Discovering Essential Code Elements in Informal Documentation

Peter C. Rigby and Martin P. Robillard
School of Computer Science
McGill University
Montreal, QC, Canada
{pcr,martin}@cs.mcgill.ca

***Abstract*—To access the knowledge contained in developer communication such as forum posts, it is useful to automatically identify the code elements contained in documents. We propose a novel traceability recovery approach to extract the code elements contained in various documents. As opposed to previous work, our approach does not require an index of code elements to find links, which makes it particularly well-suited for the analysis of informal documentation. When evaluated on 188 StackOverflow posts containing 993 code elements, the technique performs with average 0.92 precision and 0.90 recall. As a major refinement on traditional traceability approaches, we also propose to detect which of the code elements in a document are salient, or germane, to the topic of the post. To this end we developed a three-feature decision tree classifier that performs with a precision of 0.65–0.74 and recall of 0.30–0.65, depending on the subject of the document.**

I. Introduction

An increasing amount of knowledge about software gets conveyed and archived in informal developer communications such as forum posts and mailing lists. Unfortunately, while the informal structure promotes conviviality and discourse, it also makes it more difficult to index, search, and analyze. A particular problem is searching for discussions of a specific software (API) element, to find good usage patterns, bug workarounds, or alternatives.

To address this problem, many recent research projects have targeted the recovery of *traceability links* between source code elements and informal documentation [1], [2]. This work comes in the wake of more general attempts at linking source code and documents [3], [4].

Recent traceability work has focused on the problem of identifying a *known* set of code terms in developer communication. For example, if a post or email message mentions `execute`, does this term correspond to (for example) an `execute` method in a code base of interest? In this case success is strongly influenced by intrinsic factors of both the message and of the names of the source code elements. In the example above, `execute` will be easier to correctly extract as a method if the author of the message includes the parentheses after the method name. Similarly, elaborate names such as `equalsIgnoreCase` are easier to correctly link than pervasive terms such as `add`.

The state-of-the-art of traceability techniques for developer resources shows very good accuracy (see Section II). However, all existing traceability approaches developed to date have two important limitations for the detection of references to code elements in developer communications: a *closed-world assumption*, and a *uniform importance* assumption.

The closed-world assumption is that all code elements of interests are known in advance. This assumption makes sense when processing documents relating to a very specific system, such as the tutorial for the JodaTime API.[1] In this case, it is possible to scan all the text and attempt to resolve various combinations of tokens against code elements of the JodaTime API. Unfortunately, the closed-world assumption breaks down when analyzing general-purpose forums such as StackOverflow, where code terms referring to various software elements can appear in sometimes odd combinations. For example, the Android tag on StackOverflow is associated with 6355 different other tags, including HttpClient and SQLite.

The uniform importance assumption is that all mentions of a code element have equal importance. For example, if 50 messages in a forum contain a mention of the method `DocumentBuilderFactory.newInstance()`, then they are all equally "linked" to this code element. In practice however, we observe that the relevance of a code element can vary widely. In the case above, the element would be highly relevant in a post demonstrating how to access XML processing services provided by the factory, but the element is boiler-plate code in most other cases. We thus consider that not all elements are equally essential to a document: some are more *salient* than others. To determine salience, traditional information retrieval concepts such as term frequency do not apply naively. As a simple example, if a code example demonstrating a GUI layout manager involves three buttons and one layout object, it does not mean that the example is more about buttons than about layouts, even if both terms appear in the same number of documents.

A very desirable goal would be to eliminate the closed world and uniform importance assumptions for code element traceability in developer communications. An ideal solution to this problem would enable us to find, among very large collections of messages and other documents, the ones that actually discuss a particular code element. As a first step in this direction, we developed a novel code element extractor,

[1] http://joda-time.sourceforge.net/userguide.html

In your question you are setting the content type on the class UrlEncodedFormEntity to JSON. While this works for other types, with JSON you should use a StringEntity and convert the JSON object toString:

```
1 // Build the JSON object to pass parameters
2 JSONObject jsonObj = new JSONObject();
3 [...]
4 // add the parameters to the POST object
5 StringEntity entity = new
      StringEntity(jsonObj.toString(),
      HTTP.UTF_8);
6 entity.setContentType("application/json");
7 httpPost.setEntity(entity);
8 HttpResponse response =
      client.execute(httpPost);
```

Fig. 1. Answer adapted from StackOverflow

ACER, that works without a pre-defined set of known elements. To explore alternatives to the uniform importance assumption, we also built a classifier that estimates whether an element is salient or not in a document.

An evaluation of our code extractor on 188 StackOverflow posts containing a total of 993 code elements showed an average precision of 0.92 and an average recall of 0.90. These numbers are just a few percentage points lower that the best *closed-world* code element linker, RecoDoc [2].

The classification of code elements as salient or not is much less well-defined and also more difficult. In this case, our classifier managed a precision of 0.65–0.74, and a recall of 0.30–0.65 (depending on the subject of the document).

These results are encouraging because they show that the close-world assumption can be shed with minimal loss of accuracy, and that a reasonable, if modest, accuracy can be obtained for the difficult task of determining whether an element is salient. Although incremental improvements in both areas are likely to follow, we feel that the performance of our initial infrastructure is already sufficient to motivate experimentation with applications such as advanced search tools.

II. Background

Documents: Our approach is designed to extract and estimate the salience of code elements in most types of developer communication. We call *document* a generic unit of developer communication. Documents include posts on a Q&A forum, email discussions, and formal documentation like tutorials and Javadoc. In this paper we use forums as our target application.

Figure 1 illustrates a typical forum post. Such documents generally include free-form text and code fragments, both of which potentially refer to code elements from different APIs.

Code Element Extraction: There are three phases involved in most code element extraction techniques (also called linking or traceability recovery techniques). Phase 1 and 2 do not depend on each other.

