There were 106 clone groups for which both authors agreed on the label. We compare these 106 ratings to the corresponding ratings of the professional developer to check whether the authors' perception matches that of an impartial third party. The authors and the professional developer agreed on 97 clone groups being valid and five being invalid. Overall, we find that the authors had a 96% agreement rate with the professional developer and a Cohen's kappa of 0.69, indicating a substantial agreement [175, 176].

Overall, after resolving all disagreements across the 126 clone groups using majority vote, we confirm that 115/126 (91%) clone groups are valid.

In Figures 6.6a-6.6e, we provide examples of two valid clone groups, one with size 10 and one with size 37. We can see that the member methods of each clone group do share common functionality. For the first clone group in Figures 6.6a and 6.6b, the shared functionality checks for the availability and connectivity of network, and for the second clone group shown in Figures 6.6c-

```
public boolean isNetworkAvailableAndConnected() {
        ConnectivityManager connectivityManager
                = (ConnectivityManager)
                      mContext.getSystemService(Context.CONNECTIVITY_SERVICE);

        NetworkInfo networkInfo = connectivityManager.getActiveNetworkInfo();
        return networkInfo != null && networkInfo.isConnected();
    }
```

(a) Clone Group 1 Method 1

```
public boolean isNetworkAvailable() {
        ConnectivityManager connectivityManager
                = (ConnectivityManager)
                      mContext.getSystemService(Context.CONNECTIVITY_SERVICE);
        NetworkInfo activeNetworkInfo = connectivityManager.getActiveNetworkInfo();
        return activeNetworkInfo != null && activeNetworkInfo.isConnected();
    }
```

(b) Clone Group 1 Method 2

```
/**
    * Indicate that the connection was lost and notify the UI Activity.
    */
   private void connectionLost() {
       // Send a failure message back to the Activity
       Message msg = mHandler.obtainMessage(LanylActivity.MESSAGE_TOAST);
       Bundle bundle = new Bundle();
       bundle.putString(LanylActivity.TOAST, "?????????");
       msg.setData(bundle);
       mHandler.sendMessage(msg);

       // Start the service over to restart listening mode
       LanylService.this.start();
   }
```

(c) Clone Group 2 Method 1

```
private void connectionLost() {
        Message msg = handler.obtainMessage(MainActivity.MESSAGE_TOAST);
        Bundle bundle = new Bundle();
        bundle.putString("toast", "Conexion perdida con el dispositivo");
        msg.setData(bundle);
        handler.sendMessage(msg);

        // Start the service over to restart listening mode
        ChatController.this.start();
    }
```

(d) Clone Group 2 Method 2

```
private void connectionFailed() {
      Message msg = handler.obtainMessage(MainActivity.MESSAGE_TOAST);
      Bundle bundle = new Bundle();
      bundle.putString("toast", "Unable to connect device");
      msg.setData(bundle);
      handler.sendMessage(msg);

      // Start the service over to restart listening mode
      ChatController.this.start();
  }
```

(e) Clone Group 2 Method 3

Figure 6.6: Examples of evaluated clone groups. Figures 6.6a-6.6b show two methods from a clone group of size = 10. Figures 6.6c-6.6e show three methods from a clone group of size = 37

6.6e, the shared functionality sends a failure message to some activity and restarts a service. We also present an example of clone groups with longer methods in Appendix B. Additionally, all the data and labels from this manual evaluation are provided in our online artifact page [166].

Inter-clone group dissimilarity validation

Setup We also verify whether the clone groups detected by FACER share any functionality with each other. The idea is to look at clone group descriptions and decide whether any two clone groups are semantically similar. Ideally, there should be minimal functionality overlap between clone groups. This evaluation is performed by myself and the same professional Android developer who performed the intra-clone group similarity validation.

Each evaluator uses their own previously written feature descriptions so that it is easier for them to perform the task and because that reflects how they perceive the clone group's functionality. For each evaluator, we first collect the feature descriptions of all 115 clone groups that we resolve as valid in the intra-clone group validation phase. To reduce manual effort and chances of incurring human error while analyzing all $\binom{115}{2} = 6555$ combinations of feature descriptions, we form a subset of the clone group descriptions of each evaluator based on TF-IDF [177] similarity. After stemming all words in the descriptions, we calculate pair-wise similarity between all feature descriptions of an evaluator using a TF-IDF similarity score. We filter out the clone group pairs with a similarity score less than or equal to 0.5 and assume that these are dissimilar. The clone group pairs that have a similarity score greater than 0.5 are the ones we need the evaluators to manually validate. We then ask the evaluator to analyze the similar pairs of clone group descriptions (i.e., those with a TF-IDF score of > 0.5) and their associated code to determine whether the clone group pairs are similar or distinct. The two evaluators discuss any disagreements on labels for commonly evaluated clone group pairs until they reach a resolution.

Inter-clone group dissimilarity results

We first execute the TF-IDF similarity calculation on the clone group descriptions (of valid clone groups) written by the professional developer. As a result, we obtain 23 clone group pairs having

> 0.5 clone group description similarity. We then execute the TF-IDF similarity calculation on the clone group descriptions (of valid clone groups) written by the first author. As a result, we obtain 27 clone group pairs having > 0.5 clone group description similarity. Both the author and the professional developer then evaluate the clone group pairs obtained from their own respective descriptions and having a TF-IDF similarity > 0.5. The professional developer manually evaluates each of their 23 clone group pairs by looking at the corresponding code for the sampled methods of a clone group and concludes that 16 of these pairs are semantically similar to each other. The first author manually evaluates each of their 27 clone group pairs by looking at the corresponding code for sampled methods of a clone group and concludes that only two pairs of clone groups are semantically similar to each other. We note that there are seven clone group pairs in common between the 23 clone group pairs obtained from the developer's descriptions and the 27 clone group pairs obtained from the author's descriptions. Thus, based on the descriptions from both evaluators, there are a total of 43 unique clone group pairs that are potentially similar based on a TF-IDF similarity threshold > 0.5.

For the seven clone group pairs evaluated by both the author and the developer, there was an agreement on only one clone group pair being similar. After resolving the six disagreements through discussion, we find that out of the 43 unique clone group pairs across both evaluators, 31 are dissimilar, and 12 are similar. Thus overall, out of 6,555 clone group pairs formed from 115 unique clone groups, only 12 clone group pairs are semantically similar. This means that 99.8% of the clone group pairs are dissimilar, which means that our clustering based on API usages works well.

Overall, our manual evaluation results for both intra- and inter- clone group validation give us confidence that common API usages can indeed be used as a proxy for similar functionality, and that the clone groups detected by FACER are meaningful. With that, we can proceed to evaluate FACER's recommendations of related functionality.

RQ 6.1: Based on a manually validated sample, 91% of the clone groups detected on the basis of API usage similarity consist of methods having similar functionality. This demonstrates that the similarity of API calls is a valid indicator of functional and feature similarity between methods. Furthermore, we observe that 99.8% of clone group pairs in our sample are functionally dissimilar which indicates that the clone groups detected are unique and rarely overlap with another clone group in terms of functional similarity.

6.4 Automated Evaluation for Sensitivity Analysis of Parameters

6.4.1 Evaluation Methodology

In a real recommendation scenario, a developer inputs a feature query and gets matching methods against the query from the FACER repository. Then, the developer selects one of those methods for reuse and based on her selection, gets additional related method recommendations. If she finds them relevant, she can reuse them as part of her application.

In an automated evaluation scenario, we need to verify that the related method recommendations are relevant by measuring their precision against a ground truth. In other words, we need a criteria for automatically specifying that a recommended method is indeed related to the input method since using human validation for different recommendations at different thresholds is infeasible due to overloading our human participants.

We thus create a proxy ground truth for automated evaluation as follows. Given an input method m from a project p in FACER's repository, we consider any method in p as the ground truth for related method recommendations for m. Thus, we consider the project to which the input method m belongs to as the ground truth project. The ground truth we use in our automated evaluation is a proxy for a subjective decision that should be made by the developer. Methods appearing in the same project typically represent related functionality, and thus conceptually match FACER's intended purpose.

As part of the recommendation process, FACER maps an input method to a clone group, gets related clone groups through examining the Method Clone Structures, and finally returns representative methods from the related clone groups. For our automated evaluation, we consider that a true positive recommendation occurs whenever FACER returns a representative method for related clone groups that happens to be from the ground truth project. Inversely, a false positive occurs when the representative method for the related clone group happens to be from a different project.

We now discuss how we select test input methods to evaluate FACER, as well as the metrics we use for the evaluation. In total, we evaluate 20 recommendation scenarios corresponding to 20 test input methods, which we also use for the manual evaluation in the next section.

To obtain related method recommendations from FACER, we need to start with a feature query and then select a relevant method from FACER's Stage 1 recommendations. Methods recommended in Stage 1 may or may not yield related method recommendations. While getting related method recommendations from Stage 2 is optional for a user, we only consider recommendation scenarios that include related method recommendations for the purpose of our evaluation. The feature query, selected method, and related method recommendations together make up a recommendation scenario. Since we have four categories of Android applications, we want to evaluate a few recommendation scenarios for each category. Thus, we evaluate a total of 20 recommendation scenarios with five for each of the four categories of applications. We need to simulate these recommendation scenarios by issuing feature queries and then selecting a method that corresponds to Stage 1 recommendations, against which FACER Stage 2 can then return related method recommendations for evaluation.

To make our evaluation as realistic as possible, we create feature queries by manually gathering a list of feature descriptions from the README files of all applications in the FACER repository. These feature descriptions are mainly short phrases that begin with an action verb. We then shortlist the feature descriptions that are common across multiple applications within a particular category. This results in a set of 10 queries. To collect an additional 10 queries, we manually locate methods from the back-end FACER repository using SQL queries that look for certain domain-specific keywords in the API calls of methods belonging to Method Clone Structures. One of the

authors then assigns a feature description to the method using its name, comments and body. This feature description is then input to FACER Stage 1 to retrieve a set of matching methods. Thus, the 20 queries are a mix of feature descriptions we get from README files and some that we manually create.

We use each feature description of the 20 queries as the feature query to FACER Stage 1 and manually examine the list of returned recommendations. We select a method from the set of Stage 1 recommendations based on the following criteria: (1) the method name, comments and variable names indicate that it implements the desired functionality, (2) it is capable of generating related method recommendations. We consider criterion 2, because we want to evaluate the quality of related methods FACER recommends, which we cannot do for scenarios where FACER makes no related recommendations. Such a scenario would occur when the input method does not belong to any Method Clone Structure, which can be solved with considering more input repositories for the mining stage. Thus, given our evaluation goals, we focus on the case when FACER can find *any* related methods and evaluate the quality of these related recommendations.

One of the authors having professional experience with Android development first selects methods against queries using the FACER tool in a way that a real developer would do by looking through the complete list of retrieved methods, then clicking on a few methods which look relevant by name, scanning method bodies to check for desired functionally, and finally checking whether these methods have any related method recommendations (without evaluating the related recommendations). This way, we obtain a set of 20 test input methods corresponding to the 20 feature queries. We then use the ID of the selected method as input to FACER Stage 2 to get related method recommendations. These related method recommendations are what we evaluate. The 20 queries, the method identifiers of the selected relevant methods for the queries, and their application category names are shown in Table 8.3.

6.4.2 Evaluation Metrics

We use the following metrics to evaluate the related method recommendations:

Precision: Precision measures FACER's ability to correctly recommend related methods. The

Table 6.4: Feature queries

No.	Feature description	Method ID	Category
1	receive paired devices name and address	33	Bluetooth chat
2	update list of paired Bluetooth devices	80	Bluetooth chat
3	do discovery of Bluetooth devices	423	Bluetooth chat
4	send message over Bluetooth	1066	Bluetooth chat
5	connect to a Bluetooth device	1161	Bluetooth chat
6	create new folder	2250	File Manager
7	browse to file or directory	2616	File Manager
8	move file	2642	File Manager
9	put file to cache	2971	File Manager
10	draw bitmap	3017	File Manager
11	set data source for media player	14435	Music Player
12	receive key press to start stop pause media	14490	Music Player
13	search for song	15214	Music Player
14	download music	22968	Music Player
15	play music	24068	Music Player
16	save forecast in database	28669	Weather
17	send Http request to get weather	29298	Weather
18	check if network connection available	29947	Weather
19	check and add permissions for location access	31838	Weather
20	create new memory cache to store weather icons	33549	Weather

precision of recommendations is calculated as the fraction of recommended methods that are relevant i.e., belonging to the ground truth project of the input method, as shown in Equation 6.1. If all the recommended methods occur at least once in the test project, we have 100% precision.

$$Precision@N = \frac{|recommended\,methods \cap test\,project\,methods|}{|recommended\,methods|} \tag{6.1}$$

Success Rate: This metric measures the rate at which the recommender can return at least one relevant recommendation against an input method. The success rate is defined as shown in Equation 8.3.

$$SuccessRate = \frac{|queries\,answered|}{|queries|} \tag{6.2}$$

where *queries* represents the set of test input methods FACER receives and *queries answered* represents the number of times FACER successfully retrieves at least one correct recommendation against a test input method.

Mean Reciprocal Rank: The mean reciprocal rank is the average of the reciprocal ranks of the results for the number of test input methods *M*. It is defined in Equation 8.4.

$$\text{MRR} = \frac{1}{|M|}\sum_{i=1}^{|M|}\frac{1}{\text{rank}_i} \tag{6.3}$$

where $rank_i$ refers to the rank position of the first relevant result for the i-th test input method.

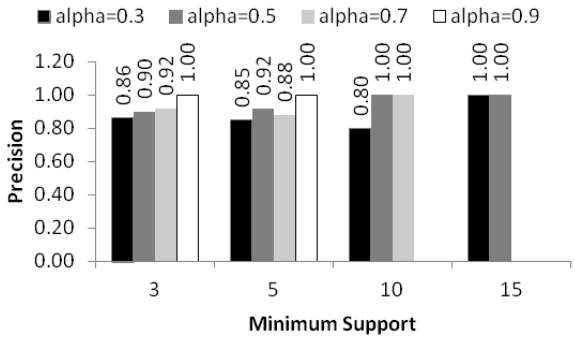
6.4.3 Automated Evaluation Results

Using the 20 test input methods we obtain, we evaluate related method recommendations from FACER using different configurations. We obtain top N recommendation sets by varying the similarity threshold α and minimum support threshold β across a range of values. Table 6.5 shows the precision (P), success rate (SR), and mean reciprocal rank (MRR) @N = $\{5, 10, 15\}$ for the recommendations obtained using different configurations of similarity threshold α=$\{0.3, 0.5, 0.7, 0.9\}$ and minimum support thresholds β =$\{3, 5, 10, 15\}$. α indicates the strength of intra-clone group similarity between methods and β indicates the minimum support a Method Clone Structure should have to be considered a source for recommending methods.

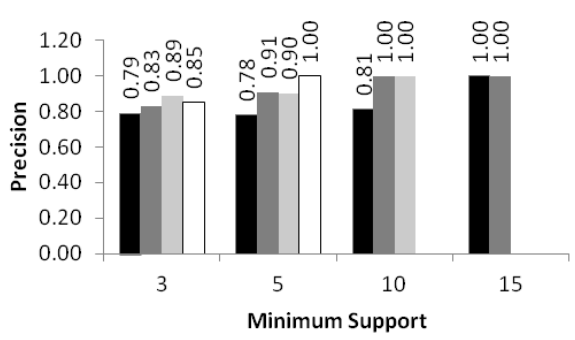
Figure 6.7 is a visual summary of the same results, showing the precision@N for the recommendations obtained using different configurations of α and β. The success rate is same across all values of N = $\{5, 10, 15\}$ and is shown in Figure 6.7d. Our objective is to determine the best

Table 6.5: Automated evaluation results using various thresholds of similarity α and minimum support β. The success rate (SR) and mean reciprocal rank (MRR) values are for all N={5,10,15}

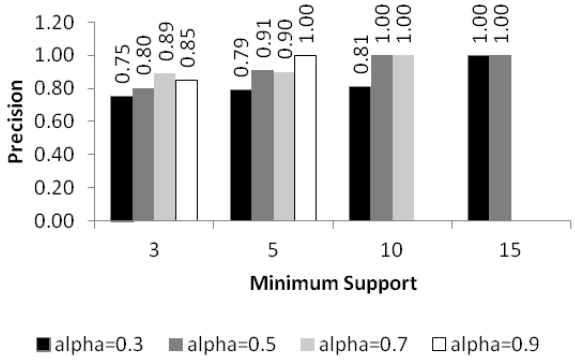
α	β	P@5	P@10	P@15	SR	MRR
0.3	3	0.86	0.79	0.75	**1.00**	0.97
	5	0.85	0.78	0.79	0.85	0.97
	10	0.80	0.81	0.81	0.45	**1.00**
	15	**1.00**	**1.00**	**1.00**	0.15	**1.00**
0.5	3	0.90	0.83	0.80	**1.00**	**1.00**
	5	0.92	0.91	0.91	0.60	**1.00**
	10	**1.00**	**1.00**	**1.00**	0.15	**1.00**
	15	**1.00**	**1.00**	**1.00**	0.15	**1.00**
0.7	3	0.92	0.89	0.89	**0.65**	**1.00**
	5	0.88	0.90	0.90	0.35	**1.00**
	10	**1.00**	**1.00**	**1.00**	0.10	**1.00**
0.9	3	**1.00**	0.85	0.85	**0.30**	**1.00**
	5	**1.00**	**1.00**	**1.00**	0.20	**1.00**



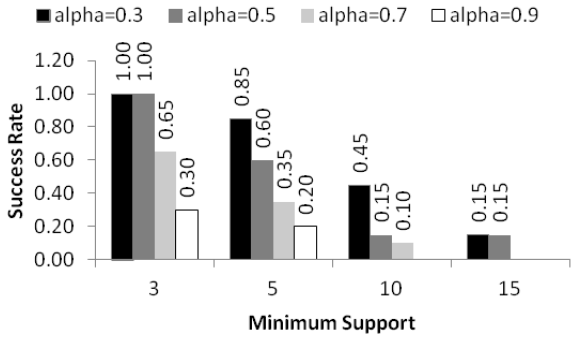
(a) Precision@5



(b) Precision@10



(c) Precision@15



(d) SuccessRate

Figure 6.7: Precision and success rate of recommendations across varying similarity threshold (alpha) and minimum support (beta)

combination of α and β that gives us a good precision without compromising success rate. From Figure 6.7a, we notice that for the same α=0.3, the precision decreases as β increases from 3 to 10, but then increases when β is further increased to 15. This is counter-intuitive but there is a reason behind the decreasing precision with increasing minimum support. In case increasing the minimum support does not yield any recommendations from the test input method itself, the recommendation algorithm switches strategy to obtain recommendations from the neighborhood of an input method. This results in a new pool of recommendations and thus a different precision value. We also observe a general trend of increasing precision as the similarity threshold α increases; however, it can have an opposite effect on success rate which decreases as α increases.

Intuitively, increasing α makes the criteria for clustering of methods into clone groups more strict and results in fewer but more precise clone groups and also fewer but more precise recommendations. This intuition is also reflected in the graphs. According to this sensitivity analysis, we choose the following configuration for our manual user evaluation: we fix N at top 5, α = 0.5, and β = 3.

6.5 Manual Evaluation of FACER's Precision

For our second evaluation, we recruit professionals and students to manually evaluate the recommended related methods and to determine the relevance of FACER's recommendations. We use the same feature queries and test methods used in our automated evaluation in Section 6.4.1. For each test method, we ask FACER to generate the top 5 related method recommendations from the optimal similarity thresholds α=0.5 and β = 3. We then ask the participants to evaluate the relevance of these recommended methods to the input query and test method considering the category of the test method's project. A recommendation against a test input method is deemed relevant by a participant if she is able to identify the functionality of the method as being relevant to the application domain of the test method.

