Towards faster method search through static ecosystem analysis

Boris Spasojević
University of Bern
Hochschulstrasse 4
CH-3012 Bern, Switzerland
spasojev@iam.unibe.ch

Mircea Lungu
University of Bern
Hochschulstrasse 4
CH-3012 Bern, Switzerland
lungu@iam.unibe.ch

Oscar Nierstrasz
University of Bern
Hochschulstrasse 4
CH-3012 Bern, Switzerland
oscar@iam.unibe.ch

ABSTRACT

Software developers are often unsure of the exact name of the method they need to use to invoke the desired behavior in a given context. This results in a process of searching for the correct method name in documentation, which can be lengthy and distracting to the developer.

We can decrease the method search time by enhancing the documentation of a class with the most frequently used methods. Usage frequency data for methods is gathered by analyzing other projects from the same ecosystem – written in the same language and sharing dependencies.

We implemented a proof of concept of the approach for Pharo Smalltalk and Java. In Pharo Smalltalk, methods are commonly searched for using a code browser tool called "Nautilus", and in Java using a web browser displaying HTML based documentation – Javadoc. We developed plugins for both browsers and gathered method usage data from open source projects, in order to increase developer productivity by reducing method search time.

A small initial evaluation has been conducted showing promising results in improving developer productivity.

Categories and Subject Descriptors

D.2.2 [**Design Tools and Techniques**]: Software libraries; D.2.7 [**Distribution, Maintenance, and Enhancement**]: Documentation; D.2.9 [**Management**]: Productivity

General Terms

Documentation

Keywords

Method search, ecosystem analysis, static analysis

1. INTRODUCTION

Libraries enable software developers to reuse implementations of common tasks and thus provide a large productivity boost. The libraries expose an application program interface (API) to the developer. Usage of APIs is one main relationship between the different projects in a software ecosystem.

In object-oriented languages an API consists of a set of classes whose instances encapsulate state and behavior provided by the library. The developer invokes methods on these objects to trigger desired behavior. Choosing the right method to invoke is often not a trivial task especially if the developer is not familiar with the API [15]. We call the process of finding the desired method "method searching". In a survey about developer needs in an ecosystem, Haenni *et al.* have shown that an important motivation for downstream users' needs is API understanding [2].

The APIs are usually documented (through explicit documentation or availability of source code) in order to help the developer understand how to use them. Looking through the documentation is a tiresome task and this is typically made worse by the fact that most object-oriented code is documented on a class by class basis, meaning that a method which can be invoked on an object of a certain class might be declared somewhere higher in the class hierarchy. Faster method search would lead to more productive developers

We propose to improve the current process of searching for methods by augmenting the data available to the developer with information derived from the analysis of the ecosystem to which the system belongs. In this paper we consider a purely technical perspective [7] of a software ecosystem as it has been done before [8, 4].

Our interest when approaching the ecosystem is to run a lightweight static analysis on the source code of all the projects that are mutually dependent or are connected through common dependencies.

We assume that the ecosystem can function as a context from which, by data mining the source code, we can discover patterns of usage for a given API. These patterns that would be unobtainable from individual projects. We can then augment documentation with information about which methods of a particular class (including the ones declared in the hierarchy chain) are more frequently used in other projects. We aim to incorporate this information in a way that does not require the developer to alter her normal development process. This means that frequency of use data needs to be presented to the developer via the same set of tools she normally uses for method search.

To gather the information on method usage frequency we propose an analysis which, for each encountered class and across all projects, produces the number of times methods of the class has been invoked on an instance of that class. The

To appear in ECSAW August 25 - 29 2014, Vienna, Austria

more times a method has been invoked by other developers the higher the probability that it is the goal for the developer searching for a method.

Even though this approach does not take in to account the context in which developer is working, our running hypothesis is that we can improve the method search process even if using only a lightweight static analysis approach.

