Using Structure-Based Recommendations to Facilitate Discoverability in APIs

Ekwa Duala-Ekoko and Martin P. Robillard

School of Computer Science, McGill University
Montréal, Québec, Canada
{ekwa, martin}@cs.mcgill.ca

Abstract. Empirical evidence indicates that developers face significant hurdles when the API elements necessary to implement a task are not accessible from the types they are working with. We propose an approach that leverages the structural relationships between API elements to make API methods or types not accessible from a given API type more discoverable. We implemented our approach as an extension to the content assist feature of the Eclipse IDE, in a tool called API Explorer. API Explorer facilitates discoverability in APIs by recommending methods or types, which although not directly reachable from the type a developer is currently working with, may be relevant to solving a programming task. In a case study evaluation, participants experienced little difficulty selecting relevant API elements from the recommendations made by API Explorer, and found the assistance provided by API Explorer helpful in surmounting discoverability hurdles in multiple tasks and various contexts. The results provide evidence that relevant API elements not accessible from the type a developer is working with could be efficiently located through guidance based on structural relationships.

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* as reuse holds promise of increased productivity. Learning how to use APIs, however, presents several challenges to both novice and expert developers [4,12,15,16]. One such challenge, referred to as the *discoverability problem*, highlights the difficulty faced by a developer looking for the types and methods of an API necessary to implement a programming task [12,16]. 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 [16]. 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

```
Properties props = System.getProperties();
props.put("mail.smtp.host", "localhost");
Session ses = Session.getInstance(props);
Message message  =  new MimeMessage(ses);
//set other attributes of Message

Transport.send(message);
```

JavaMail API

```
SimpleEmail message = new SimpleEmail();
message.setHostName("localhost");
//set other attributes of email

message.send();
```

Apache Wrapper for JavaMail

Fig. 1. Apache Commons wrapper for the JavaMail API placed the "send" method on the main-type, and simplified the process of creating an email message object by providing a default constructor.

at combining multiple objects when relevant API methods and types were accessible from the main-type [16]. 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 [4].

A potential solution to improving discoverability in APIs is to restructure an API to make the methods and types essential to the use of a main-type discoverable. For instance, moving the "send" method from `EmailTransport` to `EmailMessage` and providing constructors for object construction instead of the Factory pattern would improve discoverability. However, such a restructuring may not always be beneficial as it could negatively impact other desirable features of an API such as its performance and evolvability, and the client code may also become broken. A second solution to the discoverability problem is to provide an API wrapper. For instance, we are aware of over six API wrappers for the JavaMail[1] API, all aimed at simplifying the process of composing and delivering an email message. One such wrapper, provided by the Apache Commons project, underscores the discoverability issues with the JavaMail API by placing the "send" method on the main-type, and by simplifying object construction through the use of a constructor (see Figure 1). The use of API wrappers to resolve discoverabilty problems is promising but may be expensive, and introduces maintenance and versioning problems. Furthermore, developing wrappers introduces the risk of unwittingly altering the behavior of the original API.

In this paper, we propose a novel and an inexpensive approach for improving the discoverability of API elements. 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(Email-`

[1] `http://java.sun.com/products/javamail`

`Message)` 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. These can be accomplished without trading off other desirable API features to make elements discoverable, or the need to create and maintain API wrappers.

To investigate our intuition, we built a recommendation system, called *API Explorer*,[2] which makes use of a special-purpose dependency graph for APIs to provide recommendations based on the structural context in which assistance is requested. We implemented API Explorer as a novel extension of the content assist feature of the Eclipse IDE. Content assist in Eclipse, or IntelliSense as it is called in Microsoft Visual Studio, is limited to showing only the methods available on the object on which it is invoked. API Explorer extends content assist with support for recommending relevant methods on other objects, locating API elements relevant to the use of a method or type, and also providing support for combining the recommended elements. We evaluated API Explorer through a multiple-case study in which eight participants were asked to complete the same four programming tasks using four different real-world APIs, each task presenting multiple discoverability challenges. The results of the study was consistent across the participants and the tasks, and show that API Explorer is effective in assisting a developer discover relevant helper-types not accessible from a main-type. The results also show 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 improve discoverability in APIs. We make the following contributions:

- We present an approach that uses the structural relationships between API elements to make discoverable helper-types not accessible from a main-type.
- We provide API Explorer, a publicly available plugin for Eclipse that embodies our approach. API Explorer is the first tool, to our knowledge, that can recommend relevant API methods on other objects through the content assist feature of an IDE.
- We present a detailed analysis of data from 32 programming sessions of participants using API Explorer with real-world APIs, showing how our approach is effective in helping developers discover helper-types not reachable from a main-type, and helping us understand the contexts in which the approach would not be effective.