6.5.1 Manual Evaluation Setup

We recruit 10 industry professionals and 39 Master's students to participate in the manual evaluation. The industry professionals have both Android and Java experience and the 39 students have some experience in Java and/or Android. This number already excludes the evaluations of two students who had no Android/Java experience. To identify relevant industry professionals, we use LinkedIn's search option which allows one to search for professionals based on their job titles. We search for Android developers and send direct messages to multiple professionals to invite them to participate in our evaluation. We also use our own professional/academic contacts to recruit additional professional participants. The students were required to participate in the evaluation as a graded instrument for the Software Development: Tools and Processes graduate course at the

Table 6.6: Participant demographics for the manual evaluation of FACER's related methods recommendation (Precision)

Type of participant	Range of experience	No. of participants
Professional developers	1-2 years	6
	3-4 years	4
Students with Android experience	<1 year	34
	1-2 years	2
Students with Java experience	<1 year	26
	1-2 years	10
	3-4 years	3

Lahore University of Management and Sciences.

The detailed demographics of our subjects for manual evaluation are shown in Table 6.6. The students have varying levels of experience with Android and Java. From the table, we see that all 39 students have Java experience and 36 of them also have Android experience.

We ask participants to imagine themselves as being in the process of developing an Android application of a certain category and that they have just written a method to implement a certain feature of the application. That method is the test method we have selected for each query. Based on that method, FACER provides top 5 related method recommendations which they need to evaluate and see whether they are relevant for the application they are developing. We assign two recommendation scenarios to each student. We assign at least 4 recommendation scenarios to each of the 10 professional developers. This guarantees that we have at least three evaluations for each scenario.

We create a set of 20 files containing the recommendation scenarios to be evaluated. Each file consists of an evaluation ID (method ID), the application category name, the code for the test method with its feature description, followed by the code for top 5 related methods retrieved by FACER. In our evaluation instructions, we ask the subjects to understand the features being implemented by the recommended methods by looking at the name of the method, its comments

(if any), its API calls, other variables, and the overall semantics. We also ask the subjects to rate the relevance of the recommended methods on a three-point Likert scale with an integer range from zero to two, where two is *relevant*, one is *maybe relevant* and zero means *irrelevant*. We also require the students to write a feature description for each method that they evaluate to make sure that they understand the functionality before rating it. The students input their evaluation in a Google form. Five professionals provide their evaluations through a Google form and the other five provide their evaluations through e-mail.

Measuring precision We calculate the relevance of a recommended method *m* as the median of the relevance ratings of the participants who evaluate *m*. We obtain separate relevance ratings for student and professional ratings and obtain a median relevance over all ratings for each query.

We calculate the precision of recommendations of a query as a fraction of relevant methods over the total methods recommended. For each query, we consider a recommendation as relevant only if its median relevance is greater than or equal to 1. We calculate FACER's overall precision as the mean precision of all queries *Q* as shown in the Equation 6.4:

$$Precision = \frac{\sum_{q=1}^{|Q|} |relevantMethods(q)/recMethods(q)|}{|Q|} \tag{6.4}$$

6.5.2 Manual Evaluation Results

We first obtain separate precision values for student and professional ratings for 14 of the 20 test queries. The remaining six test queries are evaluated by professionals only and two are evaluated by students only. We then perform a paired samples Wilcoxon test [178] on the ratings of the 14 queries to test our null hypothesis which asserts that the medians of the precision values for students and professionals are identical. A p-value of 0.33 means that we cannot reject the null hypothesis, and accordingly there is no statistically significant difference between both groups. We therefore combine the relevance ratings of the 20 test queries of all student and professional participants to obtain the median value for a recommendation. We calculate the number of relevant recommendations for a query considering those median values that are greater than or equal

Table 6.7: Manual Evaluation of FACER's related method recommendations (Relevant = no. of recommendations that are relevant, Recommended = total no. of system generated recommendations, Precision = Relevant/Recommended)

Method ID	Recommended	Relevant	Precision
33	5	5	1.0
80	5	5	1.0
423	5	5	1.0
1066	5	5	1.0
1161	5	5	1.0
2250	5	3	0.6
2616	5	4	0.8
2642	5	4	0.8
2971	5	5	1.0
3017	5	0	0.0
14435	5	5	1.0
14490	5	5	1.0
15214	5	4	0.8
22968	5	4	0.8
24068	5	4	0.8
28664	5	3	0.6
29298	2	2	1.0
29947	5	5	1.0
31838	5	1	0.2
33549	2	1	0.5
Average			**0.795**

Table 6.8: Demographics of the professional participants who participated in our user survey

Type of participant	Range of experience	No. of participants
Professional Android developers	1-2 years	3
	3-4 years	8
	5-6 years	3
	7-10 years	1
Professional Java developers	1-2 years	1
	3-4 years	2
	7-10 years	2

to one. Table 6.7 shows the precision of related method recommendations for each query. Recall that we use FACER's top 5 recommendations. For 18 of the scenarios, FACER returns 5 related recommendations, while for the two remaining scenarios, FACER produces only 2 related recommendations. Overall, the evaluation contains 94 related method recommendations over the 20 scenarios. We obtain an average precision of 79.5% over the 20 recommendation scenarios.

RQ 6.2: Based on a user evaluation involving 10 professional developers and 39 students, we find that FACER's precision for recommending related method recommendations is approximately 80%.

6.6 Developers Code Search and Reuse Practices

In RQ 6.3 we want to understand professional developers' code search and reuse practices to make sure that FACER can serve real needs. To investigate these points, we survey 20 professional developers including 15 Android developers and 5 Java developers. The detailed demographics of our professional survey participants are shown in Table 6.8.

As part of our survey, we first capture the developer's profile which includes the number of years of experience, and also the types of applications they have previously developed. We then ask them about their current code search and reuse practices.

6.6.1 Survey Design

We focus on understanding developer practices while searching for (related) features. We ask about those practices before they review specific recommendations from FACER to avoid biasing their opinion in any way. Specifically, we capture the frequency of performing the following 7 activities on a scale of 1 - 5 (where 1=never, 2= rarely, 3= sometimes, 4= often, 5= always).

1. Whenever I need to implement a new feature for the application I am developing, I start by searching for code examples.

2. When I search for a code example to help me implement a feature, I find what I am looking for in the results of the first search query.

3. If I get the desired code after a successful online search, I need to search again for related functionality to proceed with development.

4. While implementing the features of my application, I need to perform repeated online searches to find code for various features.

5. I reuse code for various functionalities from my previously developed applications.

6. While writing code for some feature, I recall that I have written similar code in the past and want to search for it again.

7. When writing a new application, I find myself reusing multiple methods which implement different functionality from a single application I have developed before.

6.6.2 Survey Results

We measure the frequency of code search or reuse activities performed by professional developers and obtain their feedback on a scale of 1 to 5 (where 1=never, 2= rarely, 3= sometimes, 4= often, 5= always). The results are shown in Figure 6.8. We plot these results in a series of 100% stacked bar charts which show the percentage of subjects responding to a certain value between 1 to 5

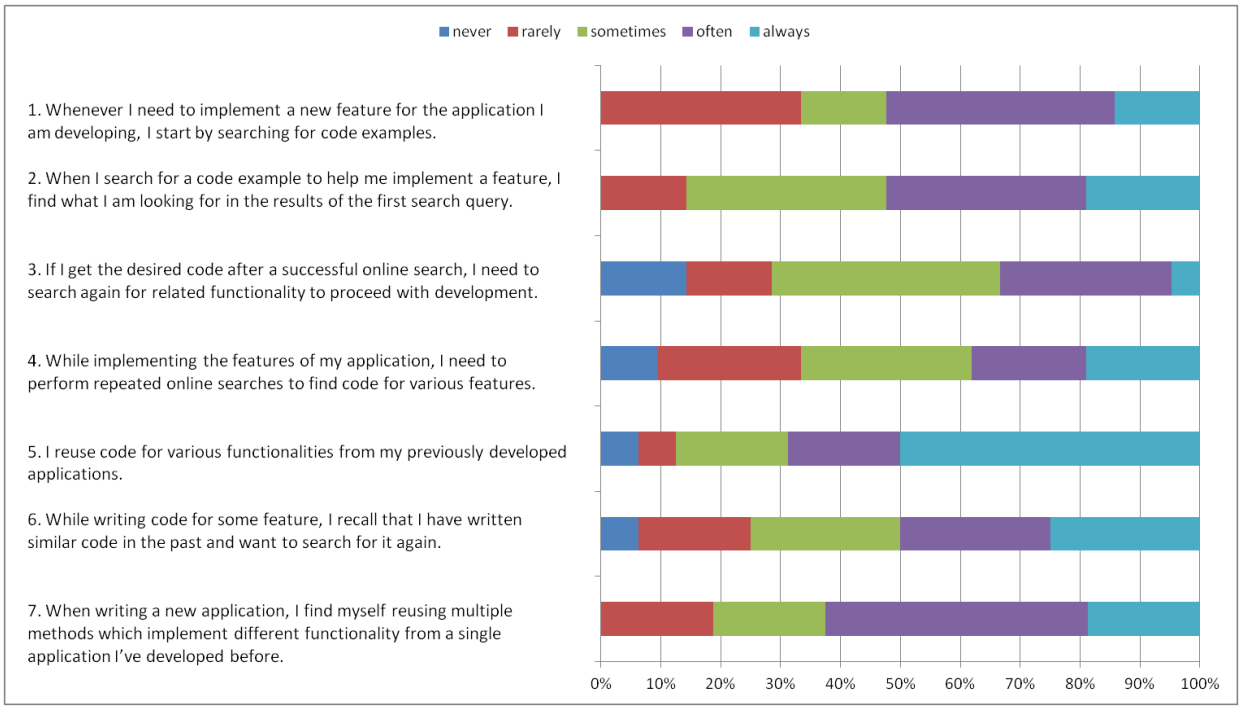


Figure 6.8: Analysing developer's code search and reuse practices

indicated by 5 different colors respectively. When reporting results to indicate that a developer does a given activity, we consider ratings 3 (sometimes) to 5 (always).

The first bar in Figure 6.8 indicates that 65% of the developers start implementing a new feature by first searching for code examples with 50% doing this at least often. This observation is in line with previous studies on developer's need to search for code examples. The second bar plot indicates that the first search attempt is successful for 85% of the developers. They do not need to reformulate their query again. Next, we investigate whether the developers need to search again for related functionality after getting code for their initial search. We observe that 70% of the developers need to find functionality related to code that is obtained from their initial search to proceed with development. This strengthens our motivation to provide developers with code for related features. From the fourth bar plot, we observe that 65% of the developers face the problem of performing repeated searches for finding code for various features of an application they are developing. This also indicates the need for a code recommender that assists developers in providing related code for their application being developed. We also investigate whether the developers tend

to search and reuse the code they already wrote in the past. Our findings from the fifth and sixth bar plot indicate that 85% of the developers reuse their own code from previously developed applications and 75% need to write code for features they have previously implemented and thus search for their already written code. This indicates that applications share some common features which is coherent with our approach of mining repeatedly co-occurring features across applications. The last bar plot in Figure 6.8 indicates that 80% of the developers reuse multiple functionality from a single application that they previously developed. Thus, building the FACER repository on an organization's code base can discover such repeatedly co-occurring functionality and using FACER can allow the developer to receive related code recommendations without explicitly searching for them.

RQ 6.3: The survey results show that 70% of the developers face the need to search for related features which supports the motivation of our work.

6.7 Usefulness and Usability Evaluation Survey

The research questions RQ 6.1 and RQ 6.2 allow us to evaluate FACER's effectiveness in terms of its clustering of similar functionality and its precision for related method recommendations, respectively. In RQ 6.4, we want to assess the usability and usefulness of our FACER tool and its recommendations. To investigate these points, we survey 20 professional developers including 15 Android developers and 5 Java developers. The detailed demographics of our professional survey participants are shown in Table 6.8.

As part of our survey, we ask the developers to review recommendations from FACER. At this point, our survey breaks into three parts based on whether the developer has Android or Java expertise and whether they opt for a short review of recommendations or a longer evaluation of recommendations. The longer evaluation includes evaluating the 20 recommendation scenarios discussed in Section 6.4.1 which we distribute across the evaluators assigning four scenarios per evaluator. Eight Android developers opt for this longer evaluation and the evaluation of five of these developers is included in the manual evaluation results we reported in Section 6.3.1, whereas

Table 6.9: Professional developers involved in reviewing recommendation scenarios for user survey

Recommendation scenarios	Dataset	No. of professionals	Professional expertise	Manual evaluation participants?
Assigned from 20 scenarios	120 Android apps	5	Android	Yes
Assigned from 20 scenarios	120 Android apps	3	Android	No
Motivating example scenario	30 Android apps	7	Android	No
Java IMS scenarios	53 Java IMS	5	Java	No

for the remaining three developers, the evaluation results were incomplete and were not included in the manual evaluation. Table 6.9 summarizes the information of the number of professional developers reviewing recommendations from various recommendation scenarios. Overall, eight Android developers are part of the longer evaluation, seven Android developers opt for a shorter evaluation where they review recommendations from our motivating example discussed in Chapter 1, Section 1.3, and five Java developers review Java recommendation scenarios. We discuss the setup for reviewing the recommendation scenario from the motivating example and the recommendation scenarios generated from Java information management systems (IMS) in Section 6.7.1. Having developers review real recommendations from FACER allows us to ask them about their perceptions of its usefulness.

We get feedback from the professional developers on the tool's interface, its potential to speed up development, their interest in future adoption of this tool, their perception of reduced time-to-search, and an overall satisfaction with the quality of recommendations. We then ask them to give their comments on our approach. Finally, we ask them to give ratings on the perceived overall usefulness and usability of FACER. Note that we also ask the 39 students who perform the manual evaluation of the 20 Android scenarios from Section 6.5 to provide an overall rating on the usefulness of FACER's recommendations. We also ask these students to provide comments on our

recommendation approach.

6.7.1 Recommendation Scenarios

In this section of the survey, we present participants with code recommendations to give them a demonstration of the capabilities of FACER. Eight Android developers opting for a longer evaluation evaluate the recommendations from the 20 scenarios discussed previously.

We present recommendations from the motivating example (discussed in Chapter 1, Section 1.3) to seven of the Android developers. The recommended methods evaluated by the developers include those for cropping an image, showing it in ImageView, as well as for resizing image and getting the uniform resource identifier (URI) of a captured image. After understanding the scenario and reviewing the recommendations, the developers provide responses against two questions. The first question asks whether the recommended methods are related to the given method (select image) and system (photo sharing application) being developed. The second question asks whether the recommended methods are useful and can be reused in the context of their method and system being developed.

We present recommendations from Java projects implementing information management systems to the developers with Java experience. We create a separate FACER repository of Java projects related to information management systems. Our choice of selecting the information management systems domain is based on our premise that most of the professional Java developers would be familiar with information management systems. Thus, they would easily be able to understand recommendation scenarios and review recommendations for information management systems. This was also evident from the profile of all our survey participants who are Java professionals (we explicitly ask the Java developers about their experience with developing information management systems to confirm that our assumption is true). We collect Java projects from GitHub using the search query "Java information management systems" and get 187 results which we sort by star ratings and select the top 53 projects. The remaining projects had no stars. We create five recommendation scenarios from the Method Clone Structures detected by FACER on these projects. These recommendation scenarios are available in our online artifact page [166].

6.7.2 Feedback on FACER Tool's Interface and its Usefulness

We use Figures 5.6 and 5.7 to survey participants how the interface of FACER looks like and present them with five statements to get feedback on the tools interface, its capacity to speed up development, their interest in future adoption of this tool, their perception of reduced time-to-search, and an overall satisfaction with the quality of recommendations. We capture all responses to these five statements on a five-point Likert scale, which allows us to measure the strength of their agreement.

1. The organization of information on the tool screens is clear.
2. I perceive that this tool can speed up my development.
3. I would be interested in using this tool.
4. This tool can reduce the need to perform repeated online searches to find code for various features of an applications.
5. Based on my evaluation of the various recommendation scenarios, on average, the recommender was successfully able to predict related functionality or set of functionalities.

We also ask the developers to provide open-ended feedback on the advantages or disadvantages of our approach and any other comments they might have.

6.7.3 Usability and Usefulness Ratings

We ask the professional participants to rate the perceived usability of the FACER tool for their development activities and the usefulness of FACER's recommendations on a scale of 1 to 5 where 1 indicates a low rating and 5 indicates a high rating. We ask the developers to provide ratings as follows:

1. Rate the usability of this recommendation tool for your development activities on a scale of 1-5.

2. Rate the usefulness of these recommendations based on their ability to provide relevant functionality for your application on a scale of 1-5.

For student participants, we only ask for a rating on the usefulness of FACER's recommendations and we ask them to provide any comments as feedback.

6.7.4 User Survey Results

Analyzing developer's feedback on FACER's recommendations for motivating example We analyze the feedback of seven professional Android developers who review the recommendation scenario from the motivating example we discussed in Chapter 1, Section 1.3. In response to whether the recommended methods are related to the given method (select image) and system being developed (photo sharing application), one developer strongly agreed, while four agreed and two were neutral. In response to whether the recommended methods are useful and can be reused in the context of their method and system being developed, six developers agreed and one developer was neutral. The high level of agreement shows that FACER can provide relevant recommendations that can be reused for the development of our motivation example of the photo sharing application.

Developer feedback on FACER tool's interface and its usefulness Figure 6.9 summarizes developer's feedback on FACER. It shows a series of bar plots which capture the percentage of developers agreeing to some statements describing the FACER tool. The levels of agreement are on a scale from 1 to 5 with 1 indicating a strong disagreement and 5 indicating a strong agreement. We observe that 75% of the developers agree (on level 4 or 5) that the organization of information on the tool screens is clear. 65% of the developers agree (on level 4 or 5) that the tool can speed up their development. 75% agree (on level 4 or 5) that they would be interested in using the tool. It is very encouraging to see that 75% of the developers perceive (on level 4 or 5) that FACER can reduce the need to perform repeated online searches to find various features of an application. 50% of the developers agree (on level 4 or 5) that overall FACER can successfully recommend related functionality.