We implement our approach as a proof of concept for two open source ecosystems written in Pharo Smalltalk[1] and Java. The two languages were selected as representatives for dynamic and static typing respectively, as implementing the approach is slightly different based on the type system of the language. We analyze 109 projects written in Pharo Smalltalk and 111 projects written in Java to generate the frequency of use data. Method search in Pharo Smalltalk is most commonly done using the built in system browser called "Nautilus[2]", and Java developers most commonly use HTML based documentation (*i.e.*, Javadoc) viewed via a web browser.

We develop plugins for Nautilus and for the Chromium[3] web browser to provide the data gathered from other projects. Using plugins enables us to provide the desired data without intruding on the method search process of the developer.

Figure 1 shows the architecture of our solution. A strong distinction is made between the server side components, discussed in detail in section 2, and the client side component discussed in section 3. The client-server architecture stems from the need to provide the client with up to date information, and enable different presentations of data, depending on the method search process of the developers. The server side components offer a service, and are unaware of the client providing data to the developer.

Information about the small evaluation of the approach is given section 4. Related work is discussed in section 5 and in section 6 and section 7 we focus on future work and conclusions respectively.

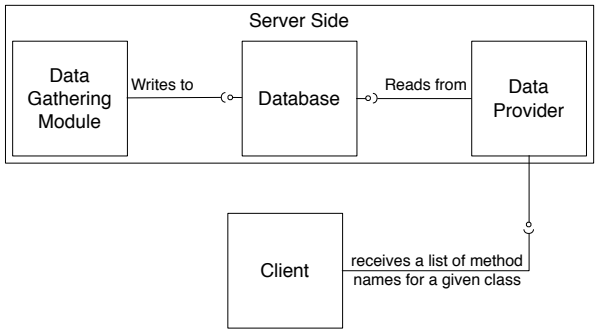

Figure 1: Architecture of the proposed solution.

2. ECOSYSTEM ANALYSIS

In order to provide the developer with the most commonly used methods of the class we first have to determine what those methods are. This information can be obtained by analyzing the source code of all the projects in the ecosystem. The module in charge of this analysis is marked as "Data gathering module" on Figure 1.

To keep the data up to date (*i.e.*, to react to introduction of new APIs to the ecosystem) this module should be re-run periodically, on be able to update the information in another way. The proof of concept does not support this at the time or writing this.

The result of this analysis is a set of triplets

$$(c, m, n) \tag{1}$$

stored in a database.

The triplet encodes the following information: in all analyzed projects the method m was invoked[4] n times on an instance of class c.

The triplets can be grouped by a given class c, which means that the number n, associated with method m summarizes the frequency of use of that method in the context of class c.

The details of the implementation of this module for statically and dynamically typed languages are given in subsection 2.1 and subsection 2.2 respectively.

A summary of the results of the data gathering module for Java (statistically typed) and Smalltalk (dynamically typed) is given in Table 1.

We use a MongoDB[5] database to store the data. MongoDB stores JSON documents, and the triplets are grouped by class before storing, resulting in documents similar to the following example[6]

```
{
 "_id" : ObjectId("51b0df6b44b1392c8e7ee0ec"),
 "className" : "Mutex",
 "selectors" :
 {
  "critical:" : 2,
  "ifNil:" : 1
 }
}
```

The triples are grouped by class name which means that each entry in the database is a different class for which frequently used methods can be provided.

The data provider shown in Figure 1 is a REST server that exposes a simple HTTP API for clients to provide the name of a class and receive a list of methods sorted based on frequency of use (n from the triplets).

2.1 Statically typed languages

In statically typed languages information on types of variables is available in source code so creating the triplets for the database consist only of parsing the source code and counting the method invocations.

