Integrating Software Project Resources Using Source Code Identifiers

Laura Inozemtseva, Siddharth Subramanian and Reid Holmes
University of Waterloo
Waterloo, ON, Canada
lminozem,s23subra,rtholmes@uwaterloo.ca

ABSTRACT

Source code identifiers such as classes, methods, and fields appear in many different contexts. For instance, a developer performing a task using the `android.app.Activity` class could consult various project resources including the class's source file, API documentation, issue tracker, mailing list discussions, code reviews, or questions on Stack Overflow.

These information sources are logically connected by the source code elements they describe, but are generally decoupled from each other. This has historically been tolerated by developers, since there was no obvious way to easily navigate between the data sources. However, it is now common for these sources to have web-based front ends that provide a standard mechanism (the browser) for viewing and interacting with the data they contain. Augmenting these front ends with hyperlinks and search would make development easier by allowing developers to quickly navigate between disparate sources of information about the same code element.

In this paper, we propose a method of automatically linking disparate information repositories with an emphasis on high precision. We also propose a method of augmenting web-based front ends with these links to make it easier for developers to quickly gain a comprehensive view of the source code elements they are investigating. Research challenges include identifying source code tokens in the midst of natural language text and incomplete code fragments, dynamically augmenting the web views of the data repositories, and supporting novel composition of the link data to provide comprehensive views for specific source code elements.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments

General Terms

Human Factors

Keywords

Traceability, semantic links, Newton

ICSE Companion'14, May 31 – June 7, 2014, Hyderabad, India
ACM 978-1-4503-2768-8/14/05
http://dx.doi.org/10.1145/2591062.2591108

1. INTRODUCTION

An Android developer performing a task might consult the source code for a class on `github.com`, its API documentation on `developer.android.com`, its discussions on `groups.google.com`, its code reviews on `android-review.googlesource.com`, its issues on `code.google.com`, and related development questions on `stackoverflow.com`. Unfortunately, each of these resources is independent: there is no easy way to navigate directly between the repositories, nor is it possible to combine the data they contain.

The disconnect between these data silos makes it more difficult for developers to build an understanding of the source code elements that are relevant to their development task. Since developers spend a considerable amount of time navigating through these web-based resources [8] and reading and navigating source code [10], the effects of small barriers to understanding and "microinterruptions" as they move between data silos are magnified. Section 2 describes several motivational scenarios that make this problem more concrete.

Previous work has attempted to solve this problem by identifying traceability links between different project artefacts. However, most previous work cannot identify references to source code elements in natural language text with high precision. Moreover, existing systems often cannot uniquely identify the code elements they find, making it impossible to do high precision linking. Section 3 describes how our work differs from previous projects in more detail. It also describes our previous work in this area [12].

Our goal is to find references to source code identifiers in structured and unstructured text with high precision, allowing us to find *semantic links* between artefacts that discuss the same code element and are thus immediately relevant to one another. We will augment existing web front ends with links to related resources, harnessing the power of hyperlinking without requiring the data silos to change. This will be done through HTML injection and dynamic page augmentation. We will also use semantic link data to support code search and to build composite pages for source code elements that describe the diverse set of repositories involved in the evolution of an element. Section 4 describes our proposed tool, NEWTON.

Concretely, our new idea is as follows: *using source code identifiers to integrate disparate data silos will permit easy navigation between these silos, supporting program comprehension and development tasks.*

2. MOTIVATING SCENARIOS

To illustrate the many ways that semantic links between project artefacts would be useful, we provide the following three motivating scenarios.

- Code review is typically facilitated with online systems. In these systems, developers are presented with a diff-style view of the change being made. In one such diff, an extra parameter was added to the `WifiConfigController` constructor.[1] Existing approaches would not link the `WifiConfigController` reference to either its documentation or its source code[2]. In contrast, NEWTON could identify these links, enabling the developer to easily navigate to these resources if they needed more contextual information for their task. NEWTON would also make it possible to click on the call to the `WifiConfigController` constructor in the code review system and see all code reviews pertinent to this method.
- Developers frequently use mailing lists to discuss feature changes and coordinate their activities. These technical discussions often include source code fragments and reference various APIs. For example, on one thread of the Blink mailing list, developers discussed two different `v8::Context` methods.[3] By linking this message thread to the two API methods, developers reading the thread could easily navigate to the relevant source code and documentation. The documentation could also be augmented with links to the mailing list, since the discussion on the list provides valuable insight into these methods.
- Stack Overflow posts often contain many source code fragments. For example, one answer explains how to reset an Android Chronometer.[4] Unfortunately, `mChronometer` is not declared in the snippet, making it impossible for existing approaches to link the example to the `Chronometer` class. If a developer reading this snippet were interested in the official documentation, they would have to infer the link to `Chronometer` and initiate a new search to find the documentation. NEWTON would provide one-click access from the Stack Overflow post to the documentation and vice-versa.