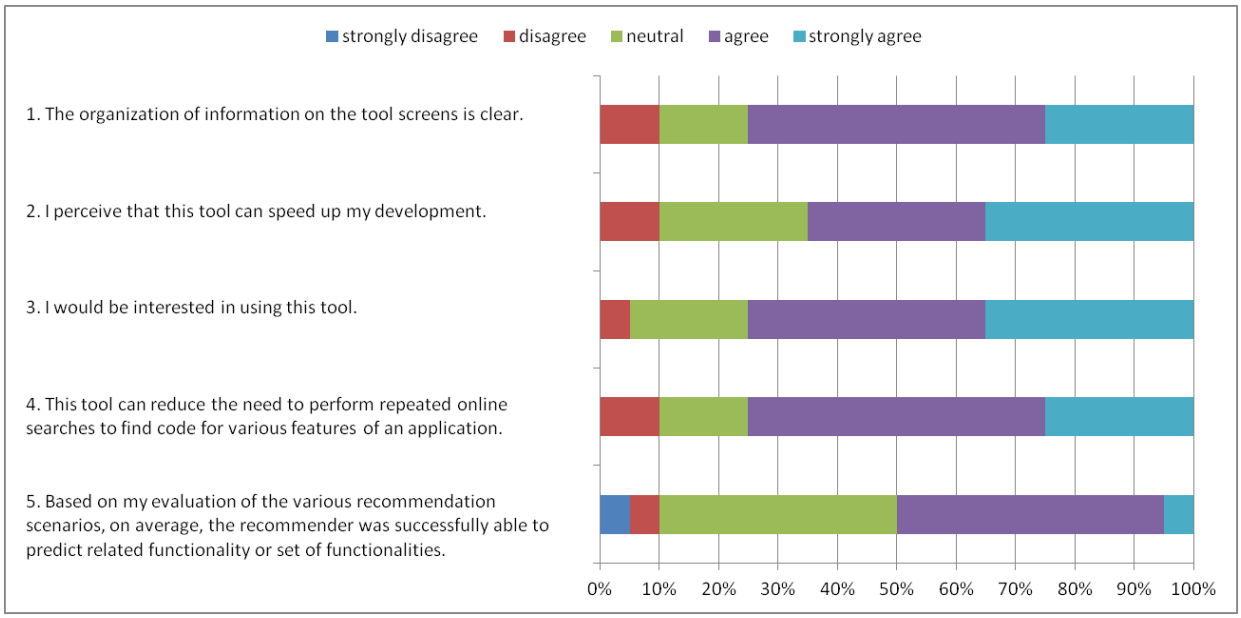


Figure 6.9: Analysing developer's feedback on FACER

Developers' comments on FACER's recommendations We received positive comments from the professional developers regarding FACER's IDE-integrated interface which eliminates the need to leave the development IDE to search for code. One professional also claims to never have seen such a recommendation approach for Android development. The following are quotes of some of the feedback we got:

- *"The best thing about this approach is that everything is on a single interface, which would make the process of searching methods quite simple and efficient. This would help generate more relevant ideas to the developers, which would improve the overall functionality of application."*
- *"This work is really good and appreciable. No such thing is available so far in Android Development. By using this user/Developer can easily search for related code/solution and implement it, using this plugin, without moving outside Eclipse/AndroidStudio"*
- *"Developer can search the related code in the IDE window rather then going to the web."*

The professional developers also commented on the ability of FACER to save time as shown

below:

- *"It can speed up the creating of basic structure of any application feature. And after that a person can customize that according to his need."*
- *"That seems quite reasonable and helpful!... This tool will save a lot of time."*
- *"Time saving. Less time will be spent on checking each link shown in Google search."*

There were some general comments of the professional subjects indicating a positive impact of the FACER tool for helping developers. One subject was of the view that FACER can not totally eliminate online searches for required code but can act as a supplementary tool to speed up development.

- *"I am really impressed by the overall idea of your tool. It will definitely help developers in the long run"*
- *"As a beginner in the industry, I used to do some online searches and check some snippets and understand the underlying objective and start implementing code as per my requirement. your FACER is good but what I'm thinking is using your tool, I don't think every suggested method solves my problem and maybe checking all suggestions and re-write the method save some time compared to my approach. so we need to both and can't replace one another, and I'd love to use it in future."*

The professional developers also made some suggestions on improving FACER's recommendations. One participant expresses the need for more relevant recommendations. To address this concern, we can look into licensing, popularity, and code quality factors discussed in existing literature [179, 180] that can improve the relevance of recommendations. Another participant suggests making predictions based on business use case, which we think can be implemented by filtering and/or prioritizing the recommendations that directly relate to the business logic of the application. Another comment refers to the need for the source of code recommendations to be always up to date, so that new solutions are available. We plan to address this need by having a continuous repository update mechanism in place.

We also received comments suggesting UI enhancements like the recommendations to appear on the right side. FACER's Eclipse plugin allows the developer to move the panel containing code recommendation to their desired position, so this is not difficult to achieve. Furthermore, we also received suggestions to show details of the parent class of the recommended methods. In future, we can integrate the ability to browse the complete class for a recommended method. One participant mentioned the need for providing alternate solutions against recommended functionalities. Sometimes developers are looking for optimal implementations of some functionality in terms of conciseness, exception handling and other quality factors. The fact that our recommended methods come from clone groups with multiple implementations of the same functionality provides a solution for the subject's requirement. In future, we can learn to distinguish between different methods of a clone group using quality parameters to provide alternate solutions for a recommendation.

Students' comments on FACER's recommendations We also received some positive comments about FACER's recommendations from the subjects who are Master's students. They indicated the usefulness of recommendations in writing code faster and thus saving time. One student commented that FACER's recommendations having concise implementations added to his knowledge of writing improved code. Some of their comments are as follows:

- *"Recommendations are quite good and can aid a developer to code [faster] given he knows where to head"*
- *"These recommendations contained concise code. In my past experience, I remember doing things [with a comparably] difficult approach"*
- *"I found 4 useful method[s] out of 5 so I like these recommendations"*
- *"[Avoids the need for] writing the whole code or searching for [a] new module... quite helpful [recommendations].... time saving..."*

The students further expressed their opinion on using FACER for future personal projects and stated its benefit of enhancing a developer's capabilities and reusability of code.

- *"Thanks for this, I hope I can use it for my future projects if [need] be"*

- *"I think it is good idea to build this system which [enhances] the capability of [a] developer"*
- *"..in OOP driven development environments, I can see this system having a lot of value in enhancing re-usability of code... can act as code auto-complete ..."*

A few student participants had some concerns regarding the recommendations. One participant pointed out that some of FACER's results are inaccurate. Another participant pointed to the fact that some method recommendations are very generic. These comments are as follows:

- *"[some] prediction[s] are very close and some ...are very [inaccurate]"*
- *"the method you have [recommended] is very generic not ... serving a specific purpose"*

The reason for inaccurate and generic recommendations may be due to the recommended methods containing API calls that do not necessarily translate to a core feature for an application's product domain, instead they may be implementing Android framework-specific code which glues together the core features of an application. In future, we want to be able to distinguish between domain-relevant features and other more generic framework-specific helper features.

Analyzing developer's ratings on the usefulness and usability of FACER As discussed in Section 6.7.3, we ask the professional subjects to rate the usability and usefulness of our approach on a scale of 1 to 5 where 1 indicates a low rating and 5 indicates a high rating. Figure 6.10 shows that 90% of the professional developers give moderate to high ratings (ranging from 3 to 5 on 5-point Likert scale) on the usability of the tool for their development activities. We also find that 95% of the professionals give a moderate to high rating (ranging from 3 to 5) on the usefulness of FACER's recommendations to provide relevant functionality for their application; 70% give a high usefulness rating (ranging from 4 to 5). This indicates that professional developers find the tool and its recommendations helpful for their development tasks.

Figure 6.11 shows ratings from the students on the usefulness of our approach. It indicates that 85% of the students give a moderate to high rating (ranging from 3 to 5) on the usefulness of FACER's recommendations to provide help in their development; 68% give a high usefulness

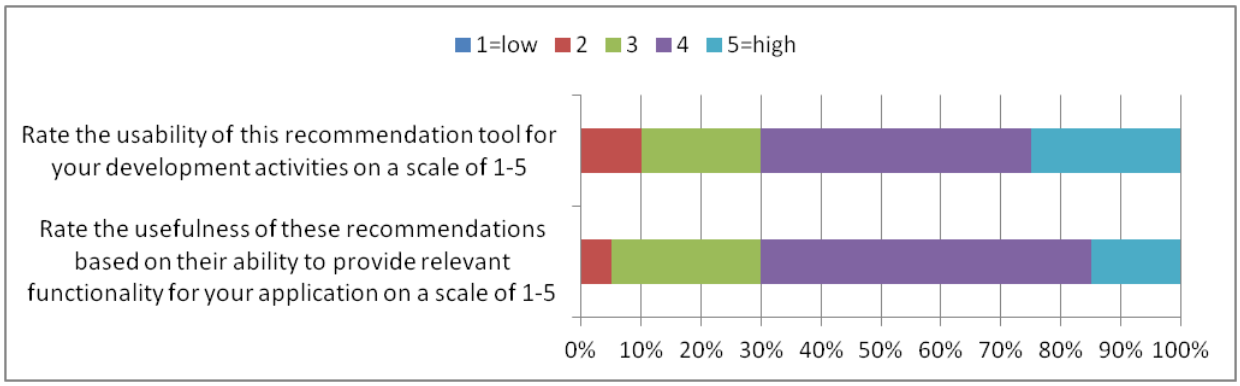


Figure 6.10: Professional developer's ratings on the usefulness and usability of FACER

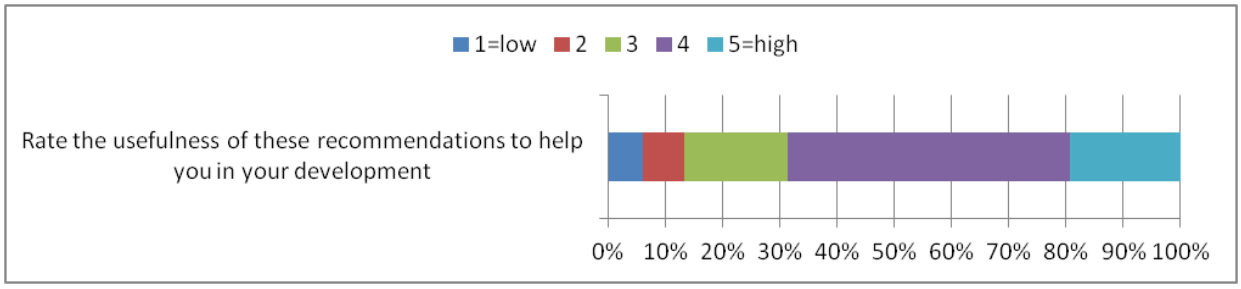


Figure 6.11: Student developer's ratings on the usefulness of FACER

rating (ranging from 4 to 5). This indicates that students, like professionals, find the tools recommendations helpful for their development tasks.

Based on the above feedback, we can conclude that, overall, participants expressed the desire to use the system for their needs and feel that it could help save time by avoiding the need to search or write code.

RQ 6.4: The survey results show that 90% of the professional developers give a moderate to high rating on the usability of FACER tool for their development activities. Furthermore, 95% of the professional developers and 85% of the student developers find the related method recommendations from FACER useful.

6.8 Threats to Validity

6.8.1 Internal Validity

We rely on third-party tools in FACER's implementation such as the JDT parser [150] and clustering algorithms. Any inaccuracies in these tools will affect our results. However, most of these tools have been widely used and tested. That said, we notice that there are certain types of Android framework-specific calls that have no object reference; such calls are not detected by the JDT parser [150] as method invocations and are, therefore, not parsed as API calls. This can result in some functionalities being ignored while clustering methods into clone groups. Furthermore, in the calculation of the number of statements of methods containing API calls, we proxied statements using line numbers which has resulted in density percentage values higher than 100% for some methods. While this can have an effect on the clustering of methods into clone groups, our manual analysis of the validity of clone groups formed as a result of the clustering algorithm gives us reassurance in the results.

For the evaluation of inter-clone group dissimilarity, the human evaluators manually validate only the clone group descriptions of pairs with a TF-IDF similarity threshold greater than 0.5. By relying on TF-IDF, we may have incorrectly automatically marked some clone groups with a TF-IDF score less than 0.5 as dissimilar. However, we decided to use this technique, because it is impossible to ask an external validator to manually validate thousands of combinations.

6.8.2 Construct Validity

The ground truth we use in our automated evaluation is a proxy for a subjective decision that should be made by the developer. There could be a method from another project that is actually relevant but that we consider as a false positive. However, our automated evaluation only helps us in determining the appropriate thresholds. We engage professional developers and students for a manual evaluation to determine FACER's precision in practice.

We do not currently report the number of times FACER is able to provide related method recommendations against all methods retrieved in Stage 1 of FACER. We only consider the success

rate for those recommendation scenarios where a method selected in Stage 1 provides a non-empty set of related recommendations, which is why the success rates are high. Currently, the FACER repository is built on 120 projects only. Once we increase the size of the repository, we can calculate the overall success rate of providing related method recommendations.

Our manual validation of the recommendations as well as user survey setup does not represent a real development scenario that a developer would go through to really use FACER. Our setup, while contrived, reduces the cognitive load and expectations of actual development from participants and allows them to focus on a well-defined task. Since we provide the description of the test method as well as the domain of the application, participants can easily determine whether a recommended method is indeed related or not. Such an evaluation gives us a fair indication of FACER's precision by human subjects. In the future, we plan to conduct a long-term user study where developers use FACER for real tasks and we evaluate how often they use the recommended related features and what their perception of the tool is.

6.8.3 External Validity

The limited number of professional developers involved in evaluating FACER is a threat to external validity. Only one professional developer performed the evaluation of 126 clone groups. We reduce the threat to validity of evaluation by ensuring that the professional evaluator is experienced and also by including myself, and three faculty members having experience in the area of software engineering, in the evaluation.

We reported the precision of FACER based on our dataset that consists only of Android applications. Applications from our datasets were chosen to have some common domains such that we can indeed find meaningful co-occurring features. While we cannot generalize beyond our analyzed applications, we do not see any conceptual reasons why FACER cannot be applied to more projects and domains with similar results. While the number of queries we use is limited, we wanted to make sure we evaluate with meaningful queries that represent actual functionality. Since FACER's Stage 2 requires that the user has selected a method from FACER's repository in Stage 1, this limits our ability to use external queries from Stack Overflow, for example, since they

may represent features not in FACER's repository.

6.9 Chapter Summary

To evaluate FACER, we extracted data from 120 open-source GitHub projects from four different domains. We first performed a manual validation of the detected method clusters to ensure that clustering methods based on API usages results in meaningful clusters with functionally similar methods. Our results show that 91% of the analyzed method clusters are valid. We then performed an automatic evaluation with different FACER settings to determine the best configuration for related method recommendations. Once we determined the best configuration, we performed a user evaluation with 10 professional and 39 experienced student participants to determine whether the related methods recommended by FACER are indeed relevant to the original method and feature. Our results show that FACER's related method recommendations are, on average, 80% precise. We also received positive feedback about FACER's functionality, which encourages us to further improve FACER and release it to developers soon. The results replication package for FACER, along with all the data from our evaluations is available on our online artifact page [166].

While FACER is shown to perform well against user queries, the capabilities of FACER in evolving development contexts need to be investigated. In the next chapter, we focus on evaluating the need for context-awareness in FACER and designing a context-aware approach for opportunistic code reuse.

Chapter 7

CA-FACER: Context Aware FACER

7.1 Introduction

Context-aware recommender systems (CARS) are designed for various real-life application domains. The purpose of CARS is to generate more relevant recommendations by adapting them to a specific contextual situation of a user [91]. A context encapsulates the characteristics of a user's task or the goals of a user [38]. Various areas of software engineering make use of context to improve a developer's experience. For code recommendation, the context of a developer can be leveraged to provide better personalized code recommendations. Existing context-aware code recommendation systems have been shown to support developers in code completion [16–29] and code reuse [26, 30–37]. Code recommendation systems can also facilitate *opportunistic reuse* [52] by providing code that represents features a developer may want to implement next [51, 57, 181]. Opportunistic reuse enables rapid application development without the need to conduct multiple searches and thus enhances developer productivity and saves time [3, 52, 57, 58]. However, there are currently no existing context-aware systems that can provide code recommendations of multiple related features for opportunistic reuse on-the-go.

With the passage of time, the activity of a developer on their project can increase the amount of code written by the developer, which results in an evolving development context. While FACER is shown to perform well against user queries, the capabilities of FACER in evolving development

contexts need to be investigated. For this dissertation, we focus on evaluating the need for context-awareness in FACER and designing a context-aware approach for opportunistic code reuse.

For the software development domain in general, possible sources of deriving a context include: (i) static artifacts, (ii) historical information (iii) dynamic execution (iv) individual developer activity and (v) team and organization activity [182]. Currently, Integrated Development Environments (IDEs) remain oriented around the static structure (i.e. files, classes etc.) of software. Although tools using historical information have been proposed, few are actually available to practicing developers [182]. To better support software development, there is a need to move beyond the limited notion of context and make use of other information available to provide working tools for developers.

We want to investigate whether context-awareness can improve the delivery of relevant method recommendations for opportunistic reuse. Our objectives are (i) to establish the need for context-awareness for code recommender tools, (ii) to create a more robust code recommender that is context-aware, (iii) to assess the effect of context-awareness on the precision of recommendations, (iv) to compare various context-aware models to propose a context-aware code recommendation strategy that improves the baseline FACER, and (v) to assess the effect of context size on the precision of recommendations.

7.2 Problem Formulation

Consider a user *u* working on a project P^u with the aim of implementing a set of features $F = \{f_1, ..., f_n\}$ for the project. Let user *u* input a query to a code search system S which returns a set of matching methods $M = \{m_1, ..., m_n\}$ from a code repository R. The user then selects and reuses a method m^r from this set. The methods that are part of the project P^u form the user's active development context C.

7.2.1 Defining Context

The active development context C is a set of methods $M^c = \{m_1^c, ..., m_n^c\}$ where M^c consists of the user-defined methods $M^u = \{m_1^u, ..., m_n^u\}$ and the reused methods $M^r = \{m_1^r, ..., m_n^r\}$ from the repository R as shown in Equation 7.1.

$$C = M^c = M^u \cup M^r \tag{7.1}$$

Given a user's active development context C and a code repository R, the aim of our system is to recommend a set of methods $M^{rec} = \{m_1^{rec}, .., m_n^{rec}\}$ from R which are likely to be an implementation for the set of features F such that $m_x^{rec} \mapsto f_x$ where $\mapsto$ denotes that the implementation of some method m_x^{rec} maps to some desired feature $f_x \in F$.

7.3 Proposed Approach for CA-FACER

7.3.1 Sources of Context

Since we are making context-sensitive recommendations of related methods against a developer's selected method, we introduce context-awareness to Stage 2 of FACER. Figure 7.1 shows three kinds of contextual data sources in shaded boxes. The details of each context source and contextual data are as follows:

1. *Active Development Profile*: We propose extracting contextual data related to developers' activity from the code they write. We identify this type of context as methods containing API usages which implement features of an active project under development.

2. *Method/feature Reuse History*: We propose capturing developers' feature reuse history H and using it as contextual data. The reuse history is composed of the feature class identifiers of the methods that a developer selects from FACER's recommendations during a project's development. A reused method that undergoes modification by a developer remains a part of the reuse history.

3. *Organization Development Activity*: We propose mining API usage-based method clone structures across an organization's source code to serve as organizational contextual data. The code recommendations that are generated using an organization's code repository can be more specific to a developer's requirements.

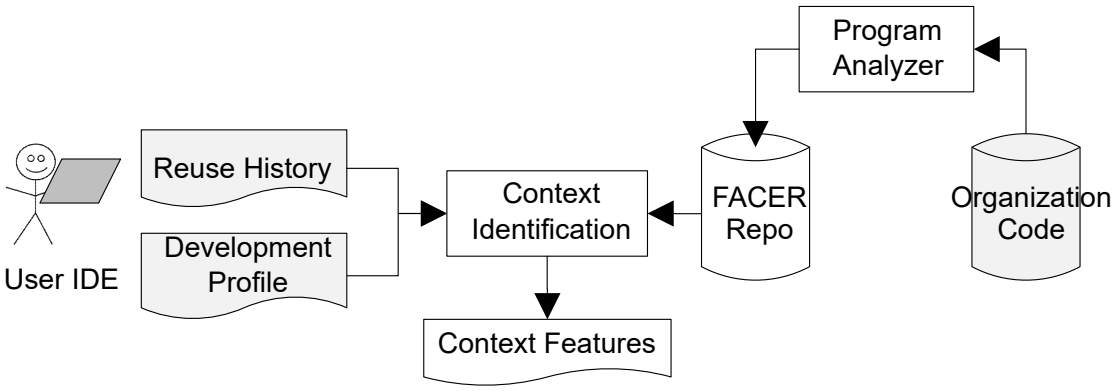


Figure 7.1: Sources of deriving context

7.3.2 Context-aware FACER Architecture

The standard paradigms of incorporating context in a recommender system are contextual pre-filtering, modeling and post-filtering [91]. We experiment with different context-aware paradigms for FACER to get the best performing context-aware approach. Thus, we incorporate context in FACER using the following three approaches.