As a representative of statically typed languages we chose Java for our proof of concept implementation. To run the analysis we used Pangea[7], a tool for running language independent analyses on corpora of object-oriented software

[1] `http://www.pharo-project.org/`
[2] `http://smalltalkhub.com/#!/~Pharo/Nautilus`
[3] `http://www.chromium.org/`
[4] We use the term "method invocation" to describe a call site in the source code, not a run-time invocation
[5] `http://www.mongodb.org`
[6] In Smalltalk jargon, a *selector* is the name of a message, *i.e.*,ifNil: or critical:, used to select the *method* to respond to the message.
[7] `http://scg.unibe.ch/research/pangea`

Language	Projects analyzed	Database entries (Classes)
Java	111	101,844
Smalltalk	109	3,332

Table 1: Summary of the results for both implementations of the data gathering module

projects. The corpus used for the Java case study is QualitasCorpus [16] version 20120401r which contains 112 systems written in Java[8].

We use QualitasCorpus as a snapshot of a software ecosystem because all the projects in the QualitasCorpus share dependencies towards a set of libraries, and some depend on other projects from the QualitasCorpus.

Pangea uses the VerveineJ[9] exporter to extract information from the source code of Java systems and export it as a FAMIX [19] meta model, and Moose[10] to analyze the data. The FAMIX meta model explicitly models method invocations, so gathering invocation data in Moose is just iterating over all invocations, exporting the class-method pairs, and summing them to form the triplets for the database.

The result of the data gathering module for Java is 101,844 entries in the database. This means that the approach can provide frequently used methods for 101,844 different classes.

Since the data collector gathers information from all invocations this also includes classes whose scope is limited to one project and are not visible in the ecosystem. This means that there is a small probability that any developer would need to search for methods of these classes. Identifying these classes does not bring any significant benefit to the approach (other than a potential performance boost due to less data in the database) so we leave this data as part of the database.

2.2 Dynamically typed languages

In dynamically typed languages information on types of variables is not available in source code. Because of this, creating the triplets for the database is slightly more complicated.

To solve this problem we run a type inference engine [12] on all instance variables of all analyzed projects, and only store to the database when a type of a variable can be unambiguously inferred. This results in a much smaller database when compared to the database built for statically typed languages.

Smalltalk is highly reflective language [14]. This enables the entire data gathering analysis to be implemented in Smalltalk itself.

The corpus used is a set of 109 projects from the Pharo Smalltalk open source ecosystem that we have already studied before [5]. The configuration browser is a tool to automatically load Smalltalk project source code and dependencies, similar to Maven[11] for Java.

[8]Our analysis infrastructure could not handle one of the systems in the corpus

[9]`https://gforge.inria.fr/projects/verveinej/`

[10]`http://www.moosetechnology.org/`

[11]`http://maven.apache.org`

As previously mentioned, variable type information is gathered by running a type inference engine. For this we implemented an existing type inference approach described by Pluquet *et al.* [13]

The approach has three steps:

1. Interface type extraction. This phase reconstructs the type of a variable of interest by using static analysis to find all messages sent to it within the context of the given class. The system is then searched for all classes that implement this set of messages.

2. Assignment type extraction. This phase reconstructs the type with respect to the assignments to the variable. This is a heuristic based analysis of the right side of assignments to the variable in question.

3. Merger. Merging the results from phases one and two into the final type results for the variable. Several different ways exist to do the merge [13], but we focus on the one that gives priority to the assignment type, and moves to interface types if an assignment type does not exist.

We chose this approach because it is simple to understand and implement, and is sufficient for this proof of concept implementation.

The result of the data gathering module for Smalltalk is 3,332 entries in the database. This is, as expected, significantly smaller than the data made available through analysis of statically typed source code.

3. PROVIDING THE FREQUENTLY USED METHODS TO DEVELOPERS

The client-server architecture of the solution enables different implementations of data presentation to the developer. This means that the need for presenting the data in a way that is appealing to the developer can be achieved by implementing a different view. We aim for the data presentation to be seamlessly integrated with the developers existing method search process.

For the proof of concept we implemented two data presenters, one for Java and one for Smalltalk. Both are implemented as plugins for tools developer use for searching for methods. This enables us to augment rather than replace the existing method search process.

The general work flow of the plugins is as follows

1. Detect which class is being viewed by the developer

2. Consult the data provider module on which are the frequently used methods of that class

3. Provide that list in a non-intrusive way.

We do not claim there is a correct way to present the data, nor do we claim any approach is superior to others. The question of optimal data presentation is out of the scope of this paper. The implementations presented in this chapter are only used as a proof of concept.

