Using Structural Relationships to Facilitate API Learning

by
Ekwa Duala-Ekoko

Doctor of Philosophy

School of Computer Science

McGill University

Montreal,Quebec

March 2012

A THESIS SUBMITTED TO MCGILL UNIVERSITY
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS OF THE DEGREE
OF DOCTOR OF PHILOSOPHY

DEDICATION

To my father, Ekoko Ben Duala, for being my inspiration and greatest fan.

ACKNOWLEDGEMENTS

Going through the Ph.D. program has been both an exciting and a challenging experience. I was fortunate to have had some great people to assist me during those challenging days. I am extremely thankful to my supervisor, Prof. Martin P. Robillard (or simply Martin — as he prefers to be called) for his insight, ideas, and commitment to my success. Martin was always available for discussions, would always provide timely and effective feedback on draft publications, and would always push for perfection. Martin's commitment to excellence and his genuine interest in both the professional and personal life of his students is a lesson that would stay with me forever.

I am also thankful to my colleagues of the Software Evolution Research Group (SWEVO): Bart, Annie, Tristan, David, and Gias. Thanks guys for scrutinizing my research ideas, for your feedback on conference practice talks, and for volunteering your time to review my conference submissions. The discussions I had with each of you and the time we spent together has helped me to be a better person.

I want to thank the members of my Ph.D. progress committee (Prof. Laurie Hendren and Prof. Jorg Kienzle) for ensuring I stayed on track. I also want to thank the IT staff (Andrew Bogecho and Ron Simpson) for helping me with the computers and responding to my late night emails whenever I needed assistance.

Lastly, and most importantly, I am thankful to my spouse, Yvonne, and my son, Daniel, for their sacrifice during this season of my life. You endured my absence and

my frustration; you were there to encourage me during those dark days in 2009. I am extremely thankful to Christ for having both of you in my life.

ABSTRACT

Application Programming Interfaces (APIs) allow software developers to reuse code libraries, frameworks, or services without the need of having to implement relevant functionalities from scratch. The benefits of reusing source code or services through APIs have encouraged the adoption of APIs as the building blocks of modern-day software systems. However, leveraging the benefits of APIs require a developer to frequently learn how to use unfamiliar APIs — a process made difficult by the increasing size of APIs, and the increase in the number of APIs with which a developer has to work. In this dissertation, we investigated some of the challenges developers encounter when working with unfamiliar APIs, and we designed and implemented new programming tools to assist developers in learning how to use new APIs.

To investigate the difficulties developers encounter when learning to use APIs, we conducted a programming study in which twenty participants completed two programming tasks using real-world APIs. Through a systematic analysis of the screen captured videos and the verbalizations of the participants, we isolated twenty different types of questions the programmers asked when learning to use APIs, and identified five of the twenty questions as the most difficult for the programmers to answer in the context of our study. Drawing from varied sources of evidence, such as the verbalizations and the navigation paths of the participants, we explain why the participants found certain questions hard to answer, and provide new insights to the cause of the difficulties.

To facilitate the API learning process, we designed and evaluated two novel programming tools: *API Explorer* and *Introspector*. The API Explorer tool addresses the difficulty a developer faces when the API types or methods necessary to implement a task are not accessible from the type the developer is working with. API Explorer leverages the structural relationships between API elements to recommend relevant methods on other objects, and to identify API types relevant to the use of a method or class. The Introspector tool addresses the difficulty of formulating effective queries when searching for code examples relevant to implementing a task. Introspector combines the structural relationships between API types to recommend types that should be used together with a seed to search for code examples for a given task. Using the types recommended by Introspector as search query, a developer can search for code examples across two code repositories, and in return, will get a list of code examples ranked based on their relevance to the search query. We evaluated API Explorer through a programming study, and evaluated Introspector quantitatively using ten tasks from six different APIs. The results of the evaluations suggest that these programming tools provide effective support to programmers learning how to use APIs.

ABRÉGÉ

Les interfaces de programmation (API) permettent aux développeurs de réutiliser du code, des bibliothques, des cadres d'application ou des services sans avoir à réimplémenter des fonctionnalités importantes à partir de zéro. Les avantages de la réutilisation de code source ou de services par des APIs ont encouragé l'adoption des APIs comme composant essentiel des logiciels modernes. Cependant, pour tirer parti des avantages des APIs, les développeurs doivent fréquemment apprendre à utiliser des APIs inconnus, un processus rendu difficile par la taille grandissante des APIs et par l'augmentation du nombre d'APIs avec lesquels les développeurs doivent travailler. Dans cette dissertation, nous avons étudié les défis que les développeurs rencontrent quand ils travaillent avec des APIs inconnus et nous avons conu et implémenté de nouveaux outils de programmation pour aider les développeurs à apprendre comment utiliser ces APIs.

Pour étudier les difficultés que les développeurs rencontrent lorsqu'ils apprennent à utiliser les APIs, nous avons conduit une étude dans laquelle 20 participants ont complété deux exercices de programmation en utilisant des APIs populaires. Par une analyse détaillée des bandes vidéo enregistrées lors des exercices et des commentaires émis par les participants, nous avons isolé vingt différent types de questions que les programmeurs ont posées lorsqu'ils apprenaient à utiliser les APIs. Nous avons aussi identifié cinq questions sur les 20 comme étant les plus difficiles à répondre par les programmeurs dans le contexte de notre étude. Notre analyse fournit des éléments probants qui expliquent la cause des difficultés observées.

Pour faciliter l'apprentissage des APIs, nous avons conu et évalué deux outils de programmation: API Explorer et Introspector. API Explorer est un outil qui a pour but de diminuer la difficulté que les développeurs rencontrent quand les types ou les méthodes nécessaire à l'accomplissement d'une tâche dans une API ne sont pas accessibles à partir du type avec lequel le développeur travaille. API Explorer tire parti des relations structurelles entre les éléments d'une API pour recommander des méthodes pertinentes sur d'autres objets et pour identifier les types d'une API pertinents pour l'utilisation d'une méthode ou d'une classe. Introspector est un outil qui a pour but de réduire la difficulté à formuler des requêtes efficaces lorsque les développeurs cherchent des exemples de code reliés à l'accomplissement d'une tâche de programmation. Introspector combine les relations structurelles entre les types d'une API pour recommander les types qui devraient être utilisés ensemble avec un germe pour chercher des exemples de code pour une tâche particulire. Un développeur peut ainsi chercher des exemples de code dans deux référentiels en utilisant les types recommandés par Introspector. En retour, l'utilisateur recevra une liste d'exemples de code triée en fonction de leur pertinence avec leur tâche courante. Nous avons évalué API Explorer grâce à une étude avec des utilisateurs et nous avons évalué quantitativement Introspector en analysant les résultats de dix tâches effectuées avec six APIs différents. Les résultats de notre évaluation suggrent que les outils de programmation que nous proposons offrent un support efficace pour les programmeurs désirant apprendre à utiliser une API.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
Introduction

Application Programming Interfaces (APIs) play a central role in modern-day software development. Software developers often favor *reuse* of code libraries or frameworks through APIs over *re-invention* since reuse increases productivity and improves the quality of the software. An informal examination of a few software systems show the wide spread use and heavy reliance on APIs by both medium-size and large scale systems (see Table 5–1). For instance, Netbeans,[1] an Integrated Development Environment, written in Java relies on over 27 different APIs; JasperReport,[2] an open source system for creating reports in a variety of document formats, makes use of over fifteen different APIs including an API for creating PDF documents, an API for generating charts, and an API for parsing XML files. The benefits of using APIs, however, do not come cheap: previous work on API usability showed that learning how to use APIs presents several barriers [28, 40, 49, 51], and that "understanding how the APIs are structured, selecting the appropriate classes and methods, figuring out how to use the selected classes, and coordinating the use of different objects together all pose significant difficulties" [50]. These difficulties are further compounded

[1] http://netbeans.org/

[2] http://jasperforge.org/projects/jasperreports

Table 1–1: API usage in sample software systems.

System	Description	Version	# of Libraries
JasperReport	A framework for generating reports.	4.0.0	15
Eclipse BIRT	A business intelligence and reporting tools designer.	2.6.2	6
Hibernate	An object-relational mapping persistence framework.	3.6.4	27
Apache Tomcat	An open source servlet container.	7.0	5
Netbeans	An Integrated Development Environment.	5.5	23

by the increasing size and complexity of APIs: for example, version 1.5 of the Java J2SE APIs is reported to contain more than 4000 classes and more than 35,000 different methods, and the Microsoft .NET 2.0 Framework contains more than 140,000 classes, methods, and fields [48]. Given the size of APIs and the increase in the number of APIs, even experienced developers must frequently learn newer parts of familiar APIs, or newer APIs when working on new tasks.

The main goal of this dissertation was to investigate why developers find certain questions about the use of APIs hard to answer, and to suggest new ideas and tools to facilitate the process of learning how to use an API. To achieve this goal, we began with an exploratory study aimed at isolating the different types of questions developers ask when learning to use an API and investigating those questions that proved difficult for developers to answer; Next, we identified areas in the API learning process where effective tool support is missing or could be improved; and finally, we

designed and evaluated new techniques and programming tools to facilitate the API learning process.

We begin with an example to illustrate the API learning process and to highlight some of the challenges a developer may encounter when using an unfamiliar API; then we present the research questions investigated in this thesis (Section 1.1). Lastly, we present a summary of the contributions of the thesis in Section 1.2, with forward references to chapters that provide a detailed presentation of each contribution.

1.1 Motivation

To help illustrate some of the questions developers ask when using unfamiliar APIs and to highlight some of the challenges they may encounter answering those questions, we use a task completed in a user study conducted as part of this thesis. In the task, we asked the participants to create a pie chart and save the chart in a graphic format using the JFreeChart API, a popular open source API for generating charts.

One participant in the study began by looking for a type[3] in the JFreeChart API representing charts in general, or pie charts in particular. The questions asked by the participant at this stage were about locating a *central-type* which embodied the concept to be implemented: for instance, *"Which class or interface of the API provides support for creating a chart?"*. The participant eventually found the `JFreeChart` class, the central-type provided by the API for representing charts. The participant soon realized that other classes, besides the `JFreeChart` class, were needed

[3] An API type for our purpose is either an API class or an API interface.

to create a pie chart. The questions asked by the participant then shifted to locating one or more *secondary-types* to be used together with the `JFreeChart` class for creating a pie chart: for instance, *"Which other classes related to the JFreeChart class are relevant to creating a pie chart?"*. The JFreeChart API provides two options for creating a pie chart: the easier and simpler way is through a convenience method on the `ChartFactory` class; the other option, which requires the coordination of several classes (`Plot`, `Dataset`, and `Title`), is through a public constructor of the `JFreeChart` class. The participant, unaware of the existence of the `ChartFactory` class (since `ChartFactory` is not directly accessible from the `JFreeChart` class), used the constructor option.

Once all the relevant classes were located, the questions asked by the participant then shifted to figuring out how to create instances of the central-type and secondary-types, and how to coordinate the use of these types to create a pie chart: for instance, *"How do I create objects of type Plot?"*; *"How is Plot related to Dataset?"*. After creating the pie chart, the participant then looked for a "save" method on the `JFreeChart` class, the last step needed to complete the task: *"Does the JFreeChart class provide a method for saving the chart?"*. However, the `JFreeChart` class does not contain a "save" method. The participant then looked through all the methods of the `JFreeChart` class, thinking that the method was named something else. However, the participant did not find any "save" method. The questions asked by the participant then shifted to locating a *helper-type* in the API for saving the chart to a file in a graphic format: *"Does the API provide a helper-type for saving charts in a graphic format?"*. The participant visited the documentation in search for a class

that could save a `JFreeChart` object in a graphic format. The participant was unable to locate a helper-type and concluded that the API simply did not provide support for saving charts. However, the API provides the helper-type `ChartUtilities`, which provides methods (such "saveChartAsJPEG" and "saveChartAsPNG") for saving charts in different formats, but `ChartUtilities` is not directly accessible from the `JFreeChart` class.

This example illustrates some of the questions developers ask when working with unfamiliar APIs and some of the challenges they encounter when trying to answer questions about the use of APIs. In general, the API learning process involves asking questions and answering them through a series of incremental discoveries such as locating the relevant types, understanding how they are related, and coordinating the use of the different types. Making these discoveries without appropriate tool support is challenging. In this thesis, we extend the body of knowledge in the area of API learning by investigating the following research questions (RQ):

I. **What are the different types of questions developers ask when working with unfamiliar APIs?** The objective of this question was to isolate and understand the needs of developers working with unfamiliar APIs, to identify questions about the use of APIs that are difficult to answer, and to understand the cause of the difficulty.

II. **How well do existing tools support developers working with unfamiliar APIs?** The objective of this question was to understand the degree to which existing tools support the questions asked by developers, and to identify areas where tool support is missing, or could be improved.

III. **How can programming tools facilitate the process of learning how to use APIs?** Having identified features that hinder the API learning process, we proposed new ideas and techniques to support API learners, and as proof of concept, implemented prototypes to evaluate the suitability of the proposed ideas and techniques in supporting developers working with unfamiliar APIs.

1.2 Contributions of Thesis

1.2.1 Asking and Answering Questions About Unfamiliar APIs

To understand what a programmer needs to know when using a new API and how best to support programmers, we conducted a study in which 20 participants worked on two programming tasks using different real-world APIs. The study generated over 20 hours of screen captured videos and the verbalization of the participants spanning 40 different programming sessions. Our analysis of the data involved generating generic versions of the questions asked by the participants about the use of the APIs, abstracting each question from the specifics of a given API, and identifying those questions that proved difficult to answer using the actions of the participants that reflected a lack of progress when looking for information. Based on the results of our analysis, we identified 20 different types of questions the programmers asked about the use of the APIs grouped in five different categories: discovering the functionality provided by an API relevant to a task, understanding the relationships between types, discovering relevant dependent types, selecting relevant API elements,[4] and

[4] An API element could be an API method, an API class, or an API interface.

understanding the behavior of API methods. We also identified the following questions as the most difficult for programmers to answer in the context of our study:

- Which keywords best describe a functionality provided by the API?
- How is the type X related to the type Y?
- Does the API provide a helper-type for manipulating objects of a given type?
- How do I create an object of a given type without a public constructor?
- How do I determine the outcome of a method call?

Drawing from varied sources of evidence, such as the verbalizations and the navigation paths of the participants, we explain why the participants found these questions hard to answer, and provide new insights to the cause of the difficulties. We believe the questions we have identified and the difficulties we observed can be used for evaluating tools aimed at improving API learning, and in identifying areas of the API learning process where tool support is missing, or could be improved. As an example, we identified some areas where support is limited from existing tools including the need for IDE based tools that would assist a programmer in discovering relevant API elements not accessible from a type the programmer is working with; and tools that would assist a programmer in easily identifying types that would serve as a good starting point for searching for examples, or for exploring the API for a given programming task. We present the details of the study, the analysis of the data from the study, and the results and observations in Chapter 3.

1.2.2 API Explorer: Coordinating the Use of Multiple Objects

Amongst the difficulties we observed, questions about the relationships between API types (such as *How is the type X related to the type Y?*, or *Does the API provide*

a helper-type for manipulating objects of a given type?) proved the most difficult for programmers to answer when working with unfamiliar APIs. The difficulties the participants encountered were due in part to a mismatch between the structuring of the APIs and the expectations of the participants. For instance, in the Chart task, most participants expected the helper-type for saving a chart to be in either the "util", "io", or "renderer" package of the API but it was somewhere else. Although most participants successfully created the chart, some did not complete the task because they could not locate the helper-type related to saving. For the second task, participants had to validate an XML file against a given schema. Most participants were able to locate the central-element for validating an XML file against a given schema (`Validator.validate(Source)`), but could not figure out how to create a `Validator` object or how to relate the `Validator` class to its secondary-type, `Schema` class. One participant attributed the difficulty of relating `Validator` to `Schema` to the absence of a "cross-reference in the API documentation that says get a `Validator` instance from a `Schema`".

We observed that, in general, the participants relied on imperfect proxies such as domain knowledge, expectations of how an API should be structured, and naming conventions as beacons for locating related API elements, or for understanding the relationships between a central-type and its corresponding secondary-types and helper-types. However, these attributes do not stay the same across different APIs and their use may be misleading, resulting in unsuccessful searches and wasted efforts. We therefore need more reliable attributes that would assist programmers

in discovering and coordinating the use of relevant helper-types. Specifically, we investigated the following:

- How can we efficiently relate an API type to its secondary-types and helper-types?
- How can we support API learners in the use of relevant API elements, and in the coordination of related types?

To provide API learners the support needed to efficiently relate API elements or to coordinate the use of multiple API elements, we proposed leveraging the structural relationships and the flow of information between API classes and methods. Our hypothesis is that the relationships between API elements such as the return types of API methods and the parameters of methods are better beacons for locating or relating relevant classes and methods than the use of domain knowledge, the expected structure of an API, or naming conventions. For instance, using static analysis on the API for the XML validation task, we can deduce that `Validator` and `Schema` are structurally related through `Schema.newValidator()`, a method which returns an instance of `Validator`. This fact can then be presented to an API learner, when requested, to illustrate how to create a `Validator` object, or how to relate `Validator` to `Schema`. To capture the relationships between API elements, we modeled an API as a special dependency graph of entities (classes, interfaces, and methods) related through structural and data flow relationships such as return types, method parameters, and subtyping. This representation, called *API Exploration Graph*, is simple

but contains enough information for inferring the relationships between an API element and its secondary-types and helper-types, and for supporting API learners in the use and the coordination of these related types.

We developed a *recommendation system*, called *API Explorer*, which uses the information on the API Exploration Graph to generate recommendations that would assist a developer to discover helper-types not accessible from a main-type based on the structural context in which help is requested. API Explorer is implemented as an extension to the content assist feature of the Eclipse IDE. Content assist in Eclipse, or IntelliSense in Microsoft Visual Studio, provides context sensitive autocompletion of variable or method names, but is limited to showing only the methods of the object on which it is invoked. Previous work to improve the content assist feature has been limited to re-ordering the list of methods [5, 38]. API Explorer is the first novel extension, to our knowledge, capable of suggesting relevant methods on other objects, identifying API elements relevant to the use of a method or class, and providing support for combining multiple objects. We use examples to illustrate how API Explorer supports the needs of API learners:

How do I construct an object of a given type?

A common frustration amongst the participants in the exploratory study relates to creating objects from classes without public constructors: "I need to create a `Schema` object and also ... a `Validator` object. The problem is these two classes are

abstract and I can't find their derived classes."[5] API Explorer provides support for creating objects from various construction patterns including declared constructors, constructors of subtypes, factory methods, public methods, or static methods.

Consider a scenario in which a developer has to implement a solution to compose and deliver an email message using the JavaMail API. Going through the documentation of JavaMail, the developer found `Message`, the main-type representing an email message. The developer then proceeds by attempting to construct an object of type `Message` from the default constructor[6] but is unsuccessful since `Message` is an abstract class. Faced with an object construction hurdle, the developer would query API Explorer for assistance by invoking content assist: the developer would enter `Message m =`, then the key sequence *Ctrl+Space*, and API Explorer would instantly display two options for creating a `Message` object from `MimeMessage` (see Figure 1–1(A)). Selecting a recommendation reveals a *hoverdoc*, containing a *rationale*, that explains why the element was recommended, and the documentation of the recommended element to help a developer determine its relevance in the given context (see Figure 1–1). Once the developer makes a selection, API Explorer automatically expands the selected recommendation into code that shows how the elements needed to create an object of type `Message` should be combined (see Figure 1–1(B)).

[5] The Validator and Schema objects are created from factory methods

[6] Three separate studies observed that most developers, both novice and experts alike, begin object construction by attempting to use the default constructor [15, 49, 51].

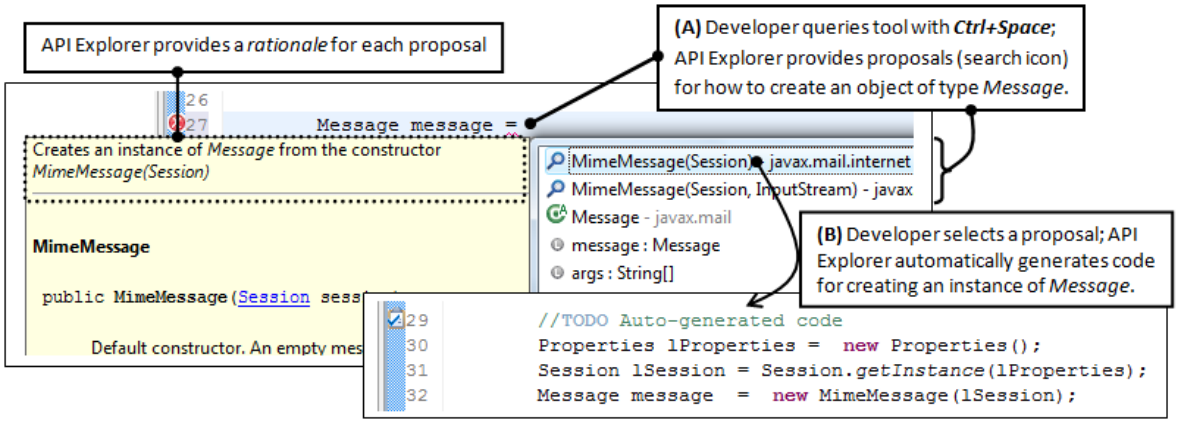


Figure 1–1: API Explorer shows the developer the types required to construct an instance of `Message` and generates code which illustrate how to combine these types.

Does the API provide a helper-type for manipulating objects of a given type?

Often, the methods API users need are not on the objects they are working with. For instance, the method for saving the chart in a graphic format is not on the `JFreeChart` class, but on the `ChartUtilities` class — a helper-type. Also, the JavaMail API, provided by Sun Microsystems, has the method for sending an email message on a different class, not the `Message` class. API Explorer supports API users by suggesting relevant methods on other objects. For instance, a developer looking for a "send" method on `Message`, will see three "send" methods of the `Transport` class recommended by API Explorer (see Fig. 1–2).

At times, a developer may search for a method prefix that does not match the name of any method on the helper-types (e.g., searching for "message.forward" instead of "message.send"). In this case, API Explorer combines structural analysis

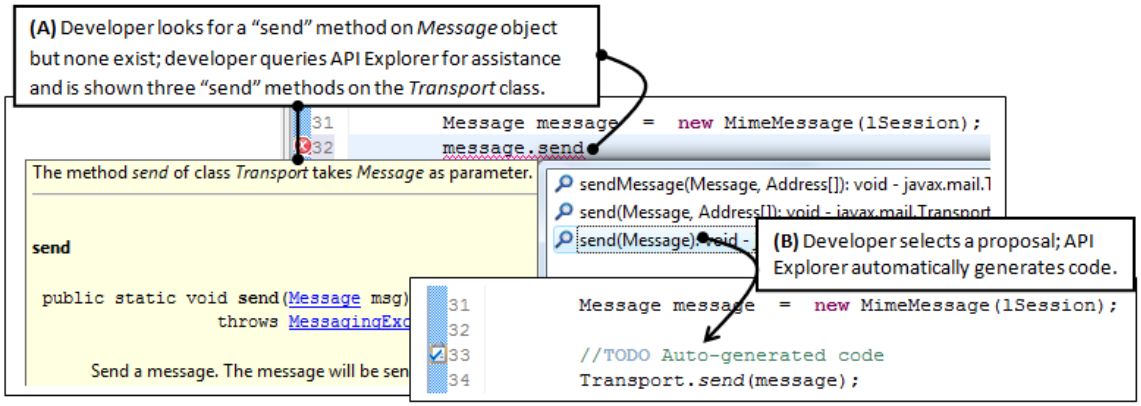


Figure 1–2: API Explorer recommends three "send" methods of the `Transport` class which can be used for sending an email `Message` object.

