Recovering Traceability Links between an API and Its Learning Resources

Barthélémy Dagenais and Martin P. Robillard
School of Computer Science
McGill University
Montréal, QC, Canada
{*bart,martin*}*@cs.mgill.ca*

***Abstract*—Large frameworks and libraries require extensive developer learning resources, such as documentation and mailing lists, to be useful. Maintaining these learning resources is challenging partly because they are not explicitly linked to the frameworks' API, and changes in the API are not reflected in the learning resources. Automatically recovering traceability links between an API and learning resources is notoriously difficult due to the inherent ambiguity of unstructured natural language. Code elements mentioned in documents are rarely fully qualified, so readers need to understand the context in which a code element is mentioned. We propose a technique that identifies code-like terms in documents and links these terms to specific code elements in an API, such as methods. In an evaluation study with four open source systems, we found that our technique had an average recall and precision of 96%.**

I. Introduction

Reusable assets such as frameworks and libraries are generally provided with resources to help software developers learn how to use these assets. Learning resources for developers typically include tutorials, reference manuals, and various mailing lists and forums. Writing, moderating, and maintaining such resources require a considerable effort [1]. For example, the Spring Framework reference manual [2] has more than 200 000 words and the Hibernate framework forum [3] has more than 180 000 messages. Even a smaller library such as HttpComponents [4], which contains 619 classes, has a developer documentation of 28 900 words divided in two manuals, and a mailing list that includes more than 8 500 messages.

Ideally, developer learning resources should be both extensive (covering all parts of the framework's API) and detailed (explaining many low-level programming patterns), while being continually maintained to keep up with feature additions, API usability problems, and community requests. For instance, we observed in a recent study on developer documentation that when a question is repeatedly asked on a mailing list, framework contributors see this as an indication that the documentation needs to be clarified [1]. In cases where there are multiple support channels (chat, mailing lists, forums) and multiple contributors operating in different time zones, the contributors are often unaware that the same code element (e.g., function) is the root cause of several questions. Specifically, because the support channels and the documentation are not explicitly linked to the API, it is difficult for a contributor to determine which code elements cause the most problems and need to be further explained in the documentation.

The main challenge in linking code elements with existing learning resources comes from the inherent ambiguity of unstructured natural languages. For example, the user guide of the Joda Time library [5] mentions in the middle of the *Date fields* section: "... such as year() or monthOfYear()". Although it is clear from this sentence fragment that a method named `year` is mentioned, there are 11 classes, not all in the same hierarchy, that declare a `year` method in Joda Time. The *code-like* term `year` could also refer to a method declared in an external library frequently used with Joda Time (e.g., Java Standard Library). In this particular case, a human reader would know that the term refers to `DateTime.year()` because the class `DateTime` is mentioned at the beginning of the section, i.e., in the *context* of the method `year()`. However, a simple mechanical match based on the method name and ignoring the context of the term, would fail. In fact, in the four open source projects we studied (Section IV), we found that a mechanical match **would have failed** to find the correct declaration of 89% of the methods mentioned in the learning resources because the methods were declared on average in 13.5 different types.

Several techniques have been previously proposed to link project artifacts. However, there is currently no technique that precisely links the documentation, the support channels and the API together at a fine level of granularity. For example, Hipikat links coarse-grained project artifacts such as code commits, emails, and bug reports based on bug numbers [6]. Bacchelli et al. devised a technique that identifies source code (e.g., code snippets) in emails and that can link classes mentioned in the email to classes declared in a codebase [7], but the technique does not work at the sub-class level of granularity.

We propose a technique that automatically analyzes the documentation and the support channel archives of an open source project and that precisely links code-like terms (e.g., `year()`) to specific code elements (e.g., `DateTime.year()`) in the API of the documented framework or library. Our technique considers the context in which a term is mentioned and applies a set of filtering heuristics to ensure that terms referring to external code elements are not spuriously linked.

978-1-4673-1067-3/12/$31.00

ICSE 2012, Zurich, Switzerland

We implemented our technique in a tool called RecoDoc and applied it on four open source systems. We found that our technique identified on average 96% of the code-like terms (recall) and linked these terms to the correct code element 96% of the time (precision). The high accuracy of our technique will enable the development of reliable approaches that can improve the learning resources based on the relationships between these resources and the API.

Our contributions include (1) a meta-model to represent documentation, support channels, code, and their relationships, and (2) a fine-grained technique to link the contents of developer learning resources with code elements, validated on an extensive collection of artifacts from three open source programs. RecoDoc is open source and publicly available [8].

We begin by presenting a meta-model to represent the various project artifacts (Section II). Then, we describe the linking technique we devised to associate the code-like terms from the learning resources to the code elements of an API (Section III). We present the evaluation we performed on four open source projects in Section 4 and we discuss the related work in Section 5.

II. Project Artifacts Meta-Model

A variety of information is needed to understand the context in which a code-like term is mentioned. In the documentation example of Figure 1, the method `getParams` is declared in eight types in the HttpClient library. We can precisely find which method declaration is referred to if we know that:

1) `getParams` is mentioned in Section 1.1.
2) Section 1.1 is part of Section 1.
3) `HttpGet` is mentioned in Section 1, so it is in the *context* for `getParams`.
4) `HttpGet` does not declare `getParams`, but inherits it from `HttpMessage`, which declares `getParams`.

Based on our previous study on developer documentation [1] and on initial prototyping with various releases of the Spring Framework (a large and complex Java project [2]), we designed a meta-model to universally represent the documentation, support channels, and API of any open source project. We use this meta-model to understand the context in which a code-like term is mentioned. The main elements of the meta-model are described in the next paragraphs and are represented in Figure 2.

Project. A project may have different *releases* and each release is associated with a particular codebase and documentation. For example, the HttpComponents project [4] has three major releases (2.0, 3.0, and 4.0) with a corresponding codebase and documentation.

Codebase. We consider that the *API* of a project consists of *code elements* (e.g., class, method, field, parameter, XML element). RecoDoc currently parses Java codebases, XML schema files and DTD files. A code element may have

Section 1
HttpGet implements the HTTP GET request in HttpClient.