1) Identifying *code-like terms*, which are sequences of characters in a document that resemble code elements.
2) Creating an index of valid code elements. *Code elements*, are types (e.g., classes, enums, annotations), methods, and fields (or their equivalents in different programming languages). For closed-world techniques, the index consists of the elements defined in the source of a software system of interest (e.g., all the classes, methods, and fields of the Java Swing API).
3) Resolving code-like terms to their corresponding code elements (and eliminating code-like terms that do not map to valid elements). Code-like terms that can map to a single code element are unambiguous. In contrast, an *ambiguous* code-like term can map to multiple code elements.

The output of the process is a list of the code elements associated with each document. The morphology of code-like terms and their (non-)ambiguity determines the difficulty of resolving them. Unambiguous terms are trivially resolved, whereas ambiguous terms may be unresolvable. We additionally distinguish between *qualified* and *unqualified* terms. *Qualified* terms are connected by a dot to another term (e.g.,`httpPost.setEntity`) and tend to be unambiguous, while *unqualified* terms always require further resolution (e.g.,`toString` in the free-form text in Figure 1).

Salience: For a code element to be salient, it must be central to an example code fragment or have some discussion defining its function or describing its use. For example, in the answer in Figure 1, we learn that when combining JSON with HttpClient, `UrlEncodedEntity` cannot be used. Instead `StringEntity` should be used and with the content type set to JSON through `StringEntity.setEntity`. The JSON object must also be converted to a string via `JSONObject.toString`. These code elements are salient to the answer. In contrast, `httpPost.setEntity` on line 7 and the constructor `JSONObject` on line 1 represent contextual code elements that provide the setup for the salient elements. In Section VII, we elaborate on the guidelines for manually determining the salience of a code element.

III. Related Work

Bacchelli et al. [1] use **lightweight regular expressions** that are based on programming language naming conventions to identify code elements contained in email discussions. Their technique, called Miler, is case sensitive and involves using camel case to identify the entities. Camel-cased terms can be divided into *compound terms* that contain two or more words (e.g.,`HttpClient`) and *non-compound terms* that are single words (e.g.,`get()` or `Intent`). Non-compound terms tend to be more ambiguous than compound terms because single terms are more likely to be a word that is commonly used in English (e.g., "To get an Intent ..."). To resolve non-compound terms, Miler searches the document for the term's fully qualified name that includes a package and class (e.g.,`android.content.Intent`), or the file name (e.g., 'Intent.java'). Miler's index of code elements is based on the source code of a reference system. The index contains only classes, so it does not recognize members like methods and

TABLE I
COMPARISON OF CODE ELEMENT EXTRACTION TECHNIQUES

Technique	Code-like terms	Code Element Index	Resolver	Avg Precision	Avg Recall
Information Retrieval [1], [3], [4]	Bag of words – text normalization	Parsed from source code	e.g., LSI	0.42	0.38
Miler [1]	Language conventions (e.g., camel case)	Parsed from source code	Exact string match	0.33	0.64
RecoDoc [2]	Language conventions and PPA [5]	Parsed from source code	Term context and filters	0.96	0.96
ACER (current paper)	Language conventions and island parser	Parsed from collection of documents	Term context and collection context	0.92	0.90

fields. Miler's average precision and recall is 0.33 and 0.64 respectively. These values vary depending on the software project under examination and the programming language.

Information retrieval techniques have been widely used to resolve the links between source code elements and documentation. For example, Antoniol et al. [3] apply a probabilistic and Vector Space Model (VSM) to resolve terms, while Marcus et al. [4] use Latent Semantic Indexing (LSI). Surprisingly, Bacchelli et al. show that lightweight regular expressions perform similarly to more complex information retrieval techniques, such as LSI, with a average precision of 0.42 and recall of 0.38, and VSM, with an average precision of 0.23 and recall of 0.31.

Information retrieval techniques and lightweight regular expressions are impractical for our purpose because they have low precision and recall and they can only identify classes.

Term context is used in RecoDoc to link code-like terms in documentation to their corresponding source code elements [2]. RecoDoc requires an index of valid code elements that it extracts from the source code of a reference system. It uses lightweight regular expressions to extract code-like terms from free-form text and uses partial program analysis [5] to extract them from code fragments. To resolve ambiguous terms, RecoDoc uses a sequence of heuristic filters. For example, it searches an ambiguous term's *context* for a possible declaring type. Here, "context" refers to additional information in various scopes surrounding the term (see in Section IV). Other filters involve name similarity matching between terms and code elements (e.g., the variable `htclient` matches the type `HttpClient`), excluding overloaded terms that exist in an external library or that represent concepts instead of code elements (e.g., 'URL' can be a concept as well as a code element), and matching terms to a declaring superclass in the class hierarchy. RecoDoc has impressive precision and recall of 0.96.

We borrow and expands upon RecoDoc's notion of term context for resolving ambiguous terms. However, for our purpose, RecoDoc has two limitations. First, its use of partial program analysis creates a dependence on the Eclipse Java compiler [5]. The compiler can handle some errors, but many errors will force it to fail. RecoDoc has no record of the terms in a code fragment with compilation errors. In tutorials, where code fragments are written by expert developers for illustrative purposes, compilation errors are less frequently a problem. However in informal documentation, code fragments represent informal questions and answers that developers quickly construct with a narrow purpose. Developers often eliminate much of the irrelevant code, making the fragment concise but not compilable. Second, RecoDoc creates an index of valid elements by parsing the source code of a software system before extracting code-like terms. This index creates a closed-world of terms, which means that it cannot identify code elements from other APIs. While a tutorial usually contains only code relating to a single API, posts on Q&A sites often refer to multiple APIs. RecoDoc ignores information on how to combine multiple APIs.

