DOCUMENTATION GENERATORS SUPPORT FOR PROGRAM COMPREHENSION: WHERE ARE WE?

Alexandre Terrasa, Jean Private and Guy Tremblay

Décembre 2015

Département d'informatique

Université du Québec à Montréal

Rapport de recherche Latece 2015-2

Laboratoire de recherche sur les technologies du commerce électronique

DOCUMENTATION GENERATORS SUPPORT FOR PROGRAM COMPREHENSION: WHERE ARE WE?

Alexandre Terrasa
Département d'informatique
UQAM
Montréal, Qc, Canada

Jean Privat
Département d'informatique
UQAM
Montréal, Qc, Canada

Guy Tremblay
Département d'informatique
UQAM
Montréal, Qc, Canada

Laboratoire de recherche sur les technologies du commerce électronique
Département d'informatique
Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal, QC, Canada
H3C 3P8
`http://www.latece.uqam.ca`

Décembre 2015

Rapport de recherche Latece 2015-2

This work was partially supported by *Research Discovery* grants from the Natural Sciences and Engineering Research Council of Canada.

Summary

Documentation generators like Javadoc are a common way to auto-document software. These tools generate API documentation by extracting annotated comments from source code. This kind of documentation is thus relatively easy to produce and can more easily be synchronized with source code.

However, previous studies have shown that such tools are a poor support for program comprehension and maintenance, and thus are generally criticized by both the documentation writers and readers for their pedagogical effectiveness [15, 16, 62].

In the current paper, we present a survey of existing auto-documentation systems, approaches and tools. First, we identify five key issues that must be addressed by auto-documenting systems to support program comprehension: lack of structure of the documentation, lack of abstraction levels, lack of support for examples, elusive information, and stale documentation. Then, we analyze 22 auto-documentation systems available for various programming languages. We group their features under the five categories of issues they try to address. Finally, we compare these tools based on their ability to support program comprehension for readers and their facility to produce documentation for writers.

The results of our comparison show that only two tools, Doxygen and Sphinx, propose features addressing each category of issues. Yet, even these two tools could benefit from additional features included in other competing tools or proposed in the literature that address those same issues. Thus, much work remains to be done to obtain a better documentation generator, and our survey is a first step in that direction, establishing a roadmap toward an improved, fully state-of-the-art, documentation generator tool.

1 Introduction

Program comprehension is essential for code usage and maintenance. When available, documentation is the first support used by programmers to obtain information from existing systems.

In 1995, Friendly [27] published a tool that could generate API documentation in HTML format from Java source code and comments. This was the birth of the—now famous—Javadoc. Since then, documentation generators have become common tools in software engineering and are being used in standard build processes. For instance, the build automation tools for Java, Ant [2] and Maven [3], both include a step dedicated to the generation of documentation with Javadoc.

Over the past twenty years, documentation generators have made it possible to produce documentation with minimum effort. But are those tools in fact helpful for program comprehension? The objective of the current paper is to answer that question.

The main contributions of this paper are twofold:

- We identify some key issues that documentation generators must overcome to support program comprehension, namely, generated documents often lack structure, abstraction levels, and examples, provide insufficient search features and can be incorrect or missing.
- We perform a comprehensive analysis of 22 mainstream tools and numerous approaches proposed in the literature based on these issues. We discuss each solution, keeping in mind the final document quality for readers and the amount of effort required by writers.

The rest of the current paper is organized as follows. Section 2 presents the issues related to documentation generators regarding program comprehension. Each of the next five sections is then dedicated to one of these specific issues: 3. Lack of structure; 4. Lack of abstraction levels; 5. Lack of examples; 6. Elusive information; 7. Stale documentation. These five sections are all organized in a similar fashion: First, the section introduces the issue; Then, common approaches used by documentation generators to solve the issue are presented, where each specific feature that tries to solve the issue appears in its own subsection. Section 8 summarizes the comparison and discusses documentation generator support for program comprehension. Finally, Section 9 concludes our survey and proposes some future work.

2 Survey Overview

This section identifies the key issues related with program comprehension that will be used, in our survey, to compare the documentation generator tools, that is, how do the various tools address those issues. It also lists the tools that will be analyzed and compared.