We continue in the next section with an example scenario that highlights typical discoverability hurdles observed in previous studies on API usability.

2 Motivation

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

[2] API Explorer is available at: `www.cs.mcgill.ca/~swevo/explorer`

an email message. The developer then proceeds by attempting to construct an object of type `Message` from its default constructor[3] and encounters the first discoverability hurdle: `Message` is an abstract class. Creating an object of type `Message` requires three helper-types (`MimeMessage`, `Session`, and `Properties`), none of which are directly accessible from `Message` (i.e., these helper-types are not referenced or reachable from any of the public members of `Message`). Eventually, after spending some time going through the documentation, or code examples on the Web, the developer would locate all the types necessary to construct a `Message` object, and also the information on how these types should be combined. Once `Message` is created and all the necessary attributes are set, the developer then proceeds to send the email and encounters the second discoverability hurdle: there is no method on the `Message` object that provides the "send" functionality. The developer must therefore spend more time looking for a helper-type with a method that could be used to send the `Message` object. The code completion feature of the IDE is not helpful because it can only display the methods available on `Message` and provides no easy way to discover the existence of a helper-type with a send method. Also, the traditional search tools that come with IDEs do not provide direct support to locate multiple helper-types from a main-type. A developer would have to combine the results from multiple tools (e.g., the type hierarchy and reference search tools) and filter out the search results before potentially finding the relevant helper-types.

This scenario describes a conceptually simple task but highlights discoverability hurdles commonly faced by developers in practice: "...in real world APIs like Java's JDK and Microsoft's .NET, it frequently seems to be the case that the classes [helper-types] one needs are not referenced by the classes [main-type] with which one starts..." [16, p.2]. Thirteen out of twenty participants in a separate exploratory study we conducted to investigate the challenges developers encounter when learning to use APIs experienced some difficulty locating the helper-types relevant to implementing a programming task [2]. We observed that the participants relied on imperfect proxies such as domain knowledge or their expectation of how an API should be structured when looking for helper-types not referenced by a main-type. These attributes are often not consistent across different APIs, and may be misleading, resulting in unsuccessful searches and wasted efforts. This observation raises two research challenges:

- How can we assist developers in efficiently discovering helper-types not accessible from a main-type?
- How can we assist developers in the process of combining these related types to implement a task?

We hypothesize that structural relationships between API elements can be leveraged as beacons to assist developers locate helper-types not accessible from a main-type, and in combining these related types. In the next section, we discuss how API Explorer, a tool developed to investigate our hypothesis, could

[3] Three separate studies observed that most developers, both novice and experts alike, begin object construction by attempting to use the default constructor [4,15,16].

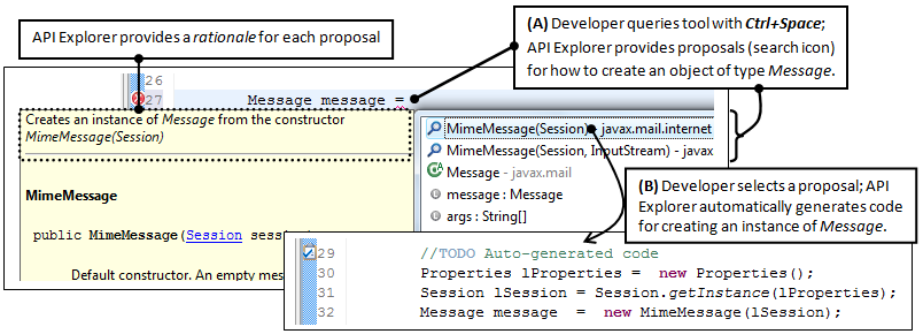

Fig. 2. API Explorer shows the developer the types required to construct an instance of `Message` and generates code which illustrate how to combine these types.

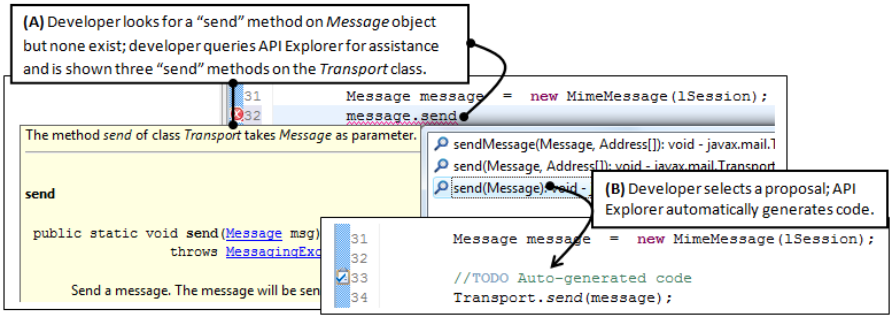

Fig. 3. API Explorer recommends three send methods on the `Transport` class which can be used for sending an email `Message` object.

have assisted the developer quickly surmount the hurdles encountered above. We present the heuristics and algorithms enabling API Explorer in Section 4.