Island grammars specify production rules for language constructs of interest (the islands, e.g., code elements), while ignoring other language constructs that are uninteresting (the water) [6]. They were originally developed to extract constructs of interest from source code that did not compile. van Deursen and Kuipers have used this technique to parse source code and automatically generate documentation [7]. As part of ACER, we select Java constructs from the language specification that contain code elements (e.g., a class definition contains the name of a class) [8], and implement an island parser that can extract these constructs from free-form text and code fragments that do not compile.

Table I compares how representative code element extraction techniques work for each stage described above, and include the performance of each approach as reported by its author. We also include a comparison with ACER, the approach we propose in this paper. We note that the performance measures were not obtained on the same benchmark, so direct comparison is not possible. However, each approach was evaluated on a benchmark appropriate to its targeted application, so the measures are at least representative of how the techniques are expected to work in practice.

IV. CODE ELEMENT IDENTIFICATION WITH ACER

Our automated code element resolution tool is called ACER. ACER can extract code elements from documents that contain free-form text as well as code fragments that may not be compilable; process an arbitrary collection of documents, so there is no dependence on a predetermined index of valid code elements; and handle large document collections with high precision and recall.

ACER performs the three stages in the code element identification process. It also has an additional processing stage.

1) It uses an island parser, to identify code-like terms from each document.
2) It creates an index of valid code elements based on stage 1.
3) It reparses each document to identify ambiguous terms that match code elements in the term index. It resolves each term using the term's context.
4) It outputs the code elements associated with each document.

Below we present each stage and the rationale for our choices. We also provide a detailed description of term context, which is necessary to understand our technique.

Stage 1: Island Parser

An island grammar only describes constructs of interest [6]. In our case, the constructs of interest are those that describe code elements. We are uninterested in language constructs, for example, that control the flow of the program. The island parser we developed is composed of a set of regular expressions that are approximations of the following constructs in the Java Language Specification [8]: qualified terms (e.g.,`HttpClient.execute()`), package names, variable declarations, qualified variables (e.g.,`client.execute()`), method chains (e.g.,`client.execute().toString()`), class definitions including inheritance, declaration and overriding of methods, inner classes, constructors, stacktraces, annotations, and exceptions.

We are able to process a document that contains compilation errors, and we do not differentiate between free-form text and code fragments. We define each regular expression, but only to the extent that is necessary to isolate code elements within a Java construct. We order regular expressions from most precise to most flexible because terms contained within a precise regular expression are more likely to be valid than those contained in a highly flexible one. To eliminate some of the ambiguity introduced by the regular expressions and to determine the kind of an ambiguous term (e.g., variable vs. class), we use regular expressions to ensure that each term conforms to the Java naming conventions (e.g., camel case) [8].

Term Context (used in all subsequent stages): The Java specification provides scoping rules that define the context for each term and "In determining the meaning of a [term's] name [...], the context of the occurrence is used to disambiguate among packages, types, variables, and methods with the same name." [8] We use these rules to resolve code-like terms contained in well-defined constructs, for example, to resolve an unqualified method that is declared within the scope of a class declaration. However, scope rules are defined for source code and are insufficient for determining a term's context inside a document that contains both free-from text and code fragments.

To solve this problem, Dagenais and Robillard [2] defined three *term contexts*: immediate or qualifying context, local context (all the terms in the same document), and global or thread context (documents in the same discussion thread). For example, in the answer post in Figure 2 with respect to the term `executeMethod` on line 2, the immediate context is the term `client`. The local context is all the terms in the document depicted by the question. The global context is all the terms contained in related documents: the question and the answer.

While these term contexts have an analog in the context defined in the Java specification, Dagenais and Robillard define them intuitively based on the following observations. First, two code elements mentioned in the same context are more likely to be related than those mentioned further apart or in another context. This concept is known as *term proximity*. Second, members are unlikely to be mentioned without their declaring type in context. The intuition behind the latter is that methods are often declared in multiple types, so their type is usually mentioned in the document, because without it they are ambiguous even to a person reading the document.

Stage 2: Index of valid terms

Unlike previous work, the index is dependent on the terms that are found across the entire collection of documents (i.e. the collection context), instead of the code elements found in the source code of a particular system. Our system must thus build the index by opportunistically collecting and validating all the terms it finds in a specified collection (e.g., a collections of posts, a mailing list, etc)

The flexibility of the parser coupled with the ambiguity of natural language, application-specific terms used by developers, and mistakes made by developers means that not all code-like terms extracted by the island parser are valid. Our intuition is that terms that occur with low frequency are less likely to be valid collection-wide elements than high frequency terms. While we considered more complex alternatives for eliminating terms, we found that an effective technique is simply to exclude terms that appear in only one thread context (one-off terms). For a term to be included in the index as a valid code element, it must appear more than once in a Java construct in a code fragment or in free-form text in a qualified manner (e.g., HttpClient.execute). Valid one-off terms must appear in a Java construct or in a qualified manner in only one document. One-off terms tend to be valid in the document in which they are found, but they introduce false positives when applied to other documents.

Variable and package names need additional processing before we can add them to the index. Package names (e.g.,`org.apache.http.client`) resemble URLs (e.g.,`www.apache.org`). While it may be possible to exclude invalid names through naming conventions, we validate packages by ensuring that each defines at least one type in the collection context (e.g.,`org.apache.http.CookieStore`). We consider package names followed by a ';' or '.*' to be valid.