Section 1.1
Call getParams() to obtain the parameters of the get request. You can call RedirectStrategy.getRedirect() to determine the redirect location from a request.

Figure 1. Documentation Example Loosely Adapted from the HttpClient tutorial.

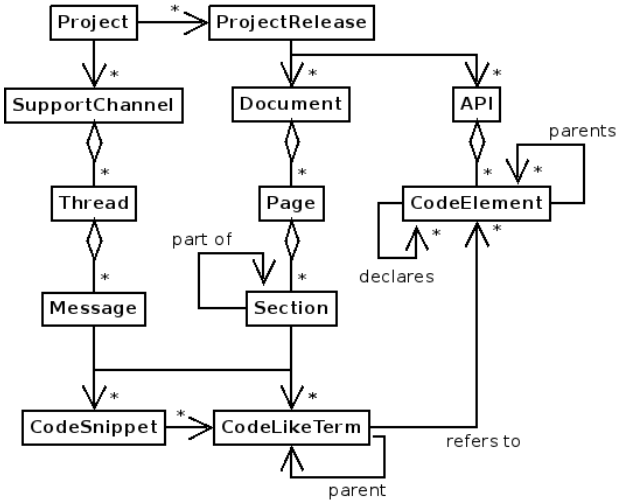

Figure 2. Documentation Meta-Model

one or more parents (e.g., a Java class implements multiple interfaces) and may declare other elements (e.g., a class declares methods). Additionally, each *kind* of code elements is internally represented by a specialized class that keeps track of its specific attributes (e.g., a `MethodCodeElement` has a list of parameters, not shown in Figure 2).

Document. The documentation of a project consists of one or more *documents*. For example, the HttpComponents project has two main documents: the HTTPClient and HTTPCore tutorials. Each document has a list of *pages* and each page has a list of *sections* (e.g., Section 1.1.2. HTTP request). A section may be part of a larger section (e.g., Section 1.1.). As we explain in Section II, we consider a documentation page to be equivalent to an HTML page and not to a printed page.

Support Channel. A project may have one or more *support channels* such as a mailing list or a forum. For example, the HttpComponents project has a mailing list, `httpclient-users`. A support channel contains a list of *support threads*, which contain a list of *messages*.

Code-like Terms and Code Snippets. Messages and documentation sections can refer to *code-like terms* and *code snippets*. A code-like term is a series of characters that matches a pattern associated with a code element kind (e.g., parentheses for functions, camel cases for types, anchors for XML elements). For example, Section 1.1 in Figure 1 con-

tains three code-like terms: `getParams`, `RedirectStrategy`, and `getRedirect`. A code-like term list, or term list, is a sequence of code-like terms. We thus consider that the term list `RedirectStrategy.getRedirect` contains two code-like terms and that the first term is the parent of the second.

A code snippet is a small region of source code that can be further divided into a list of code-like terms. For example, in a Java code snippet, all method calls would be represented by code-like terms.

Finally, a code-like term may refer to one or more code elements in the codebase. For example, the term `println` from the term list `System.out.println` might refer to all overloaded declarations of `println` in `java.io.PrintStream`.

Context. We consider that there are three levels of context that can be associated with code-like terms. The *immediate context* contains all the code-like terms in a term list. The *local context* contains all the terms in the same documentation section or the same support message. The *global context* contains all the terms in the same documentation page or in the same support thread. For example, the immediate context, the local context, and the global context for `getRedirect` in Figure 1 are respectively: {`RedirectStrategy`}, {`RedirectStrategy, getParams`}, {`RedirectStrategy, getParams, HttpGet`}. We consider that a code-like term `A` is *closer* to a term `B` than to a term `C` if `B` is in a more specific context than `C`. For example, `getRedirect` is closer to `getParams` than to `HttpGet`.

Generating Models. We generate a documentation model from a set of artifacts and recover the links between code-like terms and code elements. Figure 3 demonstrates this process.

Artifacts Collection. Our technique takes as input (1) the source code of a system, (2) the URL of the documentation index such as the table of contents of a reference manual, and (3) the URL of a support channel archive such as the first page of a forum. We then crawl the documentation and the support channel archives to download the relevant HTML pages (i.e., documentation pages, emails, forum threads). All documentation tools and archives we are aware of can produce an HTML output.

Model Generation. We use an extensible parsing infrastructure to generate the model from the project artifacts. For example, the HTML output of documentation tool DocBook [9] differs from the HTML output produced by the Maven tool [10], so we created a `MavenParser` and a `DocBookParser` that both extend a `DocumentationParser`. We parse the Java source code using the Eclipse compiler [11].

Content Classification. Once the model is generated, the parsing infrastructure classifies the content of the documentation and the support channel: it identifies the code-like terms, the code snippets, and their probable kind (e.g., class, method, XML element, Java code snippet, XML code snippet). We relied on existing techniques described in the literature [7] and in our previous work [12] to implement the content classification step. A brief description of the classification process and the evaluation of its accuracy is presented in a technical report [13].

Snippet Parsing. We further parse snippets to identify the code-like terms within them. For example, we identify all calls, declarations, and references in Java snippets. We use Partial Program Analysis (PPA) to parse Java snippets [12]. PPA accepts partial Java programs (e.g., method bodies) and produces type-resolved Abstract Syntax Trees by inferring the missing declarations.

Linking. Finally, we attempt to link the code-like terms to code elements in a specific project release. The linking process is described in Section III.

Because a code-like term not identified by the content classification step will not be considered by the linking step, our parsing infrastructure favors recall over precision.

III. Linking Technique

We define the process of matching code-like terms to code elements as a traceability link recovery process. We derived this process by studying the Spring Framework [2] learning resources and manually linking the code-like terms. The code-like terms in Spring's documentation and forum are very difficult to link and we reasoned that if our technique was accurate for Spring, it would be accurate for most Java libraries and frameworks. For example, the class hierarchy of Spring is deep (maximum depth of 8) and the framework wraps many external libraries, so numerous code-like terms actually refer to these external libraries and not to Spring.

