University Institute of Lisbon

Department of Information Science and Technology

Type relationship graphs for exploring APIs

Henrique Manuel da Costa Gomes Ferreira

A Dissertation presented in partial fulfillment of the Requirements
for the Degree of
Master in Computer Science

Supervisor
André L. Santos, Professor Auxiliar
ISCTE-IUL

June 2017

"If I have seen farther than others, it is because I have stood on the shoulders of giants."

Isaac Newton

Resumo

Actualmente, o uso de APIs, com diferentes graus de complexidade, tamanho e estrutura, é inevitável no desenvolvimento de software. As APIs são usadas para comunicação, registar informação, desenhar gráficos entre uma miríade de outras funcionalidades. Com este aumento na importância do uso de APIs, os programadores são muitas vezes confrontados com vários problemas de usabilidade quando usam uma API pela primeira vez. Propomos uma abordagem que visa ajudar na mitigação destes problemas através de sugestões de código, obtidas a partir da informação proveniente de uma análise estrutural a APIs que tem como objectivo encontrar relações entre tipos. Estas relações são armazenadas em grafos que irão ser usados para navegar entre os diversos tipos da API. Os primeiros passos de um programador quando usa uma API são, numa boa parte das vezes, uma das fontes de dificuldade. Por esta razão, a nossa abordagem disponibiliza um conjunto de pontos iniciais da API, de forma a auxiliar o programador no uso inicial da mesma. A fase seguinte no uso de uma API normalmente consiste na utilização de tipos da API, de forma a aceder ou criar outros tipos da API. A nossa abordagem sugere possíveis composições de tipos com base na informação do contexto actual de desenvolvimento. Dependendo da API, a quantidade de relações extraídas pode ser avassaladora, se sugeridas directamente ao programador. Para conseguir oferecer sugestões significativas, foi criado um sistema de filtragem e uma heurística de ordenação. A nossa abordagem foi testada analisando 5 diferentes API e simulando sugestões através da informação extraída. Estas sugestões foram comparadas com exemplos de utilização destas APIs. Os resultados evidenciam que a nossa abordagem é uma solução possível para os problemas de usabilidade de APIs, e que a análise estrutural a uma API permite obter o conjunto de informação necessária para gerar e disponibilizar sugestões a programadores.

Palavras-chave: Usabilidade API, Sugestões de código, IDE, Análise estrutural.

Abstract

In the present day, the use of APIs, varying in complexity, size and design are pervasive, in the development of software. They are used to communicate, to log information, to draw charts and a myriad of other purposes. With this increasing importance in API usage, developers often struggle when using an API for the first time and are faced with various API discoverability problems. We propose an approach that aims to help mitigate the discoverability problems by providing code suggestions based on data extracted from structurally analyzing APIs for type relationships. The relationships are stored in a graph, which is used to navigate between the API types. The first steps when using an API are often one of the difficulties that developers face. For this reason, our approach provides API starting points to help the developer kick-start the use of the API. The next stage in API usage usually consists in using API types to access or create other API types. Our approach suggests possible type compositions based on the types available in the current development context. Depending on the API, the amount of extracted relationships can be overwhelming if directly suggested to the developer and for this reason a filtering mechanism and ranking heuristic were created, in order to provide more meaningful suggestions. Our approach was tested by analyzing 5 different APIs and simulating suggestions from the extracted data, comparing those suggestions to API usage examples. The results provide some evidence that our approach is a possible solution to the API discoverability problems and that the structural analysis to a given API can provide considerable amount of the information required to create and deliver suggestions to developers.

Keywords: API Usability, Code completion, IDE, Structural Analysis.

Acknowledgements

I would like to thank my family and friends for supporting me during this process and for always being inquiring about the progress of my dissertation. A special thanks to my mother, my father, my brother and his wife and to my girlfriend for having the patience to cope with me in this stressful times and for being a source of constant and valuable motivation. It's also important to recognize the education that was given to me by multiple people during my life, but specially from my parents, that made me be a persistent and goal driven individual. I would also like to thank my bosses for being understanding and supporting me in this venture.

Finally, a big thank you to André L. Santos for the time and wisdom provided throughout this process and for being always available to meet and discuss solutions and to evaluate progress, even with my crazy schedules. It's safe to say, that without him it wouldn't be possible to finish this work and I'm very grateful for all the help and interest that came from him.

Contents

List of Figures

List of Algorithms

List of Tables

List of Code Snippets

Chapter 1

Introduction

1.1 Motivation and contextualization

In the day to day work of a software developer there is often a need to use one or more Application Programming Interface (API). This need comes for one major reason, reduce the development and testing time. There is no need to "reinvent the wheel", if someone has had the same problem in the past and has developed code to solve that problem it's more efficient to use it instead of starting from scratch. However, using an API sometimes can be challenging, not just because the code wasn't written by the developer, but also because the tools available are not always helpful and, in some cases, there is no documentation available.

Most software developers use tools known as integrated development environments (IDE) to write their code. An IDE is a code editing tool that allows developers to code with some extra features such as syntax highlighting and code completion. Code completion is one of the most handy features of the IDE because, not only helps developers to write code faster but, more often than not, can be used to look for functions, methods and attributes of a given a class. Although this is useful in several cases, however, some APIs are developed using more complex code patterns or have a different code structure therefore, they are meant to be used differently and are not what most developers expect and look for when

using an API. For example, if a developer is using a reference *s* of type String in Java and types "s." (s followed by a dot), the IDE will suggest what operations or fields are available for this type. In this case two of the possible, and more common suggestions, would be the operations *s.length()* and *s.chartAt(index).*

If a given API has a static class [1] called *StringUtils*, that offers a myriad of static methods for string operations, the code completion mechanism will not be of much help. This is one of the types of operations that traditional code completion mechanisms won't be able to find and it's one of the main causes of the API discoverability problem. [8]

As stated before, APIs are often used by developers to solve problems efficiently and they require less time to use than to develop from scratch. However, the fact is that not all APIs have the same kind of design, and therefore, they are meant to be used in different ways.

The difference between APIs depends on several factors, some are different because the person who wrote the API is used to writing code in a different way than what's expected for a given programming language (developer used to programming language A creates API with programming language B). Others differ because their goals also differ (a API that helps developers handle Strings, like in the previous example, will be very different from an API that is meant to send emails), but mainly they differ because with the passing of time, new code writing idioms emerge, others become obsolete or even a new software design pattern [2] is introduced.

When most object oriented software developers, which are the vast majority of software developers, use an API, they are accustomed to use APIs in a certain way. For example, suppose a developer is trying to use an hypothetical API to send a tweet via Twitter and, for some reason, there is no documentation available but the developer tries to discover how to do it.

[1] A class that only has static methods.
[2] Reusable solution for a comment software design problem

What the developer could expect

```
1 Tweet t = new Tweet("I love APIs!!! <3");
2 t.send();
```

How the API would be used

```
1 Tweet t = TweetFactory.createTweet();
2 t.setMessage("I love APIs!!! <3");
3 TwitterNetwork.getInstance().sendTweet(t);
```

FIGURE 1.1: Distinct examples of API design for an hypothetical Twiter API

From the example from Figure 1.1 we can see two snippets of code that have the same purpose but are written in a different way. The problem resides in the fact that, not only the code is different (which is common, each developer has their code style) but, the underlying API design is also different.

The code that many object-oriented software developers will expect (Figure 1.1) will make use of object constructors (line 1) and object methods (line 2) which is very different from the actual TwitterAPI usage. The TwitterAPI uses a Factory Pattern [18] (software pattern that centralizes the creation of objects via a static class called factory) to create the Tweet object (line 1), then defines the message via a setter method and lastly, sends the tweet via a method (line 3) available in an object of the type TwitterNetwork which in turn was developed using a Singleton Pattern (pattern that only allows one instance of a given object to exist and be used).

Although the example described above is made up, this differences in API design occur often, two examples of this are the Java Mail API [4] and the Guava API (Google Java API).

1.2 Approach

In order to aid developers cope with the problems mentions in Section 1.1 when using APIs, a structural analysis will be executed per API to extract information

from the API Types. The extracted information will be used to produce API sentences. This structural analysis has no dependencies on code source, examples or documentation and it's a fully automated process.

The API sentences produced from the extracted API information should be provided to the developer alongside the existing mechanisms.

1.3 Research Questions

The main research questions are:

1. Is it possible to create valid API Sentences [3] based on a structured analysis?
2. Is it possible to accurately identify the most likely starting points (API type) from a given API?
3. Can an automatic system meaningfully rank a set of possible API Sentences base on structural analysis?

1.4 Objectives and research methodology

A stated before, the goal is to aid the developer discover how an API is used by providing API sentences generated from the information extracted from a structural analysis to the API.

The development is divided in three phases, the first is to create and implement an algorithm that analyses APIs and extracts the relationships between API types and stores them in a graph. The second phase consists on creating a filtering mechanism and a ranking heuristic. The filtering mechanism filters out relationships that are possible given a context. The ranking heuristics takes the filtered relationships and sorts them by relevance.

[3] Chain of code instructions that make use or create API types

To test the algorithm, the filtering mechanism and the ranking heuristic, different APIs, with known usability issues, will be used.

The third, and final phase of the development process, consists of making a detail analysis of several APIs with a set of metrics and comparing the results with real-world examples for each API.

1.5 Document Structure

The remainder of this document is structured as follows:

- Chapter 2 analyses similar work that either validate the existence of the discoverability problem, support our approach to solve the problem or provide a tool that attempts to solve the problem.
- Chapter 3 explains how the API Type relationship graph is constructed to support our approach.
- Chapter 4 describes the analysis done to several to several APIs and compares the results with real usage examples and analyses the results.
- Chapter 5 presents our conclusions, discusses the shortcomings of our research, and outlines future developments.

Chapter 2

Related Work

Some work and research has been made in the area of API discoverability and this approach is based on some of that work. The work can be divided into three areas:

- Understanding of API Usability/Discoverability
- Measuring API Usability
- Tools or mechanisms to improve API usage and discoverability.

The end of this chapter also include a brief explanation of the different types of Graphs.

2.1 Understanding of API Discoverability

One of the projects that aimed to understand why APIs are easy or hard to learn, placed programmers, with various levels of experience, to execute the same set of tasks using the same API (ABEL – A Better EiffelStore API). Due to the ambiguous nature of usability the way used to measure the results was via comparison of a given user result with his expectation. The testers followed thinking-aloud protocol [1] in order to better evaluate their experience. This protocol consists

in having testers verbally report their mental process as it develops, this will also include doubts and questions that testers face. [10]

The usage of thinking-aloud protocol allows researchers to follow the mental process of the tester and better understand the way he thinks and acts. Instead of just having data indicating success or failure in a given task, this method provides extra information that can be analyzed. For instance, without this methodology, if most developers think of doing the same think when executing a task and don't succeed at first and move on to another strategy, it would be hard to understand that, although they were successful, via another strategy, that was not their initial approach.

Another approach to the understanding of the difficulty of learning APIs was made by Martin Robillard in [11] where a series of interviews to Microsoft .NET developers where conducted in order to gather information about their experiences learning APIs and identify specific areas where the problem could reside.

The last example is from the research done by Ekwa Duala-Ekoko and Martin P. Robillard [4] where, in a similar way to [10] , a group of testers attempted to use two unfamiliar APIs in two tasks. The first task (Chart-Task) objective was to create a pie chart using JFreeChart API and save the chart to a graphic file format. The second task (XML-Task) consisted of using JAXP API to create a solution that given a xml file and a xml schema would return true if the file was in conformity with the given schema. Both this tasks could be challenging due to the way the objects are created in the APIs. The testers had documentation and access to the web for examples of usage. As expected, due to the architecture of the APIs, completing the tasks wasn't straightforward. Even with the help given by the IDE, the documentation and online examples, some had problems using the APIs be it in the creation of objects or in handling execution failures.

The research from [3, 5, 8, 20, 18] show examples of patterns or internal API architectures that, using the available tools, can prove to be hard to learn and will be used as examples in the development and testing phases alongside with other APIs. One of them as stated before is the FactoryPattern, that only allows the

creation of object instances via a static class called Factory. [19] This pattern can make the discoverability process harder because it's not what developers usually search for. [3]

2.2 Measuring API Usability

In a similar way to the works above, Thomas Frill et al [6] created a heuristic evaluation to identify problems with the APIs used for the tests and cross reference that information with the feedback from testers that used the APIs. Some problems where identified both by the heuristic algorithm and the testers however the problems related to runtime or user performance issues weren't detected by the heuristic algorithm due to their nature.