In the case of variables, each one must be resolved to its declaring type. The answer post in Figure 2 contains three different variables that must be resolved. First, variables that are declared in the local context (e.g.,`InputStream is`) or in the global context (e.g.,`client` is resolved

Question

I've figured out how to create a client and how to GET responses, using the getResponseBodyAsStream in the GetMethod class, can someone show me how to POST [...]

```
HttpClient client = new DefaultHttpClient();
HttpMethod method = new
    GetMethod("http://www.apache.org/");
[...]
```

Answer

Here's a brief code example that should help.

```
//method is a PostMethod
client.executeMethod(hostconfig, method);
method.setFollowRedirects(true);
InputStream is =
    method.getResponseBodyAsStream();
[...]
```

Fig. 2. Resolving variables and unqualified terms. The question and answer posts are adapted from StackOverflow.

to `HttpClient client`) are trivially resolved to their type. Second, the declaration of contextual classes is often removed by developers. To resolve these variables, we determine which members are associated with a variable (e.g.,`method.getResponseBodyAsStream`, `method.setFollowRedirects`). We then determine which types declare a variable with a similar name in the collection context (e.g., one post declares `GetMethod method` and another `PostMethod method`). We assign the variable to the type that has the largest number of members (`PostMethod` declares both members, while `GetMethod` only declares `getResponseBodyAsStream`). In the case of a tie, we select the most frequently used type in the collection context.

Stage 3: Reparse documents and resolve ambiguous terms

With an index of valid terms, we reparse all documents and extract unqualified, ambiguous code-like terms that match a code element in the index. Below we describe how we use the term's context and the collection context to resolve or discard each term.

For example, for each unqualified member (e.g., `getResponseBodyAsStream` in the first sentence in the question in Figure 2), we determine whether or not there is a valid type (e.g.,`GetMethod`) that declares the member and is present within the local context. If it is not in the local context, we repeat this step for the thread context. If the defining type is present, we mark the member and type as valid code elements for the document. In the case of more than one possible type, we choose the type that has the closest proximity to the member. In the case of local context, proximity is the distance in characters between terms, and for the thread context, proximity is the closest absolute difference in document dates (e.g., the difference in time between when a question was posted and an answer was posted).

When we were evaluating ACER, we found that unqualified compound types and unqualified non-compound methods yielded a high number of false negatives and false positives, respectively. We include all ambiguous, unqualified compound types that match a type in the index because the increase in recall outweighed any decrease in precision (See Section VI). We also exclude all unqualified non-compound method names that are not followed by a parenthesis because many of these terms occur in natural discourse, which increases the number of false positives. For example, we exclude the word 'execute' but include `HttpClient.execute` and `execute()`. Other researchers have found that while compound camel cased terms usually represent code elements, single word terms tend to be ambiguous [1], as a result stricter matching rules are justified when resolving unqualified single word terms.

Stage 4: Final output

ACER outputs the index of valid terms for the collection and the terms that are valid for each individual document. The index contains no one-off terms, but a document does contain one-off terms, which are extracted from valid Java constructs. For example, a developer may create a `Bank` class that sends information using HttpClient. If we exclude this `Bank` class, which is a one-off term, we will introduce a false negative.

We also associate the following additional features with each term: its kind (e.g., class, member), its context (e.g., it may have been an unqualified, ambiguous term), its frequency across the local, immediate, and collection contexts, and its location in the document. We use these features in Section VII to help classify the salience of a code element.

V. Data Source and Benchmark

StackOverflow is a Q&A forum for computer programming [9].[2] Developers ask and answer questions as well as vote on the quality of a post. Each question contains one or more tags that indicate its topic. We process all question and answer posts on StackOverflow, between August 2008 and September 2011, related to the following three project tags: HttpClient, Hibernate, and Android. These three project tags crosscut a diverse set of topics. Each project tag is associated 384, 1610, and 6366 distinct other tags, respectively. We choose three project tags to ease manual validation. Randomly sampling across all StackOverflow posts would have made manually validating code elements difficult because we would have needed a base knowledge of many APIs. Although all three projects are principally written in Java, they vary dramatically in domain and size. HttpClient provides an API for communicating with a server.[3] StackOverflow contains 1051 documents tagged with 'HttpClient'. Hibernate's primary function is to allow developers to model and persist Java objects in a relational database.[4] There are 26 695 documents tagged with 'Hibernate' on StackOverflow. Android is a mobile platform.[5] Developers

[2]http://stackoverflow.com
[3]http://hc.apache.org
[4]http://www.hibernate.org
[5]http://developer.android.com/about/index.html

can provide apps and games for Android users. StackOverflow contains 230 836 documents tagged with 'Android'.

Benchmark: From randomly sampled answer posts, we manually identify the code elements contained in each post and mark the code elements as salient or non-salient for the purpose of the post. Validating the code elements identified by ACER was much less time consuming than understanding the post and identifying the salient code elements. We stopped sampling when we had coded at least 300 code elements per project. For HttpClient, Hibernate, and Android we coded 70, 53, and 65 random answer posts which contained 325, 325, and 343 code elements. We found that we could not accurately code the salience of code elements in questions because developers indiscriminately dump stacktraces and code fragments; they do not know where to focus and often mistakenly assume a code element relates to their problem. As a result, we leave the salience of elements in question posts to future work and only code answer posts.

Our unit of analysis is the code element. We consider the following kinds of code elements: packages, annotations, types, methods, and fields. As with previous work, we do not consider variables to be code elements, but we use them as intermediaries when resolving members to their valid types. When ACER incorrectly resolves a variable, all of the code elements qualified by that variable are also invalid. Our guidelines for determining code element salience are discussed in Section VII.

VI. Evaluation of ACER

We evaluate ACER by processing documents on StackOverflow and answering the following questions.

1) Does ACER identify valid code elements with high precision and recall?
2) We considered unqualified compound types as valid. What impact does this special case have on precision and recall?
3) Can ACER process a large collection of documents? How many code elements can be an unambiguously identified using an island parser, how many must be resolved, and how many were dropped because they do not represent code elements?

Precision and Recall: We compare the code elements that we manually identify with those identified by ACER. There are three possible outcomes. A false positive (FP): ACER identifies an invalid code element. A false negative (FN): ACER fails to identify a code element. A true positive (TP): ACER identifies a valid code element. A misclassified code element is considered to be both a false positive and a false negative because ACER identified an invalid code element and failed to identify a valid code element.