3.1 The Chromium plugin for Javadoc

Documentation for the majority of Java APIs is generated by a tool called Javadoc. Javadoc is a tool for generating

API documentation in HTML format from documentation comments in source code.

The fact that documentation is generated as a HTML document means that developers use a web browser to view the documentation and for method search. To augment the documentation we developed a plugin for Chromium, a popular web browser.

The plugin is triggered when the user has opened an HTML page generated by Javadoc. The page is identified as a Javadoc generated page by detecting a specific HTML comment in the Document Object Model (DOM) [20] of the page. This comment is generated by the Javadoc tool for all generated pages. Once triggered the plugin extracts the fully qualified name of the documented class from the DOM. Javadoc generated pages based on a pattern, so the class name is easily extractable from the DOM, as it is always on the same place on the page. After consulting the data provider module, the list of frequently used methods is presented by modifying the DOM.

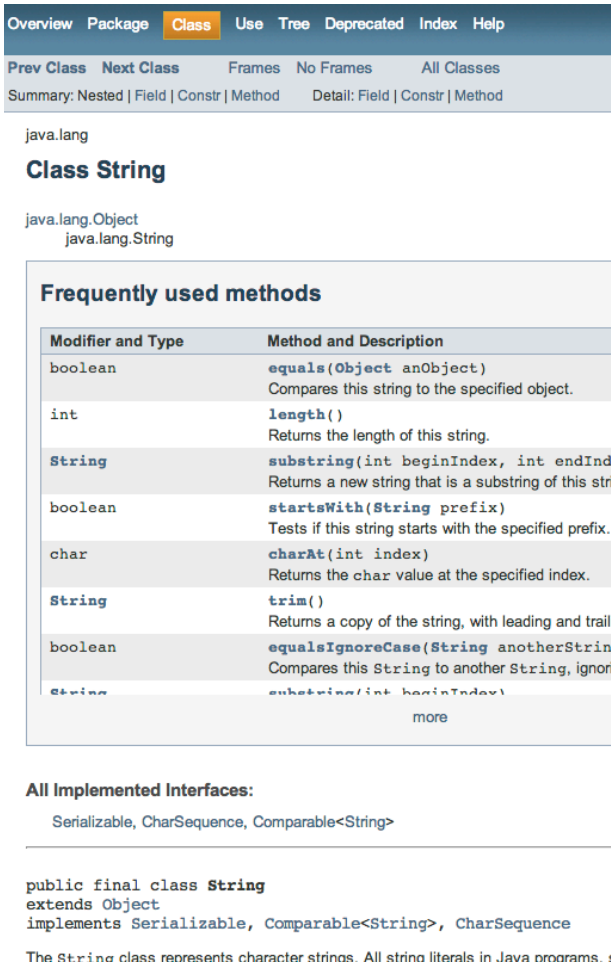

Figure 2: The browser plugin augments the documentation for the *java.lang.String* class with the "Frequently used methods" block

Our initial solution for the proof of concept implementation shows a list of frequently used methods, with the short description given in the documentation, at the top of the Javadoc page. The list shows a customizable number of methods with the ability to show the entire list on user request. Every element of the list is a HTML link to a detailed description of the method, just like in the default alphabetically sorted list of methods. The look and feel of the enhancement is identical to the original documentation generated by Javadoc. A segment of the augmented Javadoc documentation for the class *java.lang.String* is shown in Figure 2.

3.2 The Nautilus plugin

Smalltalk libraries include the source code so searching for methods is done in the source code itself. The default tool for browsing the source code in Pharo Smalltalk is Nautilus – an advanced implementation of the original Smalltalk system browser [17].

Nautilus provides a framework for developing plugins. As a built in feature of this framework plugins can register to be notified of certain events by Nautilus. One of these events is triggered when the user selects a class, and information about that class is presented. Our plugin, triggered by this event, consults the data provider module and presents the frequently used methods.