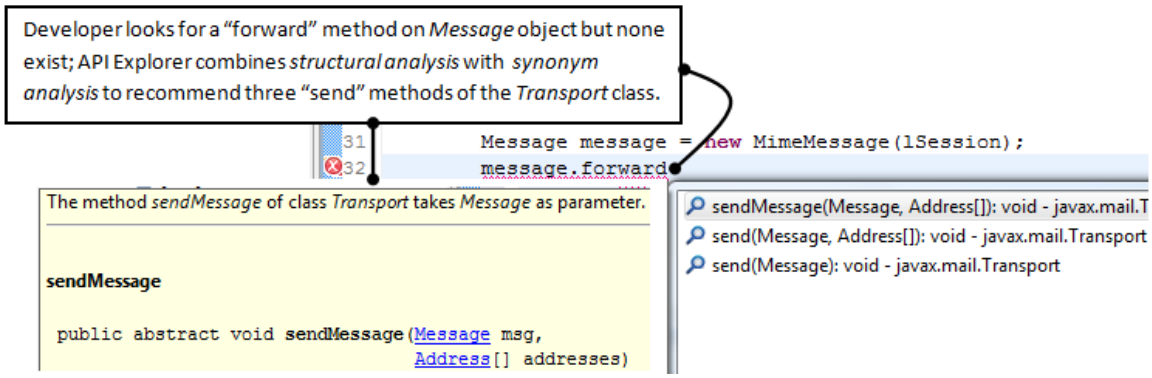


Figure 1–3: API Explorer combines structural analysis with synonym analysis to recommends three "send" methods of the `Transport` class when a developer looks for a "forward" method on the `Message` object.

with synonym analysis to recommend methods with a name similar to what the developer is looking for (see Figure 1–3).

Evaluation. We evaluated API Explorer through a multiple-case study in which eight programmers were asked to complete the same four programming tasks using four different real-world APIs, each task presenting multiple API learning challenges.

The results of the study were consistent across the participants and the tasks: the participants successfully used API Explorer to discover relevant helper-types not accessible from a main-type, and experienced little difficulty selecting API elements relevant to a given programming scenario when presented with a list of possible helper-types. The results also showed that the use of structural relationships, combined with the use of content assist to generate and present recommendations, could be a viable, and an inexpensive, alternative when seeking to make helper-types discoverable from objects a programmer is working with. We present the API Exploration Graph, the algorithms of API Explorer, and the evaluation of API Explorer in Chapter 4.

1.2.3 Introspector: Facilitating Effective Query Formulation

Learning from code examples is one of the strategies developers employ when learning how to use an API. Locating relevant code examples on the Web or in source code repositories presents two major challenges: formulating the right search query and evaluating the relevance of the search results to locate a suitable code example. Searching for suitable code examples is an iterative and a refinement process: a developer formulates a search query (Query Formulation), retrieves some code examples, and looks through the code examples for a relevant results (Determine Relevance of Results); the entire process is repeated if no suitable example code was found (Figure 1–4, Hill et al. [20]).

Query formulation is difficult because developers often lack a clear understanding of what they need when beginning a new a task, and even when they do, they have difficulty using keywords that correspond to those used by the API designers.

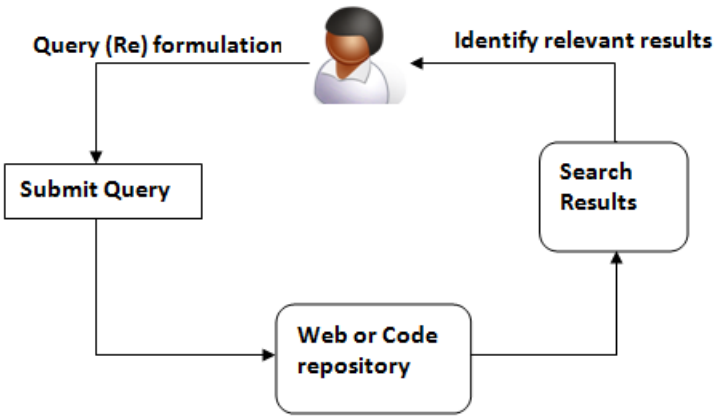


Figure 1–4: An overview of the query formulation and code search process.

This difficulty, referred to as the vocabulary mismatch problem (multiple words for the same topic) has been identified as a major challenge to query formulation [16, 18]. Research has reported a word agreement of less than 15% between the queries formulated by users and the words used in the source code [16]. Our study corroborates the vocabulary mismatch problem : we identified the question *"Which keywords best describe a functionality provided by the API?"* as being difficult for programmers to answer when working with unfamiliar APIs.

To assist developers in formulating effective queries, researchers have employed natural language techniques that automatically extract, categorize, and recommend word usage from the underlying source code [20, 37, 44]. Given one or more keywords from the developer, these techniques would recommend alternative words to help the developer explore the word usage in the given source code, and to use these alternative words to reformulate search queries. Some of these techniques also associate the recommended words with the program elements they describe so the developer can see the context of the matches to the query words, and determine the

relevance of each program element to the search [20]. However, these techniques are still within the natural language domain — the developer still has to provide one or more keywords as the *seed*, and therefore, they share to a certain extent some of the limitations of the vocabulary mismatch problem.

We proposed a technique based on API types to help facilitate the process of query formulation. Our idea is that instead of trying to guess the most appropriate keywords to use as the seed of a query, the developer would provide an API type relevant to the scenario to be implemented, then our technique would recommend relevant types strongly associated with the usage of the given type. The developer can then use the recommended types as a starting point for searching for code examples, or for exploring the API for a given programming task. To investigate our technique, we developed a programming tool called *Introspector*, implemented as an Eclipse plug-in. Consider an example of a developer working on a task for sending an email using the JavaMail API. Instead of trying to guess the most appropriate keywords to use as a query for locating code examples, the developer can simply provide an API type relevant to sending an email. For instance, the developer may provide the type `javax.mail.Message` — by selecting `Message` in the editor and invoking Introspector (see Figure 1–5).

Introspector uses five different heuristics that combine structural relationships and type usage information to identify types that are strongly associated with the use of `Message` (the seed). In this case, we identified six types that are strongly associated with the usage of `Message`. To assist the developer in selecting types for query formulation, we provide a confidence value and a rationale for each recommended type. A

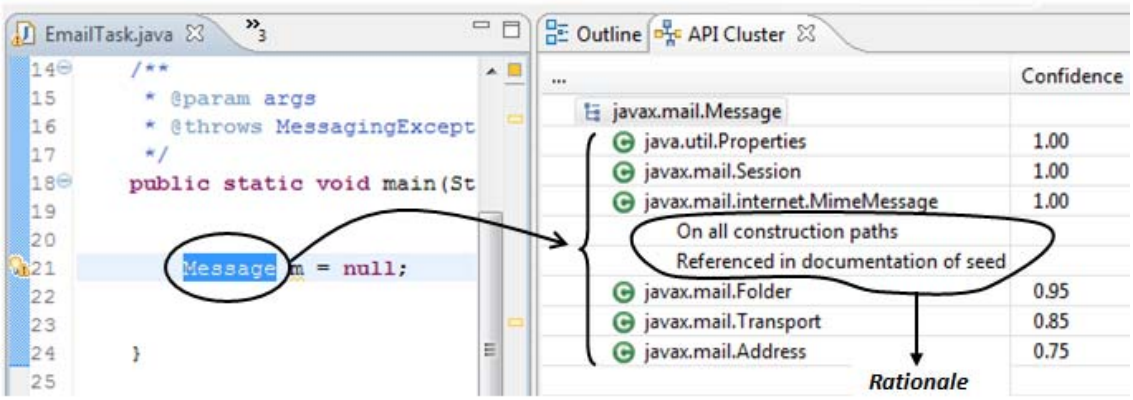


Figure 1–5: Identifying API types strongly associated with the use of `Message`.

confidence value of 1.0 implies the given type is associated with every possible use of the seed, and each rationale corresponds to one of the heuristics of Introspector. For instance, `MimeMessage` has two rationales: *On all construction paths* (to indicate that it is associated with every usage of `Message`) and *Referenced in documentation of seed* (to indicate that `MimeMessage` was singled out in the documentation of `Message`). The developer may then examine some of the recommended types using their Javadoc, and then select the types that should be used for retrieving code examples. In this case, the developer may select all the types with a confidence value of 1.0 and the `Transport` class since the task is about sending an email (see Figure 1–6).

Once the developer selects the types for the query, Introspector forwards the search query to a code repository. Introspector currently supports the Google Code Search[7] and the Koders[8] repositories. The results from the code repository are

[7] www.google.com/codesearch/

[8] www.koders.com

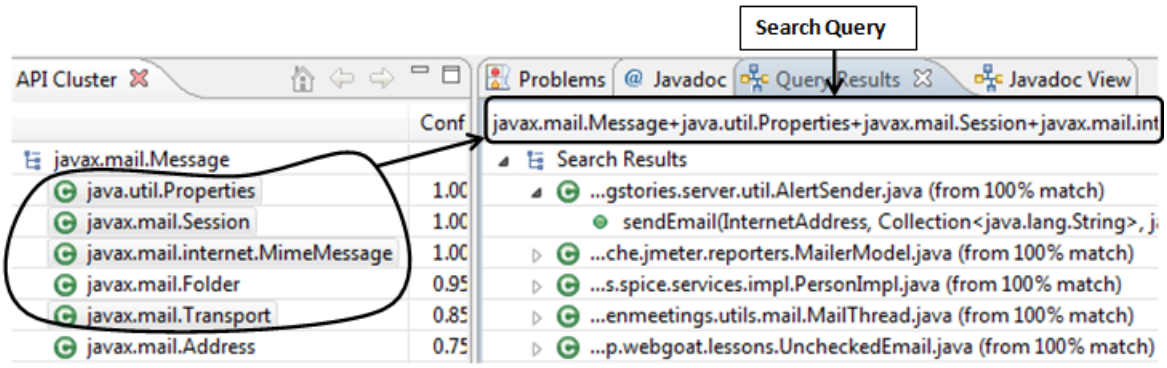


Figure 1–6: Formulating a search query from the recommendations of Introspector.

source code files, however, not every source file contains examples relevant to the search query. To assist the developer to quickly identify relevant results, we used the partial program analysis framework [10] to identify type usage within each method of the source files, then we compared the overlap between the types in the search query and the types references within each method of a given source file. An overlap of 100% implies every type in the search query is referenced within a given method. We then ranked both the source files, and the methods within each source file based on the degree of overlap, and display only the methods with an overlap of 50% and above. In the case of this example, the "sendEmail" method of the first results provides the exact code needed to send an email message. A developer can examine the code corresponding to a method or class by double clicking on the corresponding node.

Evaluation. We used ten tasks from six different APIs to evaluate our approach. The description and solution of each task form part of the documentation provided by the API designers, and were used as an oracle for evaluating the relevance of the code

examples recommended by Introspector. We began the evaluation by selecting API types to be used as the input for each task using three different seed selection schemes; then, we ran Introspector for each task, and for each input type, to formulate a search query. Subsequently, we searched for code examples using the formulated search queries, and verified whether the solution provided for each task was amongst the top three code examples recommended by Introspector. We also compared our approach to the use of natural language queries when searching for code examples. We observed that our approach is effective in identifying API types associated with the usage context of a seed, and that our query formulation strategy is more effective than the use of natural language based queries when searching for code examples. We present the heuristics of Introspector and its evaluation in Chapter 5.

1.2.4 Summary of Contributions

This thesis makes the following contributions:

I. A catalog of the different types of questions programmers ask when working with unfamiliar APIs, an objective criterion for determining hard-to-answer questions, and a catalog of qualitative evidence explaining why developers find certain questions hard to answer. These questions complement what is known about the needs of API users, and can be used to identify areas of the API learning process where tool support is missing, or could be improved.

II. The definition of a special dependency and data flow graph for APIs, called an API Exploration Graph, and algorithms which leverage the structural relationships between API elements to provide support for relating an API type to

its secondary-types and helper-types, and support on how to use and combine these related types.

III. A technique to recommend API types strongly associated with the use of a given type using heuristics based on structural dependencies and frequency measures. The recommended types could serve as a starting point for query formulation, or as an initial point for further exploration of the API for a given task.

IV. Two programming tools (API Explorer and Introspector) embodying the ideas and techniques proposed in this thesis, and evidence that these tools facilitate the process of learning how to use unfamiliar APIs.

1.3 Organization of Thesis

We continue in Chapter 2 with related work and a discussion of how this thesis relates to previous efforts aimed at supporting programmers make use of APIs. In Chapter 3, we present the details of the programming study, the analysis of the data from the study, and our results and observations. We present the API Exploration Graph, the algorithms of API Explorer, and the evaluation of API Explorer in Chapter 4. The heuristics and the evaluation of Introspector is presented in Chapter 5, and we conclude the thesis in Chapter 6.

CHAPTER 2
Related Work

This thesis builds on previous work related to programming studies, the information needs of programmers, and tools to help a developer working with unfamiliar APIs.

2.1 Programming Studies

2.1.1 API Usability Studies

Previous studies on API usability sought to identify factors that hinder the usability of APIs and to understand the trade-offs between design options. Ellis et al. conducted a study to compare the usability of the Factory pattern in contrast to constructors for object creation using five programming tasks and twelve participants [15]. Ellis et al. observed that the participants experienced difficulty and required significantly more time to construct an object with a Factory than with a constructor. Stylos et al. conducted a user study in which the usability of parameterless constructors was compared to constructors with parameters using thirty professional programmers and six tasks: they reported that programmers strongly preferred and were more effective with APIs that provide parameterless constructors [49]. In another study examining the placement of methods (that is, the class to which a method belongs), Stylos et al. reported that participants were significantly faster at identifying relevant dependencies and combining objects when the methods of a starting class referenced its dependencies [51]. Clarke uses the "Cognitive Dimensions" [7], a framework for describing API usability problems, to identify specific

usability issues with Microsoft APIs, and to help inform the design of more usable APIs. Other studies have looked at the role of web resources in learning how to use APIs: in a lab study involving twenty participants and five tasks, Brandt et al. observed that programmers used the Web primarily for just-in-time learning of new skills, and to clarify or remind themselves of previously acquired knowledge [3]. In a different study, Stylos and Myers identified several challenges developers encounter when using the Web and proposed Mica, a tool for identifying web pages with code examples [50]. Prior studies have either focused on the usability of different API design choices (e.g., Factory pattern versus constructors), or the usability issues of a specific API, or a learning resource. Our study complements previous efforts by looking at the specific questions developers ask when working with unfamiliar APIs, the difficulties they encounter answering these questions, and the degree to which existing tools support developers working with a new API.

2.1.2 Information Needs of Programmers

Several contributions have been made in the area of the information needs of programmers in general. Ko et al. conducted a study in which forty novice programmers were asked to complete several tasks using Visual Basic .NET [28], and identified learning barriers and information needs that must be satisfied for the programmers to compete the tasks. In a different study, Ko et al. observed and transcribed the activities of seventeen developers working on different tasks during a ninety minutes session [27]. Ko et al. analyzed the transcripts for the type of information that developers sought, the sources that they used, and the situations that prevented information from being acquired: they identified twenty one different information

needs of programmers, grouped into seven categories: writing code, submitting a change, triaging bugs, reproducing a failure, understanding execution behavior, reasoning about design, and maintaining awareness. They also observed that the most difficult needs to satisfy were questions about the rationale for design decisions, and that questions about APIs that could not be answered using the documentation or tools, were answered by consulting coworkers. Sillito et al. identified forty four different types of questions asked by programmers when maintaining software code, and investigated the degree to which existing tools support the questions programmers ask when modifying the source code [45]. In contrast, our study focused on the questions programmers ask when working with unfamiliar APIs, the factors that hinder programmers from answering questions about the use of an API, and the sufficiency of existing tool support.

2.1.3 Empirical Studies of Programming Strategies:

Work in the area of program comprehension has identified different strategies used by programmers to understand programs [47]. Some have argued that programmers use a top-down strategy to understand programs: that is, they work from higher level abstractions to the code [4]; Others hold that programmers use a bottom-up strategy by working from the code to the higher level abstractions [35]. Whether top-down or bottom-up, programmers either work systematically (study a system in detail to gain a global understanding of its structure) or opportunistically (focus only on areas related to their task). Clarke calls these different work styles personas, and observed that work styles are independent of a developer's level of experience or educational background [7]. We observed comparable work styles in the context of learning how

to use APIs, although the goals were different from those of program comprehension. We borrow terminologies from the program comprehension literature to help explain some of our observations.

2.1.4 Information Foraging:

Pirolli and Card proposed the information foraging theory to help explain how information-seekers search for information [36]. They observed that information-seekers use the same strategies used by predators in the wild when making decisions about where to look for information, what search strategies to use, or which information to consume. Central to the theory are *information scents* (cues which guide information-seekers to relevant information), *information patches* (the information sources), and the *information diet* (the decision of how to select the most profitable patch). Pirolli and Card observed that information-seekers would adapt their search strategies and environment, if need be, to maximize the gains of relevant information per unit cost. In the user study, we make observations consistent with the information foraging theory: programmers use cues (for instance, special words such as tutorials or examples) in the documentation and the Web to locate relevant classes and methods, and would adapt their strategies (for instance, reformulate a query) to optimize the gains of relevant information.

2.1.5 Distinction from Previous Programming studies

In our study (Section 1.2.1), we isolated specific questions programmers ask when working with unfamiliar APIs, investigated factors that prevented programmers from answering questions about the use of APIs, and looked at areas of the API learning process where tool support is missing, or could be improved. The contributions of our

study are similar to that of Sillito et al. that looked at the questions programmers asked when maintaining the source code [45], but ours is the the context of API learning. In addition, we utilized a more objective criterion for determining hard-to-answer questions, provide a catalog of qualitative evidence explaining why developers find certain questions hard to answer, and used more varied sources of evidence (such as navigation paths, verbalizations, and the time spent on various micro tasks) in our analysis than either Sillito et al., or Ko et al [27]. Our work also complements previous efforts on the needs of API users (which has mainly focus on a high-level perspective of the challenges programmers encounter when working with unfamiliar APIs) by investigating low-level questions asked by programmers when learning to use unfamiliar APIs.

2.2 Tools To Support API Usage

2.2.1 Improving Code Completion Tools

IDEs, such as Eclipse and Microsoft Visual Studio, provide code completion systems that support programmers when learning how to use APIs. These systems are triggered by special characters, such a period after the name of an object, or by special key combinations, such as *Ctrl+Space*, after a text prefix. The code completion system displays an alphabetical list of accessible members of the object on which the request is made, and progressively narrows the list of suggestions to match the prefix entered by the user. Previous work on code completion systems focused on re-ordering the list of suggested methods, or on predicting the method most likely to be called next in a given context. Robbes and Lanza [38] modeled the change history of systems as atomic operations and used this change history to predict the

method of an object most likely to be called next. Bruch et al. used example code in code repositories to improve the ordering of the list of suggested methods [5]. These previous works share a common limitation: they can only suggest the members of the object on which code completion is requested. API Explorer(Section 1.2.2) is the first novel extension of code completion, to our knowledge, capable of suggesting relevant methods on other objects, identifying related API elements of a given class or method, and providing support to API users for combining multiple objects.

2.2.2 Documentation Improvement Tools

Some researchers have investigated ideas and techniques for improving the API documentation to reduce the difficulties of learning how to use APIs. Kim et al. proposed eXoaDocs [26], a tool that integrates code snippets mined from the source code search engine Koders into the Java API documentation (Javadoc). eXoaDocs queries the Koders search engine for code examples that use a given API method, eliminates segments from the code examples not relevant to the use of the API method, and integrates the resulting code snippet in the description section of the method in the API documentation. Jadeite [52], proposed by Stylos et al., uses usage statistics of the classes and methods of APIs from code examples on the Web to help programmers find commonly used API elements from the documentation, and also integrates code snippet on how to create instances of API classes in the documentation. Jadeite also introduced the concept of *"placeholders"*, a feature which enables API designers or users to annotate the API documentation with classes they expect to exist in a given package of the documentation, or methods they expected to exist on a given class, and to add forward references to the actual parts of the APIs that should be used in

lieu of the "placeholders". Dekel and Herbsleb proposed eMoose [11], a tool which makes programmers aware of important usage guidelines from the documentation of API methods which if violated could lead to bugs.

Jadeite and eXoaDocs used techniques that rely on the existence of large collections of example usages of an API to be effective. Consequently, newer APIs or non-popular parts of existing APIs may not have sufficient example usages for Jadeite and eXoaDocs to be beneficial. API Explorer, in contrast, leverages the structural relationships between API elements, not collections of code examples, to assist a developer in discovering relevant types and in coordinating the use of multiple API elements. Jadeite's "placeholders" concept is similar to API Explorer's feature which recommends relevant methods on other objects, but Jadeite relies on API designers or user to first add the "placeholders", and to subsequently link "placeholders" to parts of the APIs that should be used instead. API Explorer, in contrast, automatically identifies relevant methods on other objects, or relevant classes in other packages, using structural relationships between API elements, and presents this information through the code completion feature of the IDE.

2.2.3 Example Recommendation Tools

Example recommendation tools leverage the proliferation of code examples on the Web and in open-source repositories to suggest source code that may be relevant to a task and API a developer is working with. These tools differ in the techniques used to retrieve code examples and in the kind of support afforded to API users. CodeBroker used comments and method signatures written by the programmer to retrieve

source code examples that demonstrate the usage of APIs element from a repository [55]. Strathcona used the structural context of the code under development such as the class in which an API element is to be used, its parent class and interfaces, and the signature of API methods to retrieve relevant code examples from a repository [21]. Prospector [31], ParseWeb [53], and XSnippet [41] take queries of the form "source-type $\rightarrow$ destination-type" as input, and recommend source code examples with method invocation sequences that show how to obtain the destination-type from the source-type. Jiang et al. [23], Salah [42], and Heydarnoori [19] proposed tools which used dynamic analysis of the interaction between sample applications and APIs to identify valid usage scenarios and valid call sequence of API methods. Code Conjurer [22] and Code Genie [29] used test cases written by programmers to retrieve example usages of APIs element from code repositories.

Prospector and XSnippet are the most similar to API Explorer because they combined the use of code examples with the structural relationships between API elements such as the return types and the parameters of methods to identify relevant method call sequences which link a source-type to a destination-type. However, the support provided by Prospector and XSnippet is limited to object instantiation only, and for both tools to work, the programmer must be aware of the destination-type, which may not always be feasible. With API Explorer, programmers can obtain instantiation support with only the source-type. Furthermore, API Explorer extends these works by providing support for locating relevant methods on different objects, and for identifying API elements relevant to the use of a class or method.

2.2.4 Distinction from Previous Programming Tools

The decision to present the information contained in the API Exploration Graph through the code completion feature of an IDE was inspired by empirical evidence that the code completion feature of IDEs is the primary way through which programmers explore APIs [51]. With API Explorer, recommendations are presented through the code completion feature of the IDE, and programmers are no longer limited to seeing just the public members of the type on which code completion is requested: we extend the toolbox of programmers by recommending relevant methods on objects different from that on which code completion is requested. Furthermore, API Explorer, unlike previous tools, is not limited by the absence of extensive code examples since it relies on the structural relationships between API elements to support programmers in relating and combining API elements.

2.3 Query Formulation Tools

2.3.1 Natural Language Techniques

Natural language techniques for query formulation seek to solve the problem: *"given one or more keywords, find the related words and the corresponding code elements in a given code base"*. These techniques use words and phrases in the source code to locate code elements that match a given query, to recommend alternative words for query reformulation, or to extract word usage in the source code for code comprehension. JIRiSS extracts identifiers and comments in the source code to form a corpus of text, then uses Latent Semantic Indexing to suggest close matches for misspelled query terms, and to suggest code elements matching a query entered by the programmer [37]. In Latent Semantic Indexing, the source code of a system is decomposed into

a corpus of text documents (the level of granularity is either the class or method), then a vector space is generated from the corpus. A document is then created from the query entered by a developer and compared against each document in the corpus to identify documents that are similar to the developer's query.

Shepherd et al. used verbs and direct object — the object a verb acts on, e.g., the identifier "sendEmail" has verb "send" and object "Email" — pairs from method signatures and comments in the source code to expand a poorly formed query into a more effective search query, and to present search results in a format that is easy to understand [44] . Hill et al. extended the work of Shepherd et al. by capturing not only verbs, but also nouns and prepositional phrases from source code identifiers [20]. Ohbas and Gondow used Term Frequency-Inverse Document Frequency (TF/IDF) to automatically extract keywords that represent major concepts in the source code to facilitate program comprehension [34]. TF/IDF is a statistical measure used to evaluate the importance of a word in a given document for a given corpus. Maskeri et al. used the Latent Dirichlet Allocation (LDA) statistical model to extract domain topics from the source code to facilitate program comprehension [32]. In LDA, a document is considered to have a mixture of topics and each word in the document is associated to at least one of the topics. Given the source code of a system as the corpus, Maskeri et al. used LDA to identify a set of topics for the corpus, and to associate the words (names of source code elements) in the document to a topic to facilitate program comprehension. Others have combined natural language techniques and structural analysis for query expansion: for instance, given an query, Exemplar uses the help documentation of APIs to identify API calls whose descriptions contain

words corresponding to the query, and forwards the API calls to a corpus of code examples to identify relevant code snippets[17].