ACER attains a high degree of precision and recall for each project. Using the standard formulas for precision and recall [10], the respective values for HttpClient are 0.96 and 0.92, for Hibernate are 0.91 and 0.91, and for Android are 0.90 and 0.86. Table II shows the breakdown by project for the true positives and false positives and false negatives associated with each kind of code element.

TABLE II
HttpClient: Precision = 0.96 and Recall 0.92, Hibernate: Precision = 0.91 and Recall 0.91, Android: Precision = 0.90 and Recall = 0.86

Project	HttpClient			Hibernate			Android		
	TP	FP	FN	TP	FP	FN	TP	FP	FN
Packages	14	0	1	18	5	0	9	3	1
Annotations	0	1	0	65	1	1	0	0	0
Types	167	7	12	127	13	15	150	18	13
Methods	112	5	9	72	6	8	118	11	31
Fields	6	0	4	14	6	5	18	1	3
Total	299	13	26	296	31	29	295	33	48

Special Case: During the manual inspection for the purpose of evaluation, we noticed that ambiguous, unqualified compound types (e.g.,`InputStream`) tend to represent valid code elements. ACER treats all unqualified compound types as valid elements. If we remove this special case for HttpClient, in the manually coded benchmark, we introduce 22 false negatives and 0 false positives, and, across the whole collection, we invalidate 287 terms. While precision remains unchanged, recall drops to 0.88 a decrease of 0.05. In the case of Hibernate, for our benchmark, we introduce 20 false negatives and 0 false positives, and invalidate 9.9K terms across the collection. Precision drops to 0.94, a one point decrease, while recall drops to 0.84, a decrease of 0.06. For Android, in the benchmark, we introduce 28 false negatives and eliminate 21 false positives and invalidate 185K terms across the whole collection. Precision increases to 0.96 a increase of 0.06, while recall drops to 0.78, a decrease of 0.08. For HttpClient and Hibernate, we confirm that the intuition that compound types tend to represent code elements is valid, and the special case introduces no new false positives in our benchmark. However, for Android, we do see an increase in false positives, but think that the decrease in false negatives justifies this special case.

Descriptive statistics: A term can be in one of three states: an unambiguous term identified by the island parser, an ambiguous, unqualified term that can be resolved to a code element in our index, and unresolved terms that are ambiguous but do not match a term in our index. We consider unambiguous and resolved terms to be valid, while unresolved terms are invalid and are dropped from ACER's final output. We report descriptive statistics for the entire collection of documents by project (not just the benchmark). Table III presents a breakdown of the terms by their kind. Most terms are types, followed by methods. Fields are proportionally the most ambiguous kind.

Across the 1K documents for the HttpClient project, ACER identified 17K valid code elements. There are 15K unambiguous code elements that can be extracted from Java constructs using the island parser. Of the 5.1K ambiguous terms, 2.1K can be resolved by ACER, while the 3K code-like terms do not represent valid code elements and remain unresolved.

Across the 27K documents tagged with Hibernate on StackOverflow, ACER identifies 355K valid code elements. The island parser identifies 300K unambiguous code elements. Of the 190K ambiguous code-like terms, 55K can be resolved to valid terms, and the remainder do not represent valid code elements. Hibernate uses many more annotations than the other

TABLE III
THE NUMBER OF UNAMBIGUOUS, AMBIGUOUS, AND RESOLVED TERMS FOR EACH PROJECT. THE NUMBER OF TERMS THAT ARE DROPPED BY ACER AS INVALID IS THE NUMBER OF AMBIGUOUS MINUS RESOLVED TERMS.

Project	HttpClient			Hibernate			Hibernate		
	Unambig.	Ambig.	Resolved	Unambig.	Ambig.	Resolved	Unambig.	Ambig.	Resolved
Package[6]	1.4K	NA	NA	53K	NA	NA	147K	NA	NA
Annotation	18	NA	NA	28K	NA	NA	1.5K	NA	NA
Type	6.6K	3.5K	1.3K	114K	148K	32K	820K	1.4M	430K
Method	3.5K	378	132	63K	25K	15K	518K	318K	212K
Fields	330	364	4	6.4K	8.3K	519	81K	60K	17K
Variable	2.7K	427	331	36K	9.4K	7.4K	381K	91K	74K
Total	15K	5.1K	2.1K	300K	190K	55K	2.0M	1.9M	733K

two project because annotations are used to indicate which classes represent entities in the database and the relationships between entities.

Across the 231K documents tagged with Android on StackOverflow, ACER identifies 2.7M valid code elements. The island parser identifies 2.0M unambiguous code elements. Of the 1.9M ambiguous code-like terms, 733K can be resolved to valid terms, and the remainder do not represent valid code elements. Android has a substantially larger number of documents and code elements than the other projects and shows that ACER can process a large collection of resources.[7]

Limitations: There is always the potential for bias in a manually created benchmark. In our benchmark, we found some cases were it was not obvious to which code element a term belongs. For example, a code fragment may contain an overridden method, but provide no indication of the class that declares the method. Despite searching the APIs of the systems involved in a post, when we could not reasonably link a term to a single API code element, we conservatively marked the element as a false positive, if ACER identified the term, or false negative, if ACER had not identified the term. We make our benchmark available for independent inspection.[8]

We chose diverse projects to test the generalizability of ACER. Since the island parser used in ACER is not intended to be a full implementation of the Java Language Specification, there is the possibility that some Java constructs are not parsed correctly. Any code element associated with incorrectly parsed constructs are marked as false positives or false negatives. Furthermore, the flexibility of the parser means that constructs in other languages that resemble Java constructs will be identified by our parser. For example, `Image.open()` could be a call to the Image class in Java or in python. Our parser is only intended to parse Java code, so non-Java code must be identified and removed. ACER currently eliminates HTML and XML code. Future work could involve extending ACER to recognize other languages.