2.1 Common Issues of Documentation Regarding Program Comprehension

de Souza *et al.* [15] surveyed a group of software maintainers, with the goal of establishing the importance they gave to various documentation artifacts for maintenance. Their results

showed that software maintainers *want and expect to need* source code and comments, abstractions (data models, class diagrams, use-cases, etc.), acceptance test plans and unit-tests, user manual and design explanations. However, de Souza *et al.*'s survey also showed that, *in practice, maintainers only rely on source code*: No other source of information can effectively be used as those sources are most often non-existent or outdated. The only textual documentation available is scattered in the comments. Sometimes, maintainers can find some usage examples from non-functional prototypes.

Forward and Lethbrige [25] present another survey about the relevance of software artifacts. They asked developers about document attributes that contribute to documentation's effectiveness. The most important attributes identified were the content (the document's information), up-to-dateness and availability, use of examples, organization (sections/subsections), use of diagrams.

Both studies considered all software documentation elements, including specifications, design documents and other documentation artifacts.

Naisan and Ibrahim [54] made a comparison between Universal Report and Doxygen from a reverse-engineering point of view. However, their comparison was limited to criteria such as supported languages, parser extensibility and diagrams generation. Also for reverse-engineering, Vestdam andNørmark [78] compared Javadoc and Doxygen against re-documenting approaches and internal documentation tools.

Apart from these papers, we found no previous study about documentation generators support for program comprehension.

Based on these studies, we chose to compare documentation generators according to how they address those issues that pertain to program comprehension:

1. lack of structure;
2. lack of abstraction levels;
3. lack of examples;
4. elusive information;
5. stale documentation.

In what follows, we analyze existing documentation generators to identify which features can be used to address these issues. We also look at the scientific literature to find other solutions, not necessarily available in any of the existing tools, for addressing these issues.

2.2 The State of the Art in Auto-Documenting Systems

Table 1 shows the tools we analyze in the present paper. Most of these documentation generators are designed for a specific programming language, except for Doxygen, Natural Docs and Universal Report that can deal with numerous languages.

We collected this list of tools from various sources, for instance the Wikipedia page that lists and compares such tools.[1] We also used the TIOBE index[2] to identify the most popular programming languages and look at their documentation generators. However, we restricted ourselves to tools satisfying the following conditions:

[1] `http://en.wikipedia.org/w/index.php?title=Comparison_of_documentation_generators&oldid=626977449`

[2] `http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html`

Language	**Tool**	**Reference**
Clojure	Audodoc	[24]
	Codox	[59]
C#	SandCastle	[51]
Go	Godoc	[70]
Haskell	Haddock	[46]
Java	Javadoc	[27]
JavaScript	YuiDoc	[84]
	JSDoc	[39]
	Doxx	[60]
	Docco	[6]
Nit	Nitdoc	[69]
Objective-C	AppleDoc	[28]
Perl	perldoc	[1]
Python	Sphinx	[29]
PHP	PhD DocBook	[71]
	phpDocumentor	[52]
	ApiGen	[79]
Ruby	RDoc	[72]
	YARD	[65]
Scala	Scaladoc	[19]
12 languages	Doxygen	[76]
19 languages	Natural Docs	[74]
26 languages	Universal Report	[73]

Table 1: Tools analyzed in our study.

1. it is a documentation *generator* — which excludes a tool such as Pandoc;
2. it targets programming languages — which excludes a tool such as SQL Documentor;
3. it has a working website, documentation, and installable executable — which excludes a tool such as Autoduck.

Unlike the lists mentioned above, the present paper does not compare the tools based on specific technical features, but rather on how they can support and improve program comprehension based on the issues mentioned earlier.

3 Lack of Structure

Documentation structure based on sections and subsections is considered a key attribute for a good documentation by readers [25]. However, the documentation typically produced by generators is usually simply an index. This index contains the elements extracted from the software like packages, classes and methods. Documentation organization is thus based on code structure. Methods are grouped by classes, classes are grouped by packages. Elements are generally sorted based on a lexicographic order of the identifiers.