1. *CA-FACER Post-filtering Technique:* We perform a filtering of the recommendations obtained from FACER by removing feature recommendations that are already present in the developer's context.

2. *CA-FACER Pre-filtering Technique:* This technique is used as a text-based feature recommendation technique in existing literature [9]. We replace text with feature class identifiers as the unit of similarity calculation to get contextually similar projects. The top K projects are selected and then Schafer's technique [183] is used to make recommendations.

3. *CA-FACER Hybrid Technique:* This is our proposed technique which aggregates recommendations from the baseline FACER and pre-filtering techniques. It also includes a contextual post-filtering step to remove recommendations that already exist in a user's context.

The architecture of CA-FACER variants is shown in Figures 7.2a, 7.2b and 7.2c. To implement contextual post-filtering, we filter the recommendations from FACER based on context as shown in Figure 7.2a. To implement contextual pre-filetring, we pre-filter the projects from the FACER repository based on contextual data and provide recommendations from a contextualized subset of the FACER repository as shown in Figure 7.2b. Finally, we propose an architecture for a hybrid context-aware approach CA-FACER to make context-sensitive recommendations as shown in Figure 7.2c.

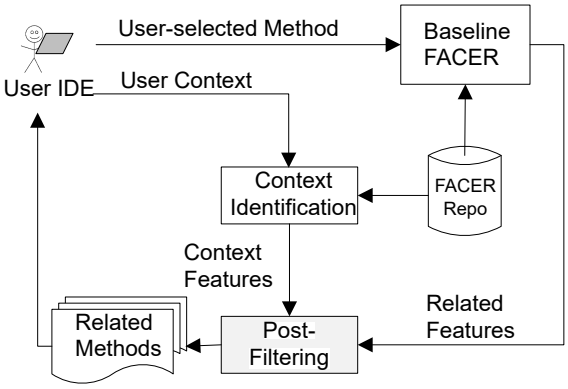
7.3.3 Context-aware FACER Approach

User Context Identification

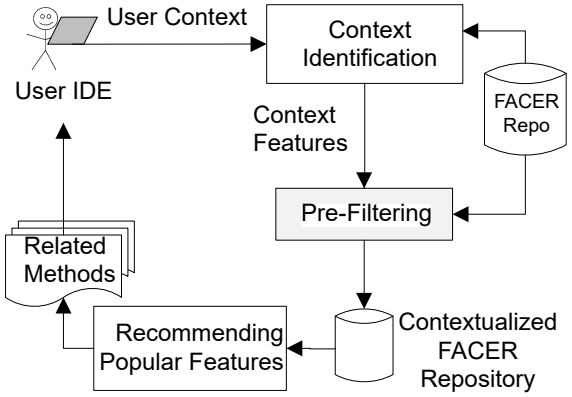
To take the context of a user into account, we consider the features already present in a user's project in terms of a set of methods $M^u = \{m_1^u, .., m_n^u\}$ and those that were reused $M^r = \{m_1^r, .., m_n^r\}$ for the current project under development. Features are mined as method clone classes in the repository R. While the method clone class identifiers for the reused methods are known, we need to assign method clone class identifiers to the set of methods M^u.

We can determine a mapping of the user methods M^u to a set of clone classes $MCC^u = \{\zeta_1^u, ..., \zeta_n^u\}$ mined as features in R such that $m_x^u \mapsto \zeta_x^u$, where $\mapsto$ denotes that a user method m_x^u maps to a method clone class $\zeta_x^u \in MCC^u$. This can be accomplished by computing the similarity between each method in M^u with the representative methods m^ζ of each clone class ζ mined in R. A user method can be assigned the method clone class identifier of the representative method to which it is most similar. We can compute the Jaccard similarity between a method written by the user m^u and each clone class member representative m^ζ from R as shown in Equation 7.2:

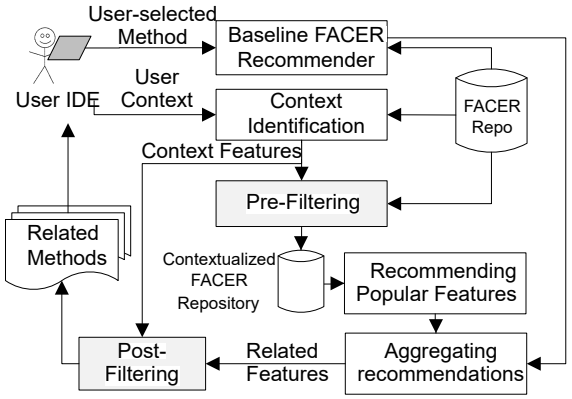
$$sim(m^u, m^\zeta) = \frac{|\overrightarrow{a^u} \cap \overrightarrow{a^\zeta}|}{|\overrightarrow{a^u} \cup \overrightarrow{a^\zeta}|} \tag{7.2}$$



(a) Contextual Post-filtering



(b) Contextual Pre-filtering



(c) Hybrid Context Aware FACER

Figure 7.2: CA-FACER Architecture Variants

where $\overrightarrow{a^u}$ is a vector of API calls of user method m^u and $\overrightarrow{a^\zeta}$ is a vector of API calls of representative method of clone class ζ. The set of methods reused by the user in their development context belong to method clone classes MCC^r. Hence, by combining method clone classes of a user's implemented methods and reused methods we can form the context C of the user as shown in Equation 7.3.

$$C = \quad MCC^c = \quad MCC^u \cup MCC^r \tag{7.3}$$

where MCC^u corresponds to a set of features F^u and MCC^r corresponds to a set of features F^r. Thus, $MCC^u \Leftrightarrow F^u$ and $MCC^r \Leftrightarrow F^r$. Thus, the context C is a set of contextual features F^c as shown in Equation 7.4:

$$C = \quad F^c = \quad F^u \cup F^r \tag{7.4}$$

Getting feature recommendations from contextually similar projects

After identifying the contextual features C of the user's project, we identify similar projects and use their feature profiles to make predictions about the existence of other relevant features in the user's project. This is accomplished by using the kNN algorithm. We calculate the similarity of the user's project and each of the existing projects in R and select the top N most similar projects as neighbors of the user's project. We compute the cosine similarity of user's project P^u with each existing repository project p as shown in Equation 7.5:

$$sim(P^u, p) = \frac{|\mathrm{F^c} \cap \mathrm{F^p}|}{\sqrt{|\mathrm{F^c}| \cdot |\mathrm{F^p}|}} \tag{7.5}$$

where F^p denotes the set of features or method clone classes of project p (MCC^p) in repository R and F^c denotes the set of features or method clone classes of the user context (MCC^c). Thus, we can also represent Equation 7.5 as Equation 7.6:

$$sim(P^u, p) = \frac{|\mathrm{MCC^c} \cap \mathrm{MCC^p}|}{\sqrt{|\mathrm{MCC^c}| \cdot |\mathrm{MCC^p}|}} \tag{7.6}$$

After finding contextually similar projects, we recommend the popular features identified as the frequently occurring API usage-based method clone classes across the projects. The prediction

scores of likelihood of a repository feature (which we identify as a method clone class ζ) being relevant to the user project P^u are calculated based on Schafer's technique [183] as shown in Equation 7.7:

$$pred(P^u, \zeta) = \frac{\sum_{n \in nbr(P^u)} sim(P^u, n) \cdot \kappa(n, \zeta)}{\sum_{n \in nbr(P^u)} sim(P^u, n)} \tag{7.7}$$

where $n \in nbr(p)$ represents a neighbor of P^u, and $\kappa(n, \zeta)$ is a function indicating whether project n contains method clone class ζ . In general, we compute prediction scores for each candidate method clone class of the neighbor projects and obtain the method clone classes Ψ with highest prediction scores above a certain threshold γ.

Getting Feature Recommendations from MCS

The aim of this technique is to find a set of features which co-occur frequently with the feature reused by the user from the repository R. Let m^r be the method reused by the user and let ζ^r be the method clone class of the reused method. We get a set of method clone structures $MCS^r = \{mcs_1^r....mcs_n^r\}$ containing ζ^rsuch that $\zeta^r \in mcs_i^r, \forall i = \{1...n\}$. If $MCS^r = \emptyset$, we get the call graph-based neighbor methods of m^r and a set of method clone classes $MCC^g = \{\zeta_1^g....\zeta_n^g\}$ assigned to the neighbors. Then, we get a set of method clone structures $MCS^g = \{mcs_1^g....mcs_n^g\}$ containing the neighbor method clone classes such that $\zeta_i^g \in mcs_j^g, \forall i = \{1...n\} \wedge \forall j = \{1...m\}$. If $MCS^g = \emptyset$, we get the file-based neighbor methods of m^r and a set of method clone classes $MCC^f = \{\zeta_1^f....\zeta_n^f\}$ assigned to the methods. Then, we get a set of method clone structures $MCS^f = \{mcs_1^f....mcs_n^f\}$ containing the neighbor method clone classes such that $\zeta_i^f \in mcs_j^f, \forall i = \{1...n\} \wedge \forall j = \{1...m\}$.

Finally, we obtain a set of method clone structures MCS^x (where $x = \{r|g|f\}$) each containing method clone classes. We create a union set of distinct method clone classes Ω and order them by a minimum support parameter β as specified by the tuple $(\zeta_1^o, ..., \zeta_n^o, \beta)$.

Aggregating and Filtering the Feature Recommendations

We take an intersection of the method clone classes in sets Ψ and Ω to get an aggregate set of method clone classes MCC^{agg} as shown in Equation 7.8.

Algorithm 3 Recommending related methods using hybrid context-aware FACER

Input: $m^s, M^u, M^r, H, R, \gamma, \beta, K, N$

Output: M^{rec}: a set of related methods

1: $F^u \leftarrow identifyContextFeatures(M^u, R)$

2: $F^r \leftarrow getReusedFeatures(M^r, H)$

3: $F^c \leftarrow F^u \cup F^r$

4: $P \leftarrow getTopKSimilarProjects(F^c, R, K)$

5: $\Psi \leftarrow getPopularFeatures(P, R, \gamma)$

6: $\Omega \leftarrow getFACERRelatedFeatures(m^s, R, \beta)$

7: $F^{agg} \leftarrow \Psi \cap \Omega$

8: **if** $F^{agg} = \emptyset$ **then**

9: $\quad F^{agg} \leftarrow \Omega$

10: **end if**

11: $F \leftarrow F^{agg} - F^c$

12: $F^{rec} \leftarrow getTopNFeatures(F, N)$

13: $M^{rec} \leftarrow getRepresentativeMethods(F^{rec}, R)$

return M_{rec}

$$MCC^{agg} = \Psi \cap \Omega \tag{7.8}$$

If $MCC^{agg} = \emptyset$, then the set of recommended features MCC^{rec} will come from Ψ. We filter the method clone classes from MCC^{rec} to exclude any features that are part of the user context to get our final set of filtered recommendations MCC^f as shown in Equation 7.9:

$$MCC^f = MCC^{rec} - MCC^c \tag{7.9}$$

Recommending Representative Methods

Finally, we get the top N features from the set MCC^f and obtain representative methods against each method clone class to recommend a set of top N methods M^{rec}.

To summarize, Algorithm 3 shows all the steps involved in using a development context to recommend a set of related methods M^{rec} from the FACER code fact repository R built on an organization's source code. The development context consists of the user's selected method m^s, the set of methods M^u written by the user for their current active project, and the set of methods M^r reused by a user from previous recommendation scenarios saved as a history H. First, we use the set of methods M^u to identify those features F^u implemented by a user which also exist in the repository R (Line 1). Next, we get the features F^r which were previously reused by the user (Line 2) and combine them with the features F^u to form the set of contextual features F^c (Line 3). We then perform contextual pre-filtering to get top K similar projects P (Line 4) and get popular features from P using Equation 7.7. We select only the features having a score above the threshold γ. We then use the FACER baseline recommendation approach to get related features against the user-selected method m^s and specify a minimum support threshold β for the frequent patterns of co-occurring features used in the recommendation process (Line 6). We then aggregate the popular features from the pre-filtered projects with those obtained using baseline FACER (Line 7). In case there are no common features recommended, we ignore the recommendations obtained from pre-filtering and use the recommendations obtained from baseline FACER (Lines 8 and 9). Next, we perform post-filtering to remove the set of features that are already a part of the user's context F^c from the set of aggregate features F^{agg} (Line 11) and get the top N features (Line 12). Finally, we get representative methods against the set of features F^{rec} and return the final set of recommended methods M^{rec} (Line 13).

7.4 Chapter Summary

Previously in Chapter 3, Section 3.4, we perform an in-depth analysis of existing context-aware code recommender systems to identify the sources, scope and triggers of context extraction, and the various contextual modeling paradigms adopted by these systems. From this analysis, we identify that there are currently no context-aware code recommenders that can provide code for related features that a developer may want to implement next. To address this gap, we take FACER as a baseline code recommender that can recommend code for related features, and plan to investigate

whether adding context-awareness to FACER can improve the delivery of relevant method recommendations. In this chapter, we discuss our proposed context-aware code recommendation approach called CA-FACER, including the three architectural variants of CA-FACER. We introduce the idea of capturing context as multiple sets of API usages representing a variety of functionality or software features, where each set of API usages comes from a single method body. Furthermore, we leverage multiple sources to obtain contextual data which include a developer's active development profile, code reuse history, and organizational development activity.

Chapter 8

CA-FACER Evaluation

8.1 Experimental Setup

The goal of our experiment is to determine if introducing context-awareness can improve the performance of FACER for recommending relevant related code examples. As part of our experiment, we devise and evaluate different implementations of context-aware FACER (CA-FACER) using different architectural models and techniques. We consider contextual elements as features implemented by developers as methods and recognized by our system as API usage sets or clone classes. We also evaluate the effect of context size on the precision of recommendations.

8.1.1 Research Questions

We aim to answer the following research questions (RQs):

- **RQ 8.1**: How does baseline FACER perform in evolving development contexts?
- **RQ 8.2**: Does post-retrieval contextual filtering improve the performance of baseline FACER for evolving contexts?
- **RQ 8.3**: Can we improve the performance of baseline FACER by making context-sensitive recommendations?

- **RQ 8.4**: Does the size of context affect the precision of recommendations?

In our previous work [51], we demonstrated the ability of FACER to provide related methods in Stage 2 against a user-selected method. The user-selected method was the only input used to generate related method recommendations without considering the context of development. In this work, we improve the performance of FACER by incorporating a novel contextual element which we define as a set of API usages in a method representing a software feature. In real development scenarios, context is ever-evolving. For RQ 8.1, we investigate the performance of baseline FACER in an evolving context. Thus, we simulate development scenarios that depict an early stage context and other more evolved development contexts for making recommendations. In evolving contexts, there is a chance that recommended results are not useful for a developer because they are already implemented. With RQ 8.2, we investigate if filtering the recommended results to remove the features that are already part of the developer's context can improve the performance of baseline FACER. The aim of RQ 8.3 is to determine whether introducing context-awareness can improve the quality of recommendations. To answer this question, we devise and evaluate various context-aware strategies and compare them against our baseline FACER system. For RQ 8.4, we want to confirm our intuition that a larger size of context enables more precise recommendations.

8.1.2 Dataset

We use the same dataset from our previous work [51]. Applications are from four different categories of Java-based Android applications: (1) music player, (2) Bluetooth chat, (3) weather, and (4) file management. 30 applications from each category result in a total of 120 applications. Overall, our dataset for the evaluation consists of 120 Java-based Android applications which we analyze in order to populate the FACER code fact repository. The time to execute the program analyzer on this dataset is almost 55 minutes on a Core i7 2.2 GHz machine with 8GB memory running Windows 10. Table 8.1 summarizes some of the key statistics of the FACER code fact repository. During the detection of clone classes, we consider only methods having a minimum of three unique API calls to ensure that we have meaningful features. Thus, the dataset consists of clone classes from 7,922 methods out of the 37,303 methods in the repository. We also ignore API

Table 8.1: FACER Code Fact Repository Statistics

Metric	**Value**
No. of applications	120
No. of files	4,369
Lines of comments	175,000
Lines of code (LOC)	498,261
No. of methods	37,303
No. of method calls	150,341
No. of API classes	2,209
No. of unique API calls	7,607
Total no. of API calls	85,386

Table 8.2: Method Clone Classes (MCC) and Method Clone Structures (MCS) detected with varying similarity threshold α

α	No. of Clone Classes	No. of MCS
0.3	1445	536
0.5	1397	107
0.7	812	37
0.9	347	11

calls involving the usage of *Log*, *Intent* and *Toast* API classes, because we want to filter out common API calls which appear in almost every application and do not contribute towards a particular feature of an application. Table 8.2 shows the number of clone classes and method clone structures against varying similarity thresholds.

8.1.3 Evaluation Methodology

We adopt an off-line automated evaluation methodology. To test the precision of recommendations, we need to simulate a user development context for input to our system and then we want to compare the recommendations against a ground truth. We take a set of 20 feature queries. For each query, we get a set of matching methods from FACER's Stage 1 code search. We select a method m^r that best implements the desired feature and select the project to which it belongs as a test project for our evaluation setup. This way we acquire a set of 20 test projects pertaining to 20 methods selected against 20 feature queries. From each test project, we simulate a user context (C) and ground truth (G) by splitting methods implementing features of a project into two parts. We test four context configurations to simulate both early and evolved development stages. The twenty queries are shown in Table 8.3.

We define context (C_0) for an initial development stage where a developer has not yet implemented any features. The ground truth (G_0) corresponding to this context configuration consists of all methods implementing features of a test project. For C_1, we take the selected method (m^r) and all those methods that belong to the same file to simulate the user-written methods m^u. Then, we take the feature class identifiers for the selected method and the simulated user methods to define C_1 as shown in Equation 8.1:

$$C_1 = \zeta^r + \bigcup_{i=1}^{n} \zeta_i^u \tag{8.1}$$

The ground truth (G_1) corresponding to this context configuration consists of all methods implementing features of a test project except the set of methods m^u. We define context configurations C_2 and C_3 to represent an evolved development stage where almost half of the features of a project are already implemented. For C_2, we use one half of features as the context and the other half as the ground truth G_2. To define context configuration C_3, we use features of ground truth G_2 and to define ground truth G_3, we use features from context configuration C_2.

For each of the 20 test projects, we assume that we have an existing user context and a method m^r was recently selected for reuse by the user. Our context aware system then generates a set of related method recommendations against the user's context C and the selected method m^r.