Conclusion: ACER can extract code elements from a collection of documents with an average precision and recall of 0.92 and 0.90, respectively. While ACER's precision and recall are 4 and 6 percentage points lower than RecDoc's, we do not require apriori knowledge of which code elements are valid. This knowledge is difficult or impossible to obtain from informal documentation. Our accuracy is substantially better than the lightweight regular expression and information retrieval approaches (e.g., [3], [1]). With average precision and recall around 0.90, we can proceed to identify the code elements that are salient to a document.

[7]On a standard desktop computer, ACER processed the 231K documents associated with Android in under two days.

[8]Please see our benchmark at http://swevo.cs.mcgill.ca/icse2013rr

VII. SALIENT CODE ELEMENTS

A code element is *salient*, if it is central to an example code fragment or if there is some discussion defining its function or describing its use. For example, if an instance of a class is created to demonstrate the correct use of a method, then it is the method that is salient, not the class. However, if the class' function is being described and example methods listed, then it is the class that is salient and not the methods. The only exception is when a code element is obviously the solution to a developer's question and no further explanation is required (e.g., "A call to `getUsernamePasswordCredentials` should fix your problem"). There are two additional types of code elements that are part of our salience guideline.

Contextual code elements are not salient. Contextual elements are those that are repeated across many posts and are required before any system features can be used. For example, developers always need an instance of a class that implements *HttpClient*, so *HttpClient* is usually contextual.

Alternative code elements: Many posts provide advice about code elements that are alternatives to others being used as part of a task. Our definition of salience requires that alternative code elements are not only listed but are also exemplified in a code fragment or discussed in free-form text. In the example below, a developer describes why a code element should not be used, describes one that should work, and then mentions two others that the developer may need to examine. The first two code elements are salient. The last two are not salient because they are not described in this post. It is probable that they are better described in a different post. Replacing one code element with another is always salient information.

> `ClientConnectionManager` is not tread safe. You must use `ThreadSafeClientConnManager` in a multithreaded application. This class manages a pool of client connections. You may want to look at how this affects `BasicHttpParams` and `DefaultHttpClient`.

Benchmark: Using the guidelines above and the randomly sampled answer posts from Section V, we manually code the

salience of the code elements contained in each post. Of the 299 code elements in the HttpClient sample that were correctly classified by ACER, 89 are salient (30%). Of the 296 code elements for Hibernate, 80 are salient (27%). Of the 303 code element for Android, 106 are salient (35%). The code elements in the benchmark are the input to our J48 decision tree classifier.

A. Features

We use the following features to classify automatically code element salience. Table IV shows descriptive statistics only for features that are important predictors in our classifier.

TF-IDF: Term frequency (TF) and inverse document frequency (IDF) are common measures of term importance in a document [11]. In our context, TF is the number of times a code element appears in a document. IDF is the inverse of the number documents a code element appears across the collection of documents (i.e. the rarity of an element in the collection). We use logarithmic scaling because it is the standard way to to reduce the distorting effect of elements that are redundantly repeated in a single document [10]. TF-IDF is calculated according to the following formula:

$$tf\text{-}idf_{t,r} = (1 + log(tf_{t,r})) * log(\frac{N}{df_t})$$

where $tf_{t,r}$ is the number of times term t occurs in document r and df_t is the number of documents in the collection containing t. TF-IDF values are not easy to interpret on their own, so we do not display them in Table IV.

Kind: We want to know if the kind (e.g., class, field) of code element impacts its salience. Table IV shows that while types are the most common kind of element, the most salient elements are methods for HttpClient and Android. Hibernate is a special case because instead of using the Hibernate API, developers write application-specific code and annotate the code elements to indicate relationships in the database. Answers tend to revolve around creating the right relationships between one-off code elements, so annotations drive the discussion. The importance of annotations to Hibernate developers is reflected in Table IV, where 45% of annotations are salient.

Context: We differentiate between the code elements that are unambiguous and those that are ambiguous and need resolving. For those that need additional resolution, we record in which context, local, global, or collection context the required type is found. We also differentiate between one-off code elements and those that are in the index. The best predictor is the ambiguity of the code element. The context and index appear to have a consistently smaller impact on salience. In Table IV, we see that, proportionally, resolved code elements are more salient than unambiguous elements. For HttpClient, Hibernate, and Android 47%, 42%, and 61% of resolved types are salient compared to 23%, 24%, and 18% of unambiguous elements.

Location and text type: We measure the position of a code element in terms of the normalized offset in characters of an element in a document. If an element appears more than once in a document, we use the average value. We use text type to refer to a code element that occurred in free-form text, a code fragment, or in both. Text type gives the type of location in a document. Text type tends to be a good predictor. For HttpClient, Hibernate, and Android 51%, 40%, and 61% of code elements in free-form text are salient and 19%, 21%, and 14% of elements in code fragments are salient. For the three projects a relatively small number of elements appeared in both free-form text and code fragments: 4%, 7%, and 8%, respectively.

TABLE IV
FEATURES USED IN CLASSIFYING THE SALIENCE OF CODE ELEMENTS IN STACKOVERFLOW ANSWERS

	HttpClient		Hibernate		Android	
	Salient	Non-Sal.	Salient	Non-Sal.	Salient	Non-Sal.
Total	89	210	80	216	106	197
Package	4	10	5	18	1	12
Annotation	0	0	30	37	0	0
Type	40	127	26	144	54	139
Method	42	70	21	65	63	99
Field	3	3	4	23	8	16
Unambig.	52	169	58	186	33	151
Resolved	37	41	22	30	73	46
Free-form	43	42	33	50	69	43
Fragment	38	163	34	158	23	144
Both	8	5	13	8	14	10

B. Classifier

The input to the classifier is our manually labeled salience values and the extracted features for each code element. We create the following four J48 decision tree classifiers and run ten-fold cross validation on each. First, a general classifier based on TF-IDF. Second, a classifier based on domain-specific features, such as code element kind. Third a classifier that includes all features. Fourth, an optimal classifier that combines the top three strongest predictors to eliminate overfitting. Table V shows the precision and recall of each classifier.