On one hand, the index structure provides fast access to the information, which is quite useful for expert users who know what they are looking for. On the other hand, the index structure does not help the beginner in his learning process. There is no logical order of reading, no starting point. The reader has to choose a page and start reading hoping to find what he is looking for.

If the index is not an optimal solution, then auto-documentation systems need to produce a richer structure that provides support for fluid and pedagogic reading sprinkled with use-cases and examples. This is the purpose of user manuals and programming guides.

3.1 Documentation as a User Manual

A user manual describes how to use the software through sections organized to improve the understanding of the API and its content. Unlike an index, a user manual's structure is more flexible and enhances the expressiveness of the writer [43, 62]. API elements can be presented in a logical order, based on use-cases and concerns instead of a fixed order linked to the structure of the code. Documentations from MSDN [50], PerlDoc [1] or PHP.net [9] are built as manuals.

Extracting a manual from code is not a new idea. In the early 1980s, Knuth [43] proposed *Literate programming*, where fragments of programs are defined inside the program documentation. A tool (called `tangle`) is then used to extract and assemble program fragments into an executable. Another tool (called `weave`) formats the documentation, generates an index, and presents all of it in a nice-looking paper format. Thereby, literate programming can help the programmer to provide a human readable view of a program that can be read as a manual.

Nowadays, it is common practice for software developers to use external supports like wikis, websites or README files to document use-cases or design decisions that can be presented in an index [63]. Manuals are generally written by hand, sometimes using a word processor. Handwritten manuals are time consuming for both initial production and maintenance.

Doxygen proposes three *grouping mechanisms*[3] to manage the documentation structure: (*i*) *Modules*: to group multiple entities (files, namespaces, classes, methods, etc.) within the same page; (*ii*) *Subpaging*: to group information into a same section within a page; (*iii*) *member groups*: to build groups of properties inside a class.

Haddock allows the writer to filter and sort the index lists using *export lists*[4]. Members from a module or a type can be manually listed to override the default index generation. This feature allows the writer to choose both the content and the order of lists. *Named chunks*[5] can be used to include arbitrary documentation blocks inside the index list.

Using PhD DocBook, one can create *chapters* within a *book*.[6] Chapters are used to structure documentation and produce a HTML output decorated with summaries and links.

SandCastle has *topic files*[7] written in XML/MAML. Topics files can be used to group documentation elements under a single conceptual concern.

[3] `http://www.stack.nl/~dimitri/doxygen/manual/grouping.html`
[4] `http://www.haskell.org/haddock/doc/html/ch03s04.html`
[5] `http://www.haskell.org/haddock/doc/html/ch03s05.html`
[6] `http://www.docbook.org/tdg5/en/html/ch02.html`
[7] `http://www.ewoodruff.us/mamlguide/html/c69a248a-bd94-47b2-90d7-5442e9cb567a.htm`

Sphinx works with file inclusions and *TOCtrees*.[8] Documentation pages are structured by including multiple documentation files. For each page, a table of contents can be generated based on the included files. phpDocumentor can do the same with the *Scrybe* plugin.[9]

Almost all tools support the inclusion of external files. This method is convenient for documenting design decisions, configuration settings or anything that is not directly tied to a source file. However, these files are not synchronized with the source code which does not solve the maintenance problem.

3.2 Reading Suggestions

Reading suggestions (*aka.* "*see also*") provide further topics that can interest the reader based on a context. This context can be the current page or the task of the reader. A basic example would be to suggest reading the `remove` method to someone reading the documentation for `add`.

Most tools provide a way to manually suggest related pages for each documentation element using links. For example, Doxygen uses the `\relatesalso` directive, APIGen, Javadoc and YARD use `@see`, and YUIDoc provides `@cross-links`. In SandCastle, topics can be linked to another topics making a cross-referenced manual using the `<seealso>` XML tag.

This manual approach offers flexibility to the writer who can think of every possible interesting topics related to a documentation element. Furthermore, because links are reified under specific directives, they can be checked at generation time. Doing so, tools like APIGen or Javadoc can detect outdated links and warn the writer.

On the negative side, such a solution is tedious for the initial writer who has to insert directives in every piece of documentation. Since code naturally evolves, this method adds an extra burden on the maintainer who has to keep adding new links to relevant information into existing documentation.