Thomas Scheller, Eva Kühn created a different approach to API usability measurement, their objective was to measure usability in an automated and objective way. Given an API usage example, their algorithm can generate a value that symbolizes the complexity of that API example. This value can in turn be compared to another API usage and conclusions can be made regarding the complexity of the API. The developed work takes into account if it's a low-level concept (LC), like calling a method or instantiating a class or if it's a high-level concept (HC) that are harder to detect and measure, like a Factory. For the LCs the algorithm calculates the complexity based on the action and its properties and for the HCs the algorithm must first recognize the HC at hand, and only then calculate the complexity. [16]

2.3 Tools or mechanisms to improve API usage and discoverability

Most of the methods of improving API usability and in turn discoverability are linked to the development process. Most of the studies indicates how it's possible

to improve usability while developing the API but those conclusions can also be used for generating code completions and to improve usability and discoverability of complex APIs.

The code completion mechanisms have a huge part to play in a software developers life, not only because it helps developers face their problems but also because it does this without the need to open a browser or consult documentation elsewhere, the experience is fully integrated with the environment that the developer uses on a day to day basis. [9, 15]

This importance is not environment or language specific, it can be observed across a broad array of languages and IDEs. Although not all platforms provide the same results and have different methods to get those results, the goal is the same: Improve discoverability. But sometimes, due to architecture or development choices made in the development stage of an API can render those tools inapt or even obsolete. [14]

The work in [3] is, of all the works referenced, the most similar. The goal is the same, improve discoverability and the approach is similar also. They started by developing a graph algorithm that analyses type dependencies of a given API as shown is Figure 2.1.

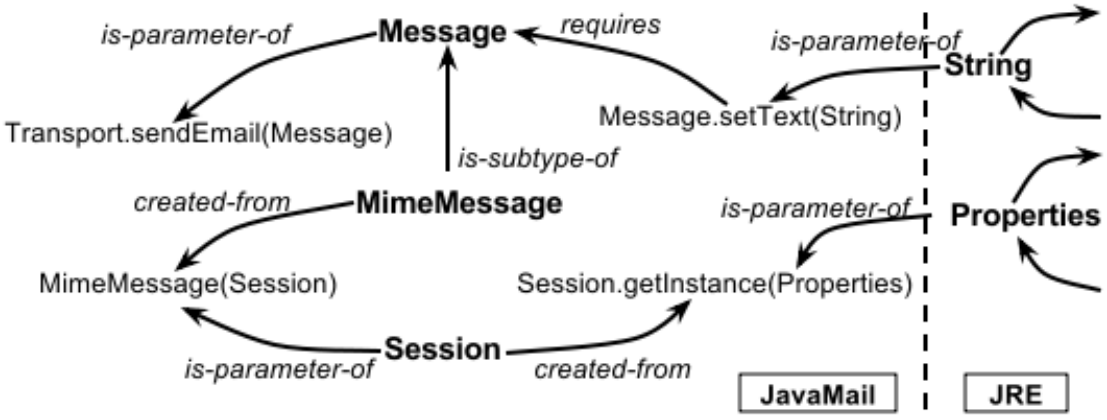

FIGURE 2.1: XGraph of simplified JavaMail API (extracted from [3])

After generating the graph for an API that information can be accessed via the autocomplete popup from the eclipse IDE that now includes suggestions for the type the user typed. An example of this mechanism is show in Figure 2.2.

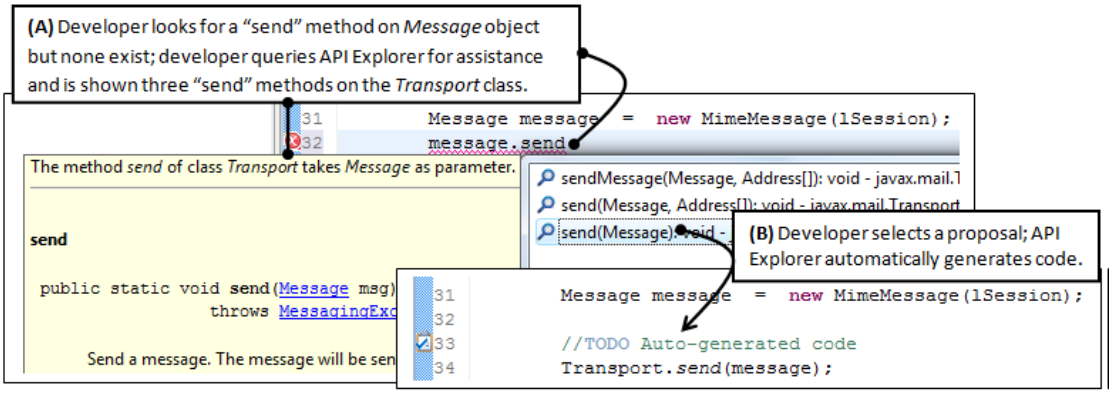

FIGURE 2.2: Example of API Explorer usage (extracted from [3])

To evaluate their mechanism in a real life context they used a case study methodology. The testers had to execute 4 different tasks, two of which, XML task and Chart task are similar to the ones used in [4]. The results show that the participants typically started exploring an API from the main-type before investigating the helper type, as was excepted, and then used API Explorer to access the helper type directly from the main type.

One example similar to this work is that of Mooty et al [7] where they develop a plugin for the eclipse IDE to improve construction of objects. The plugin adds suggestions to the code completions popup from the eclipse IDE as shown in Figure 2.3.

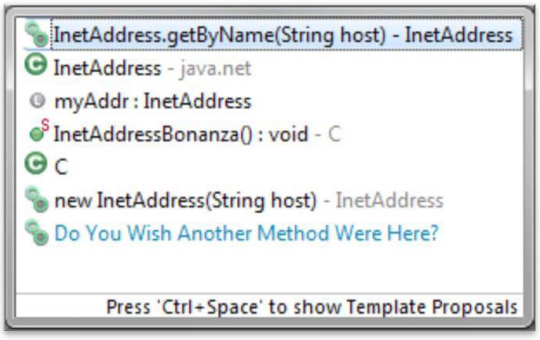

FIGURE 2.3: Example of eclipse popup using Calcite (extracted from [7])

After development, the plugin was tested by 10 participants and the results show improvements on the number of tasks completed by the participants and in the time they took to complete those tasks. This approach is similar to the second phase of this project but focuses on the creation of objects (via constructors and factories) and the code completion suggestions are gathered from code examples and not from the actual API.

The work from [17] aims to improve the quality of the existing code completing features on actual IDEs by enhancing the results list. Instead of displaying the code completions alphabetically the goal was to order them by relevance in the context. To calculate the relevance, the designed system learns from existing code repositories. The authors used the information from code repositories in three different ways:

- A frequency based code completion
  - The more used methods in the example code are the more relevant
- An association rule based code completion
  - Machine learning technique to find associations among items in the data Example: after a Message object is created, setText setter method usually follows
- The best matching neighbors code completion
  - Modification of the k-nearest-neighbor (kNN) machine learning algorithm

After the development of all code completion systems in the Eclipse IDE the findings of the evaluation indicated that the intelligent code completion system(s) outperformed the standard one and in turn, can improve API usability and developer's productivity

Another way to improve usability and discoverability is to improve the documentation available for the API [2]. Jadeite is a Javadoc-like API documentation

that provides extra information to ease the discoverability process of the developer. This system allows users to create fake classes and methods to be displayed in the documentation with the annotations that they insert. The Figure 2.4 and Figure 2.5 show the documentation before and after the creation of one fake method (called send) to send a message using the JavaMail API. The send method doesn't exist but has been created in Jadeite to store information on how to send a Message. It also takes into account the most commonly used classes to improve the time the developer is searching for a class in the documentation and displays them in a list inspired by the tag cloud. Jadeite also stores examples of constructor implementations to improve the discoverability process of object type instantiation. The evaluations made to the system showed that developers are three times faster using Jadeite instead of the normal JavaDoc [2].

PasswordAuthentication	**requestPasswordAuthentication**(InetAddress addr, int port, String protocol, String prompt, String defaultUserName) Call back to the application to get the needed user name and password.
synchronized void	**setDebug**(boolean debug) Set the debug setting for this Session.

FIGURE 2.4: Documentation before Jadeite placeholder method (extracted from [2])

PasswordAuthentication	**requestPasswordAuthentication**(InetAddress addr, int port, String protocol, String prompt, String defaultUserName) Call back to the application to get the needed user name and password.
void	**send**(Message message) edit Placeholder method for sending a message
synchronized void	**setDebug**(boolean debug) Set the debug setting for this Session.

FIGURE 2.5: Documentation after Jadeite placeholder method send (extracted from [2])

A different approach is described in [13] where the discoverability problem is solved by training a N-Gram Language model with code snippets from of Github projects that use the same API. If the model is trained with sufficient data, it is

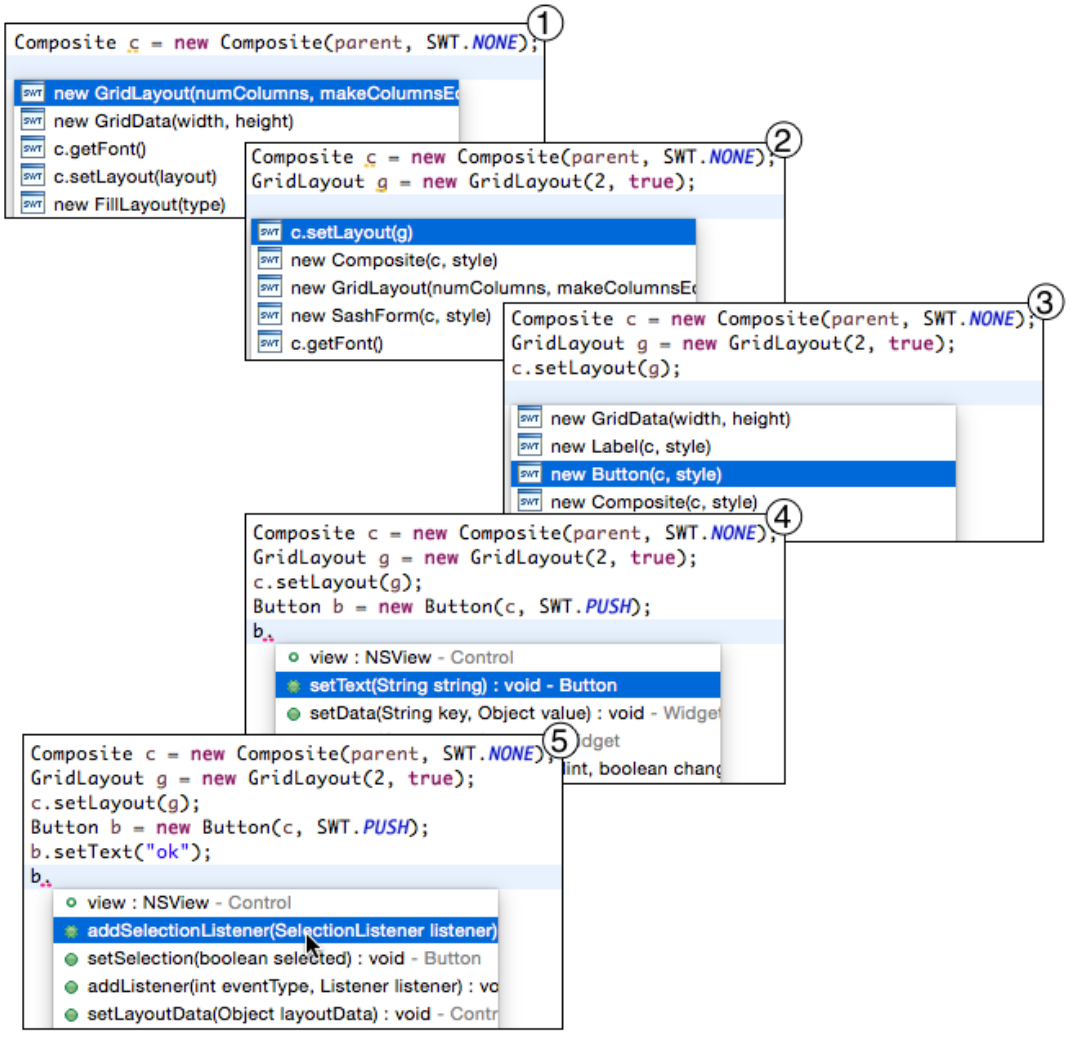

FIGURE 2.6: API Sentence generation example using N-Gram Language Models (extracted from [13])

possible to provide the developer with API Sentences based on the code examples for that API. The Figure 2.6 exemplifies a API Sentence creation using this approach.

Lastly, the work in [12] describes a different approach to solve this problem that focuses more in the development. This approach uses code annotations to complement the API development. This annotations are then processed by the IDE that assists the developer when using the API. The Figure 2.7 shows and example of an annotation for the Factory Pattern.