2.3.2 Structural-Based Techniques

Structural-based techniques use the relationships between source code elements to generate queries, or to suggest elements relevant to a given query. Strathcona uses the structural context of the code under development (the containing class, its parent classes and interface, methods, and fields) to generate queries, and to recommend code examples relevant to the current context of the programmer [21]. SNIFF combines the API documentation and code from open source systems to facilitate query formulation: it begins by annotating method calls in the source code with their corresponding Javadoc description, then matches incoming queries to relevant code snippets using the generated annotations. SNIFF also ranks the returned snippet based on their relevance to the search query [6]. Several other techniques have been proposed to solve the problem: *"given a particular function, find the related functions"*: Altair takes the signature of a method as input, and recommends other related API methods based on the extent to which the data they access overlap [30]; Robillard used program elements (methods and fields), and relationships between elements (method calls and field access) to recommend code elements of interest during code maintenance [39]; Saul et al. extended the work of Robillard by using a callgraph and an algorithm based on the steady state of a random walk on the callgraph [43].

2.3.3 Distinction from Previous Query Formulation Tools

Our work on query formulation can be framed as *"given a particular API type, find the related types"*: where the related types can be used as a starting point to formulated effective queries, or to further explore an API. In this sense, our work is related to most of the previous works cited in this section: we all share the goal of looking for code elements that match a given query and search criteria. On the other hand, our work has several differences: first, our work is in the context of API comprehension while most of the previous efforts are concerned with code comprehension and maintenance. Second, we work at the level of API types — classes and interfaces, and are interested in recommending types, as opposed to API methods, that are strongly associated to the use of a given API type. Lastly, our approach relies only on the binary of the API to recommend related types. This is a significant advantage since the binary is always available to the programmer, but other types of information such as the client code of an API may not always be available.

CHAPTER 3
Asking and Answering Questions About Unfamiliar APIs: An Exploratory Study

3.1 Introduction

To understand the challenges of learning a new API and how best to support developers, we conducted an exploratory study in which 20 participants worked on two programming tasks using two different real-world APIs. We had three main goals for the study: first, to understand the nature of the type of questions developers want answered when learning to use APIs; second, to identify the questions developers have difficulty answering when using a new API and to investigate the cause of the difficulties; and third, to suggest ideas on how to improve the tool support available to developers learning to use APIs. We begin by presenting the study methodology (Section 3.2), then we present the analysis of over 20 hours of screen captured videos and the think-aloud verbalizations of the participants generated by the study (Section 3.3), and finally, we present the results and observations of the study, together with the supporting evidence (Section 3.3.3). We conclude the chapter with the implications of our observations on API design, documentation, and tool design.

3.2 Methodology

We conducted a laboratory study in which 20 participants worked on the same two programming tasks using real-world APIs. All the 20 participants were unfamiliar with the APIs used in the study.

3.2.1 The Participants

We recruited participants from the student population of the department of Computer Science at McGill University using on-campus posters and mailing lists, and promised a monetary compensation of $20. Respondents were prescreened using a questionnaire that asked potential candidates about their programming experience and their knowledge of Java and Eclipse.

We selected 20 participants from the respondents for our study. The selected participants reported a minimum of 1 year programming experience with Java, 1 year experience working with the Java API documentation (i.e., Javadoc), and some experience programming with Eclipse. Our participants reported between 1 and 6 years of experience programming with Java, with a median of 3.5 years, and an average of 1.5 years of paid programming experience. Five of the 20 participants were female, and our participant pool included 4 Ph.D. students, 11 M.Sc. students, and 5 senior undergraduate students. Although all of our participants were students, they are representative of the population of interest and their expertise level is comparable to that of recent graduates in software development positions, which is our target population since our work aims to support novice programmers.

3.2.2 The Tasks

We asked each participant to complete two programming tasks: the first task using the JFreeChart[1] API, and the second task using the Java API for XML Processing

[1] www.jfree.org/jfreechart/

(JAXP).[2] JFreeChart is a popular open-source API for generating charts. We used version 1.0.13 of the JFreeChart API, which has 37 packages and 426 non-exception classes. JAXP is an API for validating and parsing XML documents, developed by Sun Micosystems. We used version 1.4 of the JAXP API, which has 23 packages and 207 non-exception classes.

We selected tasks that involved combining multiple objects since previous work on API usability observed that developers experienced the most difficulty performing such tasks [51]. We reasoned that tasks requiring the combination of multiple objects are more likely to reveal a variety of questions developer want answered and typical challenges they encounter when learning to use APIs. The participants were given a maximum of 35 minutes per programming task.

Chart-Task.

We asked the participants to use the JFreeChart API to construct a pie chart with three slices (45% Undergrads, 35% Master's, and 20% Ph.D.s), and to save the chart to a file in a graphic format. To complete this task, a participant needed to construct and configure at least five API types (`JFreeChart`, `PiePlot`, `PieDataset`, `ChartFactory`, and `ChartUtilities`), and had to discover key relationships between the types: for instance, the relationship between `JFreeChart`, the type for representing charts, and `ChartUtilities`, the type needed to save the chart.

XML-Task.

We asked the participants to use the JAXP API to verify whether the structure

[2] http://jaxp.java.net

of an XML file conforms to a given XML schema. The participants were provided with both an XML file and an XML schema file, and were asked to implement a solution that returns true if the XML file conforms to the given XML schema, and false otherwise. This task required the combination of at least four API types (`Schema`, `Validator`, `SchemaFacotry`, and `Source`) and was selected because of the unique challenges it presents to object construction — all the required types are abstract with no subtypes; the types must be created from factory or public methods.

3.2.3 Study Setting

Participants completed the study using the Eclipse IDE (version 3.4) and were permitted to use any of the features of the IDE. Two main information sources were used in the study: the documentation of the APIs and the Web, which provides access to example usages of the APIs. These information sources have been reported to be the primary learning resources for API users [50, 52]. We provided the participants with the Firefox browser to access these information sources, and disabled the browser's history feature to prevent any learning effect between participants.

The programming studies were conducted individually in our research lab. The participants began each study by watching a four-minute video tutorial about the think-aloud protocol. Participants were then given time to practice thinking aloud while working on a web search task. Soon after, the participant was given the instructions for the Chart-Task and was given a maximum of 5 minutes to go over the task requirements and to ask questions relating to the requirements. To avoid influencing the strategy of the participants, we did not identify the classes or packages of the APIs required to complete the tasks, as was the case with previous studies [51].

Also, the participants were advised to proceed as they would typically do when learning a new API.

Once the participant was satisfied with the task requirement, we loaded an Eclipse project which contained a class with an empty main method and the libraries of the relevant API. We then showed the participant how to use the Firefox browser to access the Javadoc pages of the APIs from the bookmark menu. At this point, the screen and voice recording software — Camtasia, version 4 — was started, and the participant was asked to begin the task. The participant was asked to move to the next task upon completion of the Chart-Task, or once the 35 minutes allocated for the task elapsed. The tasks were completed in the same order by all the 20 participants.

3.2.4 Data Collection

We used three data collection techniques in our study: the think-aloud protocol, screen captured videos, and interviews. In the think-aloud protocol [2], participants are asked to verbalize their thought process while solving a given task. Having participants think-aloud was particularly useful in our study as it permitted us to obtain an insight into the participants' understanding of the structure of the APIs, to identify the types of questions participants ask when learning to use APIs, and to understand why a participant may have difficulty answering a given question. We also conducted semi-structured post-study interviews in which the participants were asked to comment about the challenges experienced during the programming study. The interviews lasted 5 minutes.

The screen contents, the verbalizations of the participants, and the interview sessions were captured using Camtasia. The study produced a total of 40 different programming sessions and about 20 hours of screen-captured videos and verbalizations of participants working with unfamiliar real-world APIs.

3.3 Data Analysis and Results

Our analysis focused on the questions the participants wanted answered about the use of an API. Our goal was to identify those questions that are difficult to answer, to understand why these questions proved difficult to answer, and to recommend programming tools that could facilitate the API learning process. Our method for analyzing the data involved three phases: identifying the different types of questions asked by the participants, categorizing the questions, and coding the exploration patterns used by the participants when searching for answers to these questions. Our analysis approach was inspired by the work Sillito et al. [45] which looked at the questions developers asked during code modification tasks. For brevity, we refer to a participant by their ID (for instance, P5 for the fifth participant) and to the tasks as T1 (for the Chart Task) and T2 (for the XML Task).

3.3.1 Identification of questions

In this phase, we went through the screen-captured videos and verbalizations to produce a list of *specific* questions asked by each participants, and to identify segments of the videos, which we called *episodes*, corresponding to the approach used to answer each question. Some questions were explicit: for instance, participant P11 asked *"How do I create a Graphics2D object?"* while working on the Chart-tasks. Other questions were easily inferred from the actions and verbalizations of the participants:

Table 3–1: Different types of questions observed during the study: Part 1

Discovering Functionality
Q.1 Which packages or namespaces of an API provide types relevant to my task? *"I'm trying to find out which package has classes for creating a pie chart"* — P5,T1
Q.2 Is there an API type that provides a given functionality? *"the task says I should create a pie chart; I'm expecting some sort of a PieChart class to be available"* — P18,T1
Q.3 Does an API type provide a method for performing a given operation? *"Is there a method on* `BufferedImage` *that helps to save?"* — P10,T1
Q.4 What is the functionality of a given API type? *"Let's look at what the* `Validator` *class does"* — P18,T2
Q.5 Can a method intended to perform operation A be used to perform operation B? *"I'm hoping that the draw method can be used to save to a file, but I'm not too optimistic about it"* — P6,T1
Q.6 Which keywords best describe a functionality provided by the API? *"I'm going to use the Firefox search to look if there's any thing involving[containing the word] "schema""* — P9,T2
Understanding Relationships
Q.7 How is the type X related to the type Y? *"How is* `Validator` *related to* `Schema`*?"* — P18, T2
Q.8 How do I get an object of type X from the type Y? *"I need to figure out how to get a* `BufferedImage` *from a* `PiePlot`*"* — P6,T1
Q.9 Which elements of the API are of the type X? *"Which classes of the API are* `Comparable`*?"* — P11, T1
Q.10 Is the object X of the type Y? *"let's see if* `RenderedImage` *takes* `BufferedImage`*"* — P1,T1

Table 3–2: Different types of questions observed during the study: Part 2

Discovering Dependencies
Q.11 Does the API provide a helper-type for manipulating objects of a given type? *"let's see if there are classes related to* `BufferedImage` *which can give me the possibility to write the image to a file"* — P10,T1
Q.12 How do I create an object of a given type without a public constructor? *"the constructor is protected; so how do I create a* `Graphics2D` *object?"* — P11,T1
Q.13 Which other API elements are necessary to use a given API type? *"I think I need something else that would save the chart to an image"* — P1,T1
Selecting Relevant Elements
Q.14 Which subtype of an interface or class is the most appropriate for my task? *"I don't know exactly which subtype of* `Source` *to use for reading an XML file"* — P6,T2
Q.15 Which types of a given domain (package, namespace) are relevant to my task? *"Which classes of the "parsers" package could be used for validation?"* — P18, T2
Q.16 Which method from a list of overloaded methods is relevant to my task? *"I'm trying to find the appropriate create-piechart method because it seems to be overloaded"* — P16,T1
Understanding Behavior
Q.17 What role do the arguments of a given method play in its usage? *"we have a* `newInstance(String)` *method that takes a* `String` *argument and I have no idea what this* `String` *is suppose to be"* — P9,T2
Q.18 What is the valid range of values for a primitive argument, such as an integer, of a given method? *"I don't know if this [double] value should be between 0 and 1"* — P10,T1
Q.19 Is NULL a valid value for a non-primitive argument of a given method? *"let's use NULL for* `Comparable` *and see if the method throws an exception"* — P1,T1
Q.20 How do I determine the outcome of a method call? *"[the method]* `Validator.validate(Source)` *returns void; how do I know the results of the validation?"* — P12,T2

for instance, P1 came across the method `ImageIO.write(RenderedImage, ...)`, and said "let's see if `RenderedImage` takes `BufferedImage`", then went ahead and used a `BufferedImage` object where `RenderedImage` was expected. The actions and verbalization of P1 in this example is phrased into the question: *"Is* `BufferedImage` *of the type* `RenderedImage`*?"*. After identifying the list of specific questions for each participant, we then developed *generic* versions of the questions that slightly abstract from the specifics of a given API. For instance, the question *"Is* `BufferedImage` *of the type* `RenderedImage`*?"* can be stated more generally as *"Is the object X of the type Y?"*. Based on these generic questions, we identified twenty different types of questions asked by the participants across both tasks (Tables 3–1 and 3–2). We also provide a specific instance for each generic question as an example. The generic questions highlight, to a certain extent, the type and scope of the information developers need when learning how to use APIs. The number of times each type of question was observed (# of occurrences) and the number of participants that asked each type of question (# of participants) are listed in Table 3–3.

Categorization of questions.

In the second phase of our analysis, we looked at the similarities, differences, and connections between the different type of questions, producing five different categories of the questions observed during the study.

- **Discovering Functionality**: The participants asked these type of questions at the initial step of API exploration. At this stage, a participant is seeking to identify the types, methods, or fields in an API relevant for implementing a given task, or to identify keywords from the problem domain (the tasks) that

Table 3–3: Frequency measures for questions Q.1 to Q.20

Question ID	# of occurrences	# of participants
Q.1	31	16
Q.2	11	7
Q.3	58	19
Q.4	32	17
Q.5	3	2
Q.6	38	13
Q.7	8	7
Q.8	2	1
Q.9	4	4
Q.10	2	2
Q.11	19	13
Q.12	57	19
Q.13	6	5
Q.14	29	17
Q.15	27	17
Q.16	4	4
Q.17	23	17
Q.18	3	3
Q.19	4	4
Q.20	15	13

maps to relevant elements in the solution domain (the documentation or code examples). We identified six types of questions in this category (Q.1 to Q.6, Table 3–1).

- **Understanding Relationships**: The participants asked these type of questions when they needed to understand the relationships between API types. At this stage, participants had identified the types of an API relevant to a task, but needed to understand how to correctly combine and coordinate the use of the relevant elements. For instance, P18 had identified both `Validator` and

`Schema` as relevant to validating an XML file against a schema (T2), but needed to understand how these types are connected: *"How is* `Validator` *related to* `Schema`*?"*. We identified four types of questions in this category (Q.7 to Q.10, Table 3–1).

- **Discovering Dependencies**: The questions in this category came up after a participant had started implementing a task: the participant suddenly realized that the types they had discovered provides only partial support for their task, and that other helper-classes were required to implement the task. For instance, P10 attempted to save the chart as an image in T1 but realized that `BufferedImage` provides no method for saving; he then looked for a class that can be used for saving `BufferedImage` image: *"let's see if there are classes related to* `BufferedImage` *which can give me the possibility to write the image to a file"*. We identified three types of questions in this category (Q.11 to Q.13, Table 3–2).
- **Selecting Relevant Elements**: The questions in this category were observed when a participant had to make a choice between two or more subtypes of a class or an interface, or select a method from a list of overloaded methods. We identified three types of questions in this category (Q.14 to Q.16, Table 3–2).
- **Understanding Behavior**: The questions in this category were observed when a participant needed to understand how a certain API method "works". What its valid inputs are, its expected behavior, or the role of each of its arguments. We identified three types of questions in this category (Q.17 to Q.20, Table 3–2).

3.3.2 Abstraction of Developer Behavior

We needed a high-level abstraction of the actions of the participants to facilitate the analysis of their behavior and the challenges they experienced when learning to use APIs. Since our analysis is centered around the questions asked about the use of the APIs, we transcribed the segments of the videos corresponding the time frame during which a participant asked and searched for answers to a given question. Specifically, for each participant and for each episode corresponding to a specific question, we transcribed the video into a series of actions that summarizes the steps taken by the participant to answer the question. We considered the following actions:

- **Browse:** the participant looked through a list of API elements (packages, types, or methods) either within Eclipse, or in the documentation, before making a selection. The Browse action has a *target* to denote the items (either packages, classes, methods, or search result) the participant was browsing through.
- **Select:** the participant selected an item from a list of API elements, or the results of a search query after browsing. The Select actions has a target — the name of the item selected, and a flag (Yes/No) to indicate if the selected element was relevant to the question being answered. The target of the Select action could also be *None*, if no item was selected.
- **Read:** the participant focused on a portion of text or code. The Read action has a target — the name of the element being read, and a section (e.g., either the introduction section, constructor section, or the method description of an API element) to indicate the location the participant focused on.

- **Navigate:** the participant followed a dependency or a link to another element. The Navigate action has a *target* — the name of the item navigated to, a flag (Yes/No) to indicate if the *target* led to information relevant to answering the question.
- **Search:** the participant performed a search of the documentation or the Web. The Search actions has a *target* (Documentation/Web), and a flag (Yes/No) to indicate if the search query contained the name of an API element.
- **Switch:** the participant moved from the documentation to the Web, or IDE, and vice versa.
- **Use:** the participant attempted to use an API element or code example found on the Web. This action has a *target* — the element or code the participant attempted to use, and a flag (Yes/No) to indicate if the participant was successful.
- **Backtrack:** the participant stepped back to a previous location of certainty, then decides to explore a different path. This action has a *target* — the location the participant backtracked to.

As an example, Table 3–4 shows a partial transcript[3] of the participant P15 looking for information on how to save a `BufferedImage`; the transcribed actions is shown under the "Action Sequence" column. P15 started by navigating to the documentation page of `BufferedImage`, browsed through its subtypes, and then selected the

[3] The entire transcripts for both tasks, and for the twenty participants are available as an online appendix at www.cs.mcgill.ca/~eduala/apistudy

Table 3–4: Transcript excerpt for participant P15 — Chart Task

Time	Question	Action Sequence
0:18:10	How do I save a BufferedImage?	**Nav**[BufferedImage]:**Browse**[subtypes]: **Select**[WritableImage,No]:**Read**[WritableImage,Intro]: **Backtrack**[BufferedImage]:**Browse**[methods]: **Read**[createGraphics,description]:**Switch**[web]: **Search**[web,Yes]:**Select**[3rd,Yes]: **Read**[ImageIO.write,code]:**Use**[ImageIO.write,Yes]

subtype `WritableImage`, not relevant to saving an image. P15 read the introduction section of `WritableImage`, then backtracked to `BufferedImage`. P15 then browsed through the methods of `BufferedImage`, focusing on the `createGraphics` method, before switching to the Web. P15 then searched the Web with a query containing an API element, selected the third results, read through the code example and discovered the `ImageIO.write(RenderedImage, ...)` method. P15 then used `ImageIO.write(RenderedImage, ...)` successful to save the image to a file.

What is a difficulty?

As part of our analysis, we intended to identify the types of questions that proved difficult for the participants to answer and to understand the cause of the difficulty. To accomplish this, we needed an objective measure as to what constitutes a difficulty in the context of API learning. We decided not to use the amount of time taken to answer a question as the main measure of difficulty since significant performance variations have been observed amongst developers with similar level of experience [8]. At a higher-level, we observed that some of the actions, or sequence of actions, of a participant that reflected a lack of progress in the search for answers to a given

question would serve as a good measure for capturing difficulty. We used the following action sequences as a definition of the difficulty participants encountered when answering questions about the use of the APIs:

- **Use**[***target*****, No**]: This action sequence captures instances in which a participant attempted to use an API element but was unsuccessful because the API does not support the given usage. For instance, the participants P6 and P8 commented *"How can I get an instance of Validator?"* after their attempt to instantiate *Validator*, an abstract class, from the default constructor failed. This object instantiation difficulty is captured by the action sequence *Use[Validator.Constructor, No]*. We observed that participants often had expectations about the design of an API and expressed frustration when the structure of an API did not match their expectations.
- **Browse**[***list***], **Select**[***target*****, No**], ..., then **Backtrack**[***list***], or **Navigate**[***target*****, No**], ..., then **Backtrack**[**...**]: The **Select**[***target*****, No**] and **Navigate**[***target*****, No**] action sequences capture instances in which a participant went down an irrelevant information search path. Once a participant realized the information on a given path was not relevant to answering their question, they backtracked to previous location of certainty, and then chose a different path to explore: captured by the **Backtrack** action. We consider action sequence **Select**[***target*****, No**], or **Navigate**[***target*****, No**], followed by a **Backtrack** as an indication of difficulty in searching for answers to a given question. The participants relied on cues in the documentation or code examples when looking for answer to a given question. At times, the clues were not available or perceivable. In the

Table 3–5: A summary of the difficulties participants experienced answering different types of questions about the use of APIs.

Question ID	# of occurrences	occurrences with difficulty	# of participants	participants with difficulty
Q.1	31	1	16	1
Q.2	11	5	7	3
Q.3	58	14	19	13
Q.4	32	3	17	3
Q.5	3	1	2	1
Q.6	**38**	**17**	**13**	**7**
Q.7	**8**	**5**	**7**	**5**
Q.8	2	1	1	1
Q.9	4	2	4	2
Q.10	2	0	2	0
Q.11	**19**	**17**	**13**	**12**
Q.12	**57**	**35**	**19**	**17**
Q.13	6	1	5	1
Q.14	29	8	17	8
Q.15	27	5	17	4
Q.16	4	0	4	0
Q.17	23	1	17	1
Q.18	3	2	3	2
Q.19	4	1	4	1
Q.20	**15**	**11**	**13**	**11**

absence of strong cues, participants were left to guess which search paths to follow, and some participants inadvertently went down irrelevant search paths. We summarize the difficulties the participants experienced answering the different types of questions in Table 3–5. For each question, we provide the number of times the question was observed (column two), the number of instances with a difficulty (column three), the number of participants who posed the question (column four), and the number of participants that experienced a difficulty answering the question.

As a baseline, we considered a question difficult to answer if all of the following conditions apply:

- At least half of the participants who posed a question experienced some difficulty answering the question.
- At least five participants experienced some difficulty answering the question.
- A difficulty was observed in about half the total number of the instances in which a question was asked.

For instance, we considered the participants to have experienced difficulty answering question Q.20 since eleven of the thirteen participants who posed the question experienced difficulty answering it, and since a difficulty was observed in eleven of the fifteen instances in which the question was asked. We identified five questions that proved difficult for the participants to answer (boldfaced in Table 3–5):

Q.6 Which keywords best describe a functionality provided by the API?

Q.7 How is the type X related to the type Y?

Q.11 Does the API provide a helper-type for manipulating objects of a given type?

Q.12 How do I create an object of a given type without a public constructor?

Q.20 How do I determine the outcome of a method call?

3.3.3 Observations

We present our findings as observations of the challenges a developer may encounter when learning to use an API, along with the supporting evidence for each observation. These observations are supported by the results of our analysis of the data from the study, and by the verbalizations of the participants.

Observation 1 (Discovering Relevant Dependencies). *Discovering relevant API types not accessible from the type a developer is working with is a major challenge to API learners.*

Three questions (Q.7, Q.11, and Q.12) of the five we identified as being difficult to answer involved a participant either looking for types related to, and relevant to the use of a type they were working with (*"let's see if there are classes related to* `BufferedImage` *which can give me the possibility to write the image to a file"* — P10, T1), or a participant seeking to discover the relationship between API types (*"How is* `Validator` *related to* `Schema`*?"* — P18, T2). Although different, these three questions illustrate a common problem: the participants experienced significant difficulty when relevant API types were not accessible from the type they were working with (i.e., these helper-types were not referenced or reachable from any of the public members of the type the developer was working with). For instance, in the Chart Task, the participants could save the `JFreeChart` object using `ChartUtilities.saveChart(JFreeChart,...)`, but most experienced difficulty locating `ChartUtilities` because it is not accessible from `JFreeChart`. Twelve of the 20 participants in our study experienced some difficulty finding `ChartUtilities` (Q.11, Table 3–5), and three of the participants were unable to complete the Chart-Task because they could not locate this relevant dependency. This observation corroborates the findings of Stylos et al. [51] that method placement — the class on which a method is placed — affects API usability. However, *Observation 1* extends beyond method placement: the participants also had difficulty discovering the relationships

Table 3–6: A comparison of the search queries with, and without, an API element.