The scenarios described above represent common development tasks. In each case, developers would benefit from being able to navigate from the source code to a reference to that code in a repository, from the repository to the source code, and between relevant repository results in order to build a full understanding of the element under investigation.

3. RELATED WORK

Several previous studies have tackled the problem of traceability between multiple repositories. However, they suffer from one of two weaknesses: either they cannot handle natural language documents, such as emails and forum posts, or they cannot uniquely identify the code elements they find, reducing the precision of the identified links. Both weaknesses would greatly reduce the usefulness of our proposed tool.

As an example, Fritz and Murphy [6] combine information from multiple repositories to answer developers' questions. However, they rely on the items in the repositories having unique identifiers that can be easily extracted (e.g. a commit id) and cannot process natural language documents.

Dagenais and Hendren [4] performed partial type inference for Java code but their approach requires a complete source code file as input. It cannot handle natural language text.

Hipikat [3] recommends artefacts from a project's history that might be relevant to the current development task. However, it does not consider the source code of the project, so it cannot be used for the kind of linking we want to do.

Venolia's tool [13] identifies allusions to software artefacts in natural language prose. However, it accomplishes this through simple regular expressions, reducing recall, and can only obtain the fully qualified name of an element if the developer writes out the full name, e.g. `android.app.Activity`.

Bacchelli et al. [1] attempted to establish links between emails and the source code artefacts they discuss. They tried to identify source code elements in the emails using regular expressions, a vector space model and latent semantic analysis. However, all three approaches had very low precision and could not identify fully qualified names.

CodeBook [2] supports a large variety of different repositories, but uses regular expressions to look for references to source code elements in natural language text.

Similarily, RecoDoc [5] supports a variety of different repositories but uses Bacchelli et al.'s approach to look for references to source code elements.

Panichella et al. [7] tried to automatically extract method descriptions from developer communications such as emails and bug trackers. However, they are only able to map a communication to a class when the fully-qualified name or the filename is given.

ACE [9] uses an island grammar to identify elements, which leads to increased precision but again cannot retrieve the fully qualified name.

It is clear that our tool, to be successful, must be able to identify references to source code elements in natural language text and resolve those references to the fully qualified name of the element. As we will discuss in the next section, we previously developed a tool that is capable of doing the latter [11, 12]. However, it is currently limited to structured text, specifically, the `<code>` blocks of Stack Overflow posts. Our primary research goal is therefore to make the tool more general by adding the ability to parse natural language documents. Our preliminary results with conditional random field models are promising.

4. SURFACING SEMANTIC LINKS

Our approach consists of three steps:

Structural element identification. First, a list of source code identifiers must be generated for a project of interest. These can generally be extracted from readily accessible project resources, such as jar files for Java or JS files for JavaScript.

Semantic link detection. Next, project silos are examined using standard web spider techniques. The text on the identified pages is compared to the structural names to identify relevant links. Natural language processing techniques are required to identify source code tokens in unstructured text.

[1]The example can be found at `http://goo.gl/vDX15u`
[2]Found at `http://goo.gl/qUUnVd`
[3]`http://goo.gl/NEGuLe`
[4]`http://stackoverflow.com/a/5345994`

Data display. Finally, identified links can be used to inject relevant content into HTML pages as developers navigate through online resources. Composite pages can also be generated for all structural identifiers enabling developers to gain a complete understanding of a given element.

Our initial prototypes [11, 12] have focused on the first two of these three steps. We will describe each step in more detail in the following sections.

4.1 Structural Element Identification

The list of structural elements acts as an oracle to guide the semantic link detection process. Generating the list of identifiers is generally straightforward for interpreted languages and languages that are compiled to bytecode. Our prototype, BAKER [12],can automatically generate identifier lists for Java (from jar files) and for JavaScript (from the source). Any fully-qualified name that can be referenced in source code is extracted.

A global identifier set is maintained in addition to project-specific identifiers. For example, if one is identifying semantic links in StackOverflow, the global list must be used as API calls from a variety of projects may be present. However, when identifying semantic links for a specific project repository (e.g., `https://android-review.googlesource.com`), only the identifiers expected to be present in that repository need be used. While past approaches have argued that the closed world hypothesis is invalid [9], for any specific project it is not unreasonable to expect that a list of structural identifiers for the project can be generated.