Both Robillard [61] and Warr and Robillard [82] present an automatic suggestion system based on program topology and structural dependencies between model elements. These approaches permit automatic selection and ranking of elements from documentation that might be interesting for the reader depending on an entry set. The entry set can be derived from the current page viewed by the reader, or can be based on the reader's task.

Stylos *et al.* [67] uses static analysis on a large corpus of clients to identify the services used most often in an API and then build a suggestion list by ranked popularity. Even if these are not reading suggestions *per se*, this approach can be used to propose a reading order.

4 Lack of Abstraction Levels

Robillard and DeLine [62] show that clients require a high level of abstraction to understand the content and goals of an API. Based on the reader's experience, various abstraction levels are needed [75]. Newcomers do not need to see *all* the details. They would like to have enough information so that they can build a mental model of the system, and zoom on the specific details they are interested in.

[8] `http://sphinx-doc.org/markup/toctree.html`

[9] `http://phpdocumentor.github.io/Scrybe/internals/table_of_contents.html`

Because the generated documentation structure is constrained by the software code structure, it can only provide the same level of abstraction as the code (or a little more). It cannot give a global view of the software code and its concerns [78]. Grouping methods by super-class, as done by Javadoc, does not give an overview of all the services available for a type [20]. Furthermore, the Java code's structure cannot represent cross-cutting concerns that can be either scattered (one concern among multiple classes) or tangled (multiple concerns into one class) [41].

This section discusses the features that can provide a higher level view of a program than the one given by its source code. Such features help the user and the maintainer to understand quickly the content of a software.

4.1 Lists and Trees

Lists and trees are the most used abstractions in documentation generators. Lists are used to group sets of similar entities like classes or methods in a linear order, making it easier to understand the overall structure. Trees can represent hierarchical entities, for instance, the classes contained in a package. Both lists and trees offer a succinct presentation of an entity generally limited to a short summary, and links are then used to access the entity's full description.

Most tools generate a list of all the entities available in the model with no other filter than private/public. ScalaDoc and YUIDoc let the reader filter the content of the list on various properties (e.g., kind, visibility, inheritance, name). This is helpful when you know what you are looking for, but can produce a lot of noise for newcomers. Furthermore, this kind of feature needs much micro-management and is not persistent across pages. Only Haddock allows the writer to manually set the content using *export lists*.

All tools list the elements following a lexicographic ordering of the identifiers. ScalaDoc allows the reader to switch from lexicographic order to linear extension based on inheritance. With Doxygen, the writer can disable the automatic sorting of elements to follow the code order.[10]

Outside of documentation generators, some approaches can be found to filter and rank model entities automatically. Denier and Gueheneuc [17] for example propose MENDEL, a set of metrics to extract interesting classes from a model.

4.2 Diagrams and Figures

Diagrams and other visual representations present informations in a simplified, yet structured way. They help the reader to quickly understand the content of a documentation, although they should also be supported by text to be more efficient [35].

Doxygen is well equipped for graphs and diagrams generation.[11] It provides support for class inheritance, module dependencies and directory structure, call graphs as well as collaboration and class diagrams. Each representation can be used for navigating through the elements, using links.

Nitdoc, phpDocumentor and ScalaDoc can also produce class diagrams. Sphinx and YARD can do the same with plugins. Godoc only provides module dependencies graphs.

[10] `http://www.stack.nl/~dimitri/doxygen/manual/config.html`

[11] `http://www.stack.nl/~dimitri/doxygen/manual/diagrams.html`

Automatically generated diagrams suffer from the same drawbacks as automatic lists: The content is not filtered so the figures that are generated can be hard to read or even hard to display in some browsers because of their size and content.[12]

From a reader's point of view, only Godoc allows the reader to filter the dependency graphs, for instance, by specifying whether to display dependencies from the standard library. Sphinx provides a set of filters that can be applied by the writer to select the members to be displayed in lists using the `automodule`, `autoclass` and `autoexception` directives.[13]

Here again, metrics can be used to filter an automatic output, as done by Denier and Gueheneuc [17]. Ducasse and Lanza [21] go further by proposing their own visualization called *Class Blueprint*. Blueprints help the reader understand the internal structure of a class with a high level of abstraction before going deeper into the documentation. Zhang [88] uses static analysis to weight call-graphs based on a control flow analysis. This can filter the call-graph content on the most often used methods.