Search Queries With an API Element	
Total Queries:	25
Reformulated Queries:	4
Successful Queries:	21

Search Queries Without an API Element	
Total Queries:	13
Reformulated Queries:	12
Successful Queries:	1

between types (Q.7, Table 3–5), or creating objects for types without a public constructor (Q.12, Table 3–5) because the relevant helper-types were not accessible from the type they were working with. Participant P4 attributed this difficulty to the lack of a *"cross-reference in the API that says get a* `Validator` *instance from a* `Schema`*"*.

Observation 2 (Query Formulation). *In the context of our study, having an API element as one of the keywords in a search query was an effective strategy for locating relevant code examples.*

We analyzed the queries the participants formulated when searching for code examples on the Web and observed that queries that contain an API element were more *successful* (that is, led to a relevant code example) than those without an API element (see Table 3–6). There were a total of 25 queries containing an API element, and of those, only four were reformulated, and 21 of the 25 queries containing an API element led to a relevant code example. On the other hand, there were a total of 13 queries without an API element: 12 of the 13 queries were reformulated, and only one of the 13 queries led to a relevant code example. As an example, participant

P18 started the XML-Task with the search query *"java xml processing tutorials"* but found no relevant code example. He then turned to the documentation where he identified the `Schema` class as relevant to the validation task. Participant P18 then reformulated the search query to *"java xml validation against schema"* from which he found a relevant code example.

A complementary observation about query formulation is the difficulty of guessing keywords that correspond to word usage in APIs, or their documentation. This difficulty was captured by the question Q.6 (*Which keywords best describe a functionality provided by the API?*): up to seven of the thirteen participants who asked this question experienced some difficulty guessing a correct keyword.

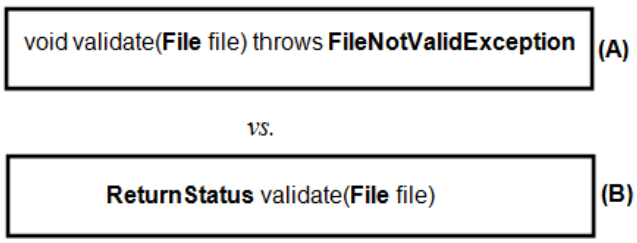


Figure 3–1: Two possibilities for communicating method-level execution failures.

Observation 3 (Exceptions). *The use of an exception to communicate the outcome of a method execution hinders API comprehension.*

There is a long standing debate in the software development community regarding whether an *exception* (as in Figure 3–1 (A)), or a *return-status-object*[4] (as in Figure 3–1 (B)) should be used to communicate method-level execution failures, and when each design choice may be appropriate [9, 25, 46]. When an exception is used to communicate the failure of the `validate(File)` method, the implication is that the `File` is considered valid if the method does not throw an exception. In other words, the exception is used to communicate the return status (*failure* or *success*) of the `validate(File)` method: the validation is said to have failed if the method `validate(File)` throws an exception, and successful otherwise. However, the extent to which this implication is apparent to a developer learning to use an API remains uncertain. In the XML tasks, the participants had to use the method `Validator.validate(Source)`, that used an exception to communicate outcome, to validate an XML file. We observed that the use of an exception to communicate outcome was problematic to the participants: 11 of the 20 participants experienced significant difficulty realizing the implication that if the method `validate(Source)` does not throw an exception, then the `Source` file is considered valid (Q.20, Table 3–5). The 20 participants spent an average of 4.2 minutes each before becoming aware of the implication that the XML file is considered valid if the validation method does not throw an exception; the average time spent to make this discovery increases to

[4] We used the word *return-status-object* to represent either an error code (a primitive such as a boolean or an integer), or an object that contains the return status of a method execution.

6.7 minutes if we consider just the 11 participants that faced a difficulty. Participants P5, P14, and P20 were unable to make the discovery within the alloted time for the task, even after spending 14 minutes, 9 minutes, and 21 minutes, respectively, on this part of the task.

We looked at the verbalizations of the participants and the post-study interviews in an attempt to understand why they could have missed the implication that the XML file is considered valid if the validation method does not throw an exception. We identified two possible reasons for this difficulty.

The expectation of the participants. The participants expected the API to provide a validation method with a return-status-object (such as in Figure 3–1 (B)), but `validate(Source)` had no return-status-object: "`validate(Source)` *does not give us something like true or false; I better look for a method that gives us a boolean*" — P1. Not finding the expected method (that is, a validate method with a return-status-object), participants would initially assume that `validate(Source)` is not the right method to use, instead of making the connection that an exception is used to communicate the success or failure of the validation. They would then spend time looking for other methods in the API with a return-status-object that could be used to validate the XML file. Not finding an alternative, the participants would then return to the `validate(Source)` method, re-read its documentation, and realize that the XML file is considered valid if the validation method does not throw an exception.

Disagreement between API designers and our participants as to what constitute an "*error condition*". The second reason expressed by the participants as to why they experienced difficulty associating exceptions to the success, or failure, of the validation relates a disagreement as to what constitute an error condition. According to expert API designers: "*if a member [method] cannot successfully do what it was designed to do — what the member name implies — it should be considered an execution failure, and an exception should be thrown*" [9, p. 218]. In other words, an execution failure is said to have occurred if the `validate(Source)` method cannot validate the XML file, and an exception should therefore be thrown. Our participants, on the other hand, seem to associate the throwing of an exception to something catastrophic:

"*if you're just trying to validate something why would it throw an exception; It doesn't make sense; I expected it [*`validate(Source)`*] to return an object*" — P14.

"*in my experience ... I find consensus that you throw exceptions only when you find an error. In a Validator you expect something to be valid or invalid. And if its invalid that should be a common occurrence just as much as it is valid. So throwing an exception for common occurrence is not a good idea*" — P11.

This disagreement between API designers and our participants reflects the debate as to when exceptions should be used. Some argue that exceptions are for "exceptional conditions"; others argue that "*exceptions should be used to report all errors*" [9, p. 212]. The software development community has yet to agree on what constitutes an exceptional condition. Our study indicates that this disagreement has the potential for influencing API comprehension.

Observation 4 (Web versus Documentation). *The use of the Web had no effect on the number of tasks successfully completed or the time taken to complete a task.*

In designing the study, we had ten of the participants use the Web and the API documentation as learning resources (the Web Group — WG), and the other ten using just the documentation (the Documentation Group — DG). We reasoned that partitioning the participants into two groups would help us identify the challenges programmers faced when looking for answer to the questions using both learning resources. We expected the participants in the Web Group to be significantly more successful since the Web provides several code examples for both tasks. However, we observed no significant advantage, either in terms of the number of tasks successfully completed or the average time taken to complete a task, between the participants of the Web Group over the participants of the Documentation Group. Six participants from the Documentation Group and seven participants from the Web Group successfully completed task T1, and six participants from the Documentation Group and five participants from the Web Group successfully completed task T2 (Table 3–7). We obtained a chi-squared statistic of 0 when we compared the number of tasks successfully completed between the two groups.

Table 3–7: The number of tasks successfully completed between the two groups.

	DG		**WG**	
	Successful	Unsuccessful	Successful	Unsuccessful
T1	6	4	7	3
T2	6	4	5	5

Table 3–8: The average task completion time (in minutes) of the participants for both tasks, and both groups.

	DG		**WG**	
	MEAN	STDEV	MEAN	STDEV
T1	29	±7	25	±9
T2	29	±8	26	±8

Looking at the task completion times (Table 3–8), the participants of the Documentation Group spent an average of 29 (±7) minutes on task T1 while participants from the Web Group spent an average of 25 (±9) minutes. We observed similar results for task T2: participants of the Documentation Group spent an average of 29 (±8) minutes while participants from the Web Group spent an average of 26 (±8) minutes. We used the Rank test to compare the task completion time between the two groups and obtained a p-value of 0.45 for task T1 and a p-value of 0.26 for task T2. But why were the participants who used the Web not significantly better than those who used the API documentation?

We observed that some participants often underestimate the time required to find code examples on the Web, extract the relevant code snippets, and to customize the snippets into the context of a task. Some participants spent a significant amount of time extracting and customizing relevant snippets. For example, participant P13 found a code example for task T2 at the 16 minutes mark, but was unable to complete the task in the remaining 19 minutes because of difficulties in extracting and customizing relevant code snippets. When asked about this in the post-study interview, participant P13 commented that *"the example had a different context from our task, so I had to translate their ideas to ours and that takes some time"*. Other

participants started with the Web but soon realized the difficulty of finding relevant code examples with no knowledge of the types in the API. For instance, P18 started with the Web but soon abandoned the Web for the API documentation after two unsuccessful searches, commenting *"having some knowledge of the classes in the API may actually be able to help me understand the information provided by the tutorials"*. In general, we observed that both learning resources provide complementary support to programmers learning to use APIs. Also, the absence of a significant difference between the two groups suggests that the time required by novice API users to find, extract, and customize code snippets from code examples may be comparable to the time needed to learn how to use APIs from the API documentation for basic tasks such as the ones in our study.

3.4 Implications

3.4.1 API Design and Documentation

Proponents of the debate on how to communicate method-level failures typically endorse either the use an exception, or the use of a return-status-object, but seldom both: "*exceptions should be used to report all errors for all code constructs*" — [9, p. 212]. Our results explain and document why the use of an exception to communicate the outcome of an operation may be problematic from an API-comprehension perspective. In such situations, it seem reasonable for API designers to consider providing both a return-status-object (to provide status information in the case of a successful operation) and an exception (to communicate method execution failure). Steven Clarke of the user experience group at Microsoft Research, and a pioneer of

the work on API usability, echoed this view in a book on Framework Design Guidelines: "*although return codes should not be used to indicate failures, you can still consider returning status information in the case of a successful operation*" [9, p. 213].

In general, our study underscores the need to investigate the impact of API design choices on API comprehension and usability before adopting a given design choice. APIs are provided to improve programmers' productivity, but poorly designed APIs may produce a counter effect. In our work with APIs, we have observed situations where programmers had to re-invent the wheel because APIs designed for their task were difficult to understand [12]. API usability studies provide a venue for identifying and fixing usability and comprehension problems before an API is made public.

3.4.2 Tool Design

The primary motivation for our study was to understand the nature of API learning and how best to support programmers learning to use a new API. We have identified 20 different types of questions the programmers asked about the use of APIs and also five questions that proved the most difficult for the programmers to answer. We believe these questions could help evaluate existing tool support, and identify areas where support is lacking. As an example, we present three areas of difficulty where support is currently limited.

Discovering relevant API elements not accessible from the type a programmer is working with. Jadeite [52] uses a concept known as a "placeholder" to allow a developer to annotate the documentation of an API type with other API

types or methods not accessible, but relevant to its use. Given a particular function, Altair [30] and FRIAR [43] use heuristics and structural relationships to find other related functions. Jadeite is the only tool, to our knowledge, aimed at helping programmers discover types or methods not accessible from a main-type. We consider Jadeite a precursor to an ideal tool for making relevant API elements not accessible from a type discoverable. Our ideal tool would automatically generate and recommend placeholders and would be integrated with the IDE, preferable with the content assist feature of the IDE.

Discovering the types of an API relevant to implementing a task. The names of API types and methods provide a common vocabulary between API users and API designers; consequently, the use of types and methods for query formulation proved to be an effective strategy (in the context of our study) for alleviating the vocabulary mismatch problem when searching for code examples relevant to implementing a task. Surprisingly, support is lacking for helping programmers discover the types of an API relevant to a task. In fact, most code recommendation tools are based on the premise that programmers already know the types of an API relevant to their tasks [21, 53, 54], but this is often not the case. Jadeite once more is the only tool aimed at helping programmers discover types relevant to a task. It leverages usage statistics from code examples on the Web to display commonly used types of an API more prominently. Jadeite has two drawbacks: it is unusable in the absence of a corpus of code examples, and its use of a popularity-based metric implies less commonly use parts of an API would not be adequately supported. We believe the API learning process would be better served by tools that use a combination of

heuristics measures, not a corpus of code examples, to recommend types relevant to implementing a task.

Unmasking the relationships between API types. Some of the difficulties we observed occurred when the dependencies between related API elements were not obvious. For instance, although the *Validator* class and *Schema* are related (a *Validator* object is created from a *Schema* object), this relationship cannot be inferred from the *Validator* class. Participant P4 referred to this as the absence of a *"cross-reference in the API documentation that says get a Validator instance from a Schema"* when commenting about the difficulty experienced relating these types. Research tools which make such hidden relationships between API elements more explicit could improve the API learning process, and facilitate API usability.

3.4.3 Threats to Validity

The results of our study are based on a systematic observation of programmers working with real-world APIs in a laboratory environment. Given this setting, there are factors which limit the generalizability of our observations.

The types of questions we observed in the study, the process of answering the questions, and the challenges the participants experienced are related to a certain extend to the tasks and the experience of the participants. Some of the questions and the difficulties we observed in the study have been observed in previous API usability studies in different settings [15, 28, 49, 51]; However, given that our study was exploratory in nature and intended to probe *why* developers experience difficulties, and also given the lab setting and pre-defined tasks, the catalog of questions cannot be considered complete, but just a starting point.

The difficulty the participants experienced in associating the throwing of an exception to the success, or failure, of the validation could have been a result of their limited programming experience. This threat was mitigated by our use of the think-aloud protocol which showed that our participants had no apparent confusion with the validation domain (they could implement a solution to validate the XML file). Rather, the difficulty they encountered was well isolated to the use of exceptions to communicate method-level failures: the implication that an operation is considered successful if the method does not throw an exception was not apparent to our participants. Even the more experienced participants (greater than three years of programming experience) encountered difficulty completing the XML task: we compared the success rate to the years of programming experience of the participants for both tasks but observed no significant difference in the performance between the participants with less than three years experience, and those with greater than three years experience. Furthermore, Steven Clarke is quoted as reporting similar observations amongst professional programmers in a book on Framework Design Guidelines: "*In one API usability study we performed, developers had to call an* `Insert` *method to insert ... records into a database. If the method did not throw an exception, the implication was that the records had been inserted successfully. However, this was't clear to the participants in the study. They expected the method to return the number of records that were successfully inserted*" [9, p. 212]. The results of our study closely corroborate Clarke's observation amongst professional programmers; the extent and reasons for the difficulty for the population of professional programmers would have to be determined by another study.

The size of our tasks, the number of tasks, and the number of participants also limits the generalizability of our observations. Although our tasks represented real usages of real-world APIs, they were limited in size to permit our participants to complete a task within the 35 minutes time frame. With only two tasks and 20 participants, the questions and the challenges observed in our study could be limited. However, given the observation that "programmers often approach larger programming tasks by focusing on smaller subtasks" [51], we believe that the different types of questions and the challenges we observed, possibly limited, would generalize to other API learning tasks.

Lastly, our study involved only Java APIs and the Java API documentation. Some of our observations may be different for APIs and documentation in other languages. Also, since our study focused on programmers learning how to use unfamiliar Java APIs, our observations may not be applicable to programmers working with familiar Java APIs. Further studies on API usability are required to verify the generalizability of our observations to these contexts.

3.5 Summary

To understand what a programmer needs to know when using a new API and how best to support programmers, we conducted a study in which 20 programmers worked on programming tasks using different real-world APIs. The study generated over 20 hours of screen captured video and verbalization spanning 40 different programming sessions. Our analysis of the data involved generating generic versions of the questions asked by the participants about the use of the APIs, and identifying those questions the proved difficult to answer using the actions of the participants that

reflected a lack of progress when looking for information. Based on the results of our analysis, we identified 20 different types of questions the programmers asked about the use of the API grouped in five different categories: discovering the functionality provided by an API relevant to a task, understanding the relationships between types, discovering relevant dependent types, selecting relevant API elements, and understanding the behavior of API methods. We also identified questions that proved the most difficult for the programmers to answer, along with the evidence supporting our observations. We believe the questions we have identified and the difficulties we observed can be used for evaluating tools aimed at improving API learning, and in identifying support that is missing from existing programming tools. As an example, we identified two major areas where support is limited from existing tools: first, IDE based tools that would assist a programmer to discover relevant API elements not accessible from an type the programmer is working with; and second, tools that would assist a programmer to easily identify types that would serve as a good starting point for searching for examples, or exploring the API for a given programming task.

CHAPTER 4
API Explorer: Facilitating Discoverability in APIs Through Structural-Based Recommendations

Empirical evidence indicates that when working on a programming task, most developers look for a *main-type* central to the scenario to be implemented and explore an API by examining the methods and types referenced in the method signatures of the main-type [51]. As a result, a developer may be at a significant disadvantage when an API method essential to a task is located on a *helper-type* not directly accessible from the main-type, or when other essential types are not referenced in the signature of the methods on a main-type. For instance, Stylos et al. observed that placing a "send" method on a helper-type such as `EmailTransport.send(EmailMessage)`, instead of having it on the main-type such as `EmailMessage.send()`, significantly hinders the process of learning how to use APIs; they observed that developers were two to eleven times faster at combining multiple objects when relevant API methods and types were accessible from the main-type [51]. A different study which looked at the usability tradeoff between the use of the Factory pattern or a constructor for object construction also reported that developers required significantly more time using a factory than a constructor because factory classes and methods are not easily discoverable from the main-type [15].

To help facilitate the discovery of API elements relevant to a task, we developed and released a novel tool called *API Explorer*.[1] Our approach is based on the intuition that the structural relationships between API elements, such as method-parameter relationships, return-type relationships and subtype relationships, can be leveraged to make discoverable the methods and types that are not directly accessible from a main-type. For instance, we can use the fact that `EmailTransport.send(EmailMessage)` takes `EmailMessage` as a parameter to recommend the "send" method of the `EmailTransport` class when a developer looks for a "send" method, or something similar, on the `EmailMessage` class. Similar recommendations can be made for object construction from factory methods, public methods, or subtypes. We implemented API Explorer as a novel extension of the content assist feature of the Eclipse IDE.

API Explorer relies on a specialized dependency graph for APIs, called an *API Exploration Graph*, and several algorithms that use the information contained in the graph to generate recommendations based on the structural context. We present the API Exploration Graph, the algorithms of API Explorer, and the evaluation of API Explorer. Portions of this chapter previously appeared in an ECOOP publication [13].

4.1 API Exploration Graph

We use an API Exploration Graph (XGraph) to model the structural relationships between API elements. In an XGraph, API elements are represented as nodes; an

[1] www.cs.mcgill.ca/ swevo/explorer

edge exists between two nodes if the elements represented by the nodes share one of several structural relationships.

Nodes: an XGraph uses two kinds of nodes to represent API elements: a node to represent API types such as classes or interfaces, and a node to represent API methods. We model a public constructor as a method that returns an object of the created type. Our model does not contain fields (e.g., constants such as `System.out`) because our approach is based on the assumption that object construction is either through a constructor, a static method, or an instance method.

Edges: an XGraph uses four kinds of edges to capture the relationships between API elements:

- *created-from* edge: this edge exists between an API type, T, and an API method, M, if the method M returns an object of type T, or has a declared return type of T. The created-from edge captures object construction through constructors, static methods, or instance methods.
- *is-parameter-of* edge: this edge exists between an API type, T, and an API method, M, if the type T is a parameter of the method M.
- *is-subtype-of* edge: this edge is used to represent subtype relationships between API types. It exists between the type T_k and the type T_m, if T_k is a subtype of T_m.
- *requires* edge: is used to distinguish instance methods from class methods. A requires edge exist between a method, M, and an API type, T, if an instance of T must exist on which the method M must be invoked.

The XGraph is simple, but by combining the information encoded in multiple edges, we are able to derive useful non-trivial facts about the relationships between API elements. For instance, consider the example API in Listing 4.1: from knowing that `MimeMessage` is-subtype-of `Message`, and that `Message` is-parameter-of the method `Transport.sendEmail`, we can infer at least three facts. First, objects of type `Message` could be created from `MimeMessage`; second, `MimeMessage` can be used whenever `Message` is expected; and third, `MimeMessage` can also be sent using the "sendEmail" method of `Transport`.

```
abstract Message {
   public void setText(String)
}

MimeMessage extends Message {
   public MimeMessage(Session)
}

Transport {
   public static void sendEmail(Message)
}

Session {
   public static Session getInstance(Properties)
}
```

Listing 4.1: A simplified version of the JavaMail API

Listing 4.1 shows a simplified version of the JavaMail API, and Figure 4–1 shows the corresponding XGraph. JavaMail uses the types *String* and *Properties* from the Java Runtime Environment (JRE). API Explorer maintains an XGraph of the JRE,

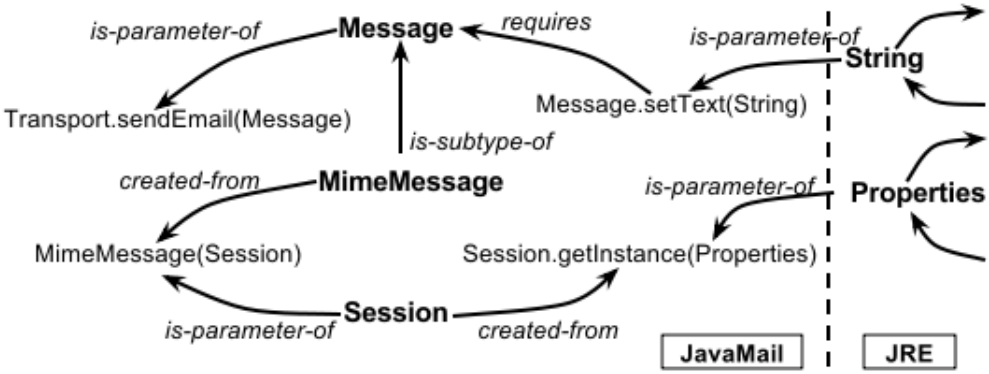


Figure 4–1: The XGraph of the simplified JavaMail API in Listing 4.1. The nodes in boldface represent API types; the other nodes represent API methods, including constructors, and the edges represent relationships between the nodes.

and automatically links it to the XGraph of APIs referencing types of the JRE, as in Figure 4–1. We generate XGraphs from the binaries of APIs using the Javaassist[2] byte code analysis library, and it takes less than one minute to create an XGraph even for large APIs such as the JRE, which includes 3000 types and 9300 methods. API Explorer uses the information in the XGraph to generate recommendations and code showing how the recommended API elements should be combined. We present the recommendation algorithms in Section 4.2, and use the sample API in Listing 4.1 and its XGraph to present examples of the algorithms.

[2] www.javassist.org

4.2 Recommendation Algorithms

4.2.1 Object Construction Algorithm

The object construction algorithm (Algorithm 1) facilitates the discovery of factory methods, static methods, or subtypes that may be needed to construct an object of a given API type, say T (input to the algorithm). The algorithm begins by looking at the *created-from* edges of the node representing T in the XGraph (lines 4 and 8). The *xgraph.getNodes(T,edgeType)* function (line 8) returns the API element (a factory method or constructor) each *created-from* edge points to, for every such edge found on T. The algorithm is designed to first search for a way of creating an object of type T that does not involve its subtypes. We did this to minimize the number of recommendations presented to a user. If no recommendation for creating an object of type T without its subtypes is found, the algorithm proceeds to look at the *created-from* edges of the subtypes of T (lines 9 to 12). The algorithm uses the *is-subtype-of* edge to locate the subtypes of T (line 10), then recursively calls the *getObjectConstructionProposals* function for each subtype (lines 11 to 12). The algorithm continues down the hierarchy until information on how to create an object of type of T is found, or all the subtypes are exhausted. Upon completion, the algorithm presents a list of recommendations showing different ways of creating an object of type T. We present the code generation algorithm in Section 4.2.4 that recursively looks for the parameters and dependencies of a selected recommendation, and generates code showing how to combine them.