All the studies referenced above are an important aid for is work because they

```
/**
 * This class creates sockets. It may be subclassed by other factories,
 * which create particular subclasses of sockets. (...)
 */
@Factory
public abstract class SocketFactory {
  /**
   * Creates an unconnected socket.
   */
  @FactoryMethod
  public Socket createSocket(...) {...}

  @FactoryMethod
  public Socket createSocket(..., ...) {...}
  ...
}
```

FIGURE 2.7: Factory code annotation example (extracted from [12])

identify the most important API design failures and provide examples of tools to cope with those failures and in turn improve discoverability.

2.4 Graphs

A graph is a representation of a set of objects that are connected between each other by links. The objects are represented as vertices or nodes and the links that connect them are represented by edges or lines. Graphs Theory is the mathematical study of graphs.

In a graph, the edges can either be directed or undirected. For example, if nodes represent people and an edge represents a friendship between two persons then this would be a undirected graph (Figure 2.8), because if person A is friends with person B, then person B is also friends with person A. An example for an undirected graph is parenting relationship, where if person A and person B are connected, it means that person A is a parent of person B.

Another type of graph is called Multigraph, that allows for multiple edges, that is when two or more edges can be connected by multiple vertices in an undirected way. In this type of graphs it's possible to allow loops to occur. A loop is an edge that connects a vertex to itself. A multigraph that allows loops is often called a

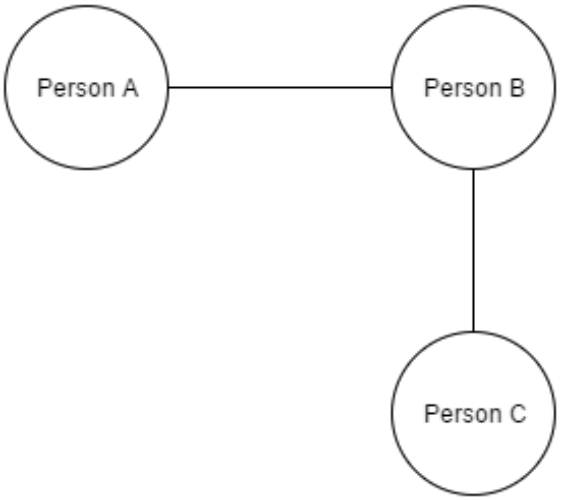

FIGURE 2.8: Simple undirected graph example

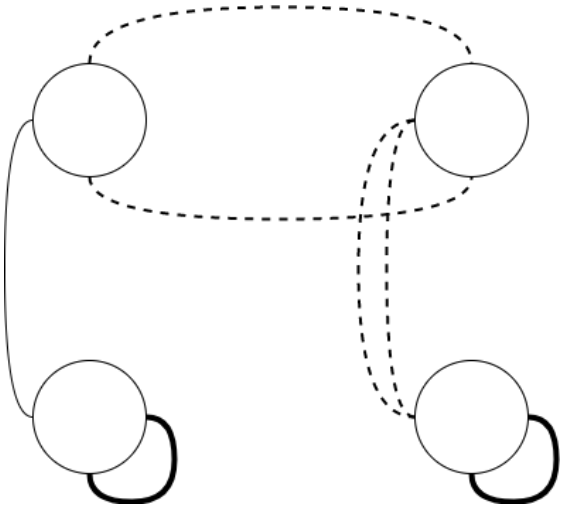

FIGURE 2.9: Multigraph example

pseudograph. The Figure 2.9 represents an example of a Multigraph with multiple edges (dashed lines) and loops (tick lines).

A quiver or multidigraph (or directed multigraph, or directed pseudograph) is a multigraph with directed edges that can allow for both multiple edges and loops.

More often than not there is a need to assign a weight to each edge and to do so Weighted Graphs are used. This weight can represent a distance, a cost or even a capacity depending on the problem at hand. These graphs are very commonly used in path solving solutions, where the weight represents the cost of traveling

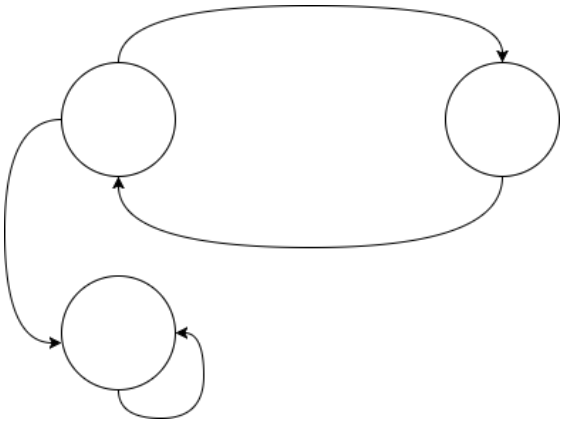

FIGURE 2.10: Directed graph example

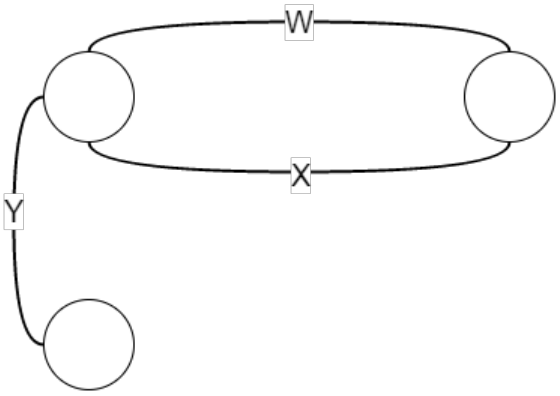

FIGURE 2.11: Weighted graph example

between two vertices, or even neuronal networks, to store the weight between two neurons (vertex).

The graph types described above are the most commonly used but, there are of course more variations.

Having all graph types in consideration, the multidgraph is the most appropriate graph type. This choice was made having into account 4 factors:

- Weighted
  - To store relationships we don't need a weight factor although we need to store the relation object itself.
- Directed

  - The relationships must have a direction. The direction indicates which type produces which.

- Loops

  - It must be possible to store a relationship where the source type and destination type are the same.

- Multiple edges

  - It must be possible to store multiple relationships between the same two types.

Chapter 3

API Type Relationship Graph

This chapter firstly shows a real example usage for the JavaX Mail API (known for it's discoverability problems) and dissects it in order to find possible problems the developer would face when using this API. The next section explains in detail how the API is analyzed for the relationship extraction and how the extracted information is stored. The last section elaborates on the filtering mechanism and ranking heuristics that take all extracted relationships and provide a subset of possible and meaningful relationships.

3.1 Running Example

Consider the Code Snippet 1 (written in Java) that sends an email message using a well known API, Javax.Mail.

Analyzing the Code Snippet 1, it's not hard to find possible problems a developer could face when attempting to use this API for the first time. This problems occur mostly due to the usage of different API code style and design patterns that the developer is used to. For instance, in order to create a MimeMessage object, it's necessary to provide a Session object, which in turn is not created in a straightforward OOP way (using a constructor). Instead, the Session object can be obtained via a static method in the Session type by providing a Properties

Code Snippet 1 Sending email message using Javax API

```
1  Properties properties = System.getProperties();
2  properties.setProperty("mail.smtp.host", "smtp.gmail.com");
3  Session session = Session.getDefaultInstance(properties);
4  try
5  {
6    MimeMessage message = new MimeMessage(session);
7    message.setFrom(new InternetAddress("source@gmail.com"));
8    message.addRecipient(Message.RecipientType.TO,
9          new InternetAddress("destination@gmail.com"));
10   message.setSubject("This is the Subject!");
11   message.setText("This email message");
12   Transport.send(message);
13 }
14 catch (MessagingException mex)
15 {
16   mex.printStackTrace();
17 }
```

object. If the developer is not familiar with the API, some time would be spent attempting to perform such a trivial task.

Another possible problem a developer could face writing this code, lies on the sending of the mail message, after it's properly initialized (from and to address, body, subject and SMTP [1] host). Most developers would expect to find a non-static method in the MimeMessage like so:

```
message.send();
```

Instead, they are faced with another static method, this one is implemented in the Transport API type. Experienced developers would easily find (via current IDE's auto completion mechanism) that the method is not available in the form explained above, but they would still struggle finding that the message is supposed to be sent via a static method available in the Transport type.

Consider now the simplified version of the UML diagram in Figure 3.1 for the the Code Snippet 1, that only shows the public types and operations by the JavaX Mail API.

[1] Simple Mail Transfer Protocol

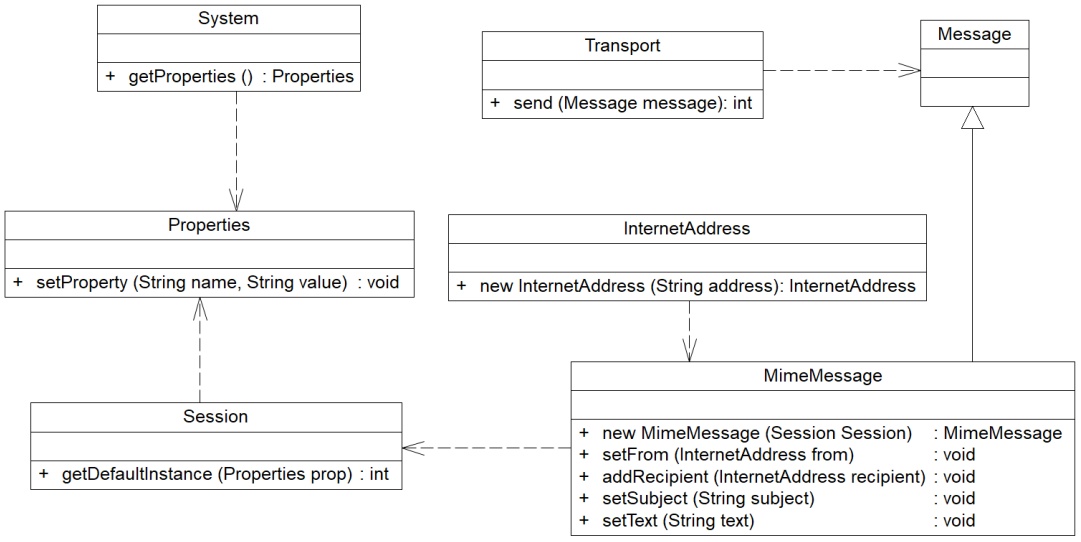

FIGURE 3.1: UML diagram of the API used in code snippet

3.2 Relationship Typology

We define a relationship to be as follows:

API type A is related/connect to API type B when it's possible to obtain an object of type B using an object of type A by means of an operation O. We can define Relationship to be O(A) → B.

Using the Javax Mail example:

Using an object of type Properties in the operation getDefaultInstance, it's possible to obtain an object of type Session. In this case the relationship could be defined as Session.getDefaultInstance(Properties) → Session.

In our approach, internal API types are significantly more important then external types taken into consideration when analyzing an API. Internal types are types belonging that are defined within the boundaries of the API. For example if we define the JavaX Mail API namespace to be javax.mail, all types found in operations in the API that don't belong in the javax.mail namespace are deemed external.

In our approach we classified relationships in 4 types of relationships:

- Parameter in a static operation
- Parameter in a constructor
- Parameter in a non-static (instance) operation
- Target instance in a non-static (instance) operation

3.2.1 Parameter in a static operation

A Parameter in a static operation relationship represents the usage of a parameter type in a static operation defined in the API. For every parameter in a static operation, one of these relationships should exist. The type that defines the operation isn't necessarily the return type of the operation.

Example #1:

```
Transport.send(message);
```

FIGURE 3.2: Graph representation - Parameter In Static Operation Example # 1

In the example #1, a Message type parameter is sent to the static method "send" of the Transport type. The resulting relationship would relate the Message type to a void type, since this is a void operation.

Example #2:

```
Session session = Session.getDefaultInstance(properties);
```

In the example #2, a Properties type parameter is sent to the static method "getDefaultInstance" of the Session type. The resulting relationship would related the Properties to the Message type.

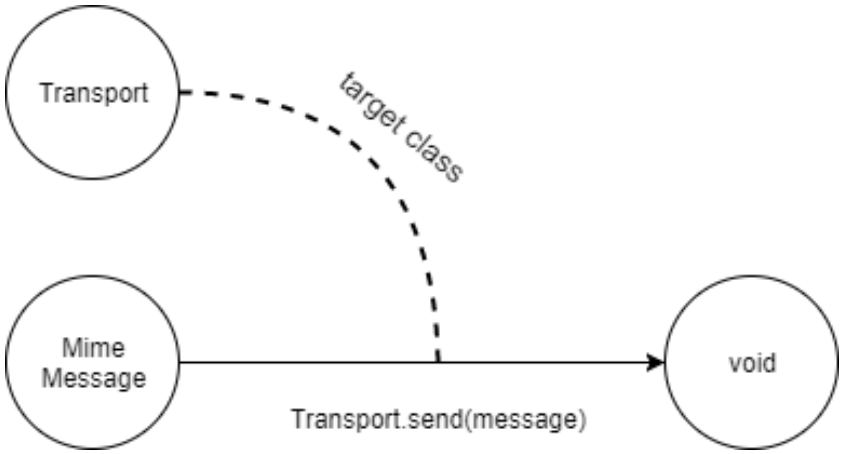

FIGURE 3.3: Graph representation - Parameter In Static Operation Example # 2

One relationship should exist for each of the parameters in a given operation, in both cases, the operations only receive one parameter so, only one relationship should exist.