TF-IDF classifier: While some documents will contain code elements with high TF-IDF scores, others will contain low TF-IDF scores. However, in both documents there are salient and non-salient code elements. As a result, absolute TF-IDF has low predictive power, so we normalize each TF-IDF score, by the maximum score for the document. In the remainder of this paper the term TF-IDF refers to normalized TF-IDF unless otherwise indicated.

$$ntf\text{-}idf_{t,r} = \frac{tf\text{-}idf_{t,r}}{max(tf\text{-}idf_r)}$$

Normalization by the maximum TF-IDF score has the effect of dividing code elements into their respective percentiles for a given document. For example, given three elements with distinct TF-IDF scores in a document, the one with the maximum TF-IDF will be in the top percentile, while the one with the lowest score will be in the bottom percentile, and the element with the middle score will be in the 50^{th} percentile. The decision tree does not necessarily make a split at a single percentile. For example, if all elements between the 25^{th} and 80^{th} percentile are salient and the others on non-salient, there would be three splits in the decision tree.

TABLE V
CLASSIFIER PRECISION AND RECALL

Project	HttpClient		Hibernate		Android	
	Prec.	Recall	Prec.	Recall	Prec.	Recall
TF-IDF (norm)	0.51	0.23	0	0	0.54	0.32
Domain-specific	0.61	0.35	0.69	0.30	0.74	0.65
All Features	0.65	0.36	0.59	0.33	0.71	0.56
Optimal	0.65	0.48	0.69	0.30	0.74	0.65

For HttpClient, Table V shows a precision and recall of 0.51 and 0.23. Code elements above the 95^{th} percentile tend to be salient. For Hibernate, TF-IDF and normalized TF-IDF do not have any predictive power. We suspect this is due to the heavy use of annotations to indicate relationships in a Hibernate database. For Android, code elements above the 85^{th} percentile tend to be salient, with a precision and recall of 0.54 and 0.32.

Domain specific features are code element kind, context, ambiguity, inclusion in the index, position, and text type (i.e. in free-form or a code fragment). For HttpClient, Hibernate, and Android, we see respective precision and recall values of 0.61 and 0.35, 0.69 and 0.30, and 0.74 and 0.65. However, the tree appears to be overfitted as there is no reasonable explanation for each split in the tree. As can be seen in Table V, combining TF-IDF with the other features has a minor impact on precision and recall, but does not make the interpretation of the classifier's output easier to understand.

Optimal classifier: We select the three features that are closest to the root of the decision tree in the classifier that contains all features and create an optimal classifier for each project. For HttpClient, this classifier has the highest precision and recall 0.67 and 0.47 and the resulting decision tree is small enough to explain. The strongest predictor is the text type. Code elements that are only in code fragments tend not to be salient. While methods contained in free-form text tend to be salient, classes tend to be salient if they are ambiguous and non-salient if they are unambiguous. Unambiguous types tend to qualify a method (e.g.,`HttpClient.execute()`), which means that they are usually contextual. Since ambiguous types appear on their own and do not qualify other code elements, they tend to be salient. TF-IDF is dropped from the classifier.

For Hibernate, the strongest predictors are text type, kind, and term context. Code elements that are only in free-form text tend to be salient if they are annotations or if they appear in the collection context. Code elements that appear in both are salient if they are annotations and not otherwise salient. Elements in code fragments tend to be non-salient.

For Android, the strongest predictor is term ambiguity followed by text type. While we included element kind, it was eliminated when we generated the classifier. Like the previous two projects, code elements that are only in code fragments tend to be non-salient. Code elements in free-form text that are ambiguous tend to be salient.

Limitations: Unlike manually resolving terms to their respective code elements, classifying the salience of code elements is less formally defined, which results in a greater potential for bias. To counteract this bias, we use a straightforward but comprehensive guideline to classify the salience of code elements in answer posts. The majority of answers on StackOverflow are short and to the point. A small number contain large multifaceted answers and sometimes contain followup questions. Code elements in followup questions are classified as non-salient to an answer post.

Code elements may be referred to indirectly during a discussion. For example in "Use the setter/getter on AbstractEntity instead of accessing the id field directly". We know that the code elements `AbstractEntity` and `AbstractEntity.id` are salient elements, but we miss the indirectly referenced, salient code element `AbstractEntity.getId()`, `AbstractEntity.setId()`. ACER is not designed to capture conceptual references to code elements. Identifying indirectly referenced code elements is left to future work. One possibility would be to capture conceptual knowledge by analyzing the natural language text surrounding code elements.

C. Discussion

Our intuition had been that TF-IDF would be a good predictor of code element salience in a document because we expected salient elements to be described in free-form text and their use illustrated in a code snippet (i.e. high TF). We expected non-salient elements to be contextual and, as a result, to be repeated in many different documents (i.e. low IDF). We had planned on supplementing TF-IDF with other features to increase the overall predictive power of our classifier. However, TF-IDF was outperformed by domain-specific features and was eliminated from our optimal classifier. This finding was unexpected because TF-IDF tends to be a good general predictor of term importance in documents [11]. Why then was it eliminated from our classifier?