Algorithm 1: Object Construction

```
   Input: T, xgraph /* the type T for which object construction
      assistance is requested and the XGraph                    */
   Output: recommendations /* a list of recommended API elements
      that could be used to create an object of type T          */
 1 Var edgeType := created-from /* a valid edge in the XGraph    */
 2 recommendations := ∅
 3 begin
 4     recommendations := getObjectConstructionProposals(T, xgraph,
       edgeType)
 5 function: getObjectConstructionProposals(T, xgraph, edgeType)
 6 begin
 7     Var proposals := ∅
 8     proposals := xgraph.getNodes(T, edgeType) /* get the nodes in the
          XGraph pointed to by the created-from edges of node T  */
 9     if proposals == ∅ then
10         Var subtypes = xgraph.getNodes(T, is-subtype-of)
11         foreach type ∈ subTypes do
12             proposals := proposals ∪
               getObjectConstructionProposals(type,xgraph,edgeType)
13     return proposals
```

Example. Consider as an example a developer looking for assistance on how to create an object of type `Message`. The algorithm begins by looking at the *created-from* edges on the `Message` node. The `Message` node has no *created-from* edge; the algorithm then proceeds by looking for subtypes of `Message` from which an object could be created. The `Message` node, in this case, has a single *is-subtype-of* edge pointing to `MimeMessage`. Next, the algorithm looks at the *created-from* edges on the `MimeMessage` node, and finds `MimeMessage(Session)`, a constructor for creating a `MimeMessage` object. The algorithm, being aware that `MimeMessage` is a subtype of

`Message`, recommends `MimeMessage(Session)` as a way of creating a `Message` object. For simplicity, our example API has types with only a single object construction option. However, in practice, an API may provide multiple ways of creating objects of a given type. For instance, the JavaMail API provides four options for creating a `Session` object. In such situations, API Explorer presents all the options to a developer to decide the most appropriate construction pattern in a given usage context. As will be seen in Section 4.3, the participants of our case study evaluation demonstrated little difficulty selecting a relevant recommendation when presented with multiple options.

4.2.2 Method Recommendation Algorithm

The method recommendation algorithm (Algorithm 2) is based on the observation that if a method a developer needs is not available on the type, T, the developer is working with, then one of the methods which take T, or an ancestor (a class, or an interface) of T, as a parameter may provide the needed functionality. The algorithm uses the *is-parameter-of* and the *is-subtype-of* edges of the XGraph to recommend relevant methods on other objects.

The algorithm begins by looking at the API methods that take T as a parameter using the *is-parameter-of* edges at the node T in the XGraph (lines 3, 12 to 15). The algorithm verifies if the name of a method that takes T as a parameter starts with the prefix entered by the user, and if so, adds that method to the list of proposals. If the list of proposals is empty once all the methods that take T as parameter have been examined, the algorithm uses synonym analysis to search for, and recommend, API methods with a name similar to what the developer is looking for. Our intuition is

Algorithm 2: Method Recommendation

```
   Input: T, prefix, xgraph /* the type for which a recommendation is
      being requested, the prefix provided by the user, and the
      XGraph                                                        */
   Output: recommendations /* list of recommended API methods     */
 1 recommendations := ∅
 2 begin
 3     recommendations := getMethodProposals(T, prefix, xgraph)
 4     if recommendations == ∅ then
 5         Var ancestors = T.getAncestors()
 6         foreach type ∈ ancestors do
 7             recommendations := recommendations ∪
 8             getMethodProposals(type, prefix, xgraph)
 9 function: getMethodProposals(T, prefix, xgraph)
10 begin
11     Var proposals := ∅
12     Var list := xgraph.getNodes(T, is-parameter-of)/* get the method
          nodes pointed to by the is-parameter-of edges of node T
          */
13     foreach method ∈ list do
14         if method.nameStartsWith(prefix) then
15             proposals := proposals ∪ method
       /* synonym analysis                                          */
16     if proposals == ∅ then
17         Set prefixSet = getSynonyms(prefix)
18         foreach method ∈ list do
19             Set methodSet = getSynonyms(method.getName())
20             if methodSet ∩ prefixSet ≠ ∅ then
21                 proposals := proposals ∪ {method}
22     return proposals
```

that, a developer looking for an API method to send an email object, if not searching

for a method prefixed "send", may be looking for something similar to "send", such as "transmit" or "deliver", instead of something totally unrelated.

The synonym analysis part of the algorithm (lines 16 to 21) re-examines all the API methods that take T as parameter. The synonym analysis begins by generating the synonym set for the prefix entered by the user (line 17); then for each method of the *is-parameter-of* edges of T, the algorithm extracts its prefix and generates its synonym set. The methods whose synonym set have one or more elements in common with the synonym set of the prefix entered by the user are added to the list of proposals (lines 20 to 21). API Explorer uses the WordNet[3] dictionary to generate the synonym sets. We also augmented WordNet with common words such as "insert", "put", and "append" often used interchangeably in APIs, and by developers, but which are not necessarily synonyms in the English vocabulary.

The method recommendation algorithm may not find a relevant method amongst the methods that take T as a parameter. In this case, the algorithm searches for API methods that take an ancestor of T as a parameter (lines 4 to 8). The algorithm uses the *is-subtype-of* edges at T to located its ancestors (line 5), and for each ancestor, calls the *getMethodProposal* function for recommendations (line 6 to 8). Upon completion, the algorithm presents a list of API methods with a prefix matching, or similar, to that entered by the developer, and with object of type T as a parameter.

Example. Consider as an example a developer looking for a "send" method on a `MimeMessage` object. The algorithm begins by looking at the *is-parameter-of* edges of

[3] http://wordnet.princeton.edu/

the `MimeMessage` node, searching for methods prefixed "send" that take `MimeMessage` as a parameter. The `MimeMessage` node, however, has no *is-parameter-of* edge; the algorithm then looks for a supertype of `MimeMessage` by moving up its *is-subtype-of* edge, and finds the type `Message`. Next, the algorithm looks at the *is-parameter-of* edges of the `Message` node and, this time, finds an edge pointing to the static method `sendEmail(Message)` on the `Transport` class. The algorithm does not terminate once the first "send" method is found; it searches for all methods prefixed "send" that can accept a `MimeMessage` object by looking at other *is-parameter-of* edges on the current node and on other nodes up the hierarchy. In this example, the algorithm would recommend the only method it found, `Transport.sendEmail(Message)`, to the developer with the knowledge that `MimeMessage` is a subtype of `Message`.

4.2.3 Relationship Exploration Algorithm

In our work with APIs, we have observed cases in which a developer has identified two or more types relevant to their programming task, but remains uncertain about how these types are related [12]. A developer wanting to verify the relationship between the types T_1 and T_2 must either combine the results of multiple search tools, or go through the documentation of at least one of the types before determining whether or not they are related. Using the XGraph, our relationship exploration algorithm (Algorithm 3) can help a developer efficiently explore the relationships between API types.

The algorithm takes as input API types and the XGraph, and outputs the relationships between the types, if any. Given a single API type, the algorithm can locate other API types related to it (lines 3 to 4). The *xgraph.getRelatedTypes(typeArray[0])*

Algorithm 3: Relationship Exploration

```
   Input: typeArray[], xgraph /* an array of API types and the XGraph
      */
   Output: relations /* a list of related API element              */
1  begin
2      relations := ∅
3      if typeArray.length == 1 then
4          relations := relations ∪ xgraph.getRelatedTypes(typeArray[0])
5      else if typeArray.length == 2 then
6          Var listOfMethods0 = xgraph.getMethods(typeArray[0])
7          relations := relations ∪
           getRelationships(typeArray[1], listOfMethods0)
8          Var listOfMethods1 = xgraph.getMethods(typeArray[1])
9          relations := relations ∪
           getRelationships(typeArray[0], listOfMethods1)
10 function: getRelationships(T, listOfMethods)
11 begin
12     Var relationships := ∅
13     foreach method ∈ listOfMethods do
14         if method.getReturnType() <: T OR T ∈ method.getParameters()
           then
15             relationships := relationships ∪ method
16     return relationships
```

function (line 4) returns a list of types related to *typeArray[0]* through the *is-parameter-of*, *is-subtype-of*, or the *created-from* edge of the XGraph. Given two API types $typeArray[0]$ and $typeArray[1]$, the algorithm looks for method-parameter or return type relationships between the types (lines 5 to 9). For $typeArray[0]$, the algorithm first retrieves the list of all the API methods defined on $typeArray[0]$ (line 6). Then, for each method on $typeArray[0]$, the algorithm checks whether the method takes an object of type $typeArray[1]$, or its ancestor, as a parameter, or has

typeArray[1], or its subtype (represented as <:) as a return type. If so, the method is added to the list of related elements (lines 7, 10 to 16). This same procedure is repeated for the type *typeArray[1]* (lines 8 to 9), and the relationships between the types are presented to the user.

Example. Consider as an example a developer wanting to explore the relationships of `MimeMessage`. The developer will begin by issuing a query to the relationship exploration algorithm to identify the types related to `MimeMessage`. The algorithm uses the edges of the XGraph to locate types related to `MimeMessage`: in this case, the algorithm would reveal that `MimeMessage` is related to both `Message` and `Transport` using the *is-subtype-of* and the *is-parameter-of* edges of the XGraph. The developer may then explore the relationship between `MimeMessage` and one of the related types (e.g., `Transport`) by selecting `Transport`. The algorithm then looks at the edges that connect `MimeMessage` to `Transport` in the XGraph to provide an explanation of how they are related. Upon completion, the algorithm would reveal that `MimeMessage` and `Transport` are related through the `Transport.sendEmail(Message)` method. Traditional "reference search" features, such as that provided in the Eclipse IDE, are unable to determine that `MimeMessage` is related to `Transport.sendEmail(Message)` because they are not inheritance-aware. Our relationship exploration algorithm therefore complements existing "reference search" tools.

4.2.4 Code Generation Algorithm

The code generation algorithm is intended to show a developer how to correctly coordinate the main-type and helper-types. This algorithm is triggered only when a recommendation is selected. If the selected recommendation is a constructor, the

algorithm first determines whether or not it has parameters. If the constructor has no parameters, the algorithm generates code showing how to use the default constructor. For constructors with parameters, the code generation algorithm first generates an identifier for the non-primitive parameters, and for each non-primitive parameter T, calls the object construction algorithm to determine how to create an object of type T. The algorithm uses the method-parameter relationship to determine how the statements should be ordered and how they relate to each other.

If the selected recommendation is an API method, the algorithm uses the *requires* edge to determine whether or not the method is static. For a non-static method, the algorithm begins by calling the object construction algorithm to create an object of the type on which the method is defined, before invoking it. Then, for each non-primitive parameter T of the selected method, the code generation algorithm calls the object construction algorithm to determine how to create an object of type T. For a static API method (i.e., method without a *requires* edge), the algorithm only has to create objects for each non-primitive parameter. The algorithm does not create objects for non-primitive parameter types already available from the context in which API Explorer was invoked — it uses variables in the context that match a given parameter type. For instance, if a developer selects `Transport.send(Message)` from the recommendations on how to send a `Message` object `m1`, the code generation algorithm will not create a new `Message` object, but will pick `m1` from the context, and output `Transport.send(m1)`.

4.2.5 Design rationale

We designed our approach with the awareness that a main-type may have several helper-types, with each helper-type relevant to a different programming scenario. For instance, the type `Message` of the JavaMail API has the method `Transport.send(Message)` as a helper-type for sending email objects, and the method `SearchTerm.match(Message)` as a helper-type for locating email objects that satisfy a given search criterion. Similarly, an API type may have several object construction patterns, with each pattern relevant to a different usage scenario. Our approach does not attempt to guess which helper-type is relevant for a given programming scenario; it recommends all valid helper-types in a given structural context (recommendations are ranked alphabetically), and allows the developer to select the most appropriate helper-type for a given programming scenario. We designed our approach this way for two reasons: first, a heuristic that attempts to narrow down the list of recommended help-types by removing those considered irrelevant in a given scenario may inadvertently hide a helper-type most appropriate for a given scenario. Such a mistake will further undermine discoverability, the very problem our approach is intended to solve. To avoid hampering discoverability, we opted for a design that relies on the developer to select the helper-type most appropriate for a given task. Second, our experience working with APIs indicates that developers have little problem selecting relevant API elements from a list of recommendations. We therefore expect that developers will have little difficulty selecting the most appropriate helper-type from a list of possible helper-types for a given programming task. We discuss the extent to which our expectations were valid in the evaluation.

4.3 Evaluation of API Explorer

Our evaluation had two goals: first, to show the extent to which our assumptions about the API exploration behavior of developers, and their ability to select relevant recommendations, are reflected in realistic API usage contexts; and second, to understand the specific circumstances in which API Explorer may be more or less helpful in discovering helper-types not accessible from a given main-type. Given that we were interested in studying how the approach supports people (as opposed to the performance of algorithms taken in isolation), we favored a qualitative evaluation methodology. We reasoned that a qualitative evaluation of our approach in the context of several programming tasks will enable us to reliably evaluate the assumptions and observations on which our approach is based, and to understand the contexts in which the approach would not be effective.

4.3.1 Case Study Design

We used a case study methodology to evaluate our approach. Yin introduces the case study methodology as "an empirical inquiry that investigates a contemporary phenomenon within its real-life context" [56, p. 13], and Easterbrook et al. explains that the case study methodology is particularly suited for evaluating software tools "where the context is expected to play a role in the phenomena" [14, p. 297], as in the case of API Explorer. For example, Holmes and Murphy used a case study evaluation to provide an in-depth understanding of how and why their Strathcona tool was helpful [21].

In the case study methodology, the cases (programming tasks, in our setting) are selected to represent the phenomenon being studied, and each case is considered as a

replication, rather than a member of a sample [14, 56]. Furthermore, our case study methodology emphasizes generalization to similar contexts (i.e, if the selected cases supports our hypotheses, then it is expected that similar cases will be supported by our approach), not statistical generalization [56, p. 31]. The goal of our case study was to answer the following questions:

Q.1 To what degree are our assumptions about the API exploration behavior of developers reflected in practice?

Q.2 In which ways can structural relationships help when trying to increase the discoverability of API elements necessary to solve a task?

Q.3 Would a developer be able to select a helper-type relevant to their task when presented with a list of possible helper-types?

Q.4 In which situations would API Explorer not be helpful, and why?

4.3.2 Programming tasks

Our approach is intended to assist developers locate helper-types not accessible from a type they may be working with. We therefore selected programming tasks that typified the discoverability hurdles our approach is intended to solve. Three of the tasks (the Email, XML, and Chart tasks) selected for the study have been the subject of previous studies that investigated the discoverability problem [12, 51].

Email task: we asked the participants to use the JavaMail API to implement a solution that would compose and deliver an email message. To complete the task, a participant needed to create and configure at least four API types, all created from factory methods or subtypes, and needed to discover a key relationship between

`Message` and `Transport` to send the email message. We used version 1.4.2 of the JavaMail API, which has five packages and 91 non-exception classes.

XML task: we asked the participants to use the Java API for XML Processing (JAXP)[4] to verify whether the structure of an XML file conforms to a given XML schema file. This task required the combination of at least four API types (`Validator`, `Schema`, `SchemaFacotry`, and `Source`); we selected this task to evaluate the object construction feature because of the unique challenges it presents — all the required types are abstract with no subtypes; the types must be created from factory or public methods (e.g., `Validator` can only be created from `Schema.newValidator()`). We used version 1.4 of the JAXP API, which has 23 packages and 207 non-exception classes.

Chart task: we asked the participants to use the JFreeChart[5] API to create a pie chart and to save the chart to a file in a graphic format. To complete this task, a participant needed to coordinate at least five API types, and had to discover the relationship between `JFreeChart`, the type for representing charts, and `ChartUtilities`, the type needed to save the chart. We used version 1.0.13 of the JFreeChart API, which has 37 packages and 426 non-exception classes.

[4] jaxp.dev.java.net

[5] jfree.org/jfreechart

PDF Task: we asked the participants to use the PDFBox[6] API to implement a solution to merge two PDF files. This task required the combination of just two API types: `PDFDocument` and `MergerUtility`. However, the relationship between `PDFDocument` and `MergerUtility` (related through an "append" method on `MergerUtility`) cannot be determined through synonym analysis since "merge" is not a synonym of "append". We were interested in investigating whether the participants would be able to use other features of API Explorer to discover this key relationship. We used version 1.2.1 of the PDFBox API, which has 31 packages and 307 non-exception classes.

4.3.3 Study participants

We recruited eight participants (henceforth referred to as P1, ..., and P8) through our departmental mailing list. Our participants reported between 1.5 and 3 years of experience programming with Java, with a median Java programming experience of 2.5 years. All the participants had at least six months experience working with the Eclipse IDE. None of the participants, with the exception of P1, had used any of the four APIs in the study; P1 had used the JFreeChart API in the past, but in a task different from ours and could not remember the types provided by the API.

4.3.4 Study procedure

We provided each participant with a tutorial of the features of API Explorer before the study began, and asked the participants to use API Explorer whenever they believed a feature it provides could be helpful. We also provided each participant

[6] pdfbox.apache.org

with a description of the tasks and the documentation of the APIs. The four tasks were completed in the same order by the participants, and the participants were allowed a maximum of forty minutes per task. We asked the participants to think-aloud whenever API Explorer was used to allow us to understand why the assistance of API Explorer was needed, why the participant selected a given recommendation, and whether or not the assistance provided by API Explorer was helpful. We also used screen capturing software to document all the actions of the participants. To avoid influencing the behavior of the participants, we did not inform them of which types of an API were relevant to each task, or which type of an API to start from; the decision of how to approach each task was left to each participant.

4.3.5 Results

The study produced a total of over 16 hours of screen captured videos and verbalizations of eight participants using API Explorer in 32 different programming sessions. Our analysis of the data focused on the questions the participants asked about the use of the APIs, how the participants used API Explorer to answer questions, and the extent to which API Explorer was helpful. We begin by presenting task-level observations that show the degree to which the API exploration behavior of the participants supports the hypothesis on which our approach is based (Q.1). For each task, and for each participant, we provide observations on how the participant approached the task, and the degree to which API Explorer was effective in helping the participant discover helper-types not accessible from a main-type. Then, we present episode-level observations: an analysis of all the instances in which API Explorer was used by each participant, the degree to which a participant was able to select

relevant recommendations, and the discoverability contexts in which API Explorer proved helpful (Q.2 and Q.3). Lastly, we look at situations in which API Explorer was not helpful (Q.4).

A. Tasks-Level Observations.

The first question (Q.1) was intended to investigate the degree to which the behavior of our participants supports our main hypothesis (*when working on a task, a developer typically starts from a main-type central to the programming scenario before looking for helper-types*) and to evaluate the degree to which API Explorer would be helpful in discovering relevant helper-types. We present a detailed outline of the observations from the Email task, and summarize the observations from the other tasks in Table 4–1.

All eight participants started the Email tasks by looking for a type representing an email message. They all found the abstract class `Message` from the documentation and proceeded to query API Explorer for assistance on how to create an object of type `Message`. API Explorer provided two recommendations: `MimeMessage(Session)` and `MimeMessage(Session,InputStream)`, both constructors from the subtype `MimeMessage`; seven of the participants selected `MimeMessage(Session)`, P5 selected `MimeMessage(Session,InputStream)` thinking `InputStream` is needed to set the email content. P5 later reverted to `MimeMessage(Session)`. After selecting `MimeMessage(Session)`, API Explorer provided four recommendations on how to create a `Session` object from factory methods, and all the eight participants selected `Session.getInstance(Properties)`, to complete the process of creating a `Message` object.

Table 4–1: A summary of how the participants approached each task, their effectiveness in using API Explorer (APIX) to locate helper-types not accessible from a main-type, and the API Explorer feature (SA — synonym analysis, EC — enhanced code completion, RE — relationship exploration, OC — object construction) used to make the discovery. The check mark (✔) represents Yes, and ✖ represents No.

	P1	P2	P3	P4	P5	P6	P7	P8
Email Task								
Started from *Message*, then looked for *Transport*	✔	✔	✔	✔	✔	✔	✔	✔
Found *Transport.send* from *Message* using APIX	✔	✔	✔	✔	✔	✔	✔	✔
Feature used	EC	SA	SA	SA	EC	SA	SA	EC
Chart Task								
Started from *JFreeChart*, then looked for *ChartUtil*	✔	✔	✔	✔	✔	✖	✔	✔
Found *ChartUtil.write* from *JFreeChart* using APIX	✔	✔	✔	✔	✖	✖	✔	✔
Feature used	EC	SA	SA	EC	—	—	EC	SA
PDF Task								
Started from *PDFDoc*, then looked for *MergerUtil*	✔	✔	✔	✔	✔	✔	✔	✔
Found *MergerUtil.append* from *PDFDoc* using APIX	✔	✔	✔	✔	✔	✔	✔	✖
Feature used	EC	RE	EC	EC	EC	EC	EC	—
XML Task								
Started from *Validator*, then looked at *Schema*	✖	✔	✔	✔	✖	✔	✔	✖
Found *Schema.newValidator()* from *Validator* using APIX	✔	✖	✔	✔	✔	✖	✔	✔
Feature used	OC	—	OC	OC	OC	—	OC	OC

The participants approached the next part of the task, sending the email message, differently. P1 started with the documentation in search for assistance on how

to send the message but did not find `Transport`. He then browsed through the methods of `Message` using the *enhanced* code completion (EC) feature of Eclipse when he noticed `Transport.send(Message)` amongst the recommendations of API Explorer. P5 and P8 also used the EC to discover `Transport.send(Message)` directly from `Message`. Participants P2, P3, P4, P6, and P7 all used the synonym analysis (SA) feature of API Explorer to query for a recommendation for "Message.send", and received four recommendations from which they discovered three different "send" methods on the `Transport` class.

We summarize the observations from the other tasks in Table 4–1. For each task, we indicate whether the participant started from the main-type before looking for the helper-type, whether the participant was able to use API Explorer (APIX) to discover the helper-type directly from the main-type, and the API Explorer feature that was used to make the discovery. For the Chart task, seven of the eight participants started from the main-type `JFreeChart` before looking for the helper-type `ChartUtilities`. Only P6 started from `ChartUtilities` before looking for `JFreeChart`, and this occurred because P6 had difficulties finding the main-type and happened to stumble on `ChartUtilities`. Six of the eight participants successfully used APIX to discover `ChartUtilities` directly from `JFreeChart`. P5 did not attempt to use APIX to look for a helper-type; he came up with an improvised solution that created a `BufferedImage` from `JFreeChart`.

For the PDF task, all the eight participants started from the main-type `PDFDocument` before looking for the helper-type `MergerUtility`, and seven of the participants successfully used APIX to discover `MergerUtility` directly from `PDFDocument`. P8

used synonym analysis with "PDFDocument.merge" but got no recommendations. He made no attempt to use other features of APIX, such as the enhanced code completion, that could have helped him discover `MergerUtility`; he came up with an improvised solution for merging the documents.

Five of the eight participants in the XML task started with the main-type `Validator`; the other three started with the helper-type `Schema`. The domain that provided support for validation had only six classes, with `Schema` at the top of the list, and `Validator` at the end: that could have influenced the three participants that started with `Schema`. Six of the participants used the object construction feature to discover how to create a `Validator` object from `Schema.newValidator()`, the other two used the documentation.

The results were consistent across the eight participants and in most of the tasks: the participants typically began exploring an API from the main-type before looking for a relevant helper-type, and successfully used API Explorer to discover relevant helper-types directly from a main-type.

B. Episode-Level Observations.

To answer questions Q.2, Q.3, and Q.4, we analyzed all the segments of the screen captured videos, which we called *episodes*, corresponding to instances in which a participant used API Explorer to discover API elements relevant to a task. In our analysis, we focused on the degree to which a participant was able to select API elements relevant to a task from the recommendations of API Explorer, the discoverability contexts in which the assistance of API Explorer was requested, and whether

or not the assistance provided was helpful. We consider the assistance provided by API Explorer *helpful* if its recommendations contains an API element relevant to a given request, and if the participant was able to recognize and select the relevant element. The results of the analysis are summarized in Table 4–2.

The third column (# of usage episodes) of Table 4–2 shows the number of episodes where API Explorer was used, per participant and per discoverability context. For instance, P1 used API Explorer 21 times: four times to discover relevant methods on other API types (row METH), 16 times to discover API elements necessary to construct an object of a given API type (row OBJ), and once to look for types related to a given API type that could be used to perform a given operation (e.g., types related to PDFDocument that could be used for merging; row ER). The participants requested the assistance of API Explorer a combined total of 161 times. The fourth column presents the average number of recommendations per episode for each of the different discoverability contexts. The average number of recommendations ranged from about 2 to 15 recommendations per episode.