While studying Spring's learning resources, we found that there are four major sources of ambiguity that make the link recovery process challenging:

Declaration Ambiguity. Because human readers can generally infer the precise code elements mentioned in learning resources by using the context in which the elements are mentioned, code-like terms are rarely fully qualified. For instance, method names are mentioned without their declaring type and package imports are omitted.

Overload Ambiguity. A code-like term representing a method is ambiguous if the method is overloaded and if the code-like term does not indicate the number or type of the parameter(s).

External Reference Ambiguity. Learning resources may refer to code elements declared in external libraries such as the Java Standard Library, a library used by the system, or a library commonly used by the users of the system (e.g., jUnit). A code-like term may also refer to a technical concept (e.g., HTTP) that has the same name as the code elements in the target system. We must avoid incorrectly linking a code-like term that refers to an external entity.

Language Ambiguity. We expect learning resources to contain errors made by users and documentation writers. These

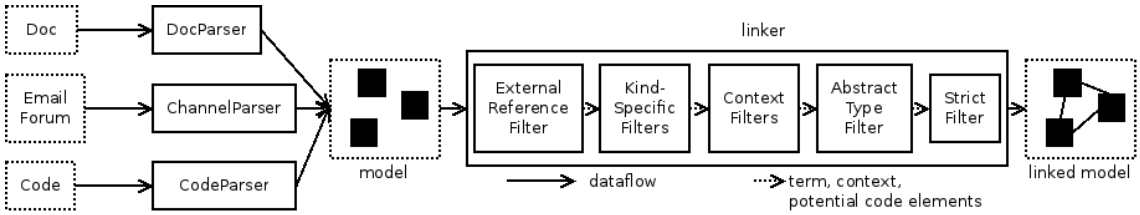

Figure 3. Parsing Artifacts and Recovering Traceability Links

errors include (1) typographical errors such as `HtttpClient`, (2) case errors such as `basiclineparser`, (3) hierarchy errors (e.g., `Collection.add()`, does not exist and potentially refers to `List.add()`), and (4) parameter errors such as forgetting a parameter in a call.

Given these sources of ambiguity and based on our experience with the Spring Framework, we make two assumptions that guide our link recovery strategy:

1) Two code-like terms mentioned in close vicinity are more likely to be related than terms mentioned further apart.
2) Members like methods and fields are unlikely to be mentioned without their declaring type also being mentioned in their context (as described in Section II).

A. Link Recovery Process

Our link recovery process takes as input a collection of code-like terms. Each code-like term is associated with a kind (e.g., method, field, class, annotation) and the other terms present in its context (immediate, local, and global, see Section II). The output of the link recovery operation is a ranked list of code elements that are potentially referred to by each code-like term. Given a collection of code-like terms, we perform the following steps:

1) Link code-like terms that are classified as types (e.g., class, annotation).
2) Disambiguate types.
3) Link members (fields and methods).
4) Link misclassified terms.

Linking Types. Given a code-like term, we find all types in the codebase whose name matches the term. We use the fully qualified name if it is present in the term. Otherwise, we search for code elements using only the simple name.

Disambiguating Classes. A code-like term that refers to a type may be ambiguous if multiple types share the same simple name (declaration ambiguity). For example, in the Hibernate library, there are two `Session` classes: one is declared in the `org.hibernate` package and the other in the `org.hibernate.classic` package.

When a term can be linked to multiple types from different packages, we count the number of types from each package mentioned in the same support message or documentation section. If a package is mentioned more frequently, the type from that package is ranked first. Otherwise, we rank the types by increasing order of package depth: we assume that deep packages contain internal types that are less often discussed than types in shallow packages.

Linking Members. Given a code-like term referring to a member (method or field), we find all code elements of the same kind that share the name of the term. For example, for the term `add()`, we find all methods named `add` in the API. Then, the potential code elements go through a pipeline of filters that eliminate some elements or re-order the list of potential elements. These filters rely on the types identified in the previous steps and we describe them in Section III-B.

Linking Misclassified Terms. Our parser may occasionally misclassify code-like terms. For example, the term `HTTP` in the HttpClient tutorial may be classified as a field (e.g., Java constants are written in uppercase). Although there is no such field in the HttpClient codebase, there is a class with that name (`org.apache.http.protocol.HTTP`).

In this step, we take all terms that were not linked to any code elements in the previous steps. Then, we search for any code element that have the same name as the term. We group the potential code elements by their kind and we attempt to link them in this order: types, methods, fields. The linking technique is the same as in the previous steps.

No Fixed Point. Even though we discover new links at each step and these links can potentially influence previous linking steps, we stop after executing the fourth step. A variation of our link recovery process would be to go through all the linking steps until no more link can be discovered or changed. We found during initial prototyping that the additional complexity introduced by reaching a fixed point is not warranted because in practice, further link recovery passes don't improve the linking accuracy. We confirmed this early observation during the evaluation of our technique: none of the linking errors could have been prevented by executing another pass, but a more accurate parser and better filtering heuristics would have improved the results (Section IV).

B. Filtering Heuristics

Because of the four sources of ambiguity mentioned in Section III, it often happens that a code-like term may be linked to many potential code elements. For example, in the evaluation of our technique, we found that on average, each

term classified as a method could be linked to 16.8 methods declared in 13.5 different types.

We devised a pipeline of filtering heuristics that attempt to resolve these ambiguities. The input of each filter is a code-like term, its context, and a list of potential code elements. Each filter eliminate potential code elements before passing the term, its context, and the remaining code elements to the next filter. Two filters, context name similarity filter and abstract type filter, reorder the potential code elements instead of eliminating them.

We say that a filter was *activated* if it modifies the list of potential code elements. All filters are thus executed, but they may not be activated if a previous filter removed all potential code elements.

We describe each category of filters in the order they are executed in the pipeline. Each category is represented in Figure 3. We indicate the type of ambiguity these filters address at the end of each filter.