Our initial solution for the way the methods are presented within Nautilus is shown in Figure 3. Nautilus is organized in 5 panes, 4 on top half, and one on the bottom. The 4 top panes contain, from left to right, a lists of all packages in the system, a list of classes in a selected package, a list of method protocols[12] in the selected class , and finally a list of methods in the selected protocol. The bottom pane is context sensitive and shows mainly the source code for a selected class or method.

The thin line of buttons between the top and bottom panes is provided by our plugin. Every button corresponds to one frequently used method, and clicking the button opens a new Nautilus window with the desired method selected, and the source code shown. The methods are sorted by frequency from left to right, so that the most popular ones could be read first. The "view all" button opens a separate window with all the frequently used methods shown in a list, similar to the default list of methods in Nautilus.

4. INITIAL EVALUATION

Our initial evaluation involves observing developers completing simple programming task using the augmented documentation. Several situations in which our approach is directly helpful have been identified in the initial evaluation. To illustrate, we present two cases.

One is the case where a popular method is, due to the alphabetical sorting of methods, located near the end of documentation prolonging the method search process. An example of this is the *substring* methods of the *java.lang.String* class. It is, according to out analysis, the third most commonly used method of the class, yet in documentation it is placed on the 48th place out of 65 methods.

The second case is when a popular method of a class is declared higher in the class hierarchy. This leads to the developer wasting time looking through the documentation of a class that does not declare the required method. An example of that is the method for concatenation[13] of *ByteStrings*

[12]Protocols are convenient groupings of related methods.

[13]The selector for this method is the comma operator i.e. *'Hello ' , 'World!'*. This is a legal Smalltalk method name.

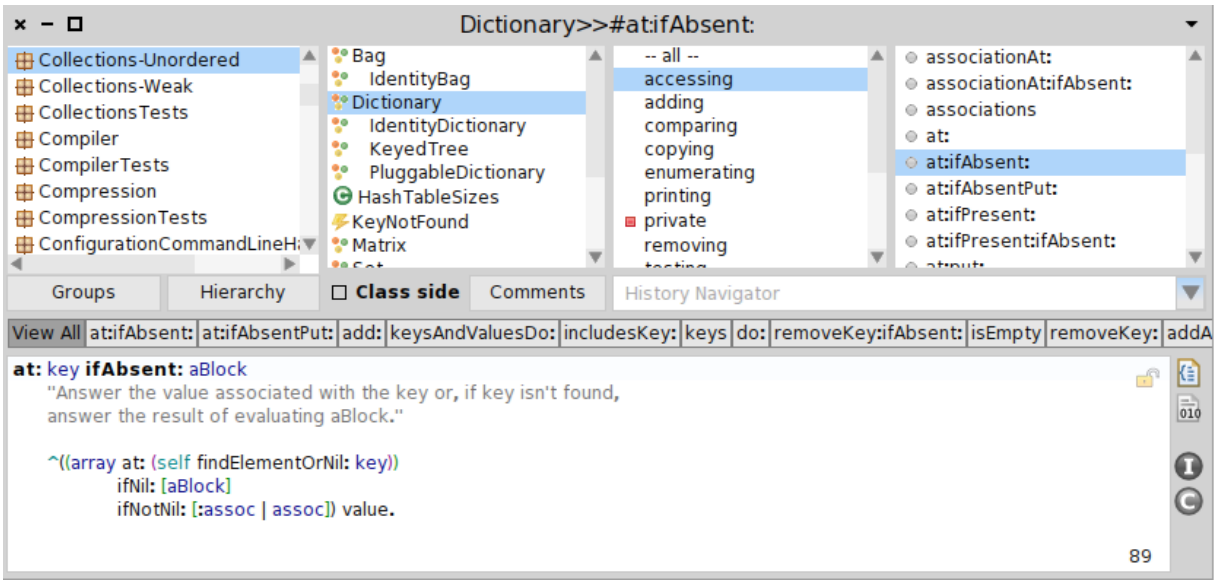

Figure 3: A sample presentation of the frequently used methods in Nautilus. The plugin provides the strip of buttons above the method source code. Each button coresponds to a frequently used method.