3 API Explorer

API Explorer generates recommendations that would assist a developer discover helper-types not accessible from a main-type based on the structural context in which help is requested.

Faced with an object construction hurdle, a developer would query API Explorer for assistance by invoking content assist after the assignment operator. For instance, in the example above, 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 2(A)). API Explorer can provide assistance for creating objects from constructors, subtypes, factory methods, public methods, or static methods. 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 2). Once the developer makes a selection, API Explorer automatically expands the selected

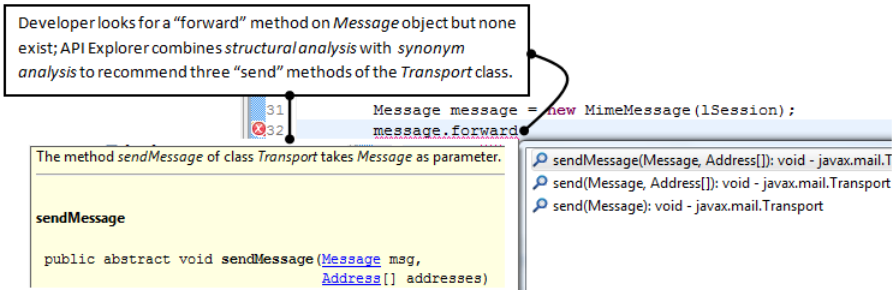

Fig. 4. API Explorer combines structural analysis with synonym analysis to recommends three send methods of the `Transport` class when a developer looks for a "forward" method on the `Message` object.

recommendation into code that shows how the elements needed to create an object of type `Message` should be combined (see Figure 2(B)).

API Explorer provides three options for discovering relevant methods on helper-types. With the first option, API Explorer display methods that take an object of type `Message` together with the public methods declared on `Message` through the code completion feature of the Eclipse IDE. To minimize confusion, we differentiated the recommendations of API Explorer using a different icon and appended them after the methods of the main-type. Thus, a developer browsing through the methods of `Message` using this *enhanced* code completion feature (i.e., code completion in Eclipse with API Explorer installed) will also come across the method `Transport.send(Message)`. The second option requires the developer to request explicit assistance from API Explorer. For instance, the developer would enter "message.send", where "message" is an object of type `Message`, and API Explorer would recommend methods named "send" on other types that take an object of type `Message` as parameter. In this case, API Explorer recommended three "send" methods of the `Transport` class, and generated code of how these types should be combined once a recommendation is selected by the developer (see Figure 3). The third option handles cases where a developer might 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 with synonym analysis to recommend methods with a name similar to what the developer is looking for (see Figure 4). We continue in the next section with the algorithms underlying API Explorer.

4 API Graph and Recommendation Algorithms

API Explorer relies on a specialized dependency graph for APIs, called *API Exploration Graph*, and incorporates algorithms that use the information contained in the graph to generate recommendations based on the structural context.

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 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.

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. 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, from knowing that `MimeMessage` is-subtype-of `Message`, and that `Message` is-parameter-of `Transport.send`, 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 "send" 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 1.1. A simplified version of the JavaMail API

Listing 1.1 shows a simplified version of the JavaMail API, and Figure 5 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, and automatically links it to the XGraph of APIs referencing types of the JRE, as in Figure 5. We generate XGraphs from the binaries of APIs using the Javaassist[4] byte code analysis library, and it takes less than one minute to

[4] www.javassist.org

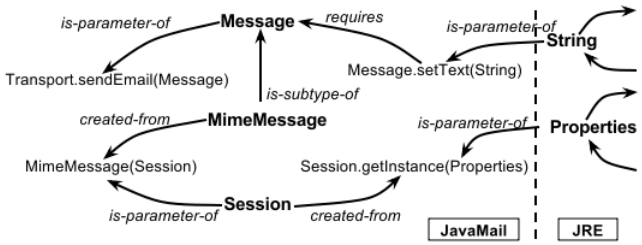

Fig. 5. The XGraph of the simplified JavaMail API in Listing 1.1. The nodes in boldface represent API types; the other nodes represent API methods, including constructors, and the edges represent relationships between the nodes.