The MCT approach [34] allows the writer to include videos and other multimedia support to code comments. Even though it provides a good support for understanding and an expressive way to comment code, video seems to be a lot harder to maintain up-to-date than text.

4.3 Source Code

Most tools make it possible for the writer to include the original source code in the documentation. Source code can then be used as a resource when the documentation is insufficient. This can also provide some support for maintainers who want to see how a software is implemented.

Code is the lowest level source of information that can be used. However, maintainers often try to avoid reading the source code, relying instead on more abstract software representations [63]. So, documentation generators must also provide features to help such higher-level understanding. Besides syntax highlighting and hyper-linking between code and documentation, only three tools support the reader in his comprehension of the code.

Godoc makes use of the Go Playground[14] to let the user edit and run pieces of code directly in the browser. The Playground tool is rather basic but supports code formatting and code sharing through URLs, whereas syntax highlighting is not supported.

Inspired by the Elucidative Programming approach [78], Docco displays the code side by side with the documentation extracted from comments and external sources.[15]

Universal Report presents the code as a navigable callgraph.[16]

Making the code more understandable for readers and maintainers is a long pursued objective. With literate programming, Knuth [43] encourages the programmer to write programs in a self-explanatory fashion, containing information on the design that will help the reader understand the source code.

[12] Doxygen's documentation in fact gives a warning about this: `http://www.stack.nl/~dimitri/doxygen/manual/diagrams.html`. Also, an example of unreadable graph can be found in the live demo of phpDocumentor: `http://demo.phpdoc.org/Responsive/graph_class.html`

[13] `http://sphinx-doc.org/ext/autodoc.html`

[14] `http://play.golang.org/p/4ND1rp68e5`

[15] `http://derickbailey.github.io/backbone.marionette/docs/backbone.marionette.html`

[16] `http://www.omegacomputer.com/flowchart.jpg`

More recently, Oney and Brandt [56] proposes the use of *codelets*, interactive code examples that can be exchanged between writers and readers. Using a DSL (Domain Specific Language), a *codelet* makes it possible for the reader to customize an example once pasted into an IDE.

Wang *et al.* [81] use heuristics based on code structure and identifiers to locate *meaningful blocks* in the code. This can help the reader detect sets of sequential instructions that work together.

4.4 Code Summarization

Presenting a library's source code is one thing, but when it comes to understanding thousands of lines of code, the reader needs better tools. To begin comprehending a large program, the reader needs to see "the big picture" first. What are the main goals of this source code? The high level mechanisms involved? A possible solution to this problem can be found in code summarization.

According to Moreno *et al.* [53], code summaries are *indicative*—they provide a brief description of some code's content—*abstractive*—they highlight information that is not directly visible in the source code—and *generic*—they only cover the key information, not the details.

Code summarization represents a powerful add-on to documentation generators as it can abstract large amount of source code in a few lines and help the reader quickly select the information he should look with more details, depending on the task at hand. We found no evidence of this kind of features in the documentation generators studied, but the scientific literature provides some interesting tools worth investigating.

For instance, Ying and Robillard [85] show that machine learning can be used to automatically generate summaries for small pieces of Java code, such as examples. Sridhara *et al.* [66], and then McBurney and McMillan [47], were able to generate summaries for Java methods using natural language processing and heuristics. At a higher level of abstraction, Moreno *et al.* [53] can generate a summary for Java classes also using natural language processing.

5 Lack of Examples

When exploring a new software, it is generally useful for developers to look at examples [25, 10, 18]. When no examples are provided by the documentation, the reader depends on the Internet to find examples linked to its use-case [56]. He then has to select the appropriate examples himself, adapt them to his needs and ensure that they match the same version of the API as the one he is using. Looking for examples can thus be a time consuming task.