4.2 Semantic Link Detection

To identify semantic links, we must identify code-like tokens in both structured and unstructured text. Structured text includes, for example, the `<code>` blocks in Stack Overflow posts, which are known to contain source code. Unstructured text includes, for example, email threads that refer to the `Activity` class without ever explicitly indicating that a code element is being discussed.

BAKER is able to correctly determine the fully-qualified identifier of a source code token in Stack Overflow `<code>` blocks with 96% precision [11, 12]. The tool currently works for Java and JavaScript. The mapping step relies on two oracles containing 20 million fully qualified names that were automatically built from the Maven repository (for the Java oracle) and a large set of non-obfuscated JavaScript files (for the JavaScript oracle). While the detailed operation of the tool will not be repeated here, it is worth mentioning that parsing source code snippets is imprecise because the snippets are generally not valid code. For example, imported code is generally not available, variables are often used without being defined, and parts of the code may be elided (`var = ...`). In addition, identifying fully-qualified names is difficult due to frequent naming collisions; for example, there are 58 methods with the short name `addHistoryListener` in the Maven repository. Finally, some languages make identifying API references challenging. In JavaScript, some data flow analyses are required along with control flow analyses to identify semantic links with high precision.

BAKER achieves its high precision in spite of these issues by flexibly querying the oracle for types, methods, and fields that match the constraints given in code snippets. For example, while 58 types have `addHistoryListener` methods, only one type also has a `getToken` method. This deductive approach allows us to identify the fully qualified names of many tokens even when we are only provided with a short name as input.

While BAKER demonstrates that our deductive approach can work effectively for both object-oriented and dynamic languages, to fully realize the vision of this project, NEWTON must have the additional ability to extract code-like tokens from unstructured text. We have tried using supervised machine learning to learn the parameters of a conditional random field model given a document corpus. Our results, while very preliminary, are promising: we have managed to obtain fairly high precision (0.77). That said, there is still much room for improvement. This will be the first step we take to continue this project.

4.3 Data Display

BAKER demonstrates that it is possible to create semantic links based on source code identifiers and code-like terms with relatively high precision. The most exciting aspect of this project is using these links to improve the development experience. We envision two primary ways of seamlessly integrating this information into the systems developers already use.

Dynamic page injection.

The simplest form of integration would employ a browser extension to dynamically augment any web page containing source code references with links back to the structural identifiers they contain.

The naming hierarchy followed by API elements is usually maintained in the URLs of the web-based front ends that host related resources. For instance, the API documentation for the `BaseInterval` class in the `org.joda.time.base` package of the JodaTime library is located at `joda-time.sourceforge.net/api-docs/org/joda/time/base/BaseInterval.html`.

Since we have access to a complete list of the fully qualified names of the elements in a library, we can build a set of curated web searches, each targeted to locate one web-based resource, and identify the top 10 returned URLs per query that maintain the naming hierarchy. The base URL for each web-based resource associated with a library can be extracted by identifying the the longest common prefix in the returned URLs over all API elements in the library. Uninteresting resources including code browsing websites like Grepcode[5] that often show up in the search results can be discarded by maintaining a blacklist of domain names.

Since BAKER (and thus NEWTON) can identify fully qualified names, it is straightforward to generate and inject links to various resources accordingly. For example, Figure 1 shows how NEWTON could augment part of a patch in a code review system. When the developer hovers over `getSystemService`, they would see links other resources that refer to this method.

The hyperlinking already present in IDEs to navigate between structural code elements could also be augmented with this data, enabling direct linking to web-based resources from within the source code. This latter feature would rely on the IDE's AST to identify which tokens should be augmented.

[5]`www.grepcode.com`

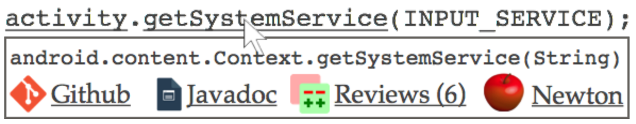
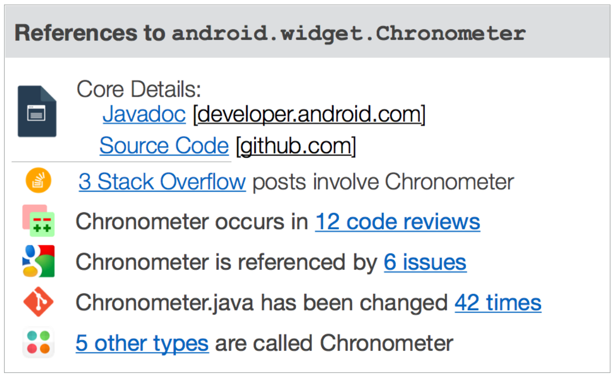

```
activity.getSystemService(INPUT_SERVICE);
android.content.Context.getSystemService(String)
```

Figure 1: Example of page injection into an online code review snippet. The hyperlink on `getSystemService` is dynamically injected into the web page; when the developer hovers over the link the fully qualified method name is shown along with links to other relevant repositories.