Table 8.3: Feature Queries

No.	Feature Description	Method ID	Category
1	receive paired devices name and address	33	Bluetooth chat
2	update list of paired Bluetooth devices	80	Bluetooth chat
3	do discovery of Bluetooth devices	423	Bluetooth chat
4	send message over Bluetooth	1066	Bluetooth chat
5	connect to a Bluetooth device	1161	Bluetooth chat
6	create new folder	2250	File Manager
7	browse to file or directory	2616	File Manager
8	move file	2642	File Manager
9	put file to cache	2971	File Manager
10	draw bitmap	3017	File Manager
11	set data source for media player	14435	Music Player
12	receive key press to start stop pause media	14490	Music Player
13	search for song	15214	Music Player
14	download music	22968	Music Player
15	play music	24068	Music Player
16	save forecast in database	28669	Weather
17	send Http request to get weather	29298	Weather
18	check if network connection available	29947	Weather
19	check and add permissions for location access	31838	Weather
20	create new memory cache to store weather icons	33549	Weather

Since the test project comes from the repository, we are aware of the feature class labels of the methods forming the user context. Therefore, we do not need to explicitly perform the context identification phase of CA-FACER. The feature class label ζ^r of m^r and feature class labels of other user context methods $\zeta_i^u, \forall i = \{1, ..., n\}$, are input to CA-FACER to get top N related method

recommendations. To summarize, we vary the size of context as follows:

- C_0: Empty context simulating a fresh start
- C_1: Methods implementing features in one file from a test project (early development stage)
- C_2: Half of the methods implementing features from a test project (more evolved context)
- C_3: Other half of the methods implementing features from a test project (more evolved context)

The aim of our evaluation is to first highlight the need and purpose of context-awareness. We do this by evaluating non-context-aware FACER for different development stages by varying ground truth. Then, we aim to compare FACER against various context-aware strategies and determine an optimal context-aware strategy for FACER. Finally, we want to check the effect of context size on the quality of recommendations. Our experimental evaluation methodology is based on varying context size and recommendation techniques.

Evaluating baseline FACER in evolving development stages

For our first experiment, we evaluate baseline FACER by varying the size of ground truth in accordance with the development stage of a developer. We use 20 test input methods to obtain top N recommendations from baseline FACER, where N = 5, 10, 15. We compare the recommendations against ground truth configurations G_0, G_1, G_2 and G_3. For this experiment, we want to understand how a varying development stage and developer expectations for relevant code can affect the performance of baseline FACER .

Evaluating Post-filtering

In this experiment, we aim to answer **RQ 8.1** to find out if context-awareness can improve the performance of baseline FACER. We evaluate a post-filtering based context-aware approach using the 20 test input methods to obtain top N recommendations from baseline FACER, where N = 5, 10, 15. We evaluate three context configurations C_1, C_2 and C_3 and compare the recommendations against ground truth configurations G_1, G_2 and G_3 respectively.

Evaluating Pre-filtering

In this experiment, we evaluate a pre-filtering based context-aware approach using Schafer's technique [183] to make recommendations. We perform evaluations for different number of projects K= 10 and K= 20 that we pre-filter to generate recommendations. We also perform evaluations for different values of prediction score threshold γ = 0.3, 0.5, 0.7, 0.9. We use the 20 test input methods to obtain top N recommendations, where N = 5, 10, 15. We evaluate three context configurations C_1, C_2 and C_3 and compare the recommendations against ground truth configurations G_1, G_2 and G_3 respectively. We also determine optimal values for K and γ.

Evaluating proposed hybrid CA-FACER

In this experiment, we evaluate a hybrid context-aware approach by combining both pre-filtering and post-filtering techniques to make recommendations from an intersection set. We use the optimal values for K and γ to obtain top N recommendations, where N = 5, 10, 15. We evaluate three context configurations C_1, C_2 and C_3 and compare the recommendations against ground truth configurations G_1, G_2 and G_3 respectively.

Comparing context-aware approaches

We compare the performance of context-aware approaches for FACER based on post-filtering, pre-filtering, and a hybrid context-aware model (hybrid CA-FACER). The results from this experiment help us answer **RQ 8.2**.

Evaluating effect of context size

For this experiment, we aim to evaluate how the different sizes of context can affect the performance of CA-FACER. We compare an early stage development context against an evolved development context for the same set of ground truth features. Thus, we evaluate context configurations C_1 and C_2 with ground truth G_2 and context configurations C_1 and C_3 with ground truth G_3. We use the optimal values for K and γ to obtain top 10 recommendations.

8.1.4 Evaluation Metrics

Precision: Precision measures the ability to correctly recommend related methods. The precision of recommendations for a particular user query and context is calculated as the fraction of recommended methods that are relevant i.e. belonging to the set of methods representing the ground truth, as shown in Equation 8.2. If all the recommended methods occur at least once in the ground truth, we have 100% precision.

$$Precision = \frac{|rec\,methods \cap GT\,methods|}{|rec\,methods|} \tag{8.2}$$

Success Rate: This metric measures the rate at which the recommender can return at least one relevant recommendation against an input method. The success rate is defined as shown in Equation 8.3.

$$SuccessRate = \frac{|queries\,answered|}{|queries|} \tag{8.3}$$

where *queries* represents the set of test input methods FACER receives and *queries answered* represents the number of times FACER successfully retrieves at least one correct recommendation against a test input method.

Mean Reciprocal Rank: The mean reciprocal rank is the average of the reciprocal ranks of the results for the number of test input methods *M*. It is defined in Equation 8.4.

$$\text{MRR} = \frac{1}{|M|}\sum_{i=1}^{|M|}\frac{1}{\text{rank}_i} \tag{8.4}$$

where $rank_i$ refers to the rank position of the first relevant result for the i-th test input method.

Wilcoxon test: We use paired and un-paired Wilcoxon tests [184] to test the statistical significance of our evaluation results. Our null hypothesis asserts that the medians of the precision values of any two recommendation strategies are identical. We perform Wilcoxon tests to compare the performance of baseline FACER with a post-filtering approach for three context configurations C_1, C_2, and C_3 for top 10 recommendations to answer **RQ 8.1**. We also perform Wilcoxon tests to compare the performance of context-aware approaches to answer **RQ 8.2**. For this setup we use optimal values of K and γ and consider the three context configurations C_1, C_2, and C_3 for top 10

recommendations. Finally, we determine the statistical significance of differences of performance for varying context sizes to answer **RQ 8.3**.

8.2 Evaluation Results

The data of empirical evaluation of context-aware FACER is available online on GitHub*. The software to replicate the results of all experiments related to context-aware FACER is available online on GitHub†.

8.2.1 FACER's performance in evolving development stages

The results of evaluating baseline FACER in evolving development stages are shown in Table 8.4. We calculate the average precision, success rate (SR) and mean reciprocal rank (MRR) across 20 test input queries assuming four development stages for which there are four different ground truth configurations G_0, G_1, G_2, and G_3. For top 5 results evaluations, the precision is highest at 0.9 when the ground truth G_0 consists of all the features of a test input project. However, precision is significantly reduced for G_1(P=.001), G_2(P <.001), and G_3(P<.001). This is because some of the methods being recommended are already part of the development context and not part of the ground truth. Since baseline FACER is not aware of methods already in the developer's context, its recommendations, while being precise, may become redundant based on developer's expectations across different development stages. Therefore, we conclude that it is important for code recommenders to be context-sensitive to filter out redundant recommendations. For P@10 and P@15, we observe a similar reduction in precision for G_1, G_2, and G_3 as compared to G_0. There is also a reduction in success rate and mean reciprocal rank for G_1, G_2, and G_3 as compared to G_0. Thus, in answer to **RQ 8.1**, we can say that in evolving contexts, baseline FACER performance is reduced because of redundant recommendations.

*https://github.com/shamsa-abid/CA-FACER_Replication

†https://github.com/shamsa-abid/CA-FACER_Replication/blob/main/CAFACER_Results_Replication.jar

Table 8.4: Comparing average precision, success rates and MRR of recommendations for FACER under evolving development stages

	Top 5			Top 10			Top 15		
	P	SR	MRR	P	SR	MRR	P	SR	MRR
C0	0.90	1.00	1.00	0.83	1.00	1.00	0.79	1.00	1.00
C1	0.58	0.90	0.88	0.52	0.90	0.88	0.48	0.90	0.88
C2	0.55	0.95	0.80	0.54	0.95	0.80	0.54	0.95	0.80
C3	0.34	0.85	0.61	0.29	0.90	0.58	0.26	0.90	0.58

8.2.2 CA-FACER's post-filtering results

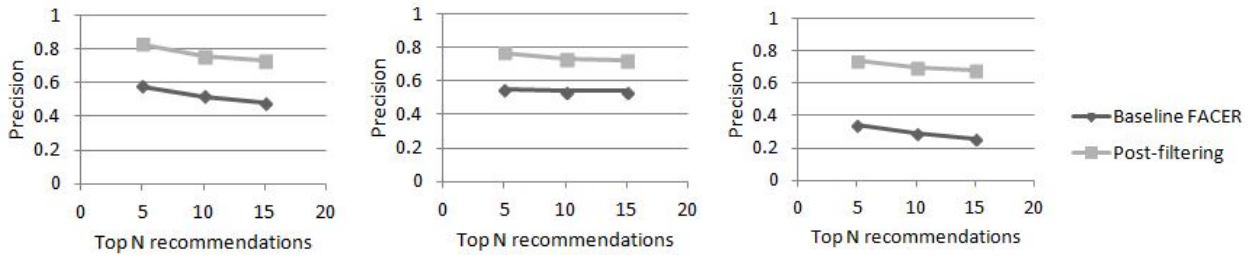
To check if context-awareness can improve the performance of baseline FACER, we add a context-identification and post-filtering step to baseline FACER. The results in Table 8.5 show that the performance of baseline FACER improves significantly with context-aware post-filtering. For development context C_1 and the corresponding ground truth G_1, P@5 is 0.83, whereas for non-context-aware FACER, it is 0.58. We also observe a rise in the mean reciprocal rank (MRR) of the post-filtered recommendations for contexts C_1, C_2, and C_3 We perform an un-paired two samples Wilcoxon test [178] on the precision of the 20 queries on top 5 recommendations to test our null hypothesis which asserts that the medians of the precision values for baseline FACER with and without post-filtering are identical. We obtain p-values of 0.012, 0.015 and 0.001 for C_1, C_2, and C_3 respectively, which are all less than the significance level alpha = 0.05. This means that we can reject the null hypothesis, and accordingly we can conclude that the performance of context-aware baseline FACER with post-filtering is significantly better than baseline FACER without context-awareness. This answers **RQ 8.2** that the post-retrieval contextual filtering does indeed improve the performance of baseline FACER by a minimum of 22% and a maximum of 40% for top 5 recommendations. Figure 8.1 shows at a glance that for the three context configurations C_1, C_2, and C_3, recommendations from post-filtering have higher precision than baseline FACER.

There are approaches which perform post-retrieval filtering or ranking to provide only those

recommendations which have some similarity with the development context. These approaches are useful for code search tools where a recommendation result is expected to be structurally or lexically similar to the code elements that are part of a developer's context, whereas for providing recommendations that are feature extensions, such a strategy may be over-restrictive. Thus, in our case post-filtering is effective in removing redundant results but it can not be used to filter or rank recommendations based on context because the related method recommendations are expected to be structurally and lexically different from the code a developer has already written.

Table 8.5: Comparing average precision of recommendations for FACER with post-filtering for context configurations C_1, C_2, C_3

	P@5	P@10	P@15	SR	MRR
C1	0.83	0.76	0.73	0.90	1.00
C2	0.77	0.73	0.72	0.95	1.00
C3	0.74	0.70	0.68	0.90	0.96



(a) Early context C_1 (b) Evolved context C_2 (c) Evolved context C_3

Figure 8.1: Comparing average precision of recommendations for baseline FACER against post-filtering

8.2.3 CA-FACER's Pre-filtering results

Tables 8.6, 8.7 and 8.8 show results from the pre-filtering approach using Schafer's technique [183] to recommend methods. We also apply post-filtering to these recommendations to remove redun-

dant recommendations and then calculate the evaluation metrics. In Tables 8.6, 8.7, and 8.8 we report the average precision, success rate and mean reciprocal rank for the 20 test input methods for different values of the number of projects (K) filtered and the prediction score threshold (γ). The general trend in all these tables is that precision increases with higher value of γ while the success rate reduces. We want to find an optimal value of *K* and γ such that we get a high precision and success rate. From Table 8.6, we observe that P@N remains the highest with γ = 0.7 while maintaining a success rate close to the highest for both K=10 and K=20. Thus, we choose γ = 0.7 and K=10 as the optimal configuration parameters. We choose K=10 to reduce the number of computations without compromising the level of performance. As compared to post-filtering, the success rate of pre-filtering approach is lower. The highest success rate for top 5 recommendations using pre-filtering with C_1 is 0.65 while it is 0.90 for post-filtered baseline FACER results. Furthermore, with a maximum success rate, the precision for pre-filtering with C_1 is 0.67 whereas it 0.83 for post-filtering. Thus, while pre-filtering can be very precise with a maximum precision of 0.97, it comes with the loss of success rate in making at least one correct recommendation. While baseline FACER makes recommendations based on feature co-occurrence, pre-filtering makes context-sensitive recommendations by using projects with features similar to the development context to recommend popular features of those projects. With these limitations of pre-filtering, we next aim to devise a hybrid solution to get more precise results.

8.2.4 Hybrid CA-FACER's results

In our hybrid approach, we recommend methods that not only frequently co-occur with the developer-selected method but also frequently occur in projects with a domain similar to that of the project being developed. Table 8.9 shows the performance results of our proposed hybrid context-aware approach (hybrid CA-FACER) using optimal values of K=10 and $\gamma = 0.7$ for context configurations $C_1, C_2,$ and C_3. The maximum average precision for top 5 recommendations is 0.94 with a success rate of 0.90 under early development stage context configuration C_1. Thus, we achieve both good average precision and success rate for top 5 recommendations. A minimum average precision of 0.72 is observed for top 15 recommendations under evolved

Table 8.6: Average precision, success rate and mean reciprocal rank of top N recommendations from pre-filtering approach for context configuration C_1 and ground truth G_1

		Top 5			Top 10			Top 15		
K	γ	P@5	SR	MRR	P@10	SR	MRR	P@15	SR	MRR
10	0.3	0.44	0.65	0.85	0.41	0.75	0.75	0.42	0.85	0.67
10	0.5	0.67	0.65	0.95	0.69	0.70	0.89	0.68	0.70	0.89
10	0.7	0.97	0.60	0.96	0.97	0.60	0.96	0.97	0.60	0.96
10	0.9	0.96	0.45	0.90	0.96	0.45	0.94	0.96	0.45	0.94
20	0.3	0.43	0.65	0.84	0.40	0.75	0.74	0.41	0.85	0.66
20	0.5	0.64	0.65	0.95	0.66	0.70	0.89	0.65	0.70	0.89
20	0.7	0.97	0.60	0.96	0.97	0.60	0.96	0.97	0.60	0.96
20	0.9	0.97	0.55	0.95	0.97	0.55	0.95	0.97	0.55	0.95

Table 8.7: Average precision, success rate and mean reciprocal rank of top N recommendations from pre-filtering approach for context configuration C_2 and ground truth G_2

		Top 5			Top 10			Top 15		
K	γ	P@5	SR	MRR	P@10	SR	MRR	P@15	SR	MRR
10	0.3	0.58	0.70	0.90	0.55	0.80	0.80	0.54	0.80	0.80
10	0.5	0.58	0.55	1.00	0.60	0.65	0.86	0.59	0.65	0.86
10	0.7	0.80	0.50	1.00	0.83	0.60	0.85	0.81	0.60	0.85
10	0.9	0.97	0.45	0.94	0.96	0.45	0.94	0.93	0.45	0.94
20	0.3	0.53	0.65	0.94	0.49	0.80	0.79	0.49	0.80	0.79
20	0.5	0.58	0.55	1.00	0.60	0.65	0.87	0.59	0.65	0.87
20	0.7	0.78	0.55	1.00	0.80	0.65	0.87	0.79	0.65	0.87
20	0.9	0.98	0.50	0.95	0.97	0.50	0.95	0.94	0.50	0.95

Table 8.8: Average precision, success rate and mean reciprocal rank of top N recommendations from pre-filtering approach for context configuration C_3 and ground truth G_3

		Top 5			Top 10			Top 15		
K	γ	P@5	SR	MRR	P@10	SR	MRR	P@15	SR	MRR
10	0.3	0.41	0.55	0.77	0.41	0.60	0.72	0.44	0.65	0.67
10	0.5	0.71	0.50	0.80	0.76	0.50	0.80	0.77	0.50	0.80
10	0.7	0.69	0.30	0.83	0.73	0.30	0.83	0.75	0.30	0.83
10	0.9	0.80	0.15	0.83	0.90	0.15	0.83	0.92	0.15	0.83
20	0.3	0.40	0.55	0.72	0.40	0.65	0.64	0.42	0.70	0.60
20	0.5	0.63	0.60	0.79	0.66	0.60	0.79	0.67	0.60	0.79
20	0.7	0.66	0.35	0.86	0.70	0.35	0.86	0.71	0.35	0.86
20	0.9	0.68	0.20	0.88	0.74	0.20	0.88	0.75	0.20	0.88

context configuration C_3. However, the success rate is still high at 0.90. We now need to compare the performance of CA-FACER against baseline FACER.

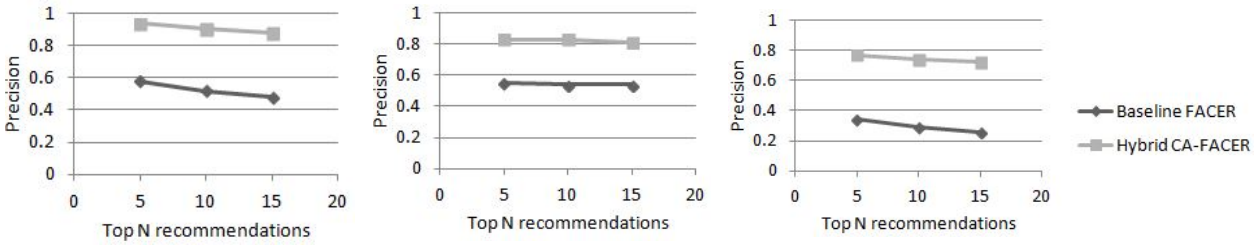
8.2.5 Results of comparison of context-aware approaches of FACER

From Table 8.4, we observe that for baseline FACER, under context C_1 and ground truth G_1 the P@5 is 0.58, whereas, from Tables 8.5, 8.6 and 8.9 we observe that P@5 is higher with 0.83, 0.97 and 0.94 for post-filtering, pre-filtering and hybrid CA-FACER respectively. While the pre-filtering approach has high precision, we observe a lower success rate than our proposed approach. On the other hand, hybrid CA-FACER is able to provide precise recommendations with a higher success rate and mean reciprocal rank. We perform an un-paired two samples Wilcoxon test [178] on the precision of the 20 queries on top 5 recommendations to test our null hypothesis which asserts that the medians of the precision values for baseline FACER and hybrid CA-FACER are identical. We obtain p-values $P<.001$, $P=.002$ and $P<.001$ for C_1, C_2, and C_3 respectively, which are all less than the significance level alpha = 0.05. This means that we can reject the null hypoth-

Table 8.9: Average precision, success rate and mean reciprocal rank of top N recommendations from our proposed hybrid CA-FACER approach for context configurations C_1, C_2, C_3

	Top 5			Top 10			Top 15		
context	P@5	SR	MRR	P@10	SR	MRR	P@15	SR	MRR
C1	0.94	0.90	1.00	0.91	0.90	1.00	0.88	0.90	1.00
C2	0.83	0.95	1.00	0.83	0.95	1.00	0.81	0.95	1.00
C3	0.77	0.90	0.96	0.74	0.90	0.96	0.72	0.90	0.96

esis, and accordingly we can conclude that the performance of hybrid CA-FACER is significantly better than baseline FACER without context-awareness. Thus, in answer to **RQ 8.3**, we can say that our proposed hybrid CA-FACER does indeed improve the performance of baseline FACER by a minimum of 36% and a maximum of 46% for making context-sensitive related method recommendations. Figure 8.2 shows at a glance that for the three context configurations C_1, C_2, and C_3, recommendations from post-filtering have higher average precision than baseline FACER.