1) External Reference Filter: This filter identifies code-like terms that are likely referring to elements outside the system's codebase or concepts with names similar to code elements. This filter considers that all types of the Java Standard Library are external references. Then, the filter tries to match the terms to a list of words that is specific to the system being analyzed. For example, the term `HttpClient` is both the name of a type and of a system. In the HttpClient mailing list, this term is almost exclusively used to refer to the system, unless the term appears in a code snippet. When this filter identifies an external reference, it eliminates all the potential code elements and the subsequent filters are never activated. Although the list of system-specific words need to be provided by the user, the number of words per system is generally low (e.g., 5 words for HttpClient) and it does not significantly impact the accuracy of our approach (see Table V in Section IV-B) (External Reference Ambiguity)

2) Kind-Specific Filters: We implemented two filters that are only activated for certain kinds of terms (methods).

Parameter Number. If the term includes the number of parameters (e.g., `put(key,value)`), the filter eliminates potential code elements that do not have the same number of parameters. (Overload Ambiguity)

Parameter Types. If the code-like term includes the types of the parameters, this filter eliminates the potential code elements whose parameter types do not match. When the parameters are given as arguments instead of types (e.g., `put(obj,obj)` vs. `put(Object,Object)`), we match the parameter types based on the name similarity: if the name of a parameter in the term matches 80% of the name of the parameter type and if more than half of the parameters match, we consider that the term matches the code element. The name similarity of two parameters is obtained by computing the number of common pairs of characters divided by the number of possible pairs. This is a metric that has been found to be robust for assessing the similarity of code-related strings [14, p.4]. We determined the thresholds during initial prototyping of the approach. (Overload Ambiguity).

3) Context Filters: These filters look at the context for the code-like term to determine which potential code element is most likely being referred to. These filters all try to find the declaring type of a term in the term's context.

Context. Types that declare a member are often mentioned in the vicinity of the member. Given a term classified as a member (e.g., method, field), this filter tries to find in the immediate context a type declaring the member. If it fails, it tries to find such type in the local context. Finally, it tries to find a type declaring the member in the global context. When the filter finds one or more type that declares the member, the filter eliminates all potential code elements that are not declared by these types. (Declaration Ambiguity)

Context Hierarchy. This filter is similar to the Context filter, but instead of looking for a type that declares a member, it looks for a type whose *ancestors or descendants* declare the member. As with the previous filter, the context hierarchy filter first searches the immediate, the local, and then, the global context. Context hierarchy filters are interleaved with context filters, so, for instance, the local context hierarchy filter is applied before the global context filter.

As an example of this filter, consider the section "Using a MutableDateTime" of the JodaTime user guide [5]. This section contains a snippet with the following code-like term: `toMutableDateTime()`. There are three potential code elements whose name match the term: {`Instant.toMutableDateTime`, `ReadableDateTime.toMutableDateTime`, `Abstract-Instant.toMutableDateTime`}. The filter thus looks in the context for the term and finds that the local context contains the following types: {`MutableDateTime`, `DateTime`}. None of these types declares the `toMutableDateTime` method, but one of their ancestors, `ReadableDateTime` does, so the filter eliminates all potential code elements except `ReadableDateTime.toMutableDateTime`

The context hierarchy filter takes into account that hierarchy errors (a form of Language Ambiguity) can happen. This is why we consider both ancestors and descendants of a type. (Declaration and Language Ambiguity).

Context Name Similarity. In many cases, a code-like term is prefixed not by its declaring type, but by a variable name. In such cases, we can rely on name similarity between the variables and the type names to disambiguate the term being linked. For example, consider the term list `ehcache.put()` from the Hibernate framework. `ehcache` does not match any type name in Hibernate and there are more than 100 `put` methods declared in various types. The Context Name Similarity filter would go through all the potential methods and rank the methods according to how similar the name of their declaring type is to `ehcache`. In this case, this filter

would rank first the method `EhCache.put()`. We use the same similarity measure as the Parameter Types filter. This filter is used only for code-like terms mentioned in English sentences: code-like terms in snippets contain the declaring type inferred by PPA. (Language Ambiguity)

Order of Execution. The context filters are executed in this sequence: immediate, immediate hierarchy, local, local hierarchy, global, global hierarchy, and context name similarity. As soon as one of the context filter is activated, the remaining filters are skipped. These filters try to find the declaring class of a term in its context. Hence, when a filter finds a declaring type close to a term, it ignores potential types that are mentioned further apart.

4) Abstract Type Filter: This filter ranks the potential code elements according to the number of descendants of their declaring type. The rationale is that a member from the most abstract type is likely to be more representative of the code-like term than the member from the most specific type. This filter privileges members from top-level types such as interfaces over intermediate types such as abstract classes implementing part of an interface. (Declaration Ambiguity)

5) Strict Filter: After we have executed all the filters, there are three potential outcomes: (1) the filters eliminated all potential code elements, so the term is not linked, (2) only one potential element remains and it is linked to the term, and (3) more than one potential element remains.

If there are more than one potential code element, we select the first element from the ranked list if at least one of the context filter was activated. This condition is based on our second assumption that a member is rarely mentioned without its declaring type. The context filters are thus highly important in our filtering pipeline: they eliminate potential code elements based on the context for a code-like term, and they also determine whether a term will be linked at all.

We refer to this last filter as *strict filtering* because it ensures that we do not spuriously link code-like terms that look like code elements. (External Reference Ambiguity)

IV. Evaluation

We implemented an infrastructure that retrieves, analyzes, and classifies the content of developer learning resources, and that recovers the links between these learning resources and the codebase of a framework or a library. To link code-like terms to code elements, we devised a pipeline of filtering heuristics that are based on the hypothesis that code elements referenced closer to each other are more likely to be related than code elements referenced further apart. These filters are responsible for resolving the four sources of ambiguities that may occur when trying to link a term to a code element. We designed this infrastructure based on our manual inspection of the Spring Framework learning resources.

We conducted a study to assess the validity of our hypothesis and the effectiveness of our filtering pipeline. The following research questions guided our evaluation efforts:

Table I
Target Systems Version

System	Version	Support Channel Dates
Joda Time	1.6	4/1/2002-12/1/2010
HttpComponents	4.1	1/1/2008-12/1/2010
Hibernate	3.5	1/1/2005-12/1/2010
XStream	1.3.1	1/1/2005-12/1/2010

1) Can we correctly link code-like terms to code elements with a high precision and recall?
2) What is the usage profile of the filtering heuristics? Are they all necessary and do they resolve all ambiguities?