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 Sections 4.2 through 4.5, and use the sample API in Listing 1.1 and its XGraph to present examples of the algorithms.

4.2 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)* method (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* method 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 4.5 that recursively looks for the parameters and dependencies of a selected recommendation, and generate code showing how to combine them.

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.

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
```

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 5, the participants of our case study evaluation demonstrated little difficulty selecting a relevant recommendation when presented with multiple options.

4.3 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 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.

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
```

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[5]

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

dictionary to generate the synonym sets. We also augmented WordNet with common words such as "insert", "put", and "append" often used interchangeable 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* method 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 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`.

4.4 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 [2]. Unfortunately, direct support for such an inquiry is unavailable. 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 a input an array of API types and the XGraph, and outputs the relationships between the types, if any. Given a single API type *typeArray[0]*, the algorithm can locate other API types related to it (lines 3 to 4). The *xgraph.getRelatedTypes(typeArray[0])* method (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, that 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.

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
```

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. The algorithm returns within a second of the query, revealing 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.

4.5 Code Generation Algorithm

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

4.6 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, 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 Section 5.

5 Evaluation

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 contexts in which API Explorer may be 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.

5.1 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" [20, 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" [3, 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, ICSE '05 [6].

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 [3,20]. 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 [20, 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?

A. 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 [2,16].

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)[6] 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[7] 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.

PDF Task: we asked the participants to use the PDFBox[8] 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.

B. 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

[6] jaxp.dev.java.net
[7] jfree.org/jfreechart
[8] pdfbox.apache.org

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.

C. 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 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.

5.2 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 from the study focused on the questions the study was designed to answer. 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. Due to space restrictions, we present a detailed outline of the observations from the Email task, and summarize the observations from the other tasks in Table 1[9].

[9] A detailed outline of how the participants approached each task, and how they used API Explorer is available at: `www.cs.mcgill.ca/~swevo/explorer/evaluation/`

Table 1. A summary of the results 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

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.

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 present a summary of the observations from the other tasks in Table 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

Table 2. A summary of all the 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**

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 2.

The third column (# of usage episodes) of Table 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 Ex-

plorer 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`, 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.
Our approach will not be helpful if the relationships between API elements can only be determined at runtime. The last column of Table 2 shows other situations in which our approach was not helpful: these involve the synonym analysis and relationship exploration features of API Explorer.

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 match, 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 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 can not 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.

5.3 Summary

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 [4,16]. 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.

5.4 Threats to validity

As indicated in Section 5.1, 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 include 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.

6 Related work

Improving Code Completion Tools. Previous work on code completion systems focused either on re-ordering the list of methods accessible on a given type, or on predicting the method of an API type most likely to be called next in a given context. Robbes et al. modeled the change history of systems as atomic operations and used this history to predict the method of an object most likely to be called next in a given context [11]. Bruch et al. used example code in code repositories to improve the ordering of the list of suggested methods [1]. These previous works can only suggest or re-order elements accessible on the object on which code completion is requested. API Explorer is a novel extension of

code completion, capable of suggesting relevant methods on other objects, and providing support for locating elements relevant to the use of a given API type.

Other IDE Tools. IDEs provide tools that could be used to search for places where an API type is referenced, and potentially, locate elements not structurally accessible from a given type. These tools are suited for code comprehension, not API exploration, and there is no evidence that a participant from any of the previous studies even attempted to use these tools when learning how to use APIs [2,4,15,16]. On the contrary, observations from previous studies indicated that the content assist feature is the most widely used when exploring APIs [15,16].