Initially, we had been concerned that one-off terms might have a negative impact on the predictive power of TF-IDF because they have high IDF values but tend to be non-salient. We had included a binary factor which indicated whether a term was a one-off. However, there was no branch in the decision tree that combined one-off terms and TF-IDF. We also created a classifier that contained only these two features. For HttpClient and Hibernate, this change had a negligible effect on precision and recall when compared to the classifier that just included TF-IDF. With Android there was an increase of 0.09 for precision and a decrease of .04 for recall. In all cases, the accuracy was well below the optimal classifier.

While creating our benchmark, we observed, especially on HttpClient which has fewer total posts, that questions with similar salient code elements in the answer were being asked repeatedly. This repetition meant that salient code elements were less rare than they would be in a tutorial which contains less repetition. Repetition is not necessarily a negative feature of Q&A sites because the repetition may show, for example, how to use the salient element in a different context. Future work is necessary to determine the impact of similar answers on TF-IDF's predictive power.

TF-IDF is an approximation of term importance, so perhaps it is not surprising that it was outperformed by domain-specific

features. For all three projects text type (whether a code element was in free form text or a code fragment) was a strong predictor. For Android and HttpClient, the classifier was strengthened by the term ambiguity. Terms that were in free-form text and were ambiguous (i.e. unqualified terms) tended to be salient because the author had specifically isolated these elements and surrounded them by descriptive text. Qualified methods in free-form text on HttpClient were also important for the same reason. For Hibernate, elements in free-form text, especially annotations, were particularly important.

Qualitative observations reveal why code elements in free-form text tend to be salient. Developers often described the code elements that were salient to a problem in free-form text and, instead of providing a code fragment to illustrate the solution, they provided a link to a code example or tutorial on another site. This allowed developers to discuss the specifics (i.e. the salient aspects) and to refer to another existing example for the details (i.e. the context). The trend was especially common on Android, which has the highest number of salient free-form text code elements (See Table IV). An investigation of the links to tutorials and other forms of documentation from informal Q&A sites is an interesting avenue for future work.

Irrelevant code elements are not only non-salient, they also adversely affect the focus and clarity of a post. For example, developers asking a question are often confused, so they tend to dump stacktraces and large code fragments into question posts. These dumps have a lower proportion of salient and contextual code elements than a well focused code example. The number of irrelevant elements in question posts were the reason why we could not develop a guideline to code them. In contrast, in an answer post, a non-salient code element is rarely irrelevant and usually is an alternative element or a contextual element. The elements contained in code fragments were labeled as non-salient by our classifier for all three projects. Non-salience does not mean that the code elements are irrelevant. We conclude that salient code elements are usually contained in free-form text, while the contextual elements, that are necessary details when solving a problem, are most often contained in the code snippet.

VIII. Conclusion

Unlike tutorials and Javadocs, StackOverflow contains a large, diverse set of questions and answers. A developer reading an answer on StackOverflow can quickly understand the context and get to its salient features. However, there are 139K answers and 569K code elements related to Android on StackOverflow. To help developers identify the salient code elements in such a large collection of documents we developed ACER, which identifies code elements, and a classifier, which identifies the salience of the code elements to a post.

ACER uses an island parser to identify code elements in documents. The advantage of this approach is that unlike previous work, it does not rely on an index of valid elements parsed from the source code of a particular system. Instead, it identifies code elements in Java constructs and creates an index of valid elements based on the elements contained in the collection of documents. ACER reparses each document extracting unqualified, ambiguous terms and resolves them to their corresponding code elements by using the term's context. Parsing a diverse sample of documents that contains over 7058 distinct tags on StackOverflow, we obtain an average precision and recall of 0.92 and 0.90. The close-world assumption [2] can be discarded with a minimal sacrifice in accuracy.

This paper introduces the notion of code element salience in developer communication and provides a technique to classify elements as salient or non-salient. Previous work assumed that each element was of equal importance. To determine the salience of the identified elements, we manually coded a random sample of documents. Based on this benchmark, we created a classifier of code element salience. The best predictors are domain specific-features. These features indicate whether an element is contained in free-form text or a code fragment, and whether it is unambiguous or needs resolving. TF-IDF was dropped from our optimal classifier. The classifier managed a precision of 0.65–0.74, and a recall of 0.30–0.65. Identifying salient code elements adds an important but complex dimension to code element traceability in documents. Despite this complexity, our results are at least as good as those seen in early work on linking code elements to source code [3]. While improvements are probable, we feel that the performance of our initial infrastructure is sufficient to justify experimentation with applications such as advanced search tools.

References

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

[2] B. Dagenais and M. P. Robillard, "Recovering traceability links between an API and its learning resources," in *Proceedings of the 34th ACM/IEEE International Conference on Software Engineering*, 2012, pp. 47 –57.

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

[4] A. Marcus and J. Maletic, "Recovering documentation-to-source-code traceability links using latent semantic indexing," in *Proceedings of the 25th ACM/IEEE International Conference on Software Engineering*, 2003, pp. 125–135.

[5] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs."

[6] L. Moonen, "Generating robust parsers using island grammars," in *Proceedings of the 8th IEEE Working Conference on Reverse Engineering*, 2001, pp. 13–22.

[7] A. Van Deursen and T. Kuipers, "Building documentation generators," in *Proceedings of the IEEE International Conference on Software Maintenance*, 1999, pp. 40–49.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha, *The Java Language Specification: Java SE 7 Edition*, 2012.

[9] L. Mamykina, B. Manoim, M. Mittal, G. Hripcsak, and B. Hartmann, "Design lessons from the fastest q&a site in the west," in *Proceedings of the International Conference on Human factors in Computing Systems*, 2011, pp. 2857–2866.

[10] C. Manning, P. Raghavan, and H. Schutze, *Introduction to Information Retrieval*. Cambridge University Press Cambridge, 2008.

[11] S. Robertson, "Understanding inverse document frequency: on theoretical arguments for IDF," *Journal of Documentation*, vol. 60, no. 5, pp. 503–520, 2004.