In the context of fully-automated linking approaches, we consider a precision and recall of 90% to be necessary for the approach to be workable. This threshold is arbitrary, but reflects its intended use, where there is little opportunity to manually correct errors.

A. Study Design

We answered the above questions by analyzing the code, documentation, and support channels of four open source systems (one release for each). For each project, we randomly selected a list of documentation sections and support messages that we then manually inspected. For each section or message, we manually identified the code-like terms and the code element the terms referred to. Finally, we executed RecoDoc on the four projects: RecoDoc parsed the documents and the support channels, generated the corresponding model, and linked the terms to the code elements. We then compared our manual inspection to the results of RecoDoc.

Target Systems. We selected four open source systems written in Java that vary in size, domain, documentation style, and support channel types. Of the four target systems, only the first system can be considered as focusing exclusively on the Java programming language, i.e., the documentation and support channel only contain references to the Java API.

Joda Time is a Java library that aims to replace the `Date` and `Calendar` Java API classes [15]. Joda Time has more than 79 KLOC. Its documentation has 13 761 words and is written using Maven Doxia [16], and its main support channel is a mailing list hosted on SourceForge. This library does not need any configuration file to be used and is mostly used by calling methods and instantiating classes.

HttpComponents is a Java library that simplifies the communication with a web server [4]. HttpComponents is split in two main components, HttpCore and HttpClient. It has more than 85 KLOC. Its documentation is written in DocBook [9] and it is split in two documents. The document we studied, HttpClient Tutorial, has 15 275 words. The main support channel for HttpComponents is a mailing list hosted on Apache. The documentation and the support channel often mention various protocols (e.g., HTTP). The library is used by calling methods, instantiating classes and by implementing interfaces.

Table II
UNITS OF ANALYSIS: RANDOM SAMPLE (S) AND POPULATION (P) CHARACTERISTICS

System	Samp. Avg. Words	Samp. Min Words	Samp. Max Words	Pop. Avg. Words	Pop. Std. Dev. Words	Samp. Avg. Terms	Pop. Avg. Terms	Samp. Avg. Elems	Pop. Avg Elems
Joda Doc.	142.4	2	951	157.5	176.2	14.8	17.2	7.9	8.3
Joda Chan.	229.6	14	1156	294.5	290.2	10.7	11.2	2.9	2.5
HC. Doc.	157.1	3	612	157.1	110.7	19.7	19.7	13.1	13.1
HC. Chan.	332.7	23	2041	373.5	592.0	12.3	13.3	2.7	1.5
Hib. Doc.	256.0	3	1155	249.8	203.2	16.5	15.4	3.9	5.7
Hib. Chan.	128.3	2	1095	116.02	253.3	19.2	11.4	2.6	1.4
XSt. Doc.	65.3	1	358	86.9	135.9	11.6	17.3	1.8	3.3
XSt. Chan.	208.8	25	800	210.2	176.6	14.1	14.1	2.6	2.1

Hibernate is an Object-Relational Mapping framework (ORM) written in Java: it enables clients to persist objects to a relational database [3]. It has more than 905 KLOC. Its documentation is split in three main documents (it was merged into two documents when we finished the study) and the document we analyzed, the main reference manual, has 70 900 words. The support channel is a forum. This framework usually requires a configuration file written in XML or a property file. Hibernate is used by calling methods, instantiating classes, making queries written in a custom language (HQL), and using Java annotations.

The fourth project, XStream, is a Java library that enables the persistence of object graphs into XML files. It has more than 14 KLOC. Its documentation is written manually in HTML and contains 25 560 words. The support channel is a mailing list hosted on Gmane. The library does not need any configuration file, but since it generates and reads XML files, many XML snippets are presented in the learning resources. The library is used by calling methods, instantiating classes, and, in some rare scenarios, by implementing interfaces.

Table I shows the version of the target system we studied and the date range we used to randomly select messages from their support channel. Although JodaTime and XStream have stayed backward compatible throughout their history, the two other systems have undergone significant changes (from 3.1 to 4.0 for HttpComponents and from 2.1 to 3.0 for Hibernate) so we only selected messages that were posted after the first beta release had been published. We wanted to make sure that a support message randomly selected had a chance to contain a term referring to a code element existing in the releases we studied. For all systems, we selected the last available message at the time of the evaluation study.

Unit of Analysis. We randomly selected 100 support messages and 100 documentation sections for each target system: we will refer to these as *units*. We inspected each unit and we manually identified the code snippets and the code-like terms that might refer to a code element in the system's codebase. We linked each code-like term to the most specific code element in the API, unless the code-like term was referring to a set of code elements (e.g., all implementations of the `save()` method), in which case, we selected the most general code element such as a method declared in an interface. We read the entire page or support thread of each unit to select the code element that was the most likely being referred to. Each unit took on average 5 minutes to inspect.

Table III
RESULTS OF LINK RECOVERY EVALUATION

System	Inspection	Recodoc	Prec.	Recall
Joda Doc.	807	763 (772)	96.2%	94.5%
Joda Chan.	291	279 (283)	96.5%	95.9%
HC. Doc.	1288	1272 (1273)	98.7%	98.8%
HC. Chan.	266	257 (260)	95.2%	96.6%
Hib. Doc.	361	349 (349)	89.7%	96.7%
Hib. Chan.	265	247 (247)	93.9%	93.2%
XSt. Doc.	175	170 (170)	95.5%	97.1%
XSt. Cha.	267	244 (255)	92.4%	91.4%
Total	3720	3581 (3609)	95.9%	96.3%

We did not identify and link code-like terms from Java exception traces because they contain many code-like terms that are more often related to bugs, and they introduce too much noise. For example, a stack trace may contain more than one hundred code-like terms whereas, on average, an email message from our four target systems contains only 11.6 code-like terms.