(a) Early context C_1 (b) Evolved context C_2 (c) Evolved context C_3

Figure 8.2: Comparing average precision of recommendations for baseline FACER against hybrid CA-FACER

8.2.6 Results of evaluating context size

Table 8.10 compares the results of average precision @10 over 20 test input methods for four different context and ground truth configurations. We compare average precision for the same

ground truth G_3 but different context configurations C_1 and C_3. The precision 0.83 is higher for the more evolved context C_3 than 0.79 for the early stage context C_1. Similarly, for the same ground truth G_4 but different context configurations C_1 and C_4, the precision is higher at 0.74 for the more evolved context C_4 than the precision at 0.59 for the early stage context C_1. We can also see that the success rate also improves by 20% with a greater context size C_4 as compared to C_1 for G_4. We perform an un-paired two samples Wilcoxon test [178] on the precision of the 20 queries to test our null hypothesis which asserts that the medians of the precision values for the same ground truth G_3 with different context sizes C_1 and C_3 are identical. A p-value of 0.454, which is greater than the significance level alpha = 0.05 means that we can not reject the null hypothesis. We perform another un-paired two samples Wilcoxon test [178] on the precision of the 20 queries to test our null hypothesis which asserts that the medians of the precision values for the same ground truth G_4 with different context sizes C_1 and C_4 are identical. A p-value of 0.284, which is greater than the significance level alpha = 0.05 means that we can not reject the null hypothesis. Thus, in answer to **RQ 8.4** we can say that precision of recommendations using a greater amount of contextual data is higher but not statistically significantly higher than using a smaller amount of contextual data.

Table 8.10: Comparing average precision, success rate and mean reciprocal rank of top 10 recommendations from our proposed hybrid CA-FACER approach for different context sizes and same ground truth

context	ground truth	P@10	SR	MRR
C1	G2	0.79	0.95	1.00
C2	G2	0.83	0.95	1.00
C1	G3	0.59	0.70	0.95
C3	G3	0.74	0.90	0.96

8.3 Threats to Validity

8.3.1 Construct Validity

The use of an artificial development context is a threat to construct validity. To reduce this threat, we create a development context from an actual project's file which contains the method selected against a query. We assume that at some point of development, a developer has written code of the file. With many possibilities for an early development stage, we opted to consider the code of the file containing the selected method as the current active file context. In reality, a developer may have written code across many files. In future, we plan to reduce this threat by capturing actual development contexts in an actual development scenario. However, this kind of evaluation is expensive considering human and time resources. Our current evaluation setup which simulates evolving contexts is equally effective in evaluating our hypotheses.

For evaluating recommendations in an evolved context, we perform a 2-fold validation. First, we select half of the features of a project including the active file features as the context and the remaining features as ground truth. Then, for simulating a different evolved context, we take the prior ground truth features as well as the active file features as context and use the remaining features as ground truth. This swapping of context and ground truth ensures that both possibilities of development scenarios are captured and threats to validity like selection bias are reduced.

The ground truth we use in our automated evaluation is a proxy for a subjective decision that should be made by the developer. There could be a method that is actually relevant but that we consider as a false positive. To reduce this threat, we plan to involve actual developers in our evaluation in future.

8.3.2 Internal Validity

We do not explicitly perform context identification to label features in a developer's context since we already have the labels available for the test project. However, using our proposed context identification process, we expect to get similar results.

8.3.3 External Validity

For our current evaluation, since we are not working with actual developers, we do not have an actual reuse history or actual development profile. We only simulate a reuse history and development profile by considering a set of methods from a complete test project. However, we vary the number of selected methods and also perform a cross-selection of methods to reduce the external threat to validity.

Currently, we use 120 projects from different categories which we download from GitHub instead of an organization's source code. We believe that our approach can be easily generalized to an actual organization setting where a software product line is being developed. Making recommendations from an organization's source code repository can yield more accurate recommendations for developers of that organization. Organizations that develop product line software have multiple projects with similar features and can benefit from our approach to enhance developer productivity.

8.4 Chapter Summary

We demonstrate how context-awareness based on API usage sets can improve the precision of FACER's code recommendations. We use a combination of pre-filtering, post-filtering, and contextual modeling to provide context-aware code examples for opportunistic reuse. From our experimental evaluation on 120 Java Android projects from GitHub, we observe a 46% improvement of precision using our proposed context-aware approach over our baseline FACER. Our experiments indicate that our proposed hybrid CA-FACER approach achieves 94% precision for top 5 recommendations with a 90% success rate for an initial development stage. We also observe that while a greater amount of contextual information improves the precision of recommendations, the improvement is not statistically significant.

Chapter 9

Conclusions and Future Work

9.1 Dissertation Summary

This dissertation focused on the topic of related code recommendation for opportunistic reuse. While several techniques have been proposed for code search and code recommendation over many years, the proactive recommendation of related code for implementing various features of an application remains unaddressed. Furthermore, with the amount of source code increasing steadily, reuse opportunities for related code are even greater.

To that end, this dissertation presented FACER, a Feature-driven API usage-based Code Example Recommender, which uses program analysis, clustering, frequent pattern mining, and context-awareness to recommend related code examples with high precision for opportunistic reuse. The basic idea of this dissertation is to **exploit the cross-project patterns of frequently co-occurring API-usage based semantic method clones to provide relevant code recommendations**.

To evaluate FACER, we extracted data from 120 open-source GitHub projects from four different domains. We first performed a manual validation of the detected method clusters to ensure that clustering methods based on API usages results in meaningful clusters with functionally similar methods. Our results show that 91% of the analyzed method clusters are valid. We then performed an automatic evaluation with different FACER settings to determine the best configuration for related method recommendations. Once we determined the best configuration, we performed a user

evaluation with 10 professional and 39 experienced student participants to determine whether the related methods recommended by FACER are indeed relevant to the original method and feature. Our results show that FACER's related method recommendations are, on average, 80% precise. We also received positive feedback about FACER's functionality, which encourages us to further improve FACER and release it to developers soon. The current prototype implementation for FACER, along with all the data from our evaluations is available on our online artifact page [166].

We then demonstrate how context-awareness based on API usage sets can improve the precision of FACER's code recommendations. We introduce the idea of capturing context as multiple sets of API usages representing a variety of functionality or software features, where each set of API usages comes from a single method body. Furthermore, we leverage multiple sources to obtain contextual data which include a developer's active development profile, code reuse history, and organizational development activity. We use a combination of pre-filtering, post-filtering, and contextual modeling to provide context-aware code examples for opportunistic reuse. From our experimental evaluation on 120 Java Android projects from GitHub, we observe a 46% improvement of precision using our proposed context-aware approach over our baseline FACER. Our experiments indicate that our proposed hybrid CA-FACER approach achieves 94% precision for top 5 recommendations with a 90% success rate for an initial development stage. We also observe that while a greater amount of contextual information improves the precision of recommendations, the improvement is not statistically significant.

This dissertation made four major contributions in the field of code recommendation for reuse. The specific contributions can be summarized as follows:

1. **FACER (Chapters 5, 6)** - Feature-driven API usage-based code recommendation is an important and novel contribution in the area of related code recommendation for opportunistic reuse. Chapter 5 presented a detailed description of our proposed approach called FACER. We describe FACER's architecture including offline repository building workflow and online recommendation workflow. Chapter 6, Section 6.3 presented a detailed evaluation of FACER's semantic clone detection, including intra-clone group similarity validation and inter-clone group dissimilarity validation. Chapter 6 Sections 6.4 and 6.5 presented the au-

tomated evaluation for sensitivity analysis of parameters and manual analysis for precision of recommendations respectively.

2. **A study on context-awareness in code recommender systems (Chapter 3, Section 3.4)** The context-aware architectures for code recommendation systems remain undocumented in current research literature. In Chapter 3, Section 3.4, we develop a better understanding of the current nature, purpose and use of context for code recommendation. We highlight each system's recommendation category, their triggers to obtain contextual data, the scope of context, the elements that constitute the context, and the contextual modeling paradigm.

3. **CA-FACER (Chapters 7 and 8)** - Since there are currently no existing context-aware systems that can provide code recommendations of multiple related features for opportunistic reuse on-the-go, Context-Aware FACER (CA-FACER) is an important contribution. In Chapter 7, we presented context-aware architectures for FACER and in Chapter 8 we investigated whether context-awareness can improve the delivery of relevant method recommendations for opportunistic reuse. We also assessed the effect of context-awareness on the precision of recommendations, compared various context-aware models to propose a context-aware code recommendation strategy that improves the baseline FACER, and assessed the effect of context size on the precision of recommendations.

4. **FACER for Eclipse and Android Studio (Chapter 7 Section 5.4, Appendix A)** - There is a lack of practical code recommender tools for developers that are integrated into the IDE. We presented FACER's IDE-integrated tools for developers to use for their daily development activities. Chapter 7 Section 5.4 described the user interface of FACER's Eclipse plugin tool and Appendix A described FACER's Android Studio plugin tool.

Given the above studies and their findings in our dissertation, we conclude the following: (1) API usage-based method clustering results in valid semantic method clones, (2) FACER's related method recommendations are highly precise, (3) context-awareness based on API usage sets can significantly improve the precision of FACER's code recommendations and a greater amount of contextual information slightly improves the precision of recommendations, (4) developers need

to search for related features and they perceive that FACER can be very useful for speeding up their development.

9.2 Future Work

Code search and recommendation has come a long way in the last decade. With our contribution towards related code recommendation we have laid a foundation, and there is room for further works in research aspects inspired by this dissertation. We conclude by identifying some of the relevant areas that are yet to be explored.

1. While FACER currently relies on clustering for semantic clone detection, recent deep learning techniques can be substituted to obtain semantic clone groups and FACER's recommendations can be evaluated to investigate any effect on performance. Similarly, FACER's pattern mining can be substituted with machine learning models to analyze the impact on quality of recommendations.

2. Sometimes developers need recommendations for optimal method alternatives. The optimization here refers to developer's need for code that is well refactored, uses memory optimally, or follows some standard coding conventions. For this problem, a solution is desired which would provide optimal method alternatives to the user in his IDE. Future work can be done to explore solutions that cater to the optimization needs of developers. The fact that our recommended methods come from clone groups with multiple implementations of the same functionality provides a solution. Future work can learn to distinguish between different methods of a clone group using quality parameters to provide alternate solutions for a recommendation.

3. For locating the most suitable code snippet in an open-source repository, other factors like licensing, popularity and code quality can affect the selection process and need to be taken into consideration. In future, we can integrate these aspects in the ranking algorithm.

4. The work can further be extended in the direction of making predictions based on business

use case. This can be implemented by filtering and/or prioritizing the recommendations that directly relate to the business logic of the application. Here the challenge will lie in identifying the business use case from available application source code and incorporating that information in the search query or using it to filter or rank search results.

5. Currently the repository is built offline, and no further update is made in it during the execution of FACER. This results in a static repository that does not evolve with experience. This need can be addressed in future by having a continuous repository update mechanism in place.

6. Sometimes inaccurate code recommendations may be due to the recommended methods containing API calls that do not necessarily translate to a core feature for an application's product domain, instead they may be implementing framework-specific code which glues together the core features of an application. Future work directions include focusing on distinguishing between domain-relevant features and other more generic framework-specific helper features for improved recommendation.

7. Ensuring that a recommended method can automatically integrate and work without problems requires a deeper understanding and awareness of the developer's context. Future work can focus on making context-sensitive recommendations that integrate easily without exceptions.

8. In addition, there is a need to have an industry-based investigation to find out whether FACER can effectively speed up development in real settings, future work is desired where developers perform some programming tasks using their conventional methods and their task completion time is compared with that of tasks completed using FACER.

Bibliography

[1] L. Heinemann, F. Deissenboeck, M. Gleirscher, B. Hummel, and M. Irlbeck, "On the extent and nature of software reuse in open source java projects," *Top productivity through software reuse*, pp. 207–222, 2011.

[2] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 266–276, IEEE, 2012.

[3] J. Brandt, P. J. Guo, J. Lewenstein, and S. R. Klemmer, "Opportunistic programming: How rapid ideation and prototyping occur in practice," in *Proceedings of the 4th international workshop on End-user software engineering*, pp. 1–5, ACM, 2008.

[4] H. Jiang, L. Nie, Z. Sun, Z. Ren, W. Kong, T. Zhang, and X. Luo, "Rosf: Leveraging information retrieval and supervised learning for recommending code snippets," *IEEE Transactions on Services Computing*, 2016.

[5] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting working code examples," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 664–675, ACM, 2014.

[6] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 111–120, ACM, 2011.

[7] S. K. Bajracharya, J. Ossher, and C. V. Lopes, "Leveraging usage similarity for effective retrieval of examples in code repositories," in *Proceedings of the eighteenth ACM*

SIGSOFT international symposium on Foundations of software engineering, pp. 157–166, ACM, 2010.

[8] H. Dumitru, M. Gibiec, N. Hariri, J. Cleland-Huang, B. Mobasher, C. Castro-Herrera, and M. Mirakhorli, "On-demand feature recommendations derived from mining public product descriptions," in *Proceedings of the 33rd International Conference on Software Engineering*, pp. 181–190, ACM, 2011.

[9] C. McMillan, N. Hariri, D. Poshyvanyk, J. Cleland-Huang, and B. Mobasher, "Recommending source code for use in rapid software prototypes," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 848–858, IEEE Press, 2012.

[10] B. Dit, M. Revelle, M. Gethers, and D. Poshyvanyk, "Feature location in source code: a taxonomy and survey," *Journal of software: Evolution and Process*, vol. 25, no. 1, pp. 53–95, 2013.

[11] J. Han, J. Pei, and M. Kamber, *Data mining: concepts and techniques*. Elsevier, 2011.

[12] B. Yi, X. Shen, H. Liu, Z. Zhang, W. Zhang, S. Liu, and N. Xiong, "Deep matrix factorization with implicit feedback embedding for recommendation system," *IEEE Transactions on Industrial Informatics*, vol. 15, no. 8, pp. 4591–4601, 2019.

[13] C. Shahabi and Y.-S. Chen, "An adaptive recommendation system without explicit acquisition of user relevance feedback," *Distributed and Parallel Databases*, vol. 14, no. 2, pp. 173–192, 2003.

[14] R. Xie, C. Ling, Y. Wang, R. Wang, F. Xia, and L. Lin, "Deep feedback network for recommendation.," in *IJCAI*, pp. 2519–2525, 2020.

[15] T. Joachims and F. Radlinski, "Search engines that learn from implicit feedback," *Computer*, vol. 40, no. 8, pp. 34–40, 2007.

[16] M. Bruch, M. Monperrus, and M. Mezini, "Learning from examples to improve code completion systems," in *Proceedings of the the 7th joint meeting of the European software en-*

gineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, pp. 213–222, ACM, 2009.

[17] R. Robbes and M. Lanza, "Improving code completion with program history," *Automated Software Engineering*, vol. 17, no. 2, pp. 181–212, 2010.

[18] A. T. Nguyen, T. T. Nguyen, H. A. Nguyen, A. Tamrawi, H. V. Nguyen, J. Al-Kofahi, and T. N. Nguyen, "Graph-based pattern-oriented, context-sensitive source code completion," in *Proceedings of the 34th International Conference on Software Engineering*, pp. 69–79, IEEE Press, 2012.

[19] L. Heinemann, V. Bauer, M. Herrmannsdoerfer, and B. Hummel, "Identifier-based context-dependent API method recommendation," in *2012 16th European Conference on Software Maintenance and Reengineering*, pp. 31–40, IEEE, 2012.

[20] S. Proksch, J. Lerch, and M. Mezini, "Intelligent code completion with bayesian networks," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 25, no. 1, pp. 1–31, 2015.

[21] T. T. Nguyen, H. V. Pham, P. M. Vu, and T. T. Nguyen, "Recommending API usages for mobile apps with hidden markov model," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 795–800, IEEE, 2015.

[22] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, "A Simple, Efficient, Context-sensitive Approach for Code Completion," *Journal of Software: Evolution and Process*, vol. 28, no. 7, pp. 512–541, 2016.

[23] A. R. D'Souza, D. Yang, and C. V. Lopes, "Collective Intelligence for Smarter API Recommendations in Python," in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pp. 51–60, IEEE, 2016.

[24] A. T. Nguyen, M. Hilton, M. Codoban, H. A. Nguyen, L. Mast, E. Rademacher, T. N. Nguyen, and D. Dig, "API code recommendation using statistical learning from fine-grained

changes," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 511–522, ACM, 2016.

[25] T. Xie and J. Pei, "MAPO: Mining API usages from open source repositories," in *Proceedings of the 2006 international workshop on Mining software repositories*, pp. 54–57, ACM, 2006.

[26] F. Mccarey, M. Ó. Cinnéide, and N. Kushmerick, "Rascal: A recommender agent for agile reuse," *Artificial Intelligence Review*, vol. 24, no. 3-4, pp. 253–276, 2005.

[27] V. Raychev, M. Vechev, and E. Yahav, "Code completion with statistical language models," in *ACM SIGPLAN Notices*, vol. 49, pp. 419–428, ACM, 2014.

[28] P. Nguyen, J. Di Rocco, D. Ruscio, L. Ochoa, T. Degueule, and M. Di Penta, "Focus: A recommender system for mining api function calls and usage patterns," in *41st ACM/IEEE International Conference on Software Engineering (ICSE)*, 2019.

[29] M. Tsunoda, T. Kakimoto, N. Ohsugi, A. Monden, and K.-i. Matsumoto, "Javawock: A Java Class Recommender System Based on Collaborative Filtering.," in *SEKE*, pp. 491–497, 2005.

[30] Y. Ye and G. Fischer, "Supporting reuse by delivering task-relevant and personalized information," in *Proceedings of the 24th international conference on Software engineering*, pp. 513–523, ACM, 2002.

[31] R. Holmes and G. C. Murphy, "Using structural context to recommend source code examples," in *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, pp. 117–125, IEEE, 2005.

[32] N. Sahavechaphan and K. Claypool, "XSnippet: Mining for sample code," *ACM Sigplan Notices*, vol. 41, no. 10, pp. 413–430, 2006.

[33] O. Hummel, W. Janjic, and C. Atkinson, "Code Conjurer: Pulling reusable software out of thin air," *IEEE software*, vol. 25, no. 5, 2008.

[34] M. Ichii, Y. Hayase, R. Yokomori, T. Yamamoto, and K. Inoue, "Software component recommendation using collaborative filtering," in *2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pp. 17–20, IEEE, 2009.

[35] R. Shimada, Y. Hayase, M. Ichii, M. Matsushita, and K. Inoue, "A-SCORE: Automatic software component recommendation using coding context," in *2009 31st International Conference on Software Engineering-Companion Volume*, pp. 439–440, IEEE, 2009.

[36] N. Murakami, H. Masuhara, and T. Aotani, "Code recommendation based on a degree-of-interest model," in *Proceedings of the 4th International Workshop on Recommendation Systems for Software Engineering*, pp. 28–29, ACM, 2014.

[37] S. Zhou, B. Shen, and H. Zhong, "Lancer: Your code tell me what you need," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 1202–1205, IEEE, 2019.

[38] M. Robillard, R. Walker, and T. Zimmermann, "Recommendation systems for software engineering," *IEEE software*, vol. 27, no. 4, pp. 80–86, 2010.