3.2.2 Parameter in a constructor

A Parameter in a constructor operation relationship represents the usage of a parameter type in a constructor operation defined in the API. For every parameter in a type constructor operation, one of these relationships should exist.

Example:

```
MimeMessage message = new MimeMessage(session);
```

FIGURE 3.4: Graph representation - Parameter In Constructor Example

In the example above a Session type parameter is sent to the MimeMessage type constructor operation. Since this is a constructor operation, the operation is

always defined in the return type and the operation name is the same of the type. The resulting would relate the Session type to a MimeMessage type.

One relationship should exist for each of the parameters in a given constructor, in this case, the operation only receives the Session type so, only one relationship should exist.

3.2.3 Parameter in a non-static (instance) operation

A Parameter in a non-static operation relationship represents the usage of a parameter type in a non-static operation defined in the API. For every parameter in a non-static (instance) method, one of these relationships should exist.

Example #1:

```
message.setSubject("This is the Subject!");
```

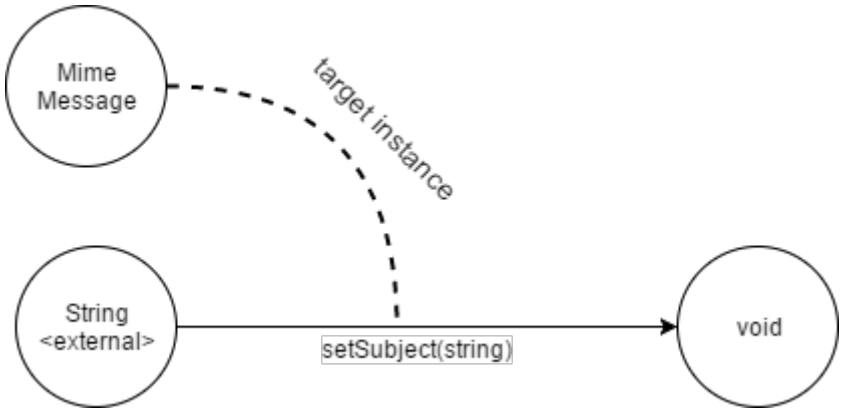

FIGURE 3.5: Graph representation - Parameter In Non-Static Operation Example # 1

In the example #1 a String type parameter is sent to the the "setSubject" method of a MimeMessage instance. The resulting relationship would relate the String type to a void type.

Example #2 (using the JavaX XML API):

```
//Supposing factory is an object of type SchemaFactory
Schema schema = factory.newSchema(new File(...));
```

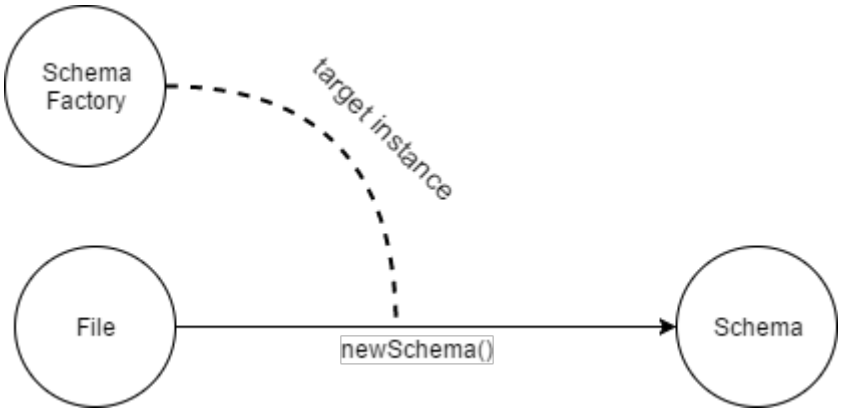

FIGURE 3.6: Graph representation - Parameter In Non-Static Operation Example # 2

In the example #2 a File type parameter is sent to the the "newSchema" method of a SchemaFactory instance. The resulting relationship would relate the File type to a Schema type.

One relationship should exist for each of the parameters in a given instance operation, in this case, only one operation should exist.

3.2.4 Target instance in a non-static (instance) operation

A Target instance in a non-static operation relationship represents the usage of the target instance type when calling a non-static operation defined in the API. For every non-static (instance) method, one of these relationships should exist.

Example #1:

```
message.setSubject("This is the Subject!");
```

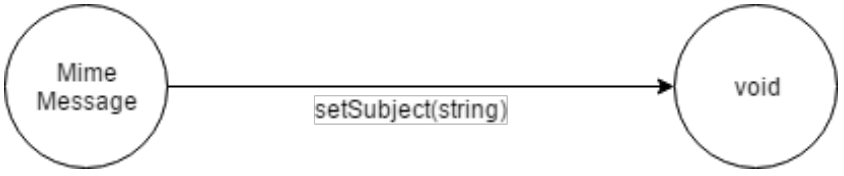

FIGURE 3.7: Graph representation - Instance In Non-Static Operation Example # 1

In the example #1 the method "setSubject" is called using the object Message. The resulting relationship would relate the MimeMessage type to void type, since this is a void operation.

Example #2 (using the JavaX XML API):

```
//Supposing factory is an object of type SchemaFactory
Schema schema = factory.newSchema(new File(...));
```

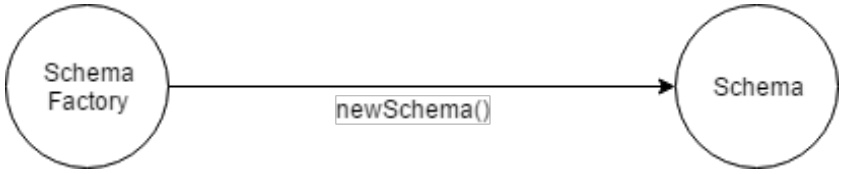

FIGURE 3.8: Graph representation - Instance In Non-Static Operation Example # 2

In the example #2 the method "newSchema" is called using the object of type SchemaFactory. The resulting relationship would relate the SchemaFactory type to Schema type.

One relationship should exist for each non-static (instance) method.

3.2.5 Summary

The following figure 3.9 describes all the relationship types and their dependency to the operation properties in a conceptual model (UML class diagram).

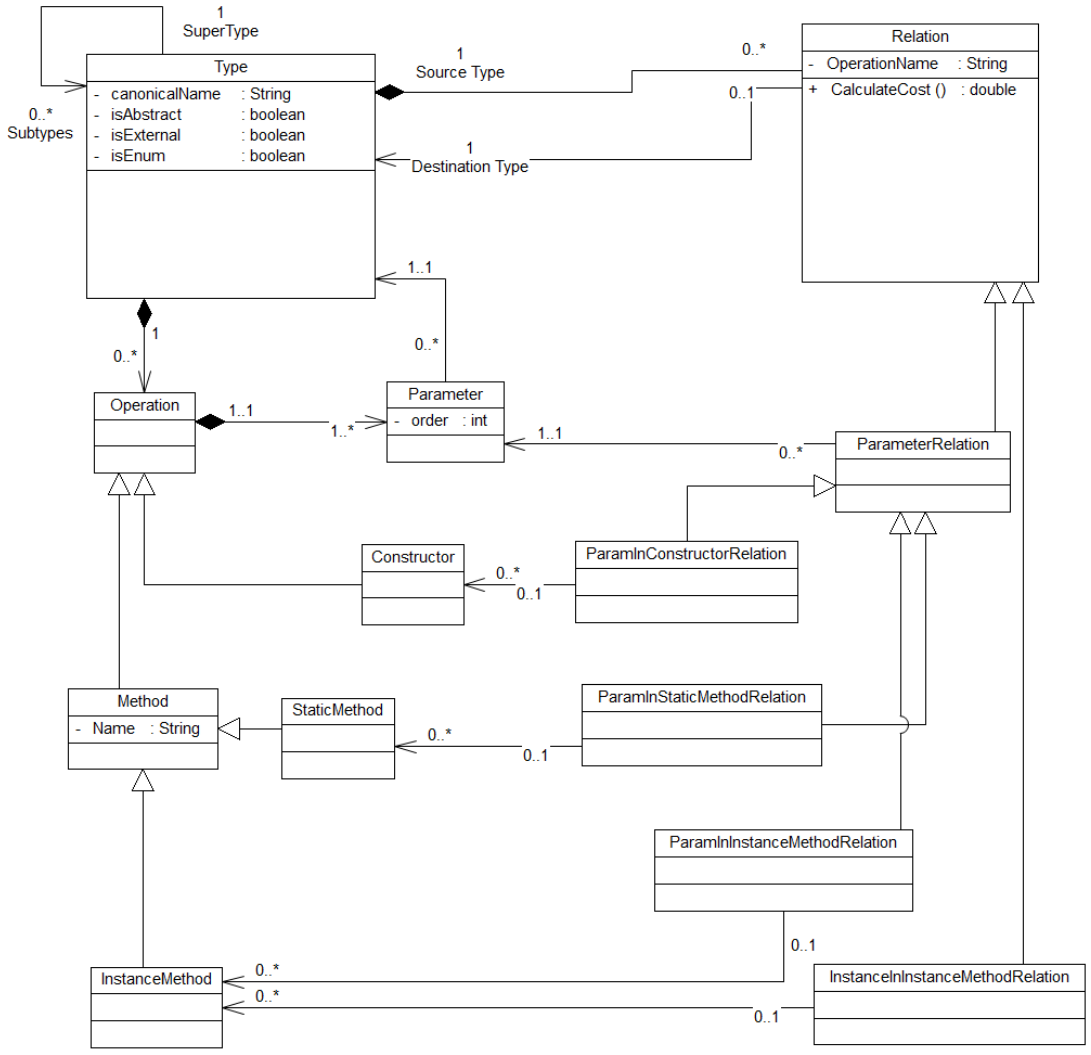

FIGURE 3.9: UML diagram of relationship types

3.3 Relationship information/details

In addition to the identification of the relationship type, we also collect all the information required to:

1. Distinguish relationships from each other (specially important when polymorphism is used)

2. Derive a code expression that encodes the use of the operation

The relationship is comprised of the following attributes:

- **Operation name**
  - Specifies the method name. In the constructor case, this is not relevant, since constructors are unnamed.
- **Source type**
  - Specifies the type that can produce the destination type. Either a parameter or an instance, depending on the relationship type
- **Intermediary type**
  - Specifies the type where the operation is available. which might be the same of the destination type.
- **Destination type**
  - Specifies the type that is produced by the relationship.
- **Parameters**
  - A list of parameters types required for the operation
- **Internal Parameters**
  - A list of the types of the internal parameters (parameters in namespace) required for the operation

The best data structure to store relationships is Graphs and, as referenced before in subsection 2.4 the graph type best suited for storing type relationships is a Pseudograph because it connects nodes using directed edges, allows multiple edges between the same two nodes and also allows loops (edge that connects node to itself).

With the information stored above it's possible to fulfill the two requirements introduced earlier. For better understanding the purpose of this information, the following subsection will use the previous examples and the all of the previously mentioned relationship types and their corresponding graph representation.

3.3.1 Parameter in a static operation

Example #1:

```
Transport.send(message);
```

Example #2:

```
Session session = Session.getDefaultInstance(properties);
```

TABLE 3.1: Describes in detail the relation object's attribute values for a Parameter used in a static operation relationship

Attribute	**Value for Example # 1**	**Value for Example # 2**
Operation name	send	getDefaultInstance
Source type	javax.mail.Message	Properties
Intermediary type	javax.mail.Transport	javax.mail.Session
Destination type	void	javax.mail.Session
Parameters	[javax.mail.Message]	[Properties]
Internal Parameters	[javax.mail.Message]	[ ]

There are two main differences between the data from the two examples. The destination type where, the Example #1 as no destination type (void) and the Example #2 outputs a Session API type and the internal parameters attribute where Example #2, although it has parameters, it doesn't use have any the internal API types as a parameter therefore, the internal parameters attribute has no types.

3.3.2 Parameter in a constructor

Example:

```
MimeMessage message = new MimeMessage(session);
```

TABLE 3.2: Describes in detail the relation object's attribute values for a Parameter used in a type constructor relationship

Attribute	**Value**
Operation name	- - -
Source type	javax.mail.Session
Intermediary type	javax.mail.MimeMessage
Destination type	javax.mail.MimeMessage
Parameters	[javax.mail.Session]
Internal Parameters	[javax.mail.Session]

Considering this is a constructor operation, the operation name is not applicable and therefore, is empty. The destination type and intermediary type of a constructor operation are always equal.