Following our manual inspection, we lauched RecoDoc, which analyzed all support messages and documentation sections of the four projects. RecoDoc has to analyze all the sections and support messages in the same page or support thread as the units in our random sample because RecoDoc may need to analyze the global context of a code-like term (see Section III-B3).

Table II shows the characteristics of the units for each target system: the average, the minimum, and the maximum number of words, per selected unit (sample), the average number of words and standard deviation for all units (population), the average number of code-like terms for the sample and the population, and the average number of code-like terms that RecoDoc linked to a Java code element for the sample and the population. The length (in words) of the units in our random sample was always within 0.2 of the standard deviation of the population. The wide range of units RecoDoc analyzed provide an evidence that our approach can be used in practice, for small or large units.

Table IV
CONTEXT FILTERS ACTIVATION DISTRIBUTION

System	Immediate Context	Immediate Hierarchy	Local Context	Local Hierarchy	Global Context	Global Hierarchy	Ctx Name Similarity	Total
Joda Doc.	40.1%	6.3%	44.0%	7.6%	1.0%	0.0%	1.0%	100.0%
Joda Chan.	34.2%	7.3%	35.3%	7.2%	7.9%	2.5%	1.0%	95.4%
HC. Doc.	75.3%	14.0%	9.0%	0.2%	0.2%	0.5%	0.5%	99.7%
HC. Chan.	44.7%	7.5%	20.3%	5.4%	5.1%	2.2%	5.9%	91.1%
Hib. Doc.	44.9%	0.3%	33.9%	4.0%	15.8%	0.1%	0.0%	99.0%
Hib. Chan.	51.8%	1.1%	26.2%	6.0%	4.9%	0.7%	1.0%	91.7%
XSt. Doc.	59.8%	0.0%	31.1%	2.9%	5.0%	0.6%	0.6%	100.0%
XSt. Chan.	62.4%	0.7%	18.1%	5.4%	6.3%	0.1%	1.4%	94.4%
Total	51.7%	1.5%	25.8%	5.9%	5.1%	0.8%	1.2%	92.0%

B. Results

During our inspection of the units, we manually linked code-like terms with code elements. We then compared our findings with the results from RecoDoc. There were five possible cases for each code-like term that we identified: (1) we linked the term with the same code element as RecoDoc (exact match), (2) we linked the term with a code element that was a descendant, an ancestor, or an overloaded version of the code element linked by RecoDoc (similar match), (3) RecoDoc failed to link a term that we manually linked (false negative), (4) RecoDoc linked a term that we did not link (false positive), (5) RecoDoc linked a term that we linked to another term (false negative and false positive).

Table III shows the results of our evaluation. The second column, *Insp.*, gives the number of code-like terms that we linked to a code element during our inspection. The third column, *Recodoc*, gives the number of terms that RecoDoc correctly linked to a code element (exact matches). The number in parentheses adds the similar matches to the number of exact matches. The fourth column, *Prec.* gives the precision of RecoDoc (exact matches divided by number of links found by RecoDoc). Finally, the fifth column, *Recall*, gives the recall of RecoDoc (exact matches divided by number of links found by our inspection).

RecoDoc Accuracy. The results from Table III clearly indicate that our technique can correctly link code-like terms to code elements with a high precision and a high recall (all over 90%, except the Hibernate documentation). RecoDoc practically always linked code-like terms in snippets to a correct code element because Partial Program Analysis recovered most of the necessary type information.

More than half of the false positives (98 out of 123 code-like terms that were incorrectly linked to a code element) were caused by code-like terms referring to a concept that had the same name as a code element (e.g., a Session). These cases were often difficult to judge during our manual inspection because it was not always clear if the writer was referring to a concept or a code element. The other false positives were caused by the linker not being able to resolve a declaration ambiguity, i.e., more than one type declaring a member were mentioned in the member's context. In these cases, the linker selected the first declaration in the list of potential declarations. This also resulted in a false negative.

The majority of the missed code elements (54 out of 104 false negatives) were caused by the parser not identifying the code-like terms in the first place. For example, in the support channels, the parser missed code-like terms mostly because of formatting inconsistencies such as words that appeared to be in a quoted message but that were in the reply.

The remaining missing code elements were caused by the linker selecting the wrong declaration because of a declaration ambiguity (25) or because the strict filter and the external reference filters were too conservative (25).

Filtering Heuristics. Of the 300 228 code-like terms that RecoDoc linked in all units (not just the sample), 160 970 were type members such as a method. On average, each of these code-like terms could potentially be linked to 16.8 members declared in 13.5 different types. This is an evidence that linking members is technically challenging. After going through all the filtering heuristics introduced in Section III-B, each code-like term could potentially be linked to only 0.7 member on average. This is an evidence that our filtering heuristics are effective at reducing the number of potential matches.

As we mentioned in Section III-B5, the context filters are the most important filters of our pipeline because (1) they eliminate potential code elements based on the context for a term, and (2) at least one contextual filter must be activated to link a term. Only one context filter can be activated for each term. Table IV shows how often each context filter was activated in the target systems (the numbers are only for the 160 970 type members). For example, in the Joda Time documentation, RecoDoc found the declaring class in the immediate context of 40.1% of the code-like terms representing methods or fields.

The sum of the usage profile of the context filters does not reach 100% because our technique can link a code-like term to a member without using any context filter. For example, in the Joda Time Channel, RecoDoc found the declaring class of 95.4% of the code-like terms in their context. RecoDoc did not find the declaring class of the other 4.6% code-like terms, but because these code-like terms could refer to

Table V
CAUSES OF CODE-LIKE TERMS NOT BEING LINKED

System	Terms	No Match	Ext. Ref.	Strict
Joda Doc.	386	89.1%	4.7%	6.2%
Joda Chan.	10059	76.8%	5.8%	17.3%
HC. Doc.	354	73.4%	10.7%	15.8%
HC. Chan.	46885	76.6%	7.0%	16.4%
Hib. Doc.	2080	41.4%	22.5%	36.1%
Hib. Chan.	885123	65.4%	8.0%	26.6%
XSt. Doc.	1250	90.4%	3.0%	6.6%
XSt. Chan.	25440	86.5%	5.0%	8.5%
Total	971577	66.6%	7.9%	25.5%

only one code element in the codebase, RecoDoc linked them nonetheless (see Strict Filtering in Section III-B).