Documentation Improvement Tools. Some efforts on facilitating the discovery of API elements have focused on improving the API documentation. Kim et al. proposed eXoaDocs [9], a tool that integrates code snippets mined from source code search engines into the Java API documentation, making factory methods or subtypes necessary for object construction discoverable. Jadeite [17] uses usage statistics of the types and methods of an API from code examples found on the Web to help developers find commonly used API elements from the documentation, and also integrates code snippets on how to construct objects of API types in the documentation. Jadeite also has a concept, similar to our method recommendation feature, known as a "placeholder" which allows a developer to annotate the documentation with the name of a method expected to be located on a given API type, and to link the "placeholder" to an actual method of the API that should be used instead. API Explorer, in contrast, automatically identifies relevant methods on other API types using the structural relationships between API elements, and presents this information through the code completion feature of the IDE. Furthermore, Jadeite and eXoaDocs require large collections of example usages of APIs. API Explorer, in contrast, is lightweight, leveraging the structural relationships between API elements, not collections of code examples, to make API elements discoverable.

Example Recommender Tools. These tools leverage the proliferation of code examples on the Web and open-source repositories to make learning how to use APIs easier; they differ in the approach used to retrieve code examples and in the kind of support afforded to API users. CodeBroker [19] uses comments and method signatures written by the programmer to recommend methods from code repositories. Strathcona [6] uses the structural context of the code under development such as the parent class of the framework type being extended, and the signature of API methods to retrieve relevant code examples from a repository. MAPO uses pattern mining techniques to identify code snippets and method call sequence that show how to use a given API method [21]. Prospector [10], ParseWeb [18], and XSnippet [13] take queries of the form "`source-type` → `destination-type`", and recommend code examples that show how to get the `destination-type` from the `source-type`. Jiang et al. [8], Salah [14], and Heydarnoori [5] proposed tools which use dynamic analysis of the interaction between sample applications and APIs to identify valid usage scenarios and valid

call sequence of API methods. Code Conjurer [7] uses 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 combine the use of code examples with the structural relationships between API elements such as return types and method parameters to identify relevant method call sequences that link a `source-type` to a `destination-type`. However, the support provided by Prospector and XSnippet is limited to object construction only, and for both tools to work, a developer is expected to provide both a `source-type` and a `destination-type`. As observed in our case study, and also in a previous API usability study [2], a developer may not even be aware of the necessary `destination-type`. With API Explorer, developers can obtain object construction support with only a `source-type`. Furthermore, API Explorer extends these works by using structural relationships to make relevant API methods not accessible from an API type discoverable.

7 Conclusion

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. This paper addresses one of the challenges developers face when learning a new API: discovering relevant helper-types not accessible from a main-type they are working with. We have proposed an approach that leverages structural relationships 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 was consistent across the participants and the tasks: API Explorer effectively assisted the participants to locate 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 provide initial 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.

References

1. 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.
2. 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.
3. 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.

4. Brian Ellis, Jeffrey Stylos, and Brad Myers. The factory pattern in API design: A usability evaluation. In *Proc. of the 29th International Conf. on Software Eng.*, pages 302–312, 2007.
5. Abbas Heydarnoori. *Supporting Framework Use via Automatically Extracted Concept-Implementation Templates*. PhD thesis, School of Computer Science, University of Waterloo, 2009.
6. Reid Holmes and Gail C. Murphy. Using structural context to recommend source code examples. In *Proc. of the 27th International conf. on Software Eng.*, pages 117–125, 2005.
7. Oliver Hummel, Werner Janjic, and Colin Atkinson. Code conjurer: Pulling reusable software out of thin air. *IEEE Software*, 25:45–52, 2008.
8. Juanjuan Jiang, Johannes Koskinen, Anna Ruokonen, and Tarja Systa. Constructing usage scenarios for API redocumentation. In *Proc. of the 15th International Conf. on Program Comprehension*, pages 259–264, 2007.
9. Jinhan Kim, Sanghoon Lee, Seung won Hwang, and Sunghun Kim. Adding examples into java documents. In *Proc. of the International Conf. on Automated Software Eng.*, pages 540–544, 2009.
10. David Mandelin, Lin Xu, Rastislav Bodík, and Doug Kimelman. Jungloid mining: helping to navigate the API jungle. In *Proc. of the International conf. on Programming language design and implementation*, pages 48–61, 2005.
11. 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.
12. Martin P. Robillard and Robert DeLine. A field study of API learning obstacles. *Empirical Software Engineering — To appear*, 2011.
13. Naiyana Sahavechaphan and Kajal Claypool. Xsnippet: mining for sample code. In *Proceedings of the 21st OOPSLA*, pages 413–430, 2006.
14. 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 *Proc. of the 21st International Conf. on Software Maintenance*, pages 155–164, 2005.
15. Jeffrey Stylos and Steven Clarke. Usability implications of requiring parameters in objects' constructors. In *Proc. of the 29th International Conf. on Software Eng.*, pages 529–539, 2007.
16. 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.
17. 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.
18. Suresh Thummalapenta and Tao Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proc. of the 22nd International conf. on Automated software Eng.*, pages 204–213, 2007.
19. Yunwen Ye, Gerhard Fischer, and Brent Reeves. Integrating active information delivery and reuse repository systems. In *Proc. of the 8th International Symposium on Foundations of software Eng.*, pages 60–68, 2000.
20. Robert K. Yin. *Case Study Research: Design and Methods*. Sage, second edition, 2003.
21. Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *ECOOP 2009*, pages 318–343. 2009.