3.3.3 Parameter in a non-static (instance) operation

Example #1:

```
message.setSubject("This is the Subject!");
```

Example #2 (using the JavaX XML API):

```
//Supposing factory is an object of type SchemaFactory
Schema schema = factory.newSchema(new File(...));
```

TABLE 3.3: Describes in detail the relation object's attribute values for a Parameter used in a non-static operation relationship

Attribute	**Value for Example #1**	**Value for Example #2**
Operation name	setSubject	newSchema
Source type	String	File
Intermediary type	javax.mail.MimeMessage	javax.xml.validation.SchemaFactory
Destination type	void	javax.xml.validation.Schema
Parameters	[java.lang.String]	[java.io.File]
Internal Parameters	[ ]	[ ]

Both examples have no internal types as parameters, the main difference is in the destination type attribute. The Example #1 produces no type and the Example #2 produces a Schema.

3.3.4 Target instance in a non-static (instance) operation

Example #1:

```
message.setSubject("This is the Subject!");
```

Example #2 (using the JavaX XML API):

```
//Supposing factory is an object of type SchemaFactory
Schema schema = factory.newSchema(new File(...));
```

TABLE 3.4: Describes in detail the relation object's attribute values for a Instance used in a non-static operation relationship

Attribute	**Value for Example #1**	**Value for Example #2**
Operation name	setSubject	newSchema
Source type	javax.mail.MimeMessage	javax.xml.validation.SchemaFactory
Intermediary type	javax.mail.MimeMessage	javax.xml.validation.SchemaFactory
Destination type	void	javax.xml.validation.Schema
Parameters	[java.lang.String]	[java.io.File]
Internal Parameters	[ ]	[ ]

The attributes for this relationship type have similar information of those referenced in subsection 3.3.3 however they differ on the Source type. In this relationship type, the source type represent the target instance type so it's always equal to the Intermediary Type.

3.4 Relation Extractor Algorithm

The Algorithm 1 describes how the extraction engine works. Every Relationship created represents an edge in the graph, a connection between two types, this step

is represented by the function *CreateEdge* which is responsible for the Relation creation process. When creating the relationships, the algorithm will retrieve all the information indicated in Section 3.3. This function receives the graph where the information edge will be stored, and the source and destination type, as well as the relationship type.

The main function of the algorithm *Analyze* receives a namespace as defined by one or more packages. This namespace allows to determine if a given type is internal or external to the API. For the running example in Section 3.1 the namespace would be defined as the package *javax.mail*.

Firstly, the relation extraction algorithm retrieves all public types for a given namespace and creates graph nodes for all of them. In the next step, the algorithm finds all the public operations in that type. These operations can either be methods or constructors. Each method can either be static or non-static, and for parameters types belonging to the API (internal types) found in given method, a relationship is created. If the method is static a *ParameterInStaticMethod* relationship is created, otherwise a *ParameterInAnInstanceMethod* is created. For all non-static methods, one *InstanceInInstanceMethod* relationship is created, representing the instance dependency for the method usage. The algorithm processes Constructor operations in a similar fashion. For each internal parameter in a constructor operation, one relationship *ParameterInConstructor* is created. The generated graph that combines all relationship types based on the Code Snippet 1 is shown in Figure 3.10.

3.5 Ranking and Filtering

One of the intended goals of the Type Relationship graph is to aid the developer via code completion when using the API. Most of the APIs, when analyzed will create a considerable amount of relationships which in turn makes the usage of this information harder. To address this issue it's necessary to create a filtering mechanism and ranking heuristics.

Algorithm 1 Relation extraction algorithm

```
1: function CreateEdge(graph, source, destination, intermediary, type)
2:     ...
3: end function
4: function Analyze(namespace)
5:     graph ← NewGraph()
6:     for type ← GetPublicTypes(namespace) do
7:         CreateNodeForType(graph, type)
8:     end for
9:     for type ← GetPublicTypes(namespace) do
10:        for method ← GetPublicMethods(type) do
11:            for parameter ← GetParameters(method) do
12:                if IsInternal(typeOf(parameter)) then
13:                    if IsStatic(method) then
14:                        CreateEdge(graph, typeOf(parameter), GetReturn-
    Type(method), type, ParameterInStaticMethod)
15:                    else
16:                        CreateEdge(graph, typeOf(parameter), GetReturn-
    Type(method), type, ParameterInInstanceMethod)
17:                    end if
18:                end if
19:            end for
20:            if ! IsStatic(method) then
21:                CreateEdge(graph, type, GetReturnType(method), type, In-
    stanceInInstanceRelation)
22:            end if
23:        end for
24:        for constructor ← GetConstructors(type) do
25:            for parameter ← GetParameters(constructor) do
26:                if IsInternal(typeOf(parameter)) then
27:                    CreateEdge(graph, typeOf(parameter), type, type, Parame-
    terInConstructorMethod)
28:                end if
29:            end for
30:        end for
31:    end for
32:    return graph
33: end function
```

The filtering mechanism's goal is to ignore all relationships that are not applicable in a given context. The context is defined by a set of types available to be used in an operation, in other words, the types of objects that the developer instantiated in the immediate context (lines of code that precede the current line). When starting out, the context is an empty set.

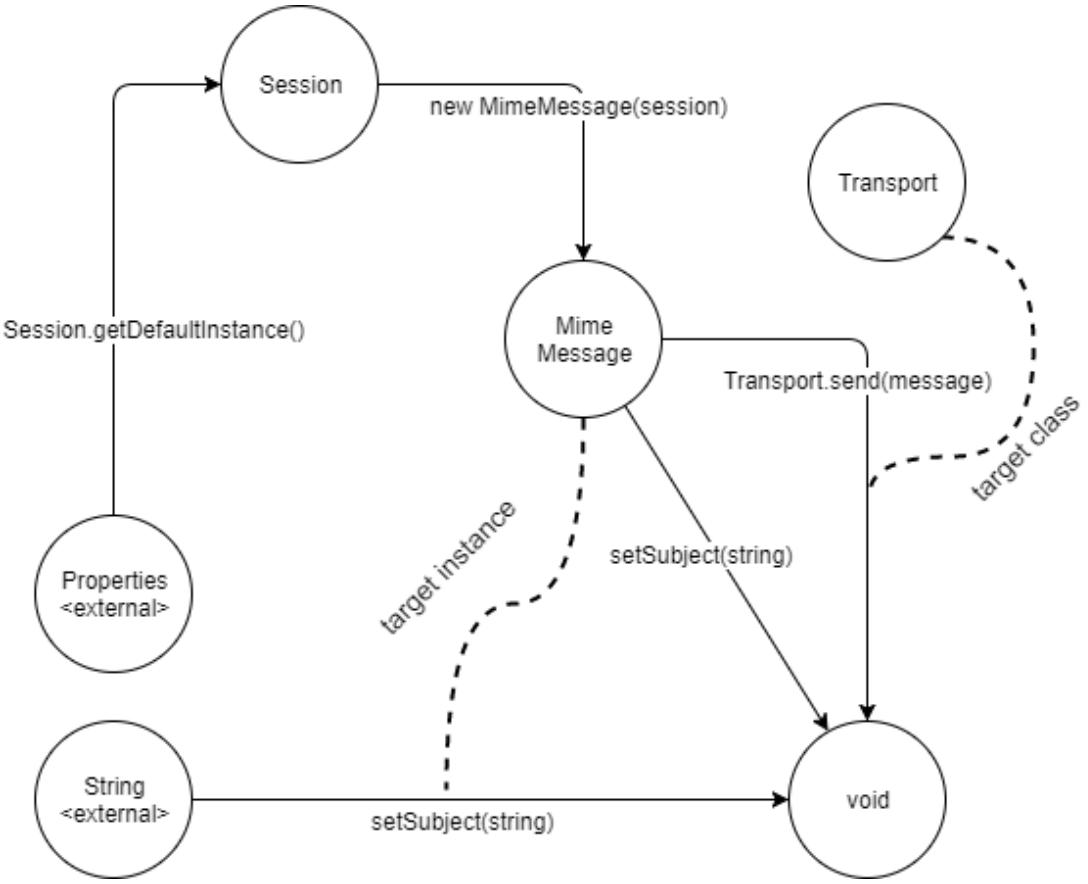

FIGURE 3.10: Generated graph for Code Snippet 1

Given the complete array of relationships, the filtering heuristic will discard all of those whose internal parameters are not all accounted for in the context. For example, if the developer is starting out using the JavaX Mail API, only the relationships that don't require any internal parameters are shown to the developer. All relationships that don't require the existence of an internal parameter from the context are called Starting Points.

The other use case the filtering heuristic, is when the context is not empty. Supposing that the context has the type javax.mail.Session, the available relationships would be the same as before, i.e. those that don't require internal parameters, plus the relationships that only require the javax.mail.Session type.

Even with the filtering heuristic, the amount of relationships can still be hard to use easily. Hence, after filtering all relationships that cannot be used, the ranking heuristic comes into play. The main goal of the ranking heuristic is to

help the developer coping with known API discoverability problems, as explained in the running example (Subsection 3.1) and in the Chapter 2. The goal isn't to help the developer with all tasks required to use an API, some of which are already addressed by IDE code completion mechanisms, but to complement those mechanisms.

The ranking heuristic purpose is to, given a subset of filtered relationships from the filtering mechanism, rank those relationships in a meaningful way, so that the most likely important relationships would be presented to the developer first. The ranking heuristic focuses on the following criteria:

1. Maximize API discoverability (API type access)
    - When exploring APIs one of the main problems is discovering which API types to use and how to instantiate them.
    - This criteria prioritizes relationships that lead to types that, when added to the context, increase the amount of possible relationships to use and in turn, maximize the number of API types that can be accessed.
2. Prioritize usage of types in the context
    - This criteria prioritizes relationships that use API types available in the current context. The goal is to use the previous choices made by the developer as information about their intent. The more recent types in context are likely to be more important than the older ones
    - The purpose of this criteria is to allow the developer to create a chain of instructions (API Sentence) that compose previous types into new types and so on.

When the developer starts using the API, the filter mechanism will only show relationships that can be used without the need of having internal API types. This small subset, of relationships without internal parameters, are the possible

Starting Points and are then passed on to the ranking heuristic. The ranking heuristic's second criteria is not applicable in this scenario because the context is empty at this point. The first criteria will emphasize the relationships that maximize API type discovery, in other words, explore relationships that, when used, allow for more relationships that weren't previously available.

After choosing the Starting Point, the developer will now use that type to explore the API in a process called Type Composition. This process can be defined by the usage of one or more API types in the context in an operation that leads to another API type, external type, or void (for void operations). This stage will have multiple iterations, depending on the API and it's the purpose of the second criteria to prioritize the usage of types in the context. Essentially the goal of the second criteria is to give importance to the previous choices of the developer when creating the API types that are now in the context.

The first criteria inspects the destination type of each relationship and queries the graph for all the outgoing relationships for that type. From this subset of relationships, the only information the criteria needs is the destination type when internal to the API, so they are extracted to a set of unique API types. Essentially, the criteria wants to know how many API types are made available when using this relationship. For each relationship an absolute number of API types are calculated and then compared to total number of API types. For example, if a relationship generates the API type Z which has 1.040 outgoing relationships (relationships originating from Z). Analyzing the destination type of these relationships, the total number of unique types is 49 and of those, the number of unique internal API types is 15. As stated before, this absolute measurement is converted to a relative one. Not only this allows both criteria to be compared side by side but also provide meaning to the measurement. It's not possible to assert that having 15 outgoing unique API types is a good or bad measurement because it depends on the API that is being tested. On the other hand, if we know that 15 types makes up 50% of the API, we know that having this type in this context, allows the developer to access 50% of all the types in API which is a considerable proportion. The only exception to this calculations are relationships regarding static operations

that return an external or primitive type (*void, int, String*, etc). This exception is only done to this type of operation due to difficulty to find such operation using standard IDE tools and also due to the fact that this operation doesn't return an internal API type, which in turn would result on constant value of 0%. For this reason these relationships are only compared using the second criteria. This change can be made due the scarcity of this relationships in the majority of the tested APIs. The following formula sums up the previous explanation of the first criteria.

$$G1(r) = \frac{\text{Number of outgoing unique types for the destination type of r}}{\text{Total number of API types}} \tag{3.1}$$

The second criteria focuses on the use of types in the context, giving more importance to the more recent ones. The criteria looks at the internal parameters required for a given relationship and relates those types to the ones present in the context at that time. The last API type in the context has the same weight as all the remaining types, that is 50%. The next one on the list has half the importance, 25%, the one following that one has half, 12,5%, and so on. Considering n to represent the index (zero-based, from the least recent to the more recent type) of the API type in the current context defined as c, we can calculate the gain for each relation (r) type present in the context used in a given relationship, defined as *G2*, to be calculated as such:

$$G2(r,c) = \sum_{n=0}^{\text{length of c - 1}} \left(\frac{1}{2}\right)^{l-n} \times \lambda \quad \text{, where } \lambda = \begin{cases} 1, & \text{if } c[n] \in \text{ParameterTypes(r)} \\ 0, & \text{if } c[n] \notin \text{ParameterTypes(r)} \end{cases} \tag{3.2}$$

The second criteria sums the calculated gains for all the relationship internal parameters present in the current context. As stated before, in order to combine both criteria they have to be in the same order of magnitude or at least the same unit of measurement. This as taken into account in both criteria. The conversion to relative values made by the criteria, solves the problem of interpreting the absolute values for each API.