Our technique found the declaring class of 86.1% (51.7 + 1.5 + 25.8 + 5.9 + 1.2) of the terms in their immediate or local context. This indicates that most documentation sections and support messages are self-contained and can be understood by readers without scanning the entire documentation page or support thread.

Unlinked Code-Like Terms. RecoDoc correctly chose to not link 971 577 code-like terms that looked like a method or a field, but that did not refer to any code element. Table V shows the reasons why RecoDoc did not link these code-like terms: (1) we did not find a code element whose name matched the code-like term, (2) an external reference filter was activated and eliminated all potential code elements of a term, or (3) the strict filter was activated. For example, for the Joda Time documentation, 89.1% of the 386 code-like terms that RecoDoc correctly did not link did not match any code element in the Joda Time codebase. 4.7% of these unlinked terms were eliminated by the external reference filters, and 6.2% were eliminated by the strict filtering pass.

Our technique did not link most code-like terms because it could not find a code element with the same name. For example, in HttpClient, the parser identified code-like terms such as PUT, and GSSAPI, but the linker correctly ignored these terms because they did not exist in the API.

The external reference filters were particularly useful in the Hibernate documentation because the API declares many methods whose name are the same as the methods in the Java standard library. For example, in section 20.5.4 of the Hibernate documentation, the term `list.clear()` refers to the interface method `java.util.List.clear()` and the Standard Library Classes filter correctly eliminated this term even though Hibernate declared methods with a similar name such as `PersistentList.clear()`.

The strict filter ensured that a code-like term, which looked like many potential members, was not linked if the member's declaring class was not in the term's context. This strategy was again useful when linking terms from the Hibernate documentation because the Hibernate codebase declares many methods that have a common name and that are declared in example code. For instance, section 1.1.3 of the Hibernate documentation mentions the term `getId()`. This term refers to the code snippet presented earlier in the section, but it can also be linked to 11 methods from 11 types in the Hibernate codebase. Because none of these types are mentioned in the page, the strict filter was correctly activated and eliminated all potential code elements.

C. Threats to Validity

The accuracy of the results is subject to the investigators' assessment of each benchmark code-like term. In some cases, the exact target of a code-like term in the learning resources was not perfectly clear. It is thus possible that we erroneously linked some benchmark terms during our manual inspection. However, every time our our manual inspection and RecoDoc results diverged on these unclear terms, we conservatively assumed that RecoDoc was wrong. Hence, the accuracy of the reported results should represent a lower-bound of the accuracy of RecoDoc. In addition, our detailed classification is publicly available for inspection [8].

We avoided the issue of overfitting by evaluating our technique on a different system than the one we used to develop and test RecoDoc. Specifically, we developed the parser and the linker based on our observations of the Spring Framework project. Although our technique works well on this large and complex system, we did not use it in our evaluation to ensure that our results were not biased.

The external validity of our evaluation is limited by the characteristics of target systems we analyzed. We selected four different systems that vary widely in their choice of documentation and support channel infrastructures, size, domain, and usage patterns (e.g. calls to a Java API vs. inheritance of a Java API and configuration files), but we did not cover all kinds of systems, such as GUI toolkits.

We executed RecoDoc on the four target systems with the same parameters. Only their parser extension and a small subset of the external reference filter differed. As we showed in the previous section, the external reference filters eliminated only 7.5% of the code-like terms, so external reference filters did not significantly impact the results. Moreover, it would be possible to reduce the usage of these custom filters by considering only the public API. For example, in Hibernate, most classes and methods are never used by users, but RecoDoc still tried to link code-like terms to elements that were named after technical concepts (e.g., Select). We chose to consider the entire codebase because there was no objective way to determine whether a type was part of the public or internal API. However, it is likely that a Hibernate developer could do this easily.

V. RELATED WORK

A variety of techniques have been proposed to link code elements to natural language queries or general project artifacts such as documents and bug reports.

For example, Natural Language Processing (NLP) and information extraction techniques frequently rely on the context of a term or the distance between two terms to extract relevant relationships [17]. The presence of a term in the context of another term is called a *discourse feature*. As opposed to our technique, users of general NLP techniques typically need to train the techniques on a corpus first to develop a reliable classifier for a specialized task.

Antoniol et al. applied two information retrieval techniques, the probabilistic model and the vector space model, to find the pages in a reference manual that were related to a class in a target system [18]. The authors found that the accuracy of both approaches was limited. Information retrieval techniques work best when the entities to be linked can be expressed by several words. Information retrieval technique are thus usually used to link coarse-grained artifacts like entire documents and classes [19], [20], [21] whereas our technique attempts to link single code-like terms to fine-grained code elements.

Hipikat is a tool that generates a project memory from a set of coarse-grained artifacts: bug reports, support messages, source code commits, and documents [6]. The tool stores and indexes the artifacts and then determines whether the artifacts are related. Hipikat uses several strategies to recover the links between the artifacts: presence of bug numbers, textual similarity (using a vector space model), etc. The tool enables developers to query the project memory by returning a set of artifacts related to the query.

XFinder is a tool that matches the steps of a tutorial to the code elements that implement each step [22]. For example, a tutorial might describe how to implement a text editor using the Eclipse platform: implement interface `X` and call method `Y`. Given a codebase implementing several text editors, XFinder will find all the text editors and will map the code elements implementing each editor to the steps of the tutorial. As opposed to our technique, XFinder expects the tutorial to be encoded in a specific format that identifies the kind of the step and the artifacts.

Dekel and Herbsleb devised eMoose, a tool that enables framework developers to annotate the API documentation of a framework to highlight "directives" such as preconditions [23]. When a developer writes code that calls a method with an annotated directive, eMoose highlights the method call in the code editor. Contrary to our technique, the links between the documentation and the API must be encoded manually by the framework developers.