[39] G. Bavota, A. De Lucia, A. Marcus, and R. Oliveto, "Recommending refactoring operations in large software systems," in *Recommendation Systems in Software Engineering*, pp. 387–419, Springer, 2014.

[40] T. Ishihara, K. Hotta, Y. Higo, and S. Kusumoto, "Reusing reused code," in *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pp. 457–461, IEEE, 2013.

[41] A. Ohtani, Y. Higo, T. Ishihara, and S. Kusumoto, "On the level of code suggestion for reuse," in *Software Clones (IWSC), 2015 IEEE 9th International Workshop on*, pp. 26–32, IEEE, 2015.

[42] "Archived Eclipse Projects." `http://www.eclipse.org/projects/archives.php`. [Online; accessed 16-May-2018].

[43] R. Hill and J. Rideout, "Automatic method completion," in *Proceedings of the 19th IEEE international conference on Automated software engineering*, pp. 228–235, IEEE Computer Society, 2004.

[44] V. Balachandran, "Query by example in large-scale code repositories," in *Software Maintenance and Evolution (ICSME), 2015 IEEE International Conference on*, pp. 467–476, IEEE, 2015.

[45] K. Kim, D. Kim, T. F. Bissyandé, E. Choi, L. Li, J. Klein, and Y. L. Traon, "FaCoY: A Code-to-code Search Engine," in *Proceedings of the 40th International Conference on Software Engineering*, ICSE '18, (New York, NY, USA), pp. 946–957, ACM, 2018.

[46] W. Takuya and H. Masuhara, "A spontaneous code recommendation tool based on associative search," in *Proceedings of the 3rd International Workshop on Search-Driven Development: Users, Infrastructure, Tools, and Evaluation*, pp. 17–20, ACM, 2011.

[47] R. Holmes, R. J. Walker, and G. C. Murphy, "Strathcona example recommendation tool," in *ACM SIGSOFT Software Engineering Notes*, vol. 30, pp. 237–240, ACM, 2005.

[48] S. Wang, D. Lo, and L. Jiang, "Autoquery: automatic construction of dependency queries for code search," *Automated Software Engineering*, vol. 23, no. 3, pp. 393–425, 2016.

[49] M. Gasparic, G. C. Murphy, and F. Ricci, "A context model for ide-based recommendation systems," *Journal of Systems and Software*, vol. 128, pp. 200–219, 2017.

[50] "Stack Overflow Question: Android: Select image from gallery then crop that and show in an imageview." `https://stackoverflow.com/questions/25490928/androidselect-image-from-gallery-then-crop-that-and-show-in-an-imagevi` [Online; accessed 16-Sep-2020].

[51] S. Abid, S. Shamail, H. A. Basit, and S. Nadi, "FACER: An API usage-based code-example recommender for opportunistic reuse," *Empirical Software Engineering*, vol. 26, no. 6, pp. 1–58, 2021.

[52] S. Jansen, S. Brinkkemper, I. Hunink, and C. Demir, "Pragmatic and opportunistic reuse in innovative start-up companies," *IEEE software*, vol. 25, no. 6, pp. 42–49, 2008.

[53] N. Mäkitalo, A. Taivalsaari, A. Kiviluoto, T. Mikkonen, and R. Capilla, "On opportunistic software reuse," *Computing*, vol. 102, no. 11, pp. 2385–2408, 2020.

[54] Y. Kim and E. A. Stohr, "Software reuse: survey and research directions," *Journal of Management Information Systems*, vol. 14, no. 4, pp. 113–147, 1998.

[55] R. G. Lanergan and C. A. Grasso, "Software engineering with reusable designs and code," *IEEE Transactions on Software Engineering*, no. 5, pp. 498–501, 1984.

[56] J. Brandt, P. J. Guo, J. Lewenstein, M. Dontcheva, and S. R. Klemmer, "Two studies of opportunistic programming: interleaving web foraging, learning, and writing code," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pp. 1589–1598, ACM, 2009.

[57] S. Abid, S. Javed, M. Naseem, S. Shahid, H. A. Basit, and Y. Higo, "CodeEase: Harnessing Method Clone Structures for Reuse," in *2017 IEEE 11th International Workshop on Software Clones (IWSC)*, pp. 1–7, IEEE, 2017.

[58] B. Hartmann, S. Doorley, and S. R. Klemmer, "Hacking, mashing, gluing: Understanding opportunistic design," *IEEE Pervasive Computing*, vol. 7, no. 3, pp. 46–54, 2008.

[59] C. K. Roy and J. R. Cordy, "A Survey on Software Clone Detection Research," *Queen's School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[60] S. Bellon, R. Koschke, G. Antoniol, J. Krinke, and E. Merlo, "Comparison and evaluation of clone detection tools," *IEEE Transactions on software engineering*, vol. 33, no. 9, pp. 577–591, 2007.

[61] M. Gabel, L. Jiang, and Z. Su, "Scalable detection of semantic clones," in *Proceedings of the 30th international conference on Software engineering*, pp. 321–330, 2008.

[62] K. A. Kontogiannis, R. DeMori, E. Merlo, M. Galler, and M. Bernstein, "Pattern matching for clone and concept detection," *Automated Software Engineering*, vol. 3, no. 1, pp. 77–108, 1996.

[63] C. K. Roy, J. R. Cordy, and R. Koschke, "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of computer programming*, vol. 74, no. 7, pp. 470–495, 2009.

[64] C. Liu, X. Xia, D. Lo, C. Gao, X. Yang, and J. Grundy, "Opportunities and challenges in code search tools," *arXiv preprint arXiv:2011.02297*, 2020.

[65] P. Bielik, V. Raychev, and M. Vechev, "Programming with" Big Code": Lessons, Techniques and Applications," in *LIPIcs-Leibniz International Proceedings in Informatics*, vol. 32, Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2015.

[66] M. Vechev, E. Yahav, *et al.*, "Programming with "Big Code"," *Foundations and Trends® in Programming Languages*, vol. 3, no. 4, pp. 231–284, 2016.

[67] C. Sadowski, K. T. Stolee, and S. Elbaum, "How developers search for code: a case study," in *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, pp. 191–201, ACM, 2015.

[68] M. Umarji, S. Sim, and C. Lopes, "Archetypal internet-scale source code searching," *Open source development, communities and quality*, pp. 257–263, 2008.

[69] E. Linstead, S. Bajracharya, T. Ngo, P. Rigor, C. Lopes, and P. Baldi, "Sourcerer: mining and searching internet-scale software repositories," *Data Mining and Knowledge Discovery*, vol. 18, no. 2, pp. 300–336, 2009.

[70] J. Shuai, L. Xu, C. Liu, M. Yan, X. Xia, and Y. Lei, "Improving code search with co-attentive representation learning," in *Proceedings of the 28th International Conference on Program Comprehension*, pp. 196–207, 2020.

[71] W. Ye, R. Xie, J. Zhang, T. Hu, X. Wang, and S. Zhang, "Leveraging code generation to improve code retrieval and summarization via dual learning," in *Proceedings of The Web Conference 2020*, pp. 2309–2319, 2020.

[72] S. Chatterjee, S. Juvekar, and K. Sen, "Sniff: A search engine for java using free-form queries," *Fundamental Approaches to Software Engineering*, pp. 385–400, 2009.

[73] X. Gu, H. Zhang, and S. Kim, "Deep code search," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pp. 933–944, IEEE, 2018.

[74] S. Sachdev, H. Li, S. Luan, S. Kim, K. Sen, and S. Chandra, "Retrieval on Source Code: A Neural Code Search," in *Proceedings of the 2Nd ACM SIGPLAN International Workshop on Machine Learning and Programming Languages*, MAPL 2018, (New York, NY, USA), pp. 31–41, ACM, 2018.

[75] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: a multilinguistic token-based code clone detection system for large scale source code," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 654–670, 2002.

[76] S. Luan, D. Yang, C. Barnaby, K. Sen, and S. Chandra, "Aroma: Code recommendation via structural code search," *Proceedings of the ACM on Programming Languages*, vol. 3, no. OOPSLA, pp. 1–28, 2019.

[77] C. Ragkhitwetsagul and J. Krinke, "Siamese: scalable and incremental code clone search via multiple code representations," *Empirical Software Engineering*, vol. 24, no. 4, pp. 2236–2284, 2019.

[78] S. P. Reiss, "Semantics-based code search," in *Proceedings of the 31st International Conference on Software Engineering*, pp. 243–253, IEEE Computer Society, 2009.

[79] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," *ECOOP 2009–Object-Oriented Programming*, pp. 318–343, 2009.

[80] S. Bajracharya, J. Ossher, and C. Lopes, "Sourcerer: An internet-scale software repository," in *Proceedings of the 2009 ICSE Workshop on Search-Driven Development-Users, Infrastructure, Tools and Evaluation*, pp. 1–4, IEEE Computer Society, 2009.

[81] J. Cambronero, H. Li, S. Kim, K. Sen, and S. Chandra, "When deep learning met code search," in *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 964–974, 2019.

[82] Y. Wan, J. Shu, Y. Sui, G. Xu, Z. Zhao, J. Wu, and P. S. Yu, "Multi-modal attention network learning for semantic source code retrieval," *arXiv preprint arXiv:1909.13516*, 2019.

[83] L. Chen, W. Ye, and S. Zhang, "Capturing source code semantics via tree-based convolution over API-enhanced AST," in *Proceedings of the 16th ACM International Conference on Computing Frontiers*, pp. 174–182, 2019.

[84] Z. Yao, J. R. Peddamail, and H. Sun, "Coacor: Code annotation for code retrieval with reinforcement learning," in *The World Wide Web Conference*, pp. 2203–2214, 2019.

[85] F. Lv, H. Zhang, J.-g. Lou, S. Wang, D. Zhang, and J. Zhao, "Codehow: Effective code search based on api understanding and extended boolean model (e)," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pp. 260–270, IEEE, 2015.

[86] Q. Zou and C. Zhang, "Query expansion via learning change sequences," *International Journal of Knowledge-based and Intelligent Engineering Systems*, vol. 24, no. 2, pp. 95–105, 2020.

[87] M. M. Rahman, C. K. Roy, and D. Lo, "Rack: Automatic api recommendation using crowd-sourced knowledge," in *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, vol. 1, pp. 349–359, IEEE, 2016.

[88] W. Liu, X. Peng, Z. Xing, J. Li, B. Xie, and W. Zhao, "Supporting exploratory code search with differencing and visualization," in *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 300–310, IEEE, 2018.

[89] W. Li, S. Yan, B. Shen, and Y. Chen, "Reinforcement learning of code search sessions," in *2019 26th Asia-Pacific Software Engineering Conference (APSEC)*, pp. 458–465, IEEE, 2019.

[90] K. Järvelin and J. Kekäläinen, "Cumulated gain-based evaluation of ir techniques," *ACM Transactions on Information Systems (TOIS)*, vol. 20, no. 4, pp. 422–446, 2002.

[91] G. Adomavicius and A. Tuzhilin, "Context-aware recommender systems," in *Recommender systems handbook*, pp. 217–253, Springer, 2011.

[92] "Apache Lucene." `https://lucene.apache.org`. [Online; accessed 28-August-2017].

[93] "Apache Lucene Core." `https://lucene.apache.org/core/`. [Online; accessed 29-Sep-2020].

[94] B. S. Baker, "A theory of parameterized pattern matching: algorithms and applications," in *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pp. 71–80, 1993.

[95] P. Yang, H. Fang, and J. Lin, "Anserini: Enabling the use of Lucene for information retrieval research," in *Proceedings of the 40th International ACM SIGIR Conference on Research and Development in Information Retrieval*, pp. 1253–1256, 2017.

[96] S. Yan, H. Yu, Y. Chen, B. Shen, and L. Jiang, "Are the code snippets what we are searching for? a benchmark and an empirical study on code search with natural-language queries," in *2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 344–354, IEEE, 2020.

[97] K. Mens and A. Lozano, "Source code-based recommendation systems," in *Recommendation Systems in Software Engineering*, pp. 93–130, Springer, 2014.

[98] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate structural context matching: An approach to recommend relevant examples," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, 2006.

[99] K. Inoue, R. Yokomori, T. Yamamoto, M. Matsushita, and S. Kusumoto, "Ranking significance of software components based on use relations," *IEEE Transactions on Software Engineering*, vol. 31, no. 3, pp. 213–225, 2005.

[100] T. Ishihara, K. Hotta, Y. Higo, H. Igaki, and S. Kusumoto, "Inter-project functional clone detection toward building libraries-an empirical study on 13,000 projects," in *Reverse Engineering (WCRE), 2012 19th Working Conference on*, pp. 387–391, IEEE, 2012.

[101] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *Proceedings of the 10th Working Conference on Mining Software Repositories*, pp. 319–328, IEEE Press, 2013.

[102] H. Niu, I. Keivanloo, and Y. Zou, "API usage pattern recommendation for software development," *Journal of Systems and Software*, vol. 129, pp. 127–139, 2017.

[103] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, "Jungloid mining: helping to navigate the api jungle," in *ACM SIGPLAN Notices*, vol. 40, pp. 48–61, ACM, 2005.

[104] S. Thummalapenta and T. Xie, "Parseweb: a programmer assistant for reusing open source code on the web," in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pp. 204–213, ACM, 2007.

[105] L. Wang, L. Fang, L. Wang, G. Li, B. Xie, and F. Yang, "APIExample: An effective web search based usage example recommendation system for Java APIs," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*, pp. 592–595, IEEE Computer Society, 2011.

[106] A. Mishne, S. Shoham, and E. Yahav, "Typestate-based semantic code search over partial programs," in *Acm Sigplan Notices*, vol. 47, pp. 997–1016, ACM, 2012.

[107] C. Lv, W. Jiang, Y. Liu, and S. Hu, "APISynth: a new graph-based API recommender system.," in *ICSE Companion*, pp. 596–597, 2014.

[108] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live API documentation," in *Proceedings of the 36th International Conference on Software Engineering*, pp. 643–652, ACM, 2014.

[109] L. Moreno, G. Bavota, M. Di Penta, R. Oliveto, and A. Marcus, "How can i use this method?," in *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, pp. 880–890, IEEE, 2015.

[110] X. Gu, H. Zhang, D. Zhang, and S. Kim, "Deep API learning," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pp. 631–642, ACM, 2016.

[111] J. Zhao and Y. Liu, "Detecting and Ranking API Usage Pattern in Large Source Code Repository: A LFM Based Approach," in *International Cross-Domain Conference for Machine Learning and Knowledge Extraction*, pp. 41–56, Springer, 2017.

[112] J. Zhang, H. Jiang, Z. Ren, and X. Chen, "Recommending APIs for API Related Questions in Stack Overflow," *IEEE Access*, vol. 6, pp. 6205–6219, 2018.

[113] F. Thung, R. J. Oentaryo, D. Lo, and Y. Tian, "WebAPIRec: Recommending web APIs to software projects via personalized ranking," *arXiv preprint arXiv:1705.00561*, 2017.

[114] F. Thung, D. Lo, and J. Lawall, "Automated library recommendation," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, pp. 182–191, IEEE, 2013.

[115] S.-K. Hsu and S.-J. Lin, "MACs: Mining API code snippets for code reuse," *Expert Systems with Applications*, vol. 38, no. 6, pp. 7291–7301, 2011.

[116] H. Jiang, J. Zhang, X. Li, Z. Ren, D. Lo, X. Wu, and Z. Luo, "Recommending new features from mobile app descriptions," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 28, no. 4, p. 22, 2019.

[117] X. Chen, Q. Zou, B. Fan, Z. Zheng, and X. Luo, "Recommending software features for mobile applications based on user interface comparison," *Requirements Engineering*, pp. 1–15, 2018.

[118] H. Yu, Y. Lian, S. Yang, L. Tian, and X. Zhao, "Recommending features of mobile applications for developer," in *International Conference on Advanced Data Mining and Applications*, pp. 361–373, Springer, 2016.

[119] Y. Yu, H. Wang, G. Yin, and B. Liu, "Mining and recommending software features across multiple web repositories," in *Proceedings of the 5th Asia-Pacific Symposium on Internetware*, p. 9, ACM, 2013.

[120] S. Maki, S. Kpodjedo, and G. El Boussaidi, "Context extraction in recommendation systems in software engineering: a preliminary survey," in *Proceedings of the 25th Annual International Conference on Computer Science and Software Engineering*, pp. 151–160, IBM Corp., 2015.

[121] Y. Malheiros, A. Moraes, C. Trindade, and S. Meira, "A source code recommender system to support newcomers," in *2012 IEEE 36th Annual Computer Software and Applications Conference*, pp. 19–24, IEEE, 2012.

[122] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A project memory for software development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 446–465, 2005.

[123] A. Lozano, A. Kellens, and K. Mens, "Mendel: Source code recommendation based on a genetic metaphor," in *2011 26th IEEE/ACM International Conference on Automated Software Engineering (ASE 2011)*, pp. 384–387, IEEE, 2011.

[124] M. P. Robillard and P. Manggala, "Reusing program investigation knowledge for code understanding," in *2008 16th IEEE International Conference on Program Comprehension*, pp. 202–211, IEEE, 2008.

[125] B. Antunes, J. Cordeiro, and P. Gomes, "SDiC: Context-based retrieval in Eclipse," in *2012 34th International Conference on Software Engineering (ICSE)*, pp. 1467–1468, IEEE, 2012.

[126] F. Long, X. Wang, and Y. Cai, "API hyperlinking via structural overlap," in *Proceedings of the 7th Joint meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pp. 203–212, 2009.

[127] A. Thies and C. Roth, "Recommending rename refactorings," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pp. 1–5, 2010.

[128] J. Cordeiro, B. Antunes, and P. Gomes, "Context-based recommendation to support problem solving in software development," in *2012 Third International Workshop on Recommendation Systems for Software Engineering (RSSE)*, pp. 85–89, IEEE, 2012.

[129] M. M. Rahman and C. K. Roy, "Surfclipse: Context-aware meta-search in the IDE," in *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 617–620, IEEE, 2014.

[130] B. Ashok, J. Joy, H. Liang, S. K. Rajamani, G. Srinivasa, and V. Vangala, "DebugAdvisor: a recommender system for debugging," in *Proceedings of the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pp. 373–382, 2009.

[131] M. Erfani, I. Keivanloo, and J. Rilling, "Opportunities for clone detection in test case recommendation," in *2013 IEEE 37th Annual Computer Software and Applications Conference Workshops*, pp. 65–70, IEEE, 2013.

[132] O. Hummel, W. Janjic, and C. Atkinson, "Proposing software design recommendations based on component interface intersecting," in *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, pp. 64–68, 2010.

[133] X. Ge, D. Shepherd, K. Damevski, and E. Murphy-Hill, "How the sando search tool recommends queries," in *2014 Software Evolution Week-IEEE Conference on Software Maintenance, Reengineering, and Reverse Engineering (CSMR-WCRE)*, pp. 425–428, IEEE, 2014.

[134] S. Lee, S. Kang, and M. Staats, "NavClus: A graphical recommender for assisting code exploration," in *2013 35th International Conference on Software Engineering (ICSE)*, pp. 1315–1318, IEEE, 2013.

[135] N. Sawadsky, G. C. Murphy, and R. Jiresal, "Reverb: Recommending code-related web pages," in *Proceedings of the 2013 International Conference on Software Engineering*, pp. 812–821, IEEE Press, 2013.

[136] R. Xie, X. Kong, L. Wang, Y. Zhou, and B. Li, "HiRec: API Recommendation using Hierarchical Context," in *2019 IEEE 30th International Symposium on Software Reliability Engineering (ISSRE)*, pp. 369–379, IEEE, 2019.