The final rank for a given relationship is calculated by averaging both criteria calculations as described previously. This rank is used to sort the relationships, those that have the higher scores would be first on the list of suggestions to the developer.

$$G(r, c) = \frac{G1(r) + G2(r, c)}{2} \tag{3.3}$$

3.6 Implementation

The extraction algorithm was implemented in Java to test the validity of the approach described in the previous sections of this chapter. The initial attempt to retrieve the types was via a static analysis of the code, through inspection of their source code and visiting specific sections of the code, like method declaration, variable declaration, etc. This approach was not successful because it required access to the source code, which is not always available.

The adopted solutions was to extract the relationships between API types using Reflection. Reflection allows access to type information, such as method declaration, constructor declaration, type dependencies, just having the compiled code. This solution requires the API type to be defined and compiled, as it would be required if a developer intended to use the API in the first place.

To ease the extraction of information via Reflection, the API FastClasspath-Scanner [2] was used. The FastClasspathScanner API eases the Reflection tasks in Java for finding types in namespaces and avoiding possible errors. The code snippet 2 shows an example of how it's possible to extract classes from a given namespace using this library.

[2]https://github.com/lukehutch/fast-classpath-scanner

Code Snippet 2 Retrieving all types from a given namespace using FastClasspathScanner API

```
/*
Define the namespace, !! is used to include
System types if they are found in namespace
*/
String[] namespace = new String[] { "javax.mail", "!!"};
FastClasspathScannerscanner =
        new FastClasspathScanner(namespace);
ScanResult scanResult = scanner.scan();
// Retrieve all class names
List<String> classNames =
        scanResult.getNamesOfAllStandardClasses();
/*
Request Class objects
(true flag to ignore classpath errors)
*/
List<Class<?>> classes =
        scanResult.classNamesToClassRefs(classNames, true);
```

We made sure that all of the following situations were detected and correctly processed by the extraction process.

- Static operation in a different type with one or more parameters
- Static operation in the same type with one or more parameters
- Non-static operation in a different type with one or more parameters
- Non-static operation in the same type with one or more parameters
- Type constructor
- Inner types
- Internal and external parameters

We created tests aimed to assert that the extraction process was working correctly and that the information stored in the relationship is enough to provide the developer with all the information he would need if he wanted to use the operation.

As previously stated, the data structure best suited for storing relationships, and the one used in the Relation Extraction algorithm is a Graph. Instead of

implementing Graphs from scratch in Java, the API JGraphT[3] was used. The following Table 3.5 displays the comparison between some of the available graph types in the JGraphT API and it highlights the DirectedPseudograph which is the JGraphT's implementation of the multidgraph data structure, that was used in our implementation.

TABLE 3.5: Comparison of all graph types available in the JGraphT API

Graph Name	**Weighted**	**Directed**	**Multiple Edges**	**Loops**
SimpleDirectedGraph	NO	YES	NO	NO
DirectedMultigraph	NO	YES	YES	NO
DirectedPseudograph	**NO**	**YES**	**YES**	**YES**
DefaultDirectedWeightedGraph	YES	YES	NO	YES
DirectedWeightedMultigraph	YES	YES	YES	NO
DirectedWeightedPseudograph	YES	YES	YES	YES

The Code Snippet 3 shows an example usage of the JGraphT API for creating a simple graph with two vertices and one edge connecting them.

Code Snippet 3 Example of a creation of a simple graph using JGraphT API

```
1  // Define the namespace
2  DirectedPseudograph<String,String> graph
3          = new DirectedPseudograph<String,String>();
4  String luke = "Luke";
5  String anakin = "Anakin";
6  //Create nodes for luke and anakin
7  if (!graph.containsVertex(luke))
8          graph.addVertex(luke);
9  if (!graph.containsVertex(anakin))
10         graph.addVertex(anakin);
11 // connect them by creating an edge
12 graph.addEdge(anakin, luke, "Father");
13 //Get Outgoing edges of a node
14 System.out.println(graph.outgoingEdgesOf(anakin));
15 //Would output the relation Father of Luke
```

[3]http://jgrapht.org/

The next Chapter explains how the filtering mechanism and the ranking heuristic were implemented in a simulator. The performance of the simulator was measured by analyzing several APIs and comparing the suggestions with usage examples. Depending on the results of the simulator, it will be possible to prepare a future integration with an IDE where the suggestions would be provided to the developer directly in the development environment. The usability of this IDE integration would have to be tested with several developers and different APIs tasks.

Chapter 4

Evaluation

This chapter describes how our approach was evaluated. In order to evaluate the filtering mechanism and the ranking heuristic performance, a simulator was developed. This simulator was used to test 5 different APIs. The first section of this chapter describes on how the API Simulator works and the second section, presents the results of the analysis performed on the 5 different APIs. Each API will be tested by comparing the simulator results with an usage example extracted on-line from the official sources for each API.

4.1 API Simulator

In order to test the quality of the extracted data, the filter mechanism and the ranking heuristic, a simulator was developed. The simulator keeps track of the current context (set of API types previously created), and simulates a real world usage scenario where the developer would request the IDE for a code completion suggestion and the IDE would keep track of the current context for proposing those suggestions. When analyzing the simulator results and comparing them with usage examples, the index of the appropriate choice (according to the example) will be the basis for our analysis. Smaller indexes indicate that the appropriate choice was closer to the top of the suggestions. This index is the result of the ranking heuristic

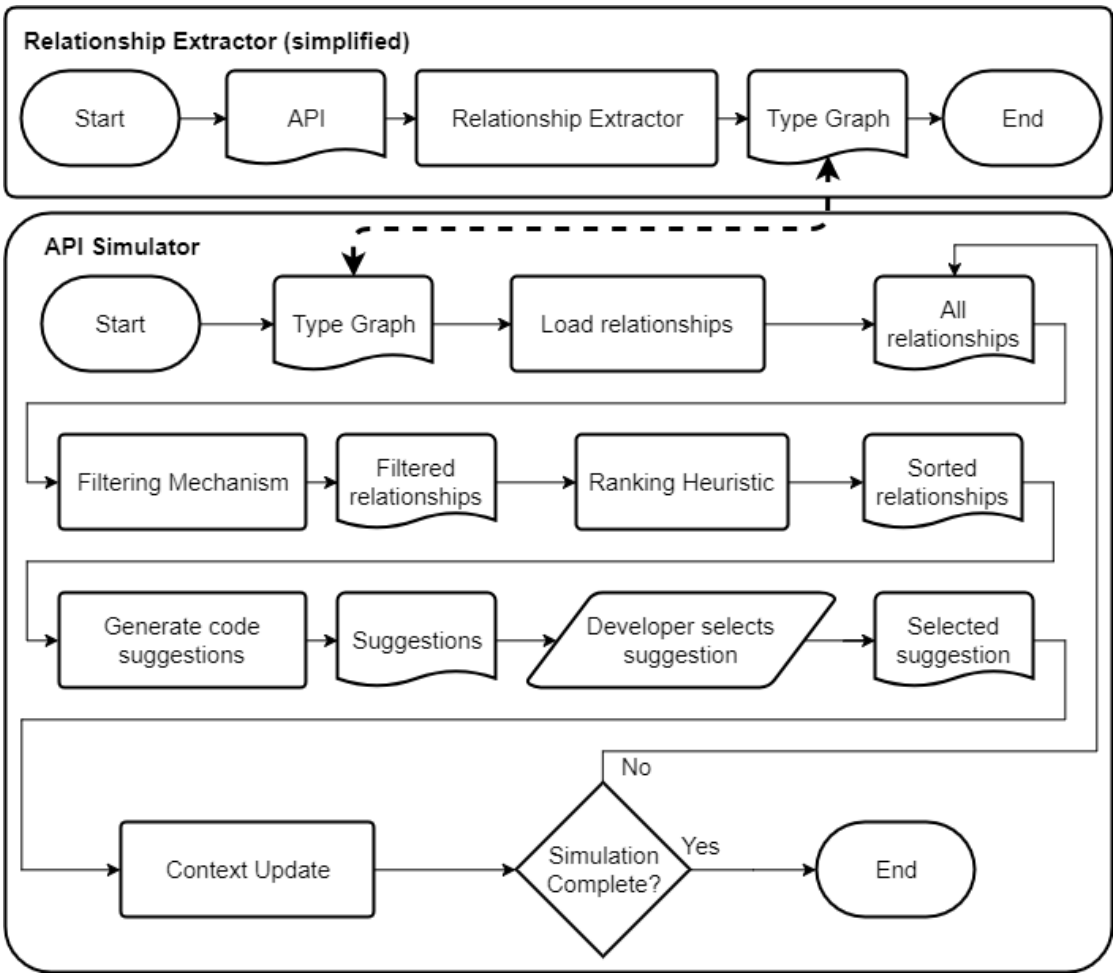

FIGURE 4.1: API simulator diagram

described in Chapter 3. The simulator takes the current context into consideration and filters the possible relationships the developer can use. After this, one may choose from a list of choices that were sorted using the ranking system described in Chapter 3. After the developer's choice, the simulator updates the context to include the type that was created on the previous step. The simulation repeats this process for a specific number of steps that can be changed. The simulator was started using the extracted graph for each API and the process described above was repeat until the usage example was reproduced or until the choices were deemed relevant. Figure 4.1 illustrates how the graph data is sent to the API simulator from the Relationship Extractor and describes the internal process in a simplified way. For each simulation step, the type graph is queried for relationships, which are filtered and ranked in order to provide code suggestions to the developer (who

```
1: Session session = Session.getDefaultInstance();
2: Session session = Session.getInstance();
3: Address address = new Address();
4: URLName uRLName = new URLName();
5: URLName uRLName = new URLName();
6: URLName uRLName = new URLName();
7: SearchTerm searchTerm = new SearchTerm();
8: Flags flags = new Flags();
9: Flags flags = new Flags();
10: ParameterList parameterList = new ParameterList();
11: ParameterList parameterList = new ParameterList();
12: MimeMultipart mimeMultipart = new MimeMultipart();
13: MimeMultipart mimeMultipart = new MimeMultipart();
14: MimeMultipart mimeMultipart = new MimeMultipart();
15: InternetHeaders internetHeaders = new InternetHeaders();
16: InternetHeaders internetHeaders = new InternetHeaders();
17: ContentDisposition contentDisposition = new ContentDisposition();
18: ContentDisposition contentDisposition = new ContentDisposition();
19: InternetAddress internetAddress = InternetAddress.parse();
20: InternetAddress internetAddress = InternetAddress.parse();
Choice:
```

FIGURE 4.2: API simulator screenshot

chooses on). If the choice made results in a new type, the context is updated. If the simulation is deemed complete, the process ends, otherwise another simulation step begins with the updated context. The simulator was developed in Java as a command line application as shown in Figure 4.2. The visual representation of the suggestions provided by the simulator was not designed to be accurate (missing parameters in operation call) but instead was designed to allow the identification of operations.

4.2 API Analysis

In this section the simulation of 5 different APIs will be analyzed and the the data will be compared with usage examples for that API. The test process for all APIs was identical. All usage examples (Hello World examples) were extracted from the official API sites. Each API analysis is divided into 4 sections:

- API description and usage example
- API type graph information analysis

- Simulation analysis
- Observations

4.2.1 JavaX Mail

The JavaX Mail API's main goal is provide cross platform email messaging functionalities to a Java application. The namespace used to define the API boundaries was *javax.mail*. All public types found in the namespace are deemed considered as API types by the relationship extraction process. The version 1.4.7 was used for the simulation.