Hill et al. built a technique that links query terms to a set of matching methods in a codebase [24]. The NLP-based technique analyzes the methods and their parameters by tokenizing their identifiers and determining their part-of-speech (POS) tags to compute multiple propositional phrases (e.g., `addItem(BookItem)` becomes "add item" and "add book"). Our technique could potentially try to match sentence fragments with these propositional phrases.

Identifying Code Snippets. The need to identify code elements in natural language documents is not recent and several techniques have been devised to this end. One technique and one study have particularly influenced our parsing infrastructure.

Island Grammars is a general technique that enables the identification of structured constructs such as code elements in arbitrary content (e.g., an email message) [25]. The main idea is to separate the content into small recognizable constructs of interest (islands) and everything else (water). Our parser implements a similar approach by first identifying the code snippets (big islands) and then, by identifying the smaller code elements (small islands) in the English paragraphs (water).

Bacchelli et al. compared various techniques to identify code elements and code snippets in email messages and found that lightweight techniques based on regular expressions performed better than information retrieval techniques such as latent semantic indexing [7]. We implemented our documentation and support channel parsers with regular expressions based on the observations of this study.

VI. Conclusion

We presented a technique for precisely linking code-like terms in developer documentation and support channels to fine-grained code elements in a system's codebase. We designed our technique based on the assumption that code elements mentioned in close vicinity are more likely to be related than code elements mentioned further apart. We identified four sources of ambiguity inherent to linking code-like terms in unstructured natural language documents, and we devised a pipeline of filtering heuristics to resolve these ambiguities.

In an evaluation study with four different open source systems, we showed that our technique could link code-like terms to code elements with a high precision and recall (96%). Additionally, our study showed that linking code-like terms in documentation sections and support messages was a difficult problem because each code-like term representing a method could be associated on average with 16.8 methods declared in 13.5 different types.

We plan to use the parsing and linking infrastructure we built to analyze developer documentation and automatically recommend improvements. For example, we can now easily detect whether a code-like term referenced in a documentation section is often mentioned in support messages: this situation could indicate that the section needs to be clarified.

Acknowledgments

The authors thank David Kawrykow and Tristan Ratchford for their valuable comments on the paper. This project was supported by NSERC and FQRNT.

REFERENCES

[1] B. Dagenais and M. P. Robillard, "Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2010, pp. 127–136.

[2] "Spring Framework," http://www.springsource.org/, accessed 31-Aug-2011.

[3] "Hibernate," http://www.hibernate.org/, accessed 31-Aug-2011.

[4] "HttpComponents," http://hc.apache.org/, accessed 31-Aug-2011.

[5] "Joda Time User Guide," http://joda-time.sourceforge.net/userguide.html, accessed 31-Aug-2011.

[6] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A Project Memory for Software Development," *IEEE Transactions on Software Engineering*, vol. 31, pp. 446–465, 2005.

[7] A. Bacchelli, M. Lanza, and R. Robbes, "Linking e-mails and source code artifacts," in *Proceedings of the 32nd International Conference on Software Engineering*, 2010, pp. 375–384.

[8] "Recodoc," http://www.cs.mcgill.ca/~swevo/recodoc, accessed 31-Aug-2011.

[9] "DocBook," http://www.docbook.org/, accessed 31-Aug-2011.

[10] "Maven," http://maven.apache.org/, accessed 31-Aug-2011.

[11] "Eclise Java Compiler," http://www.eclipse.org/jdt/, accessed 31-Aug-2011.

[12] B. Dagenais and L. Hendren, "Enabling Static Analysis for Partial Java Programs," in *Proceedings of the 23rd Conference on Object-Oriented Programming Systems Languages and Applications*, 2008, pp. 313–328.

[13] B. Dagenais and M. P. Robillard, "Recovering Traceability Links between an API and its Learning Resources," School of Computer Science, McGill University, Tech. Rep. SOCS-TR-2011.6, 2011.

[14] Z. Xing and E. Stroulia, "Umldiff: an algorithm for object-oriented design differencing," in *Proceedings of the International Conference on Automated Software Engineering*, 2005, pp. 54–65.

[15] "Joda Time," http://joda-time.sourceforge.net/, accessed 31-Aug-2011.

[16] "Maven Doxia," http://maven.apache.org/doxia/index.html, accessed 31-Aug-2011.

[17] M.-F. Moens, *Information Extraction: Algorithms and Prospects in a Retrieval Context*. Springer, 2006.

[18] G. Antoniol, G. Canfora, G. Casazza, A. D. Lucia, and E. Merlo, "Recovering traceability links between code and documentation," *IEEE Transactions of Software Engineering*, vol. 28, no. 10, pp. 970–983, 2002.

[19] X. Chen, "Extraction and visualization of traceability relationships between documents and source code," in *Proceedings of the International Conference on Automated Software Engineering*, 2010, pp. 505–510.

[20] A. De Lucia, R. Oliveto, and G. Tortora, "Adams re-trace: traceability link recovery via latent semantic indexing," in *Proceedings of the 30th International Conference on Software Engineering*, 2008, pp. 839–842.

[21] J. Hsin-Yi, T. N. Nguyen, C. Ing-Xiang, H. Jaygarl, and C. K. Chang, "Incremental latent semantic indexing for automatic traceability link evolution management," in *Proceedings of the 23rd International Conference on Automated Software Engineering*, 2008, pp. 59–68.

[22] B. Dagenais and H. Ossher, "Automatically locating framework extension examples," in *Proceedings of the International Symposium on Foundations of Software Engineering*, 2008, pp. 203–213.

[23] U. Dekel and J. D. Herbsleb, "Improving API Documentation Usability with Knowledge Pushing," in *Proceedings of the International Conference on Software Engineering*, 2009, pp. 320–330.

[24] E. Hill, L. Pollock, and K. Vijay-Shanker, "Automatically capturing source code context of nl-queries for software maintenance and reuse," in *Proceedings of the of the International Conference on Software Engineering*, 2009, pp. 232–242.

[25] L. Moonen, "Generating robust parsers using island grammars," in *Proceedings of the of the Working Conference on Reverse Engineering*, 2001, p. 13.