The fifth column presents the number of episodes in which a participant was *unable* to select or recognize an API element relevant to a task from the recommendations made by API Explorer. We observed only two instances in which a participant was unable to select a relevant API element from the recommendations of API Explorer. In the first instance, P3 had requested the list of API types related to `PDFDocument` while looking for a type that could be used for merging PDF files. API Explorer provided a list with 12 API types, including `MergerUtility`,

Table 4–2: A summary of all instances in which API Explorer was used by each participant for the various contexts (object construction [OBJ], looking for relevant methods on other types [METH], and exploring the relationships between types [ER]).

		# of usage episodes	average # of recommendations	*unable* to select	API Explorer *not* helpful
P1	**OBJ**	16	6.3	0	0
	METH	4	5	0	0
	ER	1	0	0	1
P2	**OBJ**	12	5.5	0	0
	METH	4	4.2	0	2
	ER	4	6	0	1
P3	**OBJ**	11	7.8	0	0
	METH	3	8.2	0	0
	ER	8	3.5	1	1
P4	**OBJ**	16	6.3	0	0
	METH	3	9.1	0	0
	ER	5	5.9	0	0
P5	**OBJ**	14	7	0	0
	METH	2	15.2	0	0
	ER	3	0	0	0
P6	**OBJ**	12	8.3	0	0
	METH	4	5.1	0	1
	ER	5	3.6	0	0
P7	**OBJ**	11	7.1	1	1
	METH	3	8.6	0	0
	ER	1	5.5	0	0
P8	**OBJ**	14	6.2	0	0
	METH	2	8.4	0	0
	ER	3	1.7	0	0
	TOTAL	**161**		**2**	**7**

but P3 failed to notice it because it was not visible, and P3 did not scroll to examine the entire list. In the second instance, P7 had requested for assistance on how to create a `Schema` object, and received eight recommendations: P7 selected `DocumentBuilder.getSchema()` instead of `SchemaFactory.newSchema(File)`, but later reverted to `SchemaFactory.newSchema(File)` when she realized a schema file was provided for the task. API Explorer was not helpful in only seven of the 161 episodes in which it was used (last column): we address these situations below where we look at the limitations of our approach.

The participants experienced little difficulty selecting API elements relevant to a given programming scenario when presented with a list of possible helper-types. API Explorer also proved mostly helpful when looking for helper-types relevant to creating an object, relevant helper-methods on other objects, and when looking for types related to a given API type that could be used to perform a given operation.

C. Limitations of our approach.

The last column of Table 4–2 shows some of the situations in which our approach was not helpful: these involve the synonym analysis and relationship exploration features of API Explorer. Our approach is also not helpful when the relationships between API elements can only be determined at runtime.

The effectiveness of our synonym analysis algorithm depends on API methods respecting naming conventions such as method names beginning with action verbs, not acronyms, and on the ability of a developer to provide a prefix that matches, or is a synonym to the name of a relevant method on a helper-type. In two instances,

P2 and P8 had sought for assistance on how to merge PDF files using synonym analysis with "PDFDocument.merge" but received no recommendation. This was expected as "merge" is not a synonym of the "append" method on MergerUtility. We had designed the PDF task to see whether the participants would be able to use other features of API Explorer to discover `MergerUtility` from `PDFDocument`. In particular, to address the limitations of the synonym analysis feature, we *enhanced* the default Eclipse code completion feature with the ability to display not only the methods defined on type T, but also the API methods that take an object of type T as a parameter. For instance, a developer browsing through the methods of `Message` using this enhanced code completion feature will also come across the method `Transport.send(Message)`. Thus, a relevant helper-method not recommended by synonym analysis will be discovered when the developer looks through the methods of T. As shown in Table 4–1 (PDF task), six of the eight participants were able to use the enhanced code completion feature to discover `MergerUtility` directly from `PDFDocument`.

Our relationship exploration algorithm has two limitations: it cannot identify the API types that throw a given exception, and can only identify direct relationships between API types. P1 had looked for types related to `SendFailedException` that could be used to send an email message but was misinformed that there was no related type, although this exception is thrown by `Transport`. This occurred because the current version of our XGraph does not support types related through thrown exceptions. However, P1 subsequently discovered `Transport.send` with the

assistance of the method recommendation feature of API Explorer. P2 was misinformed that `Document` is not related to `Source`, although they are related through `DOMSource(Document)`, a constructor of a subtype of `Source`. This occurred because our relationship exploration algorithm does not consider indirect relationships between API elements. We plan on extending our XGraph and algorithms to show API types that throw a given exception and to support indirect relationships between API elements.

4.3.6 Summary of the evaluation

Overall, the results of the study were consistent across the participants and for most of the tasks: the participants began exploring the APIs from a main-type before looking for the helper-types, and were mostly successful at using API Explorer to locate helper-types not accessible from a main-type. The participants also experienced little trouble selecting relevant elements when presented with multiple recommendations. Our understanding of the domain enabled us to select tasks from real-world APIs with discoverability hurdles typical to those that have been identified in the literature [15, 51]. We therefore expect our observations to generalize to similar contexts, namely, when seeking to make API elements not directly accessible from a given API type more discoverable. The participants expressed four reasons why they considered the assistance provided by API Explorer helpful:

- *Saves time* (P2, P3, P4, P5, P7, P8): "It would have taken me a lot of time to go [to the documentation] and find which class will have a merge functionality. Using the tool, I could find `MergeUtility` directly from `PDFDocument`" – P2.

- *Increases awareness* (P1, P4, P6, P7, P8): "this is another thing I really like. A lot of times when you look at an API, you look at just the first constructor and use that. API Explorer shows me other better options that I wouldn't have looked for." – P1.
- *Serves as a reminder* (P1): "I couldn't remember the proper way of using it [the `JFreeChart` class] and was reminded by the tool" – P1.
- *Unmasks hidden relationships* (P1, P2, P4, P5, P7, P8): "If you want to save something, you would like to say `object.save()` but that option is usually not provided; usually, it is `something.save(object)` [that is provided]. It [API Explorer] is useful because it can make the association between the object you want to save and the method that you need to call" – P1.

API Explorer recursively shows a participant how to create and relate objects necessary to use a selected recommendation, even if the required objects comes from commonly used types. Two participants (P4 and P6) complained that this was not necessary for commonly used types such as the `String` class: "telling me how to construct a `String` might not necessarily be the most helpful thing because it is commonly used." – P6.

4.3.7 Threats to validity

As indicated in Section 4.3.1, our method of choice for evaluating API Explorer was the case study, which emphasizes exploration of the relation between a phenomenon and its context as opposed to generalization. In particular, the diversity of APIs and programming languages present factors which limits the generalizability of the results of our study. First, API Explorer will not be helpful for APIs without helper

types, or APIs without indirect object construction patterns such as the Factory pattern. The same is true for an API with a well-written API documentation that includes actual usage examples. History, however, suggests that we are far from these ideals: there are situations where it seems reasonable to provide a Factory, instead of a constructor, and to provide helper-types. It is for such situations that we envisage tools such as API Explorer to remain helpful in facilitating discoverablity in APIs. Second, although the tasks used in our evaluation were drawn from real-world APIs, it is likely that they did not uncover every discoverability hurdle that could occur in practice. In particular, very few indirect relationships, a feature not currently supported by API Explorer, were uncovered by the evaluation. As future work, we plan on extending API Explorer to support indirect relationships and to conduct further studies to evaluate this feature. Lastly, some APIs have the notion of an internal API, intended to be used by the designers only, and the public API, for general use. The current version of API Explorer does not take these differences in account when making recommendations; there is therefore the possibility that recommendations made by API Explorer may be from the internal API, a practice discouraged by API designers.

4.4 Summary

Learning how to use APIs is major part of a software developer's job — even experienced developers must learn newer parts of an existing API, or newer APIs when working on a new project. We proposed an approach to address one of the challenges a developer faces when learning a new API: discovering relevant helper-types not accessible from a main-type the developer is working with. Our approach leverages

the structural relationships in APIs to make relevant API elements not accessible on a given API type discoverable. We implemented our approach in a tool called API Explorer, and evaluated the approach through a multiple-case study in which eight participants replicated four programming tasks with several discoverability hurdles. The results of the study were consistent across the participants and the tasks: API Explorer effectively assisted the participants in locating helper-types not accessible from a main-type in different discoverability contexts. The participants also experienced little difficulty selecting relevant API elements from the recommendations of API Explorer. The results of our evaluation provide evidence that the use of structural relationships to make API elements discoverable could be a viable, and an inexpensive, alternative to API wrappers or API restructuring when seeking to improve discoverability in APIs.

CHAPTER 5
Introspector: Facilitating Effective Query Formulation

Learning from code examples is one strategy developers employ when learning how to use an API. In the absence of appropriate examples in the documentation of an API, developers turn to the Web or to code repositories such as Koders, or Google Code Search in search for relevant code examples. To locate code examples relevant to a given task, a developer has to formulate a search query, execute the search query against a code repository or the Web, and go through the search results to identify a suitable example. Formulating an *effective* search query (that is, a search query that would return code examples relevant to a given a task) is however difficult because developers often lack a clear understanding of what they need when beginning a new task, and even when they do, they have difficulty providing keywords that corresponding to those used by the API designers. Research has reported a likelihood of less than 15% of having two people choosing the same keyword for a familiar object [18]. The difficulty of selecting common keywords for familiar objects, referred to as the vocabulary mismatch problem (that is, multiple words for the same topic), has been identified as a major challenge to query formulation [16, 18].

To help developers formulate effective search queries for locating code examples, we propose an approach based on the use of API types as keywords in a search query. Our approach takes as input an API type — the *seed* — and recommends a list of other types that should be used together with the seed to formulate a search

query. Our approach is based on the intuition that the names of API types provide a common vocabulary between API users and designers, and therefore the use of API types in query formulation reduces the ambiguity as to the kind of code example that a user is searching for. Consider the example of a developer working on a task for sending an email using the JavaMail API (see Section 1.2.3). With our approach, a developer would provide an API type relevant to sending an email: for instance, the developer may provide `javax.mail.Message` as the seed. However, the seed alone may not be sufficient to locate relevant code examples. For instance, the top five results of using `javax.mail.Message` as a search query on Google Code Search do not contain a relevant example for sending an email. Code repositories index the source code of not only how an API may be used, but also the code of the API itself. Therefore, the top results of the search query "`javax.mail.Message`" simply point to different locations of its source file.[1]

We observed that we can greatly improve the chances of locating a relevant code examples by combining the seed with one or more other API types that are associated with its usage context. For instance, given the seed `Message`,[2] our approach combines several heuristics to identify and recommend other types in the API that are often associated with its usage: for instance, our approach would recommend several types

[1] The search was performed on October 12, 2011: see Figure 6–1, Appendix B, for the screen shots of the search results.

[2] For brevity and to enhance readability, `javax.mail.Message` is represented by its *simple name* `Message`. Future references to other types will also be represented by their simple names.

including `Address` and `Transport`. The developer may then formulate and execute the search query "`Message`, `Address`, `Transport`" (interpreted as "retrieve code examples that reference the types `Message`, `Address` and `Transport`") against a code search engine (Google Code Search, or Koders) supported by our tool, Introspector. The first search results in this case is a code example which demonstrates how to send an email using the Java Mail API.

We present our heuristics for identifying and recommending API types that should be used together with a seed for searching for code examples, and the evaluation of our approach.

5.1 Recommending Related Types

A given API type (henceforth referred to as the *seed*) is typically related to several other types in an API. Our intuition is that through the use of association-based heuristics and the relationships between API types, we can identify a core set of types that are often associated with the usage of a seed. We use a two phase approach to accomplish our goal: in phase one, we begin by identifying an *initial set* of types that are structurally related to the seed; in phase two, we apply five different heuristics to the initial set of types from phase one to recommend the types of an API that are frequently associated with the usage of the seed, and therefore suitable for searching for code examples involving the seed.

5.1.1 Phase I — Identifying an Initial Set of Types Related to the Seed

The goal of this phase is to isolate a subset of the API that would be suitable for a more detailed analysis of the types frequently associated with the usage of a seed. We begin by introducing two concepts that are relevant to identifying the initial set:

Distance: this is a measure of how far an API type is from the seed, for a given structural relationship. For instance, assuming that the type A (the seed) is the super class of the type B, and B is the super class of the type C: then we say that B is at a distance of one from A, and C is at a distance of two from A, given the subclass relationship. We use the edges of the API Exploration Graph (see Section 4.1) to measure the distance from an API type to a seed: a distance of one corresponds to an edge between the seed and a type in the API. We use the *distance* from the seed to determine how far to traverse a tree along edges corresponding to a relationship type when generating the initial set.

Working set: this represents the set of API types that have been identified after applying a given relationship from the seed to the types that are at a given distance from the seed. For instance, given the subclass relationship example, the working set will initially contain just the seed: {A}; for distance one, the set will now include B, the subclass of the seed: {A, B}; for distance two, the set will include C, the subclass of B: {A, B, C}. The given relationship is *only* applied to the new members of the working set: that is, the types that were identified by the previous iteration.

We use the following structural relationships to identify an initial set of API types for a given seed, within a specified distance:

- *Parent*: the parent class or interfaces of the seed, and of the types added to the working set for each iteration, within a given distance. For instance, if `MimeMessage` is the seed, this relationship would identify only its parent class `Message` for distance one. However, for distance two, this relationship would

also identify the parent class or interfaces of `Message` (a type added to the working set by the first iteration).

- *Subtype*: the subtypes of the seed, and of the API types added to the working set for each iteration, within a given distance.
- *Return-type*: the API types with methods that have the seed, or a type added to the working set, as a return-type.
- *Incoming Parameter*: the API types that are a parameter to a method defined on the the seed, or methods defined on a type added the working set.
- *Outgoing Parameter*: the API types with methods that take the seed, or a type added to the working set, as a parameter.
- *Polymorphic*: the relationships presented above capture only simple relationships between a seed and its related types. For instance, the *subtype* relationship can identify only the subtypes of the seed, and the *outgoing parameter* relationship can only identity types that take the seed as a parameter. However, types that take a subtype of the seed as a parameter would not be identified by any of the relationships presented above. The polymorphic relationship is an aggregation of the *subtype* and the *outgoing parameter* relationships, and is used to identify the API types with methods that take subtypes of the seed as a parameter, within a given distance. For instance, if `MimeMessage` is the seed, and `MimeMessage` is a subtype of `Message`, then given the method `Transport.send(Message)`, `MimeMessage` would also be considered related to `Transport`.

- *Object-construction*: this relationship identify types related to the seed, or a type added to the working set, through object instantiation. This relationship is an aggregation of the *return-type*, *subtype*, and the *parameter* relationships. For instance, given that `Message` can be created from the constructor of its subtype `MimeMessage(Session)`, then `Message` is also considered related to `Session`.
- *Common friends*: If the type A is related to the type B, and the type C is related to B, then A would be considered related to C for distance two. For instance, in the example below, `Message` is related to `MessageWrapper`, and `Transport` is related to `MessageWrapper`, therefore `Message` is also related to `Transport` through the common friends relationship.

  ```
  MessageWrapper wrapper = new MessageWrapper(Message);
  Transport.send(wrapper);
  ```
- *Used together*: this relationship identify types that are commonly used together as method parameters. For instance, given the method `Transport.send(Message, Address)`, we consider the types `Message` and `Address` to be related through the used-together relationship.

Most of these relationships are borrowed from the API Exploration Graph; we extended the graph with two new relationships (Common friends and Used together) not captured by the original version. Given a seed as input, we use the above-mentioned relationships to generate an initial set of types to be used in phase two to identity API types that are frequently associated with the usage context of the seed.

5.1.2 Phase II — Heuristics

The API types identified in phase one are structurally related to the seed, but most of the types in the initial set are not always associated to its usage. To eliminate types not relevant to the usage of a seed, and to identify types often associated with the usage contexts of the seed, we apply five different heuristics to the initial set of types. Each heuristic captures a different form of association between a seed and its related types, and the results of the heuristics are eventually combined to recommend API types with a strong association to the seed.

I. Tree-Based Heuristic

Object construction is typically the first step in making use of an API type. Therefore, the API types necessary to construct an instance of a seed are associated with its usage context. However, an API type may have multiple instantiation patterns involving several types. The goal of the construction chain heuristic is to identify those API types that are present on all of the construction paths of the seed. To achieve our goal, the heuristic begins by creating a *construction-chain tree* with the seed as the root (see Figure 5–1). The construction-chain tree represents all the possible instantiation options for a given seed, and also of the types necessary to create an instance of the seed. For each level in the tree, we compute the likelihood (referred to as the level-based likelihood) of encountering each type at that level. The likelihood is a function of the structure of the API, and assumes a uniform distribution of all the branches at each node of the tree. To compute the likelihood that a given API type would be required to create an instance of the seed, we combine the level-based likelihoods for the path from that type to the root of the tree.

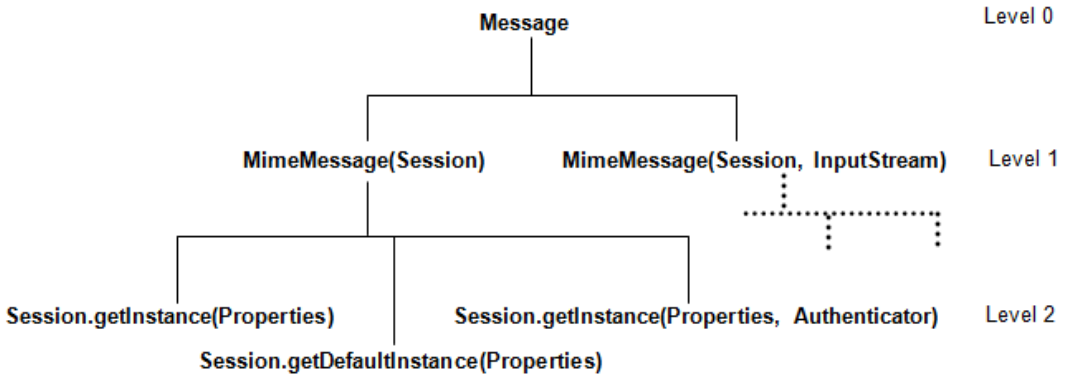


Figure 5–1: Partial construction tree for the type `Message`.

Assume that the seed is `Message` (its construction tree is presented in Figure 5–1). Level 1 of the tree present two options of creating the type `Message`: `MimeMessage(Session)` and `MimeMessage(Session, Inputstream)`. There are three distinct types at this level: `MimeMessage`, `Session`, and `Inputstream`. The level-based likelihood of encountering `MimeMessage` is 1.0 because it is on both paths; the likelihood for `Session` is also 1.0. However, the likelihood for `Inputstream` is 0.5 because it is on only one of the two paths. At level 2, we have three options of creating an instance of `Session` involving two types: `Properties` and `Authenticator`. In this case, the level-based likelihood of encountering `Properties` is 1.0 since it is on all three paths, but the likelihood for `Authenticator` is 0.33 since it is on a single path. Once all the level-based likehoods have been generated, we then proceed to compute the likelihood that a type in the tree would be necessary to instantiate an object of the seed — `Message`: we call this value the tree-based likelihood.

To compute the tree-based likelihood for type T, we multiply the level-based likelihoods of each type along the path from the node T to the root of the tree. For

instance, the tree-based likelihood that the type `Properties` would be required to create an instance of `Message` is the product of level-based likelihoods for `Message`, `MimeMessage`, `Session`, and `Properties` which gives us a value of 1.0. In contrast, the tree-based likelihood of the type `Authenticator` is the product of level-based likelihoods of `Message`, `MimeMessage`, `Session`, and `Authenticator`, which gives 0.33. Observe that a type with a tree-based likelihood of 1.0 must be present to create an instance of the seed, and therefore associated with the usage context of the seed. For instance, in our example, the types `MimeMessage`, `Session`, and `Properties` must be present to create an instance of `Message`. This heuristic outputs the set of API types with a tree-based likelihood of 0.5 or above (that is, the types that are on 50% or more of the construction paths of a seed).

II. Documentation Heuristic

The documentation heuristic is based on the observation that API designers often make reference to other types relevant to the use of a given type in the documentation, when such API documentation is available. For instance, the description section of the Javadoc of the `javax.mail.Message` makes reference to both the `MimeMessage` and the `Transport` types necessary for creating an instance of `Message` and for sending an email, respectively (see Figure 5–2). Our intuition is that for cases where the Javadoc of an API is available, the types mentioned in the description section of the documentation are relevant to the use of the seed.

The documentation heuristic begins by extracting the simple name of the API types in the initial set of types, then it checks whether the simple name of a given type is mentioned in the description section of the Javadoc of the seed. We use regular

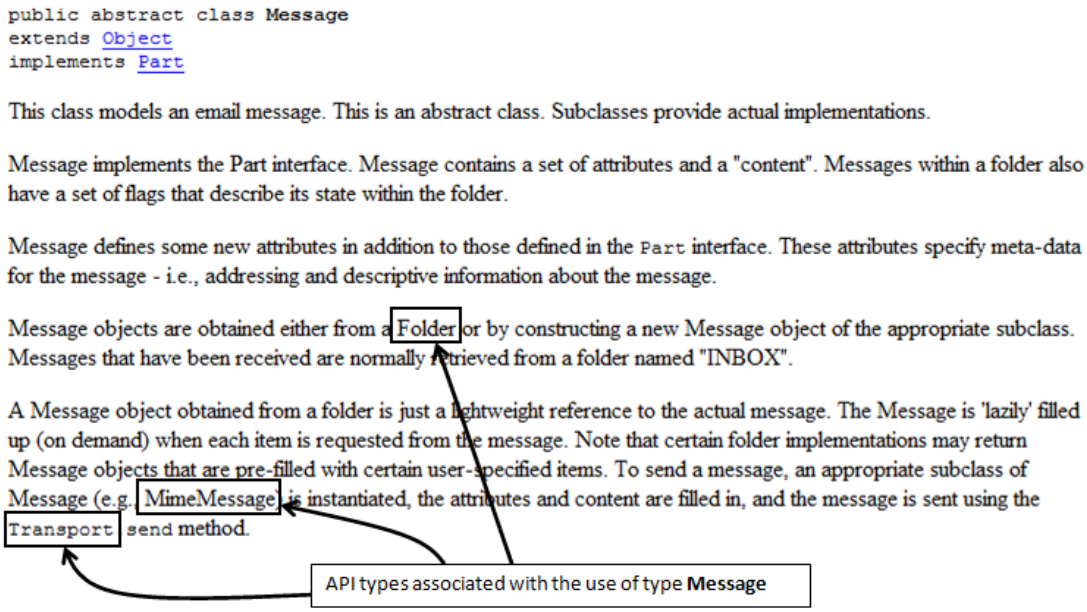


Figure 5–2: An example of API types referenced in the description section of the Javadoc of the type `Message` of the JavaMail API (source: http://download.-oracle.com/javaee/6/api/javax/mail/Message.html, Oracle Corporation; retrieved on September 25, 2011).

expressions to search through the text of the description section for references to a given simple name. This heuristic outputs the API types in the initial set that are also mentioned in the description section of the Javadoc of the seed; it does not keep track of the number of times an API type was mentioned.

We use the simple name of an API type to determine whether or not a given type is referenced in the documentation of the seed because our observations suggest that designers typically use simple names, not absolute names, when referring to API types in the Javadoc (see Figure 5–2). The use of the simple name to look for types referenced in the Javadoc is imprecise, and raises the possibility for both false positives (for instance, if the simple name is a prefix to a word in the documentation)

and false negatives (for instance, if references to an API type is misspelled). However, we consider the simplicity of this heuristic appropriate for our purpose (a simple check for the presence or absence of a given word).

III. Data Flow Heuristic

This heuristic returns a list of API types that often flow in and out of the methods defined on the seed. For instance, `Address` would be considered an incoming type of `Message` because of the method `Message.setFrom(Address)`, and `Folder` would be considered an outgoing type because the method `Message.getFolder()` returns a `Folder`. Given a seed such as the type `Message`, the data flow heuristic computes the number of times in which an API type was either an *incoming parameter* or a *return-type* of the methods defined on `Message`. The output of this heuristic is a set of API types that occurred two or more times as a parameter or return type of methods defined on the seed. Our intuition is that types that occurred two or more times as either an incoming or outgoing type of the seed are highly relevant to its usage.

IV. Collaborator Heuristics

We defined a *collaborator* as any type with a method that takes the seed as a parameter. For instance, `Transport` is considered a collaborator of `Message` because of the method `Transport.send(Message)`. For each collaborator, we count the number of times the seed occurred as a parameter on one or more of its methods. Our intuition is that collaborators that have the seed as a parameter on two or more of their methods are highly relevant to the use of the seed. This heuristic computes, for each

collaborator, the number of times the seed occurred as a parameter of its methods. The output of this heuristic is a set of collaborators in which the seed occurred at least twice as a parameter.