TABLE 4.1: JavaX Mail API Type Graph Information

Measurement description	**Value**
Number of API Types	84
Number of relationships by relationship type:	
Parameter in a constructor	159
Parameter in static method type	57
Parameter in a non-static method	777
Target Instance in a non-static method	816
Total	1.809
Average number relationships per type	21,54
Outgoing graph edges:	
Average number of outgoing edges	10,04
Average number of outgoing edges (unique types)	3,42
Maximum number of outgoing edges	67
Maximum number of outgoing edges (unique types)	13
Incoming graph edges:	
Average number of incoming edges	3,81
Average number of incoming edges (unique types)	1,75
Maximum number of incoming edges	40
Maximum number of incoming edges (unique types)	11

Table 4.1 contains the information extracted from the API type graph. The first possible conclusions to extract from this data e indicates that the API is not very big in with respect to the number of the API types. The next piece of information in Table 4.1 is related to the amount of relationships of each type and the total amount of relationships. Regarding this information, it's possible to observe that the majority of relationships are non-static, either Parameter or

Target instance. The final two sections of Table 4.1 analyze the incoming and outgoing edges with respect to the average and maximum number of edges. To better understand these measurements, for both average and maximum, the number was also calculated considering the amount of unique types (not the amount of relationships). Analyzing the average data for both incoming and outgoing edges (unique types) it's possible to conclude that most of the API types connect to a small number of API types.

On an initial analysis to the JavaX Mail API there are 98 possible starting points, about 5% of all relationships in this API. Considering that this is not an API with a lot of types (84), 5% is a considerable amount of starting points for the developer to filter and select when developing with this API.

Table 4.2 shows the position of the API instructions in the simulator suggestions. The Context column of the Table 4.2 shows the value of the simulator context used to filter and rank the suggestions. The Index column of Table 4.2 indicate the index of the the instruction in the suggestions compared with the total number of suggestions. As it's possible to see in Table 4.2 the starting point that matches the usage example was first on the list and instructions that follow were also located in the top 3 positions of the suggestions list. This shows that the ranking heuristic managed to sort the relationships from the universe of possible relations at each state, also indicated in the Index column.

The instructions shown in Table 4.2, are highlighted in gray in Code Snippet 4 and were chosen because they are known API discoverability problems for this API [3, 12, 4]. Code Snippet 4 is an example usage of the API to send an email message via SMTP, this Code Snippet is the same as the one introduced in Chapter 3. The remaining instructions weren't the target of the analysis due to the lack of complexity involved in these instructions or because they relate to API types, like the Properties object created in line 1 of Code Snippet 4.

Considering all of the information provided above the simulation results are very positive. All the highlighted code instructions were able to be generated from the simulator's suggestions and the suggestion Index was low for all instructions.

Code Snippet 4 JavaX API Example: Sending email message using Javax API

```
1  Properties properties = System.getProperties();
2  properties.setProperty("mail.smtp.host", "smtp.gmail.com");
3  Session session = Session.getDefaultInstance(properties);
4  try
5  {
6    MimeMessage message = new MimeMessage(session);
7    message.setFrom(new InternetAddress("source@gmail.com"));
8    message.addRecipient(Message.RecipientType.TO, new
       InternetAddress("destination@gmail.com"));
9    message.setSubject("This is the Subject!");
10   message.setText("This email message");
11   Transport.send(message);
12 }
13 catch (MessagingException mex)
14 {
15   mex.printStackTrace();
16 }
```

Table 4.2: JavaX Mail API Usage Ranking Results

Line of Code	Index	Context
Session session = Session.getDefaultInstance();	1 / 98	-
MimeMessage message = new MimeMessage(session);	2 / 112	Session
Transport.send(message);	1 / 193	Session MimeMessage

4.2.2 JavaX XML Validation

The JavaX XML Validation API's main goal is to, given a XML file and a XSD schema file, validate the XML file against the XSD Schema. The namespace used to defined the API boundaries was *javax.xml.validation*. The version 1.4.2 was used for the simulation.

From the data in Table 4.3 it's possible to determine that this is a small API with only 15 types and a total of 144 relationships. Even with a small number of types and relationships, the task of validation a XML file with a XSD schema as been proven to a be difficult one for developers when using the API for the first time [12, 3], and this is the main reason to choose this API to be analyzed in this stage.

Table 4.3: JavaX XML Validation API Type Graph Information

Measurement description	**Value**
Number of API Types	15
Number of relationships by relationship type:	
Parameter in a constructor	10
Parameter in static method type	2
Parameter in a non-static method	66
Target Instance in a non-static method	66
Total	144
Average number relationships per type	9,6
Outgoing graph edges:	
Average number of outgoing edges	6,63
Average number of outgoing edges (unique types)	2,00
Maximum number of outgoing edges	18
Maximum number of outgoing edges (unique types)	4
Incoming graph edges:	
Average number of incoming edges	3,38
Average number of incoming edges (unique types)	1,63
Maximum number of incoming edges	10
Maximum number of incoming edges (unique types)	3

Looking at the data from the outgoing and incoming edges, it's possible to conclude that the average number of edges (both incoming and outgoing) are small. Essentially this indicates that when using a type from this API, this type will relate to a small number of other types. The types with the maximum number of unique type relationships for outgoing and incoming are only connected to 4 and 3 other types, respectively.

The results for the simulation in Table 4.4 are very promising. The number of possible starting points is small, 7 to be exact, which is excepted from an also small API. The starting point that matches the usage example in Code Snippet 5 was ranked first in the list of suggestions. The next two instructions were also ranked in the top of the suggestion list, in second and third respectively. As indicated in the Index column the number of total suggestions was always rather small, ranging from 7 to 17, comparing with other APIs. This occurred due to the filtering mechanism that filters out every relationship that can't be used, based on the current context, at each stage of the simulation.

The last instruction rank was higher than the previous, ranking 16 out of a possible 17. The high rank value can be explained by the void return value on this non-static operation of the *Validator* type. In our approach, relationships similar to this one, when compared to all the remaining relationships will most likely have a lower score in the ranking heuristic, due to the fact that the first criteria, explained in Section 3.5, will always rank with a value of 0% since there are no types made available by the void type. However, operations like this one are easily found using the existing IDE code completion mechanisms and are not the main focus of our approach.

Code Snippet 5 JavaX XML Validation API Example: Validate XML file

```
1  try
2  {
3    SchemaFactory factory = SchemaFactory.newInstance(...);
4    Schema schema = factory.newSchema(new File(xsdPath));
5    Validator validator = schema.newValidator();
6    validator.validate(new StreamSource(new File(xmlPath)));
7  }
8  catch (IOException | SAXException e)
9  {
10         System.out.println("Exception: "+e.getMessage());
11 }
```

Table 4.4: JavaX XML Validation API Usage Ranking Results

Line of Code	Index	Context
SchemaFactory factory = SchemaFactory.newInstance();	1 / 7	-
Schema schema = factorye.newSchema();	2 / 14	SchemaFactory
Validator validator = schema.newValidator();	3 / 11	SchemaFactory Schema
validator.validate();	16 / 17	SchemaFactory Schema Validator

Code Snippet 5 illustrates an example usage of this API where the XML file, loaded from the variable *xmlPath* is validated against an XSD file, loaded from the variable *xsdPath*.

With everything considered, the simulation to JavaX XML Validation API, described in Table 4.4 was able to reproduce the steps from the usage example

in Code Snippet 5 with low rank scores in the corresponding suggestions that, as stated before, have been identified as problematic for developers using this API.

4.2.3 JFreeChart

The JFreechart API's main goal is to create charts of several types and output those charts to final formats or display them inside a Java application. The namespace used to defined the API boundaries was *jfree.chart* and *org.jfree.data.general*. The version 1.0.13 was used for the simulation.

TABLE 4.5: JFreechart API Type Graph Information

Measurement description	**Value**
Number of API Types	544
Number of relationships by relationship type:	
Parameter in a constructor	634
Parameter in static method type	459
Parameter in a non-static method	23.998
Target Instance in a non-static method	23.998
Total	49.089
Average number relationships per type	90,24
Outgoing graph edges:	
Average number of outgoing edges	63,18
Average number of outgoing edges (unique types)	8,74
Maximum number of outgoing edges	689
Maximum number of outgoing edges (unique types)	62
Incoming graph edges:	
Average number of incoming edges	13,07
Average number of incoming edges (unique types)	3,96
Maximum number of incoming edges	935
Maximum number of incoming edges (unique types)	77

From the data in Table 4.5 it's possible to determine that this is a large API, with 544 API types and a total of 49.809 relationships. The vast majority of relationships are related to non-static operations. Analyzing the number of parameter in static methods indicates that there is a strong possibility to exist some sort of Factory or Utility type in the API.

Drilling down into the outgoing edges maximum and average values that consider only unique types, some conclusions can be drawn. From the maximum

outgoing edges (unique types), 62 to be more exact, it's possible to indicate that, at least one type connects to a large number of API types. This type or types are likely to have an important role when using this API in a myriad of different tasks. Analyzing the average number of outgoing edges, with a calculated value of 8,74, is a further indication that the maximum value of 62 is likely to be abnormal. The data for the incoming edges is similar and is consistent with these findings.

Table 4.6 contains the simulation results for this API that aims to replicate the Code Snippet 6. The Code Snippet 6 describes an example usage of this API where the a Pie Chart is created and exported to a PNG image file.

Due to the size of this API, be it in number of types or relationships, the possible number of starting points, 574, is very high. This is one of the main known discoverability problems that developers face when using this API. [3, 4] There are a lot of different ways to execute the same task in this API. For example, it's possible to create a JFreechart type (the main chart API type) via the ChartFactory with a dataset or create it directly via the constructor providing the appropriate plot type. This is one of the main reasons of the high rank value (140, for the starting point. This high rank value could be reduced in a future IDE mechanism with the addition of keyword filtering of suggestions. In this scenario, the developer, knowing that he or she wants a Pie Chart, he or she could type the keyword "pie" and the filtering mechanism would output have fewer suggestions.

After having the *DefaultPieDataset* type, even though the number of possible relationships increased, the rank value decreased drastically to 8 in the second code instruction. The decrease in value is a result from the second criteria of the ranking heuristic, that ranks relationships based on the context usage, as explained in Section 3.5. This positive behavior can also be observed in the last instruction where the appropriate suggestion was first on the list of suggestions.

In summary there were some issues regarding starting points in this API due to it's size and known structure problems, as previously explained. [3] The top starting points, of the possible 574, would also allow to execute the same task as the one in Code Snippet 6 but using different API operations. Therefore if the

Table 4.6: JFreechart API Usage Ranking Results

Line of Code	Index	Context
DefaultPieDataset pieDataset = new DefaultPieDataset();	140 / 574	-
JFreeChart jFreeChart = ChartFactory.createPieChart3D(pieDataset);	8 / 598	DefaultPieDataset
ChartUtilities.saveChartAsPNG(jFreeChart);	1 / 639	DefaultPieDataset JFreeChart

Code Snippet 6 JFreechart API Example: Create Pie Chart

```
1  //Prepare the data set
2  DefaultPieDataset pieDataset = new DefaultPieDataset();
3  pieDataset.setValue("Coca-Cola", 26);
4  pieDataset.setValue("Pepsi", 20);
5  pieDataset.setValue("Gold Spot", 12);
6  pieDataset.setValue("Slice", 14);
7  pieDataset.setValue("Appy Fizz", 18);
8  pieDataset.setValue("Limca", 10);
9
10 //Create the chart
11 JFreeChart chart = ChartFactory.createPieChart3D("Title",
12         pieDataset, true, true, true);
13 //Save chart as PNG
14 File file = new File(...);
15 ChartUtilities.saveChartAsPNG(file, chart, 400, 300);
```

starting point is not the same as the one in the Code Snippet 6, the developer would still be able to be assisted in this task. The remaining two steps show positive results with low index for the appropriate suggestions.

4.2.4 SWT

The Standard Widget Toolkit (SWT) provides access to cross platform user interface widgets to use in Java applications. The namespace used to define the API boundaries was *org.eclipse.swt.widgets*, *org.eclipse.swt.layout* and *org.eclipse.swt.graphics*. The version 4.3 was used for the simulation.

From the data in Table 4.7 it's possible to determine this is a medium sized API with 105 types and a total of 7.812 relationships, most of the being related to non-static operations. Analyzing the graph information, both average and maximum

Table 4.7: SWT API Type Graph Information

Measurement description	**Value**
Number of API Types	105
Number of relationships by relationship type:	
Parameter in a constructor	185
Parameter in static method type	21
Parameter in a non-static method	3.753
Target Instance in a non-static method	3.853
Total	7.812
Average number relationships per type	42,23
Outgoing graph edges:	
Average number of outgoing edges	46,65
Average number of outgoing edges (unique types)	8,17
Maximum number of outgoing edges	216
Maximum number of outgoing edges (unique types)	42
Incoming graph edges:	
Average number of incoming edges	20,11
Average number of incoming edges (unique types)	6,75
Maximum number of incoming edges	455
Maximum number of incoming edges (unique types)	48

number of incoming and outgoing edges seem to indicate the dependency in some API type or types, as some of the previous APIs analyzed in this Section.