Pharo Smalltalk. This is, according to out analysis, the most commonly invoked method of the class *ByteString*, yet it is declared in the *SequenceableCollection* class, which is 4 levels higher in the class hierarchy.

5. RELATED WORK

Leveraging information from other projects to make API usage easier for the developer is not a novel idea and much work has already been done in this field, but to our knowledge this is the first attempt to use ecosystem analysis to help developers find the most frequently used methods of a class faster.

A common approach to improving API usage is to provide code snippets to the developer. The snippets are most commonly mined from open source repositories [21, 3, 6] but other sources are also used i.e. Google code search [18]. Several approaches are used present the code snippets to the developer. These include search engines [18], IDE augmentations [21], adding code snippets to documentation [10] and others. Providing code snippets to the developer works on a different granularity when compared to our approach. We aim to help the developer find the right method, while code snippet suggestion provides a whole block of code. The usefulness of each depends on the developers use case.

Mileva *et al.* mined software repositories to observe popularity of APIs, and help developers choose a popular API rather then an unpopular one, claiming that the "wisdom of the crowd" can be used to measure the quality of an API [9]. Our approach assumes that the API is known, and aims to improve the usage of the API, not the selection.

Several papers [11, 1] deal with mining frequent call sequences from API. The goal of this is to use other projects to predict the sequence of method calls the developer wishes to write based on one or two initial methods. This differs from our approach because API sequence prediction requires input of one or more method invocations to predict a sequence, and our approach aims to help the developer find individual methods that are frequently used.

Some approaches to help developers use APIs relay on the context in which the developer works [21, 11, 1], commonly gathered by analyzing the source code the developer is working on. Our approach ignores this kind of context in order to reduce complexity, as we aim to test the hypothesis that a substantial increase in productivity can be gained with minimum effort.

6. FUTURE WORK

The main part of future work for this approach is to do a user evaluation. Initial evaluation with individual developers shows promising results in regards to boosting productivity, and ease of adoption, but a larger developer base is needed to establish a solid and measurable claim.

We plan to do this evaluation by exposing developers to an unfamiliar environment – a new API, or even a new language – and measuring the time needed for the each developer to solve a set of coding tasks. A different group of developers, also with no knowledge of the environment, will be solving the same tasks, but with the documentation augmented by our approach.

We hope to see a measurable reduction in time developers using the augmented documentation take to solve the tasks when compared with the group using the standard documentation.

The proof of concept implementation can be improved in several ways. Expanding the number of analyzed projects would cover a larger number of APIs available in the ecosystem. Using active projects and keeping the data up to date would give a timely sample of the state of the ecosystem.

Data gathering for dynamically typed languages could benefit from more precise type inference engines or type information being gathered through dynamic analysis. Also, the way the data is presented could greatly benefit from user feedback.

7. CONCLUSION

In this paper we present an approach to augment existing method search tools with information about the most frequently invoked methods on a per class basis. This is done in order to improve the process of method search, and thus increase developer productivity.

Frequently invoked methods are discovered by analyzing a large set of projects – a snapshot of the ecosystem. This results in a snapshot of the most popular APIs and the most popular methods of those APIs.

This data is then stored, and made available for client applications – augmentations of existing tools – to present to the developers.

We implemented a proof of concept for Java – as representative of statically typed languages, and Smalltalk – as a representative of dynamically typed languages. This paper presents the implementation details of both.

An initial evaluation, done by observing developers using the plugins, shows promising results, supporting our hypothesis that a substantial improvement in developer productivity can be achieved with minimal complexity. A larger user case study is needed to provide a measurable and representative result.

Acknowledgment

We gratefully acknowledge the financial support of the Swiss National Science Foundation for the project "Agile Software Assessment" (SNSF project Np. 200020-144126/1, Jan 1, 2013 -Dec. 30, 2015)

We also thank Cédric Reginster, for his contribution to the proof of concept implementation.