[137] T. K. Landauer and S. T. Dumais, "A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge.," *Psychological review*, vol. 104, no. 2, p. 211, 1997.

[138] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *Proceedings of the 4th international conference on Aspect-oriented software development*, pp. 159–168, ACM, 2005.

[139] S. Abid and H. A. Basit, "Towards a structural clone based recommender system," in *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, vol. 3, pp. 51–52, IEEE, 2016.

[140] H. A. Basit and S. Jarzabek, "A data mining approach for detecting higher-level clones in software," *IEEE Transactions on Software engineering*, vol. 35, no. 4, pp. 497–514, 2009.

[141] H. A. Basit, U. Ali, S. Haque, and S. Jarzabek, "Things structural clones tell that simple clones don't," in *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, pp. 275–284, IEEE, 2012.

[142] X. Xia, L. Bao, D. Lo, P. S. Kochhar, A. E. Hassan, and Z. Xing, "What do developers search for on the web?," *Empirical Software Engineering*, vol. 22, no. 6, pp. 3149–3185, 2017.

[143] M. Mayo, "Frequent Pattern Mining and the Apriori Algorithm: A Concise Technical Overview." `https://www.kdnuggets.com/2016/10/association-rule-learning-concise-technical-overview.html`, October 2016. [Online; accessed December 2019].

[144] "Stack Overflow Developer Survey 2020: Most Loved, Dreaded, and Wanted Platforms." `https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-platforms`. [Online; accessed 16-September-2020].

[145] R. Abdalkareem, E. Shihab, and J. Rilling, "On code reuse from stackoverflow: An exploratory study on android apps," *Information and Software Technology*, vol. 88, pp. 148–158, 2017.

[146] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A large-scale empirical study on software reuse in mobile apps," *IEEE software*, vol. 31, no. 2, pp. 78–86, 2013.

[147] S. McIlroy, N. Ali, and A. E. Hassan, "Fresh apps: an empirical study of frequently-updated mobile apps in the google play store," *Empirical Software Engineering*, vol. 21, no. 3, pp. 1346–1370, 2016.

[148] L. Pascarella, F.-X. Geiger, F. Palomba, D. Di Nucci, I. Malavolta, and A. Bacchelli, "Self-reported activities of android developers," in *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pp. 144–155, IEEE, 2018.

[149] "GitHub." https://github.com/. [Online; accessed 28-August-2020].

[150] "Eclipse Java Development Tools." https://www.eclipse.org/jdt/. [Online; accessed 28-Sep-2020].

[151] "Lucene Document." https://lucene.apache.org/core/7_2_0/core/org/apache/lucene/document/Document.html. [Online; accessed 29-Sep-2020].

[152] "Apache Lucene - Scoring: Score Boosting." https://lucene.apache.org/core/3_5_0/scoring.html#Score%20Boosting. [Online; accessed 27-Sep-2020].

[153] "Abstract Syntax Trees." https://www.eclipse.org/jdt/core/r2.0/dom%20ast/ast.html. [Online; accessed 28-Sep-2020].

[154] "Java Class Libraries." https://docs.oracle.com/javase/8/docs/api/allclasses-frame.html. [Online; accessed 28-Sep-2020].

[155] "Java Development Kit." https://www.oracle.com/java/technologies/javase-downloads.html. [Online; accessed 28-Sep-2020].

[156] "Android SDK classes." https://developer.android.com/reference/classes. [Online; accessed 28-Sep-2020].

[157] "Android SDK classes." https://developer.android.com/studio. [Online; accessed 28-Sep-2020].

[158] J. Kanwal, O. Maqbool, H. A. Basit, and M. A. Sindhu, "Evolutionary perspective of structural clones in software," *IEEE Access*, vol. 7, pp. 58720–58739, 2019.

[159] P. Jaccard, "Étude comparative de la distribution florale dans une portion des Alpes et des Jura," *Bull Soc Vaudoise Sci Nat*, vol. 37, pp. 547–579, 1901.

[160] D. Defays, "An efficient algorithm for a complete link method," *The Computer Journal*, vol. 20, no. 4, pp. 364–366, 1977.

[161] R, "Heirarchical Clustering, R language documentation (Version 4.0.4)." https://rdrr.io/r/stats/hclust.html. [Online; accessed December 2019].

[162] G. Grahne and J. Zhu, "Fast algorithms for frequent itemset mining using fp-trees," *IEEE transactions on knowledge and data engineering*, vol. 17, no. 10, pp. 1347–1362, 2005.

[163] P. Fournier-Viger, "Mining Frequent Closed Itemsets using the FPClose Algorithm." https://www.philippe-fournier-viger.com/spmf/FPClose.php. [Online; accessed 1-Feb-2019].

[164] S. Luan, D. Yang, K. Sen, and S. Chandra, "Aroma: Code Recommendation via Structural Code Search," *arXiv preprint arXiv:1812.01158*, 2018.

[165] S. E. Robertson, S. Walker, S. Jones, M. M. Hancock-Beaulieu, M. Gatford, *et al.*, "Okapi at TREC-3," *Nist Special Publication Sp*, vol. 109, p. 109, 1995.

[166] "FACER Artifacts." https://github.com/shamsa-abid/FACER_Artifacts.

[167] J. Han, M. Kamber, and J. Pei, "9 - classification: Advanced methods," in *Data Mining (Third Edition)* (J. Han, M. Kamber, and J. Pei, eds.), The Morgan Kaufmann Series in Data Management Systems, pp. 393 – 442, Boston: Morgan Kaufmann, third edition ed., 2012.

[168] U. Yun and J. J. Leggett, "WLPMiner: weighted frequent pattern mining with length-decreasing support constraints," in *Pacific-Asia Conference on Knowledge Discovery and Data Mining*, pp. 555–567, Springer, 2005.

[169] "binaryDist." https://github.com/NikNakk/binaryDist/, June 2015.

[170] "Sparse Matrix Clustering." https://stackoverflow.com/questions/30944701/clustering-a-large-very-sparse-binary-matrix-in-r/30945176?noredirect=1#comment106303086_30945176, June 2015.

[171] S. K. Alexander Eckert, Lucas Godoy, "parallelDist: Parallel Distance Matrix Computation using Multiple Threads." https://cran.r-project.org/web/packages/parallelDist/index.html. [Online; accessed 14-Sept-2020].

[172] R. D. Venkatasubramanyam, S. Gupta, and H. K. Singh, "Prioritizing code clone detection results for clone management," in *2013 7th International Workshop on Software Clones (IWSC)*, pp. 30–36, IEEE, 2013.

[173] J. Svajlenko, I. Keivanloo, and C. K. Roy, "Scaling classical clone detection tools for ultra-large datasets: An exploratory study," in *2013 7th International Workshop on Software Clones (IWSC)*, pp. 16–22, IEEE, 2013.

[174] "Fiverr - Freelance Services Marketplace for Businesses." `https://www.fiverr.com/`. [Online; accessed 3-Feb-2021].

[175] J. Cohen, "A coefficient of agreement for nominal scales," *Educational and psychological measurement*, vol. 20, no. 1, pp. 37–46, 1960.

[176] J. R. Landis and G. G. Koch, "The measurement of observer agreement for categorical data," *biometrics*, pp. 159–174, 1977.

[177] C. Sammut and G. I. Webb, eds., *TF–IDF*, pp. 986–987. Boston, MA: Springer US, 2010.

[178] "Wilcoxon Test." `https://www.rdocumentation.org/packages/stats/versions/3.6.2/topics/wilcox.test`. [R Documentation Online; accessed 18-Oct-2020].

[179] D. Spinellis, "How to select open source components," *Computer*, vol. 52, no. 12, pp. 103–106, 2019.

[180] R. Capilla, T. Mikkonen, C. Carrillo, F. A. Fontana, I. Pigazzini, and V. Lenarduzzi, "Impact of opportunistic reuse practices to technical debt," in *2021 IEEE/ACM International Conference on Technical Debt (TechDebt)*, pp. 16–25, IEEE, 2021.

[181] S. Abid, "Recommending related functions from API usage-based function clone structures," *Proceedings of the 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1193–1195, 2019.

[182] G. C. Murphy, "Beyond integrated development environments: adding context to software development," in *2019 IEEE/ACM 41st International Conference on Software Engineering: New Ideas and Emerging Results (ICSE-NIER)*, pp. 73–76, IEEE, 2019.

[183] J. B. Schafer, D. Frankowski, J. Herlocker, and S. Sen, "Collaborative filtering recommender systems," in *The adaptive web*, pp. 291–324, Springer, 2007.

[184] F. Wilcoxon, "Individual comparisons of grouped data by ranking methods," *Journal of economic entomology*, vol. 39, no. 2, pp. 269–270, 1946.

[185] "FACER-AS Plugin Installer." `https://github.com/AymanAbaid/FacerPlugin/blob/master/build/distributions/FACER-AS-1.03.zip`.

[186] "FACER-AS Resources." `https://bit.ly/3zltzSE`.

[187] "FACER-AS Source Code." `https://github.com/AymanAbaid/FacerPlugin`.

Appendices

Appendix A

FACER-AS: An API Usage-based Code Recommendation Tool for Android Studio

A.1 Installation and Setup

To setup and install FACER-AS, a developer needs to download the FACER-AS plugin [185] and a *resources* folder [186]. The user can follow the instructions on the tool's GitHub page [187] to complete the setup and installation.

A.2 FACER-AS Plugin Usage

Once the plugin is installed in the IDE, a new menu option titled *FACER* appears on the menu bar of the Android Studio IDE, as shown in Figure A.1. Following are the various actions a developer can perform using the plugin:

1. **Plugin Configuration:** A first-time user needs to configure FACER-AS with the path to the *resources* folder using the *Configuration Setup* dialog. The user can i) click on the FACER menu option in the menu bar, and select "Configure FACER", or ii) press *Ctrl+2 key* combination, and then enter the path.

2. **Triggering FACER-AS:** A developer can write a query in the IDE's editor, select the query text, and invoke the plugin using one of two methods. They can i) click on the FACER menu option in the menu bar, and select "Enable FACER", or ii) press *Ctrl+1 key* combination. A popup appears inside the editor next to the selected text. The user can then select the option *Get FACER Recommendations* (Figure A.1: Label 1) from the popup to get Stage 1 method recommendations corresponding to the selected text.

3. **Viewing FACER-AS Tool Window:** When the user clicks on the option *Get FACER Recommendations* (Figure A.1: Label 1), a new window for the plugin appears docked in the IDE. There are two lists for displaying the names of recommended methods on the left side of the tool window. Recommendation results from a query search are displayed in the *Query Results* list (Figure A.1: Label 2), whereas related method recommendations are displayed in the *Related Methods* list (Figure A.1: Label 6). A *code view panel* on the right side is used to display the code. Small icons on the top right corner of the tool window are used for various functions as discussed here.

4. **Viewing Method Bodies:** When a user double-clicks on a method name (Figure A.1: Label 3), its body is displayed in the *code view panel* in a new tab (Figure A.1: Label 4). This allows opening multiple method bodies in multiple tabs at the same time.

5. **Getting Related Methods:** When a user opens a method body in the *code view panel*, and clicks on the *magic wand* icon (Figure A.1: Label 5) displayed on the top right corner of the *code view panel*, the recommendation of related methods (Stage 2) gets invoked. A list of methods related to the currently selected method appears in the *Related Methods* list (Figure A.1: Label 6).

6. **Copying Code:** To copy a method body into the editor, a user can i) press the key combination *Ctrl+c* to copy selected code to the system clipboard and manually copy at the desired location or ii) click on the *clipboard* icon (Figure A.1: Label 7), which pastes the method body at the cursor position inside the editor.

7. **Viewing Source File Code:** We enable programmers to read the context of recommended methods by providing the option of viewing the complete source code of a recommended method's host file. The *view source file* (Figure A.1: Label 8) icon can be used to view the source code of the host file of a method currently open in the *code view panel*. The code for the file opens in a new tab in the *code view panel* (Figure A.1: Label 9).

8. **Getting Called Methods:** A method currently open in the *code view panel* might contain calls to methods from the host application. The *called methods* icon (Figure A.1: Label 10) retrieves these called methods and displays them in the *Related Methods* list.

9. **Upvoting Related Methods:** An *upvote* icon (Figure A.1: Label 11) appears when related methods are recommended. The upvote feature is currently for evaluation purposes. For our user study, we ask the developer to upvote related methods regardless of their need to immediately reuse a code recommendation.

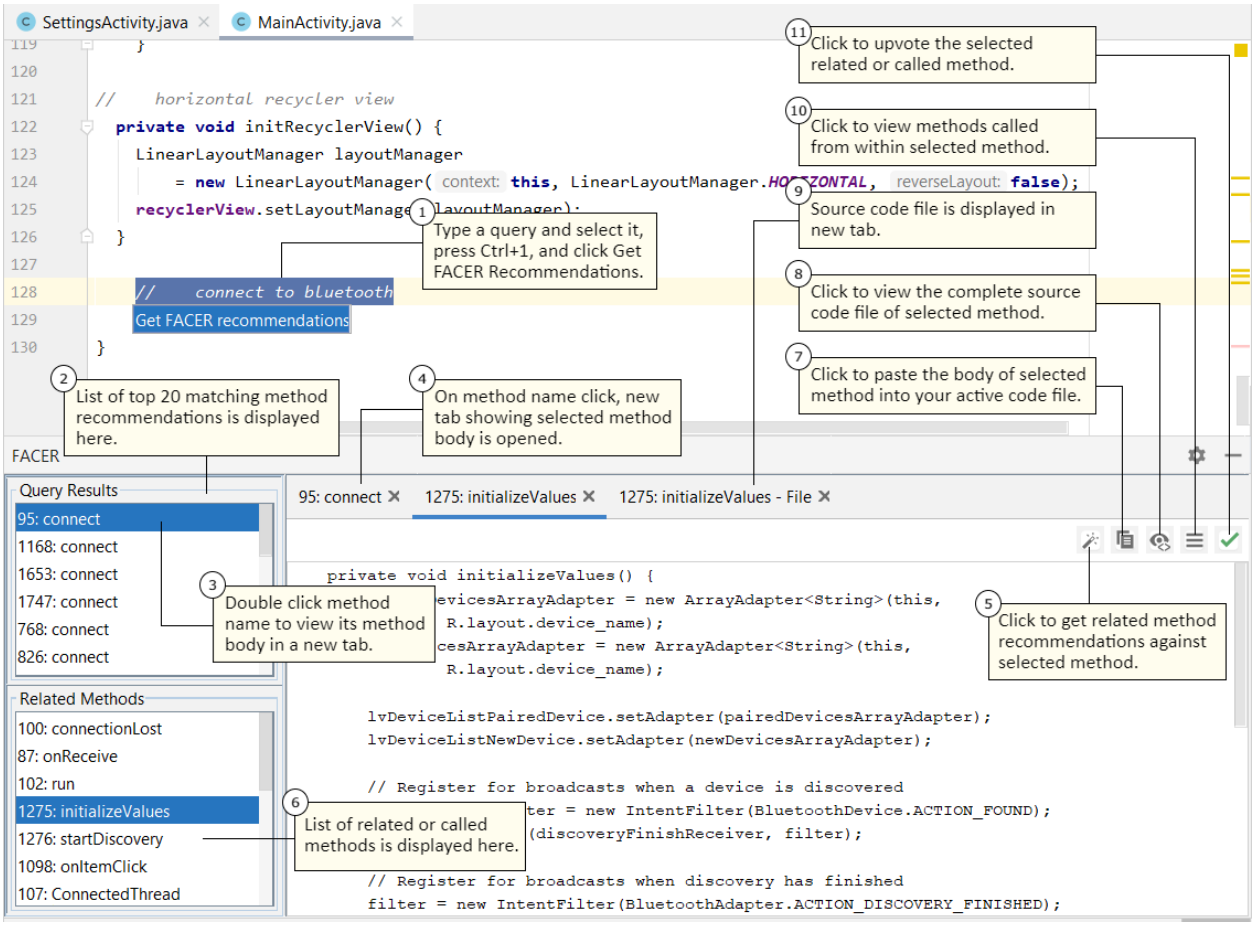


Figure A.1: FACER-AS Interface

Appendix B

Example Clone Group With Long Method Bodies

********** Cluster ID: 445 **********

Project ID:36

Method ID:2484

Method Name:copyDirectory

File Name: $F : /FACER\_2020/RawSourceCodeDataset/ClonedNew/file\_2bb85ea657e0a014ee3ad82095e4be85d1aafd71/android-filemanager-2bb85ea657e0a014ee3ad82095e4be85d1aafd71/src/com/example/myapp/ASyncCopyFiles.java$

```
public void copyDirectory(File sourceLocation, File targetLocation) throws
    IOException {
  if (sourceLocation.isDirectory() && sourceLocation.exists()) {
    if (!targetLocation.exists()) {
      targetLocation.mkdir();
    }

    String[] children = sourceLocation.list();
    for (int i = 0; i < children.length; i++) {
      copyDirectory(new File(sourceLocation, children[i]), new File(
      targetLocation, children[i]));
    }
  } else {
```

```
    InputStream in = new FileInputStream(sourceLocation);
    OutputStream out = new FileOutputStream(targetLocation);

    // Copy the bits from instream to outstream
    byte[] buf = new byte[1024];
    int len;
    while ((len = in.read(buf)) > 0) {
      out.write(buf, 0, len);
    }
    in.close();
    in = null;
    out.flush();
    out.close();
    out = null;
  }
}
```

Listing B.1: Method 2484 of Project 36 and Cluster 445 implementing the *copy directory* feature

Project ID:39

Method ID:2728

Method Name:copy

File Name:

$F : /FACER\_2020/RawSourceCodeDataset/ClonedNew$

$/file\_369c9cd4750d42ab4c3cc2e5a69cc37f3edd4dbe$

$/FileManager - 369c9cd4750d42ab4c3cc2e5a69cc37f3edd4dbe$

$/src/main/java/com/bmeath/filemanager/FileHelpers.java$

```
public static boolean copy(String[] paths)
{
  InputStream in;
  OutputStream out;
  File src, dst;
  String srcPath, dstPath;

  try {
    dstPath = paths[paths.length - 1];
    dst = new File(dstPath);

    for (int i = 0; i < paths.length - 1; i++) {
      srcPath = paths[i];
      src = new File(srcPath);

      if (src.isDirectory()) {
```

```
        if (!dst.exists()) {
          dst.mkdirs();
        }

        String[] files = src.list();
        // recursively copy all sub-items
        for (int j = 0; j < files.length; j++) {
          copy(new File(src, files[j]).getCanonicalPath(), new File(dst,
              files[j]).getCanonicalPath());
        }
      } else {
        in = new FileInputStream(srcPath);

        if (new File(dstPath).isDirectory()) {
          dstPath += File.separator + src.getName();
        }

        if (new File(dstPath).exists()) {
          dstPath = renameCopy(dstPath);
        }

        out = new FileOutputStream(dstPath);

        byte[] buffer = new byte[1024];
        int read;
        while ((read = in.read(buffer)) != -1) {
          out.write(buffer, 0, read);
        }

        in.close();
        out.flush();
        out.close();
      }
    }
    return true;
  } catch (FileNotFoundException fnfe) {
    fnfe.printStackTrace();
  } catch (Exception e) {
    e.printStackTrace();
  }
  return false;
}
```

Listing B.2: Method 2728 of Project 39 and Cluster 445 implementing the *copy directory* feature

Common API Calls:

File.isDirectory

File.exists

File.list

File.new

FileInputStream.new

FileOutputStream.new