References to `android.widget.Chronometer`

Core Details:
Javadoc [developer.android.com]
Source Code [github.com]

3 Stack Overflow posts involve Chronometer

Chronometer occurs in 12 code reviews

Chronometer is referenced by 6 issues

Chronometer.java has been changed 42 times

5 other types are called Chronometer

Figure 2: An example of a composite view that could be presented in response to a search for `android.widget.Chronometer`.

Composite pages.

By combining the semantic link data for a specific source code identifier from many different artefact repositories, a composite element page can be dynamically generated. This page could either be standalone, much like a Javadoc page, or could be presented in the form of a widget embedded in some other context. For example, Figure 2 shows a faceted search view that could be presented at the top of a source code search query, much as Google search offers unit conversions or other augmented data. This same view could also be embedded within an IDE to be displayed whenever a developer hovers over an API call. Composite pages could be built that incorporate vast amounts of detail about a source code element. For instance, such a page could include snippets from the documentation, details about all of the times the element has been changed in the version control repository, active and past code reviews for identifying design rationale, related issues, and a collection of code examples from Stack Overflow that demonstrate the use of the API.

5. CONCLUSION

Many development tasks affect resources that are not co-located with the code. These resources, such as documentation, mailing lists, issue trackers, and version control repositories, among others, are stored in separate silos; however, they are conceptually linked by the source code elements they contain. Since these resources are often accessed through web-based interfaces, we can expose the conceptual links between resources by identifying structural identifiers in the text stored within the resources and augmenting their web interfaces with this information.

In this paper we proposed NEWTON, a platform that will use this linking approach on a variety of web-based development repositories to increase the accessibility of the information in these repositories. By unobtrusively augmenting the systems developers are already using, we hope to assist them with their day-to-day programming tasks.

6. REFERENCES

[1] A. Bacchelli, M. Lanza, and R. Robbes. Linking e-mails and source code artifacts. In *Proceedings of the International Conference on Software Engineering*, 2010.

[2] A. Begel, Y. P. Khoo, and T. Zimmermann. Codebook: Discovering and exploiting relationships in software repositories. In *Proceedings of the International Conference on Software Engineering*, pages 125–134, 2010.

[3] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth. Hipikat: A project memory for software development. *Transactions on Software Engineering*, 31(6):446–465, 2005.

[4] B. Dagenais and L. Hendren. Enabling static analysis for partial Java programs. In *Proceedings of the Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, pages 313–328, 2008.

[5] B. Dagenais and M. P. Robillard. Recovering traceability links between an API and its learning resources. In *Proceedings of the International Conference on Software Engineering*, 2012.

[6] T. Fritz and G. C. Murphy. Using information fragments to answer the questions developers ask. In *Proceedings of the International Conference on Software Engineering*, 2010.

[7] S. Panichella, J. Aponte, M. Di Penta, A. Marcus, and G. Canfora. Mining source code descriptions from developer communications. In *Proceedings of the International Conference on Program Comprehension*, 2012.

[8] C. Parnin, C. Treude, L. Grammel, and M.-A. D. Storey. Crowd documentation: Exploring the coverage and the dynamics of API discussions on Stack Overflow. Technical Report GIT-CS-12-05, Georgia Tech, 2012.

[9] P. C. Rigby and M. P. Robillard. Discovering essential code elements in informal documentation. In *Proceedings of the International Conference on Software Engineering*, pages 832–841, 2013.

[10] M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *Transactions on Software Engineering*, 30(12):889–903, 2004.

[11] S. Subramanian and R. Holmes. Making sense of online code snippets. In *Proceedings of the Working Conference on Mining Software Repositories*, pages 85–88, 2013.

[12] S. Subramanian, L. Inozemtseva, and R. Holmes. Live API documentation. In *Proceedings of the International Conference on Software Engineering*, 2014.

[13] G. Venolia. Textual allusions to artifacts in software-related repositories. In *Proceedings of the International Workshop on Mining Software Repositories*, pages 151–154, 2006.