The main struggles that a developer would face when using the SWT API are related to the initial setup, that consists on creating a *Display* object and a *Shell* afterwards. This two types are very important because all widgets that the developer would be need, he or she would able to find by name (like *Button, Label, Text*, etc) but would require those objects to be created first. Knowing this, the starting point for this API is one of the main focuses and had good results, with a rank value of 6 of a possible 58 starting points as described in Table 4.8. Having the *Display* type in the context, the *Shell* was also one of the top suggestions, with a rank value of 5 of a total of 129 relationships.

Code Snippet 7 illustrates an example usage of this API where a simple user interface with a button and a text widgets. The top suggestions of the next set of relationships the simulator provided were mostly widgets constructors, like the *Button* and *Text* type created indicated in the last two instructions of Table 4.8. These two suggestions also had good results (19 of a possible 746 and 13

of a possible 831 respectively) considering that the API provides a considerable amount of widgets to be created. The keyword filtering suggested in Subsection 4.2.3 would also work in this scenario.

There are two important instructions (line 6 and 14) highlighted in the Code Snippet 7 required to be present when creating a user interface using the SWT API. However, as stated before, in our approach, the relationships that represent these instructions will most likely have a low rank value when compared to the remaining relationships. The instruction in line 6 (*shell.setLayout(new RowLayout(...));*) sets the layout of the Shell and without it the interface will not be display correctly and the one in line 14 (*shell.open();*), opens the shell, displaying the user interface. Even though our approach doesn't suggest these operations (with a high rank value), the existing IDE code completion mechanisms, given the *Shell* object will suggest them to the developer and will allow him or her to still be able to use this API.

Table 4.8: SWT API Usage Ranking Results

Line of Code	**Index**	**Context**
Display display = new Display();	6 / 58	-
Shell shell = new Shell(display);	5 / 129	Display
Button button = new Button(composite);	19 / 746	Display Shell
Text text = new Text(composite);	13 / 831	Display Shell Button

Analyzing the overall performance of the simulation for this API, the suggestions provided during the simulation, presented in Table 4.8, would be able to aid the developer when using this API for the first time, specially in the creation of the *Display* and *Shell* types. As previously mentioned, the two instructions that weren't directly solved by our approach, would be solve by the existing IDE code completion mechanisms.

Code Snippet 7 SWT API Example: Create a window with a button and a text widgets

```
1  //Initiate display and shell
2  Display display = new Display();
3  Shell shell = new Shell(display);
4  shell.setSize(300, 200);
5  shell.setText("Button Example");
6  shell.setLayout(new RowLayout());
7  //Create button and add to the shell
8  Button button = new Button(shell);
9  button.setText("Click Me");
10 //Create text and add to the shell
11 Text text = new Text(shell);
12 text.setText("Hello World SWT");
13 //Open window
14 shell.open();
```

4.2.5 Swing API

Similarly to the SWT API analyzed in Subsection 4.2.4, the Swing API provides a set of components to be used to design cross platform user interface in Java applications. The namespace used to define the API boundaries was *javax.swing*. The version was used for the simulation was the included in the Java Development Kit (JDK) 1.8 build 131.

Analyzing the data in Table 4.9 and comparing to previous API analysis, it's possible to classify Swing as a large API, the largest that tested, with 881 API types and a total 61.176 relationships. Analyzing the data related to the graph, we can observe that the maximum number of outgoing edges and incoming edges for unique types, is considerable, the biggest from all the APIs analyzed. The average number of outgoing edges with respect to unique types is also the maximum from all APIs tested. The reason behind the high numbers in Table 4.9 is hard to determine but, although the domain for Swing and SWT APIs is the same, the design of the API is very different.

Our approach for the ranking system was not able to meaningfully rank the suggestions when comparing to the usage example in Code Snippet 8. This is due, in part, to the high number of starting points, 779 to be precise, but mainly due

Table 4.9: Swing API Type Graph Information

Measurement description	**Value**
Number of API Types	881
Number of relationships by relationship type:	
Parameter in a constructor	952
Parameter in static method type	1.002
Parameter in a non-static method	29.577
Target Instance in a non-static method	29.645
Total	61.176
Average number relationships per type	69,44
Outgoing graph edges:	
Average number of outgoing edges	44,58
Average number of outgoing edges (unique types)	9,85
Maximum number of outgoing edges	1315
Maximum number of outgoing edges (unique types)	70
Incoming graph edges:	
Average number of incoming edges	8,32
Average number of incoming edges (unique types)	3,26
Maximum number of incoming edges	406
Maximum number of incoming edges (unique types)	82

to the way that the widget composition (*JButton, JLabel, etc*) was redesigned in the Swing API. Most of the components in the Swing API are able to instantiated without any parameters and for this reason, one of the types that would be a valid starting point *JFrame* is not ranked high in the list of suggestions. This happens because all the components in this API have more outgoing unique types than *JFrame*.

This is a clear design difference comparing with the SWT API in Subsection 4.2.4 that only allows the creation of widgets after the parent is instantiated. Using the Swing API, the components are passed to the *JFrame* or the to appropriate *JPanel*. Even if the ranking system's first criteria was changed to somehow better rank the *JFrame* type, the problem would surface in the operation that adds a component to the *JFrame* because *add(..)* is a void non-static operation and as referenced in the previous analyses operations like these are not the goal of our approach.

Given that Swing API is used in a lot of projects, it has been shown to be a good API to be used in an example based approach where multiple examples are

analyzed from open-source repositories in order to create API Sentences. [13]

Code Snippet 8 Swing API Example: Create a window with a label

```
1 //Create and set up the window.
2 JFrame frame = new JFrame("HelloWorldSwing");
3 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
4 //Add the label the content pane
5 JLabel label = new JLabel("Hello World");
6 frame.getContentPane().add(label);
7 //Display the window.
8 frame.pack();
9 frame.setVisible(true);
```

4.3 Discussion

After analyzing the suggestions from the simulation for all APIs referenced in the Section 4.2 it's possible to conclude that the filtering mechanism and ranking heuristic performed well in the majority of cases. The two main exceptions are the starting point for the the JFreechart API in Subsection 4.2.3 and the Swing API in Subsection 4.2.5. As indicated before, the chosen APIs to undergo this analysis all have known discoverability problems and they vary in complexity, dimension and purpose. All the index of the suggestions extracted from the simulator and used in the previous section were not changed in any way, which means that if a given operation as multiple signatures, they appeared in the suggestions. One example of this is the Figure 4.2 in Section 4.1 where the same operation name appear multiple times for operations that have more than one signature (parameters and return value). When developing an IDE suggestion mechanism, one possibility that can help with this situation is to merge these operations into one and, when the developer selects that suggestion, all the possibilities could appear so that the developer can choose the exact one he or she desires.

With the exception of JFreechart, the excepted Starting Points suggestions were all located in the TOP 10 of the suggestions list provided by the simulator. The high index value for the JFreechart API's Starting Point is mainly due to the amount of possible starting points and the multiple ways the API provides the

same functionality. As it was previously explained, it's possible that a keyword filtering would improve mitigate this issue.

For the steps after the Starting Points, where ranking heuristic's second criteria takes the current context into account, the results where positive and consistent throughout the analyzed APIs. Even though the total number of possible relationships increase with the amount of different types in the context, the ranking value stays very low and, in some cases, even improves with each new simulator step.

Chapter 5

Conclusions and Future Work

The use of APIs in the development work flow is inevitable therefore, providing an aid to API discoverability problems is useful. Our approach, of generation suggestions based on graph data extracted from structurally analyzing APIs for relationships shows promising results. In the APIs we've tested, with the exception of the Swing API, it was possible to recreate an usage example with the suggestions provided from our simulator.

One of the novelties of our approach is the ability to find and rank all of the possible starting points for a given API. The starting points essentially are the first code instructions that the developer can write when using an API. The value of this feature is not to find the starting points but instead, to rank them in a meaningful way to better aid the developer. This is useful for big APIs, which have a lot of starting points and for small APIs that have fewer starting points and a more strict design structure.

Another novelty of our approach is the ability to, given a set of types in a current context, suggest all possible Type Composition code instructions available in the API ranked with respect to their meaningfulness. A Type Composition operation consists of using an API type to access/create another API type. This is useful when the developer has create an API type and wants to know where to

use that type. This mechanism helps the developer find related API types that otherwise would be harder to find.

In the future, more APIs could be analyzed using the created simulator in order ascertain the validity of our approach for more types of APIs. It's unlikely that our approach would have successful results for all APIs, however the best way to improve our approach, is by discovering it's flaws and attempting to evolve the approach accordingly. The most likely changes to take place in our approach will be focused on the ranking heuristic, in order to improve the suggestions ranking for API designs that do not fit well into our ranking method.

Another possible iteration on our work is the integration of the simulator's behavior in a IDE to test if these suggestions can be seamlessly used in the development environment. This integration would focus more on the usability of the delivery of suggestions and not the quality of the information. One possible integration solution could give some suggestion tuning abilities to the developer. This tunning would allow the developer to change the importance of the ranking heuristic's criteria. As it stands now, the ranking heuristic averages the scores for both criteria. In this approach, the developer could choose to focus on the first criteria, where the suggestions that originate types that expand the discovery of new API types are given more importance, or focus on the second criteria. If he or she opted for the second criteria, the main focus would be to use the types in the current context essentially, give more importance to all relationships that make use of the current types in the context.

Bibliography

[1] Ted Boren and Judith Ramey. Thinking aloud: Reconciling theory and practice. *IEEE transactions on professional communication*, 43(3):261–278, 2000.

[2] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, pages 213–222. ACM, 2009.

[3] Ekwa Duala-Ekoko and Martin P Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *European Conference on Object-oriented Programming*, pages 79–104. Springer, 2011.

[4] Ekwa Duala-Ekoko and Martin P Robillard. Asking and answering questions about unfamiliar APIs: An exploratory study. In *Proceedings of the 34th International Conference on Software Engineering*, pages 266–276. IEEE Press, 2012.

[5] Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *Proceedings of the 29th international conference on Software Engineering*, pages 302–312. IEEE Computer Society, 2007.

[6] Thomas Grill, Ondrej Polacek, and Manfred Tscheligi. Methods towards api usability: A structural analysis of usability problem categories. *Human-Centered Software Engineering*, pages 164–180, 2012.

[7] Mathew Mooty, Andrew Faulring, Jeffrey Stylos, and Brad A Myers. Calcite: Completing code completion for constructors using crowds. In *Visual Languages and Human-Centric Computing (VL/HCC), 2010 IEEE Symposium on*, pages 15–22. IEEE, 2010.

[8] Brad A Myers and Jeffrey Stylos. Improving API usability. *Communications of the ACM*, 59(6):62–69, 2016.

[9] Cyrus Omar, YoungSeok Yoon, Thomas D LaToza, and Brad A Myers. Active code completion. In *Proceedings of the 34th International Conference on Software Engineering*, pages 859–869. IEEE Press, 2012.

[10] Marco Piccioni, Carlo A Furia, and Bertrand Meyer. An empirical study of API usability. In *Empirical Software Engineering and Measurement, 2013 ACM/IEEE International Symposium on*, pages 5–14. IEEE, 2013.

[11] Martin P Robillard. What makes APIs hard to learn? answers from developers. *IEEE software*, 26(6), 2009.

[12] André L Santos and Brad A Myers. Design annotations to improve API discoverability. *Journal of Systems and Software*, 126:17–33, 2017.

[13] André L Santos, Gonçalo Prendi, Hugo Sousa, and Ricardo Ribeiro. Stepwise API usage assistance using n-gram language models. *Journal of Systems and Software*, 2016.

[14] Thomas Scheller and Eva Kuhn. Influencing factors on the usability of API classes and methods. In *Engineering of Computer Based Systems (ECBS), 2012 IEEE 19th International Conference and Workshops on*, pages 232–241. IEEE, 2012.

[15] Thomas Scheller and Eva Kühn. Influence of code completion methods on the usability of APIs. In *Proceedings of the IASTED International Conference Software Engineering (SE 2013)*, 2013.

[16] Thomas Scheller and Eva Kühn. Automated measurement of API usability: The API concepts framework. *Information and Software Technology*, 61:145–162, 2015.

[17] Jeffrey Stylos, Andrew Faulring, Zizhuang Yang, and Brad A Myers. Improving API documentation using API usage information. In *Visual Languages and Human-Centric Computing, 2009. VL/HCC 2009. IEEE Symposium on*, pages 119–126. IEEE, 2009.

[18] Jeffrey Stylos and Brad A Myers. The implications of method placement on API learnability. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 105–112. ACM, 2008.

[19] John Vlissides, Richard Helm, Ralph Johnson, and Erich Gamma. Design patterns: Elements of reusable object-oriented software. *Reading: Addison-Wesley*, 49(120):11, 1995.

[20] Minhaz Zibran. What makes APIs difficult to use. *International Journal of Computer Science and Network Security (IJCSNS)*, 8(4):255–261, 2008.