V. Type Usage Heuristic

This heuristic is based on the observation that the implementation of some collaborators reference not only the seed but also other API types that may be relevant to the use of the seed. For instance, although not evident from its interface, the `Transport` class of the JavaMail API references the types `Address` (which contains the email address of the sender and recipients) and `Session` (which contains the mail server information), both relevant to the use of `Message`. We therefore expect some overlap between the types referenced within the collaborators and the types most relevant to the use of a seed. We used bytecode analysis to generate the type usage information for a given seed since our approach is based on the assumption that the source code of an API may not be available to the developer. This heuristic begins by looking for all the types referenced within each collaborator of the seed: that is, all the fields declared by a collaborator and all the types referenced within methods of the collaborator class. Then, the heuristic computes the intersection between the set of types referenced within each collaborator and the initial set of types for a given seed. We consider types that are common between both sets to be associated with the use of the seed.

Recommending API types *strongly* associated with the usage of a seed.

Each of the five heuristics captures a different form of association between a seed

and its related types: for instance, assuming the seed is `Message`, we know from the Documentation heuristic that the type `Transport` may be associated with the use of the type `Message`. However, a single association is typically not enough to recommend an API type as *strongly* associated to the usage context of a seed. We use the number of heuristics that support a given association between a seed and a related API type as a measure of the strength of the relationship. The relationships between a seed and an API type that are supported by all five heuristics are ranked first, relationships that are supported by four heuristics are ranked second, relationships that are supported by three heuristics are ranked third, relationships that are supported by two heuristics are ranked fourth, and those supported by only a single heuristic are ranked last. We recommend only API types that were identified by two or more of the five different heuristics as being associated to the usage context of a seed. The types identified by the construction chain heuristics with a likelihood of 1.0 are treated differently — i.e., they do not need additional support from other heuristics — since such types are on every instantiation path of the seed, and by default, are ranked first.

5.2 Evaluation

Our approach is based on the intuition that the use of API types to formulate search queries can mitigate the challenges a developer encounters when searching for code examples relevant to implementing a programming task. Specifically, we claimed that the types recommended by Introspector can assist a developer in locating code examples relevant to implementing a given programming task. We designed our evaluation to investigate the following research question:

Are the API types recommended by Introspector for a given seed effective for searching for code examples relevant to a given task?

In our evaluation, we considered a search query to be *effective* if the solution of a given task is amongst the top three code examples recommended by Introspector for that query.

To investigate our research question, we used ten programming tasks from six different real-world APIs (see Table 5–1). The description and the solution of each task was provided by the API designers and form part of the documentation and tutorials that came with the respective API. We selected tasks that required at least four types from the corresponding API, and some of the tasks and APIs we selected have been used in previous studies (the Eclipse Java Development Tool and Resources APIs for Task 1 and Task 2 were used by Holmes et al. [21] and Bajracharya et al. [1]; the Apache POI API was used by Jureczko et al. [24]; the Twitter4J API was used by Nita and Notkin [33]). We used the task description and the source code provided by the API designers for each task as an oracle for evaluating the effectiveness of our approach.

Seed Selection

Our approach takes as input an API type — the *seed* — and recommends additional types that should be used together with the seed to search for relevant code examples for a given task. To minimize seed selection bias and to ensure that the results of the evaluation are not based on arbitrary seed selection, we used three different seed selection schemes to evaluate our approach:

Table 5–1: Tasks Description.

Task ID	Task Description	API	API Description
1	Create a Java project.	eclipse.jdt.core, eclipse.core.resources	Eclipse Java Development Tool and resources APIs.
2	Create an AST.	eclipse.jdt.core, eclipse.core.resources	
3	Create a cell in Excel.	Apache POI	API for manipulating excel files.
4	Add an image to an Excel file.	Apache POI	
5	Search for Tweets.	Twitter4J	Java library for Twitter.
6	Asynchronously update status.	Twitter4J	
7	Obtain cache statistics.	JCache	General purpose caching API for Java applications.
8	Convert a Word document to PDF.	JODConverter	Java library to convert documents between different office formats.
9	Send a text message.	Smack	An instant messaging API.
10	Check availability ("presence") of other users.	Smack	

I. Central-type scheme. Previous work observed that when working with an unfamiliar API, developers begin with a central-type representing the concept to be implemented before looking for other relevant types [12, 51]. For instance, developers would begin with the type `PDFDocument` when working on a task for manipulating PDF documents. For tasks that have been the subject of a previous study, we used the same seed as in previous work. For instance, Holmes and Murphy used the type `ASTParser` for Task 2 [21], and we did the same. For tasks that have not been used in previous studies, we selected a type we considered to be central to each task as the seed using the observation from previous work. The seed used for each task, for the Central-type scheme, is presented in Table 6–1, Appendix A. Even with the inclusion of tasks from previous studies, the central-type scheme remains subject to investigator bias. To help minimize the seed selection bias and to provide important comparison points for our results, we applied two more seed selection schemes: Keyword-based and Random.

II. Keyword-based scheme. Developers working with unfamiliar APIs often use keywords from the task description to help locate types relevant to their task [12]. For instance, given a task description such as *"Compose and send and email message using the JavaMail API"*, a developer may look for an API type containing the keyword "email" or "message". We used this strategy to look for keywords in the task description that correspond to types in the solution for each task, and used those types as the seed to our approach. For instance, the type `javax.mail.Message` would be used as the seed for the task of composing and sending an email message since this type contains the keyword "message". The keywords used for selecting

the seeds for the different tasks, and the seed used for each task, are presented in Table 6–2, Appendix A.

III. Random scheme. At times, the task description may not contain keywords that point to types relevant to implementing a given task. We used the random seed scheme to reflect instances where a developer may begin with any of the types relevant to implementing a task. For the random scheme, we consider each type necessary to implement a task as a potential seed, and average the results across the different seeds. For instance, five API types (`IProject`, `IJavaProject`, `JavaCore`, `IWorkspace`, and `IWorkspaceRoot`) are needed to implement Task 1: in our evaluation, we used each of the five types as a seed to recommend additional API types to be used for searching relevant code examples. We would report the effectiveness of the Random Scheme for Task 1 to be 2/5, assuming that only two of the five seeds lead to a relevant code example. There were a total of 47 different seeds across the ten tasks: we present the seeds for each task in Table 6–3, Appendix A.

Methodology.

We applied the three seed selection schemes to each of the ten tasks, and for each task, we obtained different API types to be used as the seeds for evaluating our approach. Next, we ran Introspector for each task, and for each seed, and obtained an expanded set of API types identified as being often associated with the usage context of the seed. We used the value three as the distance from the seed when generating the expanded set. We selected a distance of three because, in our initial investigation, we found that a distance above three created a much larger initial set

with types less relevant to the seed, but did not increase the effectiveness of our heuristics. We generated three search queries for each seed by combining the seed with the top K (K = 1, 3, and 5) API types identified as relevant to the use of the seed. Finally, using Introspector for each task and for each K, we executed the search query against the Google Code Search repository, and obtained a list of code examples ranked based on their relevance to the search query.

To evaluate the effectiveness of the search queries for each task, we manually compared the top three code examples returned by Introspector for each K to the solution provided in the documentation for the task. We also compared the solution provided for each task with the results of using just the *seed alone* as a search query. In our comparison, we considered a code example relevant to a given task if it was an exact match to the solution provided by the API designers for the task, or if it contained code snippets that could be extracted to form a solution for the task. We present the results of the Keyword and Central-types schemes, for the different values of K, in Tables 5–2 to Tables 5–5. The results of the Random scheme are summarized in Table 5–6.

Comparison to Natural Language Queries.

As part of our evaluation, we compared our approach to the use of natural language queries when searching for code examples. To formulate the search query for each task, we used the verbs, nouns, and direct-objects (that is, the object a verb acts on; for instance, the phrase "remove the attribute" has verb "remove" and object "attribute") [44] of the description of the task. This approach for formulating search queries from task description was inspired by previous work [20, 44]. We also included

Table 5–2: The relevance of the code examples recommended by Introspector for the *Keyword* and *Central-type* schemes: **Seed alone**.

	Keyword(40%)		**Central-type(50%)**	
	In top 3?	**Quality**	**In top 3?**	**Quality**
Task 1	Yes	Snippet	Yes	Snippet
Task 2	Yes	Exact	Yes	Snippet
Task 3	Yes	Snippet	Yes	Snippet
Task 4	Yes	Snippet	Yes	Snippet
Task 5	No	—	No	—
Task 6	No	—	No	—
Task 7	No	—	No	—
Task 8	—	—	Yes	Snippet
Task 9	No	—	No	—
Task 10	No	—	No	—

the programming language of the API (Java, in our case) and the name of the API as part of the search query to avoid ambiguity. The natural language query used for each task is presented in Table 6–4, Appendix A. We executed the query of each task on the code search engines (Google Code Search and Koders) supported by Introspector, and compared the top three code examples returned by each code search engine to the solution provided for the task. We used the same approach as explained above to determine the relevance of the code examples returned by the search engines. We summarize the results on the use of natural language queries in Table 5–7.

5.2.1 Results

Introspector — Seed Selection Schemes.

Table 5–3: The relevance of the code examples recommended by Introspector for the *Keyword* and *Central-type* schemes: $\mathbf{K = 1}$.

	Keyword(70%)		**Central-type(90%)**	
	In top 3?	**Quality**	**In top 3?**	**Quality**
Task 1	Yes	Snippet	Yes	Snippet
Task 2	Yes	Exact	Yes	Snippet
Task 3	Yes	Snippet	Yes	Snippet
Task 4	Yes	Snippet	Yes	Snippet
Task 5	Yes	Snippet	No	—
Task 6	No	—	Yes	Exact
Task 7	Yes	Snippet	Yes	Snippet
Task 8	—	—	Yes	Exact
Task 9	Yes	Snippet	Yes	Snippet
Task 10	No	—	Yes	Snippet

Table 5–4: The relevance of the code examples recommended by Introspector for the *Keyword* and *Central-type* schemes: $\mathbf{K = 3}$.

	Keyword(80%)		**Central-type(90%)**	
	In top 3?	**Quality**	**In top 3?**	**Quality**
Task 1	Yes	Snippet	Yes	Snippet
Task 2	Yes	Snippet	Yes	Exact
Task 3	Yes	Snippet	Yes	Snippet
Task 4	Yes	Snippet	Yes	Snippet
Task 5	Yes	Snippet	No	—
Task 6	No	—	Yes	Exact
Task 7	Yes	Snippet	Yes	Snippet
Task 8	—	—	Yes	Snippet
Task 9	Yes	Snippet	Yes	Snippet
Task 10	Yes	Exact	Yes	Snippet

Table 5–5: The relevance of the code examples recommended by Introspector for the *Keyword* and *Central-type* schemes: $\mathbf{K = 5}$.

	Keyword Scheme		**Central-type Scheme**	
	In top 3?	**Quality**	**In top 3?**	**Quality**
Task 1	Yes	Snippet	Yes	Snippet
Task 2	Yes	Exact	Yes	Snippet
Task 3	Yes	Snippet	Yes	Snippet
Task 4	Yes	Snippet	Yes	Snippet
Task 5	—	—	—	—
Task 6	No	—	—	—
Task 7	—	—	—	—
Task 8	—	—	Yes	Exact
Task 9	Yes	Snippet	Yes	Snippet
Task 10	Yes	Exact	Yes	Snippet

The second and third column of Tables 5–2 to 5–5 presents a summary of the results of the *Keyword* and *Central-type* schemes: the "In top 3" column indicates whether the solution for a given task is amongst the top three code examples recommended by Introspector; the "Quality" column indicates whether a recommended code example can be used without the need for customization: "Exact" implies the recommended code example can be used without customization, and "Snippet" implies the recommended code example has to be customization to get a solution for the task. We used a dash (—) in the "In top 3" column to indicate instances in which a search query could not be generated for a given scheme, or for a given value of K. For instance, we could not generate a search query for the Keyword scheme for Task 8 because there was no keyword in the task description that matched the

Table 5–6: A summary of the proportion of seeds, for each task, with a relevant code example amongst the top three examples recommended by Introspector for the *Random* scheme. For instance, for Task 1, three of the five (3/5) seeds lead to a relevant code example when the *seed alone* was used as a search query. However, across all the ten tasks, only 15 (that is, 31%) of the 47 seeds led to a relevant code example given the *seed alone*.

	Seed Alone	**K = 1**	**K = 3**	**K = 5**
Task 1	3/5	4/5	4/5	4/5
Task 2	2/4	3/4	4/4	4/4
Task 3	4/6	6/6	6/6	6/6
Task 4	2/6	2/6	2/6	—
Task 5	0/5	2/5	3/5	—
Task 6	2/5	3/5	4/5	—
Task 7	0/4	2/4	3/4	—
Task 8	2/4	4/4	3/4	—
Task 9	0/4	3/4	4/4	4/4
Task 10	0/4	1/4	3/4	3/4
Average:	15/47 (31%)	30/47 (63%)	36/47 (76%)	—

name of an API type. Also, we could not generate a search query in ten cases for K = 5 because the number of API types recommended by Introspector was below five (see Table 5–5). Lastly, the value next to the name of each scheme represents the proportion of the search queries, for the given value of K and for a given seed selection scheme, with a relevant code example amongst the top three examples recommended by Introspector. For instance, when the seed alone was used as the search query, four of the ten (that is, 40%) search queries of the Keyword scheme provided code examples relevant to the corresponding task (see Table 5–2). In general, for the Keyword and Central-type schemes, we observed that the effectiveness of the search

queries improved as we increased the value of K from the seed alone to the top one, and top three recommended types. The effectiveness of the search queries for the Keyword based scheme improved from 40% (seed alone) to 70% (K=1), then to 80% (K=3); and the Central-type scheme improved from 50% (seed alone) to 90% (K=1), and remained unchanged for K=3. We did not provide the proportion of effective queries for K=5 since search queries could not be created for ten cases.

Table 5–6 provides a summary of the results for the Random scheme. For each task, and for each value of K, we provide the proportion of the seeds with a relevant code example amongst the top three examples recommended by Introspector. For instance, we observed that three of the five (that is, 3/5) seeds of Task 1 led to a relevant code example when the seed alone was used as a search query. However, when we consider all ten tasks, only 15 (that is, 31%) of the 47 seeds led to a relevant code example when just the seed was used as a search query. When we used the first API type (that is, K=1) recommended by Introspector together with the seed as the search query, the effectiveness of the Random schemes, across all the ten tasks (that is, the 47 seeds), increased to 63%. Code search engines index the source code of not only how an API is used, but also the source files of the API. We observed that most of the top three code examples point to different source files of the API when the seed alone is used as a search query. By providing an additional API type associated with the usage context of the seed, we significantly improved the quality of the search results.

The first API type recommended by Introspector was not always associated with the usage of the seed. This explains why the effectiveness of the Random scheme,

for K=1, was 63%. When we increased the value of K to the top three API types recommended by Introspector, the effectiveness of the Random scheme, across all ten tasks, increased to 76%. The reason for the increase is that most of the search queries generated by Introspector for each of the 47 different seeds now contain at least one additional API type relevant to the usage context of the seed. The improvement in the effectiveness of the queries given the increase from K=1 to K=3 highlights the robustness of our approach: for 36 of the 47 different seeds across the ten tasks, there was at least one API type associated to the usage context of a seed amongst the top three API types recommended by Introspector.

Notwithstanding the demonstrated usefulness of Introspector to increase the performance of searches, there are some factors which limit the effectiveness of our approach: API evolution, incorrect recommendations, and multiple usage contexts.

API evolution: as APIs evolve, API designers are faced with the challenge of supporting existing client code while making changes to support changing requirements. With time, an API may provide both a deprecated approach, and a newer approach for implementing a given task. However, our approach cannot differentiate between the deprecated and the new approach of implementing a task, resulting in irrelevant recommendations. For instance, for task 4 ("Add an image to an Excel file"), Introspector recommended no code examples: on closer examination, we observed that the API provides support for both the older and newer Excel file formats; however, the search query generated by Introspector was for the older Excel file format for which code examples may not be available.

Incorrect associations: Our approach recommends related types based on association heuristics, and the strength of the relationship with the seed (that is, the number of heuristics that support the relationship between a seed and its related types). At times, the associations inferred by the heuristics may be incorrect, resulting in irrelevant recommendations. The relationship between the seed and a relevant type must be supported by two or more of the five heuristics, to be recommended by our approach. We observed that there are cases where an API type relevant to the use of a seed may not be recommended because the support from the heuristics is limited — that is, the relationship between a seed and a relevant API type is supported by only a single association heuristic. For instance, Introspector did not recommend API types to be used with the seed as a search query for task 5, for the Central-type scheme. On closer examination, we observed that the relationship between the seed and the relevant types for the task had support only from a single heuristic. The results of the evaluation however suggest that the number of cases where our approach may omit relevant related types is limited.

Multiple usage contexts: An API type may have multiple usage contexts: some usages may be related to its primary purpose, and there may be other secondary usage contexts. For instance, the primary usage context of the type `Message` of the JavaMail API is creating and sending an email message, and this fact is reflected in its Javadoc which references the type `MimeMessage` (for instantiating a `Message` object) and the type `Transport` (for sending an email message). However, the type `Message` may also be used in other contexts, such as for retrieving email messages from a mail server. Our approach is based on measures (such as, the instantiation

Table 5–7: The relevance of the code examples returned by the code search engines using natural language queries.

	Google Code (20%)		**Koders (30%)**	
	In top 3?	**Quality**	**In top 3?**	**Quality**
Task 1	No	—	No	—
Task 2	No	—	No	—
Task 3	No	—	Yes	Snippet
Task 4	No	—	Yes	Snippet
Task 5	Yes	Snippet	No	—
Task 6	No	—	No	—
Task 7	No	—	No	—
Task 8	No	—	No	—
Task 9	No	—	Yes	Snippet
Task 10	Yes	Exact	No	—

pattern of a seed, and the types referenced in the Javadoc of the seed) that favor the primary usage of an API type. Consequently, the expanded set generated for a given API type may not lead to relevant code examples for tasks which involve uncommon, secondary usages of an API type.

Natural Language Queries (NLQ).

Table 5–7 presents the results of the use of NLQ to search for code examples for the ten tasks using Google Code Search and Koders. We found a solution for only two of the tasks amongst the top three code examples returned by Google Code Search, and for three of the tasks amongst the top three code examples returned by Koders. Code search engines retrieve code examples by matching keywords in a search query to the source code in their repository. The word usage in a task description, or the

keywords that may be used by a developer to formulate a search query may not reflect the word usage in the API, or code examples that make use of the API. This explains why the search queries from NLQ were not effective in retrieving relevant code examples.

5.2.2 Discussion

In general, our approach — the use API types to formulate search queries — outperformed the use of NLQ when searching for code examples relevant to a task. Observe that the results for the Random scheme takes into account the seeds of both the Central-type and the Keyword-based schemes: the results of the Random scheme is therefore an average across all the ten tasks and the three seed selection schemes. We observed that the results of using the seed alone as a search query (that is, 31% effectiveness) is comparable to the use of NLQ. However, there is a significant improvement in the effectiveness of the search queries when the results of the NLQ are compared to the results for K=1 and K=3. For instance, for the Random scheme, we found a relevant solution for 61% (K=1) to 76% (K=3) of the 47 different seeds across all the ten tasks. In contrast, we found a solution for only two of the tasks with Google Code Search, and for three of the tasks with Koders using NLQ.

The use of natural language queries to search for code examples is ineffective when API users have a hard time guessing the words that were used by the API designers to implement a given feature of an API. API types provide a common vocabulary between API users and API designers; thus, the use of API types to formulate queries seems to be an effective strategy when searching for code examples relevant to a programming task.

Symmetry of expanded sets. A potential concern about our approach is the possibility that the performance of our tool is independent of the choice of seed for a given task. In other words, if the overlap between the expanded sets of the types relevant to implementing a task is high, then it matters less what the choice of seed is for a given task. To investigate this concern, we generated the pairwise overlap between all the possible seeds for each task (that is, all the types of an API relevant to implementing a given task), for the ten tasks. Specifically, for each possible pair of seeds, for each task, we began by generating the expanded set for the two seeds; next, we computed the intersection of the expanded sets; then we computed the average size of the expanded set for the given pair.

We summarize the results of the this analysis using bar charts: the results of the first, the middle, and the last task are presented in Figure 5–3; the results of the other tasks are presented in Appendix C. Each group of blue and red bars represents a summary of the overlap for a pair of seeds for a given task. The blue bars represent the average size of the expanded set for a given pair, and the red bars represent the overlap between the expanded set for a given pair. Blue bars without a corresponding red bar indicates that there is no overlap between the expanded sets for that pair. For instance, there was no overlap between the expanded sets in three of the ten pairs of seeds for Task 1. We observe that, in general, the overlap between the expanded sets for the seeds in each task was low: therefore, the performance of our approach is mostly sensitive to the choice of the seed.

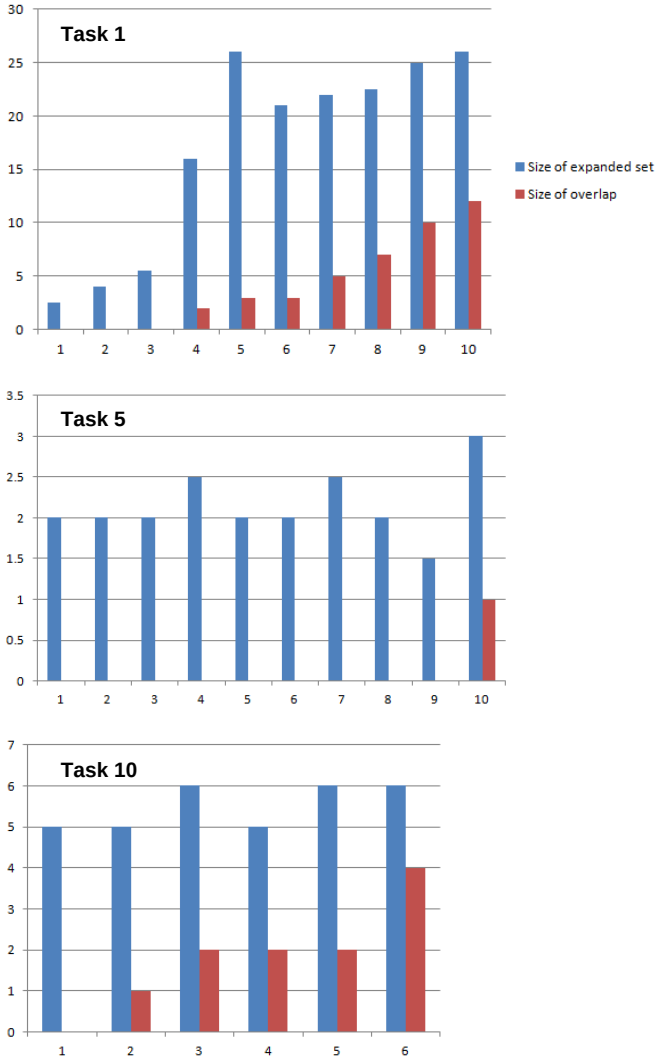


Figure 5–3: A summary of the pairwise overlap between the expanded sets of the seeds for the first (Task 1), middle (Task 5), and last task (Task 10). The blue bars represent the average size of the expanded set for a given pair, and the red bars represent the overlap between the expanded set for a given pair.

5.3 Summary

Searching for code examples for a programming task is a common strategy used by developers working with unfamiliar APIs. However, formulating effective search queries for locating relevant code examples is challenging because of the vocabulary mismatch problem — multiple words for the same topic. We proposed a technique based on API types to help facilitate the process of query formulation: our idea is that instead of trying to guess the most appropriate keywords to use as the search query, the developer would provide an API type relevant to the scenario to be implemented, then our technique would recommend relevant types strongly associated with the usage of the given type. The developer would then use the recommended types as a starting point for searching for code examples. We implemented our approach as an Eclipse plug-in called Introspector, and evaluated the approach quantitatively using ten tasks from six APIs. We also compared our approach to the use of natural language based queries. We observed that our approach is effective in identifying API types associated with the usage context of a seed, and that our query formulation approach is the most effective when the top three API types recommended by Introspector are combined with the seed to search for code examples. The evaluation also suggest that our query formulation strategy is more effective than the use of natural language based queries when searching for code examples.