8. REFERENCES

[1] M. Acharya, T. Xie, J. Pei, and J. Xu. Mining api patterns as partial orders from source code: From usage scenarios to specifications. In *Proceedings of the the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 25–34, New York, NY, USA, 2007. ACM.

[2] N. Haenni, M. Lungu, N. Schwarz, and O. Nierstrasz. Categorizing developer information needs in software ecosystems. In *Proceedings of the 1st Workshop on Ecosystem Architectures*, pages 1–5, 2013.

[3] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of ICSE'05*, pages 1–10, 2005.

[4] M. Lungu. *Reverse Engineering Software Ecosystems*. PhD thesis, University of Lugano, Nov. 2009.

[5] M. Lungu, R. Robbes, and M. Lanza. Recovering inter-project dependencies in software ecosystems. In *ASE'10: Proceedings of the 25th IEEE/ACM International Conference on Automated Software Engineering*. ACM Press, 2010.

[6] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: Helping to navigate the api jungle. *SIGPLAN Not.*, 40(6):48–61, June 2005.

[7] K. Manikas and K. M. Hansen. Software ecosystems - a systematic literature review. *J. Syst. Softw.*, 86(5):1294–1306, May 2013.

[8] D. G. Messerschmitt and C. Szyperski. *Software Ecosystem: Understanding an Indispensable Technology and Industry*. The MIT Press, 2005.

[9] Y. M. Mileva, V. Dallmeier, and A. Zeller. Mining api popularity. In *Proceedings of the 5th International Academic and Industrial Conference on Testing - Practice and Research Techniques*, TAIC PART'10, pages 173–180, Berlin, Heidelberg, 2010. Springer-Verlag.

[10] J. E. Montandon, H. Borges, D. Felix, and M. T. Valente. Documenting apis with examples: Lessons learned with the apiminer platform. In *WCRE*, pages 401–408, 2013.

[11] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. N. Nguyen. Graph-based mining of multiple object usage patterns. In *Proceedings of the the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC/FSE '09, pages 383–392, New York, NY, USA, 2009. ACM.

[12] J. Palsberg and M. I. Schwartzbach. Object-oriented type inference. In *Proceedings OOPSLA '91, ACM SIGPLAN Notices*, volume 26, pages 146–161, Nov. 1991.

[13] F. Pluquet, A. Marot, and R. Wuyts. Fast type reconstruction for dynamically typed programming languages. In *DLS '09: Proceedings of the 5th symposium on Dynamic languages*, pages 69–78, New York, NY, USA, 2009. ACM.

[14] F. Rivard. Smalltalk: a reflective language. In *Proceedings of REFLECTION '96*, pages 21–38, Apr. 1996.

[15] C. Scaffidi. Why are apis difficult to learn and use? *Crossroads*, 12(4):4–4, Aug. 2006.

[16] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. The qualitas corpus: A curated collection of java code for empirical studies. In *Software Engineering Conference (APSEC), 2010 17th Asia Pacific*, pages 336 –345, Dec. 2010.

[17] L. Tesler. The Smalltalk environment. *Byte*, 6(8):90–147, Aug. 1981.

[18] S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, ASE '07, pages 204–213, New York, NY, USA, 2007. ACM.

[19] S. Tichelaar. *Modeling Object-Oriented Software for Reverse Engineering and Refactoring*. PhD thesis, University of Bern, Dec. 2001.

[20] L. Wood, J. Sorensen, S. Byrne, R. Sutor, V. Apparao, S. Isaacs, G. Nicol, and M. Champion. *Document Object Model Specification DOM 1.0*. World Wide Web Consortium, 1998.

[21] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei. Mapo: Mining and recommending api usage patterns. In S. Drossopoulou, editor, *ECOOP 2009 - Object-Oriented Programming*, volume 5653 of *Lecture Notes in Computer Science*, pages 318–343. Springer Berlin Heidelberg, 2009.