CHAPTER 6
Conclusions

Learning how to use APIs has become a major part of a software developer's job. The goal of this thesis was to identify and understand some of the challenges a developer encounters when working with unfamiliar APIs, and to design and implement novel tools to facilitate the API learning process. To achieve this goal, we started with an exploratory study to investigate the different types of questions developers ask when working with unfamiliar APIs, to identify those questions that are difficult to answer, and to investigate the cause of the difficulty. Our study involved twenty participants working on the same two programming tasks using real-world APIs. The study generated over 20 hours of screen captured video and the verbalization of the participants spanning 40 different programming sessions.

Our analysis of the data involved generating generic versions of the questions asked by the participants about the use of the APIs, abstracting each question from the specifics of a given API, and identifying those questions that proved difficult for the participants to answer. Based on the results of our analysis, we isolated twenty different types of questions the programmers asked when learning to use APIs, and identified five of the twenty questions as the most difficult for the programmers to answer in the context of our study. Our analysis also provides evidence to explain the cause of the observed difficulties. The different types of questions we identified and the difficulties we observed can be used to evaluate tools aimed at improving

the API learning process, and to identify areas of the learning process where tool support is lacking or could be improved. For instance, we observed that tool support to assist a programmer in identifying types that would serve as a good starting point to search for code examples or to explore an API for a given task is limited.

In the second phase, we designed and evaluated programming tools (*API Explorer* and *Introspector*) to address some of the questions we identified as being difficult for developers to answer when working with unfamiliar APIs. The API Explorer tool was designed to address the difficulty a developer faces when the API elements necessary to implement a task are not accessible from the type the developer is working with. API Explorer leverages the relationships between API elements to recommend relevant methods on other objects, to identify API elements relevant to the use of a method or class, and to support a developer in combining multiple objects. We implemented API Explorer as a novel extension of the content assist feature of the Eclipse IDE, and evaluated API Explorer through a case study in which eight participants completed four programming tasks with several discoverability hurdles. The participants experienced little difficulty selecting relevant API elements from the recommendations of API Explorer, and found the recommendations of API Explorer helpful in locating helper-types in several discoverability contexts.

The Introspector tool was designed to address the difficulty of formulating effective queries when searching for code examples relevant to implementing a task. Introspector combines the structural relationships and the flow of information between API types to recommend types that should be used together with a seed to search for code examples for a given task. Using the recommended types as search

query, a developer can search for code examples across two code repositories using Introspector, and in return, will get a list of code examples ranked based on their relevance to the search query. We evaluated Introspector quantitatively using ten tasks from six different APIs. We also compared the recommendations of Introspector to the use of natural language based queries when searching for code examples. The results of the evaluation suggest that the use of API types to formulate search is a more effective strategy than the use of natural language based queries when searching for code examples.

APIs have become the major mechanism of reusing source code. Before benefiting from reuse, developers have to learn how to use APIs. This thesis expands on the body of work on API learning by investigating the causes of the difficulties developers encounter when working with unfamiliar APIs, and by designing and evaluating new programming tools to facilitate the API learning process. A major distinction of this thesis is our use of the structural relationships between API types and methods to generate recommendations on how to use an API. Previous efforts on tools for API learning have predominantly focused and relied on the existence of a large corpus of source code examples to generate recommendations. However, there are situations where a corpus of source code examples for an API may not be available: for instance, new APIs, and less frequently used parts of existing APIs typically do not have a repository of examples. In the absence of examples, tools that need a corpus to generate recommendations will not be helpful. The contributions of this dissertation therefore complement previous tools that rely solely on the existence of code examples; this dissertation also demonstrates that in the absence of a corpus of

code examples, we can leverage the relationships between API elements to provide a measure of support to developers learning to use APIs.

This dissertation is one small piece of the research effort to understand and support API learning, but there are other areas on API learning and usability worth investigating. The contributions of this thesis are based on observing programmers working with a strongly typed programming language: an interesting venue of future work would be to investigate the usability challenges of APIs based on untyped languages such as Python or JavaScript. It would also be worthwhile to investigate the extent to which the approaches proposed in this thesis are applicable to untyped languages. In our work with APIs, we focused predominantly on code libraries; we did not look at code frameworks.[1] Studies to investigate the cause of the challenges developers encounter when learning how to extend code frameworks will no doubt reveal other challenges not observed in our work, and may also provide avenues of interesting research and new programming tools. We also observed through our study that integrating code examples found on the Web, or code repositories, into a developer's project is difficult, particularly for novice API learners. However, tool support for customizing and integrating code examples into a developer's project is

[1] *Inversion of Control* — the decision as to when an API method should be called — is typically used to differentiate between *code frameworks* and *code libraries*. Code libraries allow the reuse of code through API types and methods, but the developer decides when a method should be called. A framework allows the reuse of both code and design, but the decision as to when a method is called is made by the framework.

limited. It is our hope that some of these observations and unanswered questions would inspire future research efforts on API usability and learning.

Bibliography

[1] Sushil K. Bajracharya, Joel Ossher, and Cristina V. Lopes. Leveraging usage similarity for effective retrieval of examples in code repositories. In *Proceedings of the 18th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 157–166, 2010.

[2] T. Boren and J. Ramey. Thinking aloud: reconciling theory and practice. *IEEE Transactions on Professional Communication*, 43(3):261–278, 2000.

[3] Joel Brandt, Philip J. Guo, Joel Lewenstein, Mira Dontcheva, and Scott R. Klemmer. Two studies of opportunistic programming: interleaving web foraging, learning, and writing code. In *Proceedings of the 27th International Conference on Human factors in computing systems*, pages 1589–1598, 2009.

[4] R Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machine Studies*, 18:543–554, 1983.

[5] Marcel Bruch, Martin Monperrus, and Mira Mezini. Learning from examples to improve code completion systems. In *Proceedings of the 7th joint ESEC/FSE*, pages 213–222, 2009.

[6] Shaunak Chatterjee, Sudeep Juvekar, and Koushik Sen. Sniff: A search engine for java using free-form queries. In *Proceedings of the 12th International Conference on Fundamental Approaches to Software Engineering: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, pages 385–400, 2009.

[7] Steven Clarke. Measuring API usability. *Dr. Dobbs Journal*, pages S6 –S9, 2004.

[8] Bill Curtis. Substantiating programmer variability. In *IEEE*, volume 69 of *7*, pages 846–846, 1981.

[9] Krzysztof Cwalina and Brad Abrams. *Framework design guidelines: conventions, idioms, and patterns for reusable .Net libraries*. Addison-Wesley Professional, second edition, 2009.

[10] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 313–328, 2008.

[11] Uri Dekel and James D. Herbsleb. Improving API documentation usability with knowledge pushing. In *Proceedings of the 31st International Conference on Software Engineering*, pages 320–330, 2009.

[12] Ekwa Duala-Ekoko and Martin P. Robillard. The information gathering strategies of API learners. Technical report, TR-2010.6, School of Computer Science, McGill University, 2010.

[13] Ekwa Duala-Ekoko and Martin P. Robillard. Using structure-based recommendations to facilitate discoverability in APIs. In *Proceedings of the 25th European Conference on Object-Oriented Programming*, pages 79–104, 2011.

[14] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for software engineering research. In *Guide to Advanced Empirical Software Engineering*, pages 285–311. Springer London, 2008.

[15] 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, 2007.

[16] G. W. Furnas, T. K. Landauer, L. M. Gomez, and S. T. Dumais. The vocabulary problem in human-system communication. *Communications of the ACM*, 30:964–971, November 1987.

[17] Mark Grechanik, Chen Fu, Qing Xie, Collin McMillan, Denys Poshyvanyk, and Chad Cumby. A search engine for finding highly relevant applications. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 475–484, 2010.

[18] Scott Henninger. Using iterative refinement to find reusable software. *IEEE Software*, 11:48–59, September 1994.

[19] Abbas Heydarnoori. *Supporting Framework Use via Automatically Extracted Concept-Implementation Templates*. PhD thesis, School of Computer Science, University of Waterloo, 2009.

[20] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings*

of the 31st International Conference on Software Engineering, pages 232–242, 2009.

[21] Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the 27th International conference on Software Engineering*, pages 117–125, 2005.

[22] Oliver Hummel, Werner Janjic, and Colin Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25:45–52, 2008.

[23] Juanjuan Jiang, Johannes Koskinen, Anna Ruokonen, and Tarja Systa. Constructing usage scenarios for API redocumentation. In *Proceedings of the 15th International Conference on Program Comprehension*, pages 259–264, 2007.

[24] Marian Jureczko and Lech Madeyski. Towards identifying software project clusters with regard to defect prediction. In *Proceedings of the 6th International Conference on Predictive Models in Software Engineering*, pages 1–9, 2010.

[25] Damien Katz. Error codes or exceptions? Why is reliable software so hard? April 2006.

[26] Jinhan Kim, Sanghoon Lee, Seung won Hwang, and Sunghun Kim. Adding examples into java documents. In *Proceedings of the International Conference on Automated Software Engineering*, pages 540–544, 2009.

[27] Andrew J. Ko, Robert DeLine, and Gina Venolia. Information needs in collocated software development teams. In *Proceedings of the 29th International Conference on Software Engineering*, pages 344–353, 2007.

[28] Andrew J. Ko, Brad A. Myers, and Htet Htet Aung. Six learning barriers in end-user programming systems. In *Proc. of Visual Languages and Human Centric Computing*, pages 199–206, 2004.

[29] Otávio Augusto Lazzarini Lemos, Sushil Krishna Bajracharya, and Joel Ossher. Codegenie: a tool for test-driven source code search. In *Companion to the 22nd OOPSLA*, pages 917–918, 2007.

[30] Fan Long, Xi Wang, and Yang Cai. Api hyperlinking via structural overlap. 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*, ESEC/FSE '09, pages 203–212, 2009.

[31] David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proceedings of the International conference on Programming language design and implementation*, pages 48–61, 2005.

[32] Girish Maskeri, Santonu Sarkar, and Kenneth Heafield. Mining business topics in source code using latent dirichlet allocation. In *Proceedings of the 1st India software engineering conference*, pages 113–120, 2008.

[33] Marius Nita and David Notkin. Using twinning to adapt programs to alternative APIs. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, pages 205–214, 2010.

[34] Masaru Ohba and Katsuhiko Gondow. Toward mining "concept keywords" from identifiers in large software projects. In *Proceedings of the 2005 international workshop on Mining software repositories*, pages 1–5, 2005.

[35] N Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[36] Peter Pirolli and Stuart K. Card. Information foraging. *Psychological Review*, 106:643–675, 1999.

[37] Denys Poshyvanyk, Andrian Marcus, and Yubo Dong. JIRiSS - an Eclipse plug-in for source code exploration. In *Proceedings of the 14th IEEE International Conference on Program Comprehension*, pages 252–255, 2006.

[38] R. Robbes and M. Lanza. How program history can improve code completion. In *Proc. of the 23rd Conference on Automated Software Eng.*, pages 317–326, 2008.

[39] Martin P. Robillard. Automatic generation of suggestions for program investigation. In *Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 11–20, 2005.

[40] Martin P. Robillard. What makes APIs hard to learn? Answers from developers. *IEEE Software*, 26(6):27–34, 2009.

[41] Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. In *Proceedings of the 21st OOPSLA*, pages 413–430, 2006.

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

[43] Zachary M. Saul, Vladimir Filkov, Premkumar Devanbu, and Christian Bird. Recommending random walks. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 15–24, 2007.

[44] David Shepherd, Zachary P. Fry, Emily Hill, Lori Pollock, and K. Vijay-Shanker. Using natural language program analysis to locate and understand action-oriented concerns. In *Proceedings of the 6th international conference on Aspect-oriented software development*, pages 212–224, 2007.

[45] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. Asking and answering questions during a programming change task. *IEEE Transactions on Software Engineering*, 34(4):434–451, 2008.

[46] Joel Spolsky. Exceptions. *http://www.joelonsoftware.com/items/2003/10/13.html*, October 2003.

[47] Margaret-Anne Storey. Theories, methods and tools in program comprehension: Past, present and future. In *Proceedings of the 13th Workshop on Program Comprehension*, pages 181–191, 2005.

[48] Jeffrey Stylos. *Making APIs More Usable with Improved API Designs, Documentation and Tools*. PhD thesis, School of Computer Science, Carnegie Mellon University, 2009.

[49] Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proceedings of the 29th International Conference on Software Engineering*, pages 529–539, 2007.

[50] Jeffrey Stylos and Brad A. Myers. Mica: A web-search tool for finding API components and examples. In *Proc. of the Visual Languages and Human-Centric Computing*, pages 195–202, 2006.

[51] Jeffrey Stylos and Brad A. Myers. The implications of method placement on API learnability. In *Proc. of the 16th International Symposium on Foundations of Software Eng.*, pages 105–112, 2008.

[52] Jeffrey Stylos, Brad A. Myers, and Zizhuang Yang. Jadeite: improving API documentation using usage information. In *Extended abstracts on Human factors in computing systems*, pages 4429–4434, 2009.

[53] Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the Web. In *Proceedings of the 22nd International conference on Automated software Engineering*, pages 204–213, 2007.

[54] Tao Xie and Jian Pei. MAPO: mining API usages from open source repositories. In *Proceedings of the workshop on Mining software repositories*, pages 54–57, 2006.

[55] Yunwen Ye, Gerhard Fischer, and Brent Reeves. Integrating active information delivery and reuse repository systems. In *Proceedings of the 8th International Symposium on Foundations of software Engineering*, pages 60–68, 2000.

[56] Robert K. Yin. *Case Study Research: Design and Methods*. Sage, second edition, 2003.

Appendix A — Supplementary Data from the Evaluation of Introspector

Table 6–1: Seeds for the Central-type scheme, for each task.

Task ID	Seeds
1	IJavaProject
2	ASTParser
3	Cell
4	Picture
5	Query
6	AsynTwitter
7	CacheStatistics
8	DocumentConverter
9	Message
10	RosterEntry

Table 6–2: Keywords from the task descriptions that were used to generate seeds for the keyword-based scheme.

Task ID	Keywords	Seeds
1	"java project"	IJavaProject
2	"AST"	AST
3	"Cell", "Excel"	Cell
4	"image", "Excel"	Picture
5	"tweets", "search"	Query, Tweet
6	"status"	Status
7	"cache statistics"	Cache, CacheStatistics
8	—	—
9	"message"	Message
10	"presence"	Presence

Table 6–3: Seeds for the Random scheme, for each task.

Task ID	Seeds (types needed for each task)
1	IJavaProject, IProject, JavaCore, IWorkspace, IWorkspaceRoot
2	ASTParser, ICompilationUnit, AST, CompilationUnit
3	Cell, Row, Sheet, WorkBook, CreationHelper, HSSFWorkBook
4	Picture, Drawing, ClientAnchor, Sheet, WorkBook, HSSFWorkBook
5	Query, QueryResult, Twitter, TwitterFactory, Tweet
6	AsynTwitter, TwitterListener, TwitterAdapter, Status, AsynTwitter-Factory
7	CacheStatistics, Cache, CacheFactory, CacheManager
8	DocumentConverter, OpenOfficeConnection, OpenOfficeDocument-Converter, SocketOpenOfficeConnection
9	Message, Chat, ChatManager, XMPPConnection
10	RosterEntry, XMPPConnection, Roster, Presence

Table 6–4: Search queries and options for the natural language query evaluation. These search queries were executed on Koders and Google Code Search.

Task ID	Search Query	Search Options
1	"create java project"	Java
2	"create AST"	Java
3	"create cell in excel"	java
4	"add image to excel"	Java
5	"search tweets"	Java
6	"update status asynchronously"	Java
7	"obtain cache statistics"	Java
8	"convert from word to pdf"	Java
9	"send text message"	Java
10	"check presence of users"	Java

Appendix B — Supplemental Figures

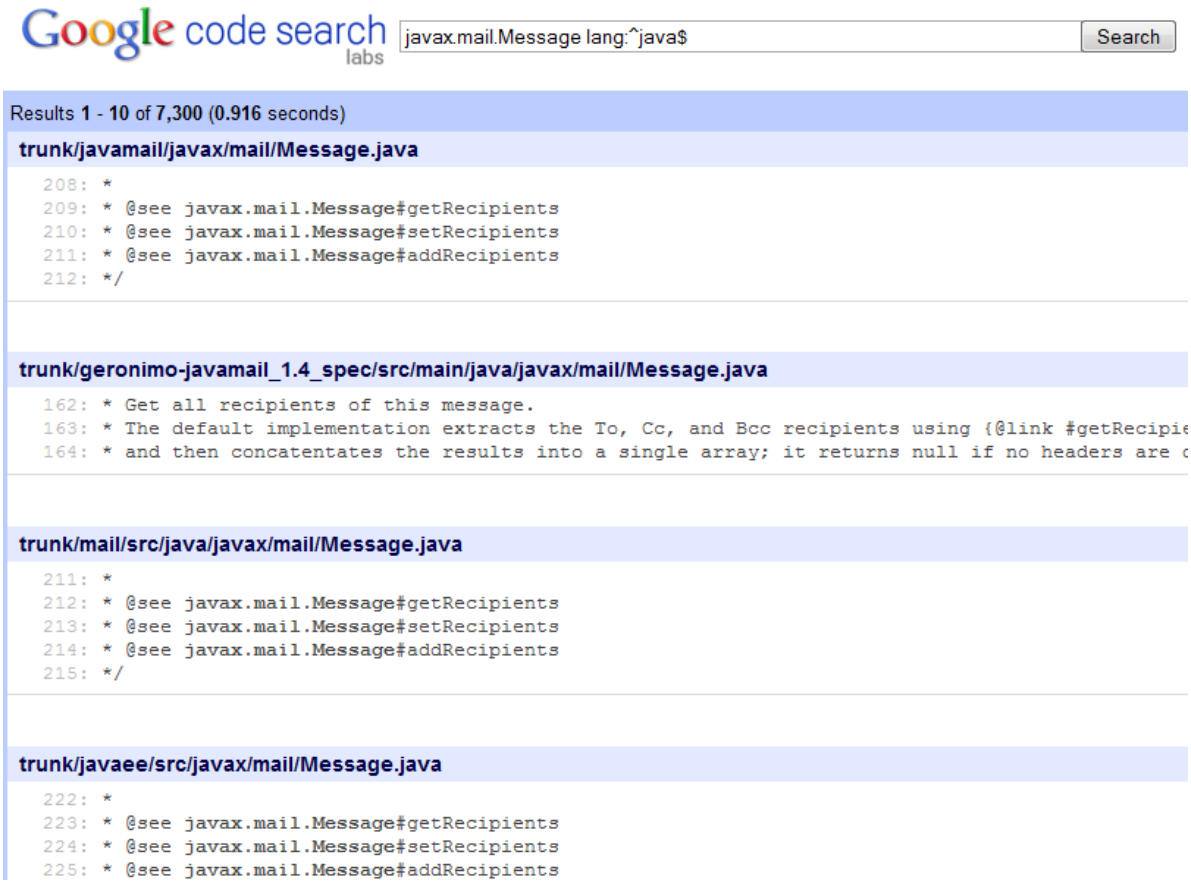
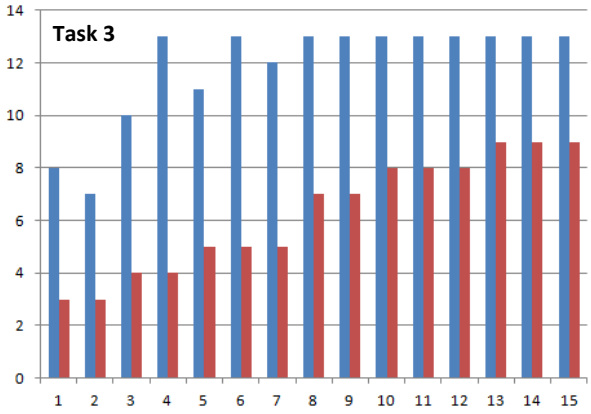


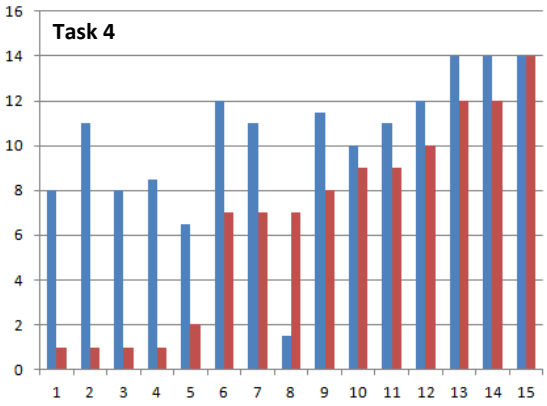
Figure 6–1: An example of using the seed `javax.mail.Message` to search for code example on how to send an email message using Google Code Search (October 15, 2011). The top search results all point to different source files of the `javax.mail.Message` class.

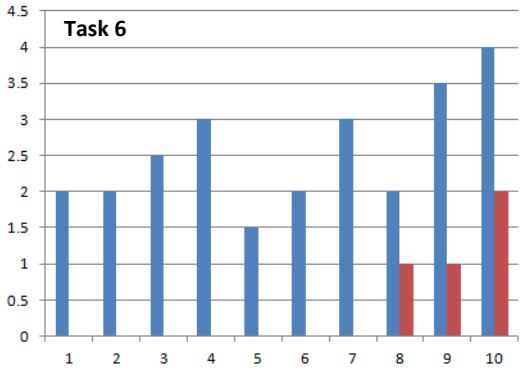
Appendix C — Supplemental Data for Symmetry Analysis

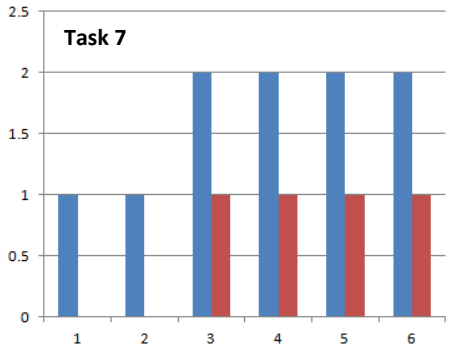
A summary of the pairwise overlap between the expanded sets of the seeds for Tasks 2, 3, 4, 6, 7, 8 and 9. The blue bars represent the average size of the expanded set for a given pair, and the red bars represent the overlap between the expanded set for a given pair.

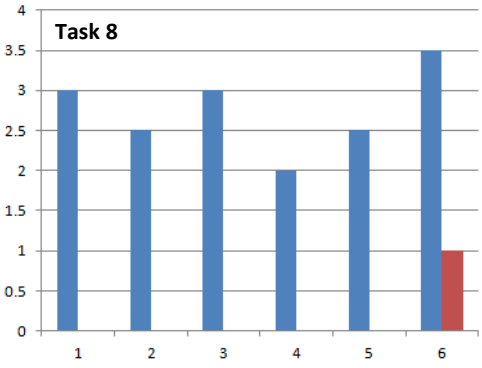

Task 2
14
12
10
8
6
4
2
0
1
2
3
4
5
6

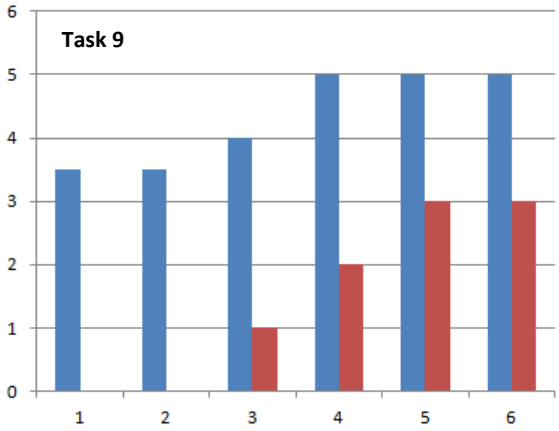

Task 3
14
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


Task 4
16
14
12
10
8
6
4
2
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15


Task 6
4.5
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6
7
8
9
10


Task 7
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6


Task 8
4
3.5
3
2.5
2
1.5
1
0.5
0
1
2
3
4
5
6


Task 9
6
5
4
3
2
1
0
1
2
3
4
5
6

Appendix D — Ethics Board Approval of User Studies

McGill University Research Ethics Board Approval of User Studies REB File #:6-0605 — Empirical Studies of Software Developers Involved in a Modification Task (see next page).