Some software writers provide examples directly in the API documentation by adding code examples in the comments. With most of the tools, those examples are not checked as they are neither compiled nor tested. If the software API changes, nothing can warn the writer that he forgot to update the examples. Thus, there is a risk of desynchronization with the source code. When trying the example, the reader must find the problem, and then apply a patch himself. This is, indeed, a hard way to learn!

Readers need examples to understand documentation and code. So, writers have to create good examples and insert them in the documentation at the right point to help

comprehension. But how can documentation generators help the writer to create such good examples? And what are good examples?

We define good examples as having the following characteristics:

1. **minimal**: they show only what is needed;

   In a study of the StackOverflow Q&A site, Nasehi, Sillito *et al.* [55] show that the best rated examples are short. In another study, Oney and Brandt [56] define the best length of an example as being between 5 and 20 lines of code. Short examples are also a main objective of Ying and Robillard [86] when they use code summarization techniques to produce examples.

2. **explanatory**: they show a use case, a behavior, or a good practice;

   Explanatory examples are another feature of good examples identified in Nasehi, Sillito *et al.*'s survey [55]. The best rated examples tend to explain a single concept, are divided in intelligible steps, and are adapted to the question's context. This is also one of the goal pursued Oney and Brandt's *Codelets* [56]: interactive examples that can be adapted to the reader's domain.

3. **maintainable**: they can be easily kept up-to-date with the source code and the documentation;

   Robillard and Deline, in their survey [62], state that "code examples are assessed in terms of how authoritative and credible they are." Examples that provide evidence they are up-to-date are best received.

4. **executable**: they can be executed by the reader without modification or fix.

5.1 Reified Examples

Doxygen, Godoc, JSDoc, SandCastle, ScalaDoc, YARD, and YUIDoc provide a special directive to tag examples, Once tagged, examples can be formatted or grouped within special index pages. AppleDoc (markdown), Nitdoc (markdown), Perldoc (perlpod) and Sphinx (reStructured Text) rely on the documentation format to display code in comments. In this case, the examples are only highlighted, i.e., no further processing is performed.

5.2 Examples Checking

Examples should be executed and automatically verified. Like code, testing examples is the only way to keep them up-to-date. We found three tools that provide automatic example checking: Godoc, Sphinx and Nitdoc.

With Godoc, examples are written in external files that are automatically linked to the documentation thanks to a strict naming convention.[17] In the code examples, expected output is specified using the "`Output:`" directive so examples can be automatically checked. Like any other pieces of code, examples can be played in the browser using the Go Playground.

With Sphinx, examples are embedded into code comments, and the output of each example is described using assertion as in unit tests. A tool called *Dexy*[18] is used to

[17] `https://godoc.org/github.com/fluhus/godoc-tricks`
[18] `http://dexy.it/`

extract code from comments, then run and test them. The same is possible in Nitdoc with *DocUnits*.[19]

These latter two tools make it possible to maintain executable examples synchronized with the source code, and thus encourage the developer to write documentation and examples. Indeed, by writing examples, the developer also writes unit tests. The Sphinx approach works particularly well with *Test Driven Development* [8] where the developer alternates frequently between writing unit tests and code.

5.3 Automatic Examples Generation

Examples can also be extracted from existing source code [25]. Unit tests contain numerous examples that meet our criteria for good examples, as they are executable, minimal and maintainable. Source code of software clients is also a good source of examples, as they can provide executable and up-to-date examples.

Although none of the studied tools can automatically produce examples from existing source code, some approaches have been proposed in the literature.

Zhong *et al.* [89] use data mining to extract call sequence examples from clients found in open source code repositories. Kim [42] and Grechanik *et al.* [31, 32] propose a search engine that can mine examples from the web.

Zhang *et al.* [87] use a database to store all the calls made to an API in a public code base. Using queries, they can then extract examples of arguments used in method calls. Even though this approach is designed to be embedded in an IDE, it could also be adapted to the needs of documentation, with the goal of explaining how to call a method or instantiate a class.

For Buse and Weimer [10], the extraction of examples from existing source code produces examples that are too complex and hard to read. Instead, they propose to fully generate the examples using static analysis, and show that their approach can generate minimal examples that are, in more than 80% of cases, as good as handwritten examples.

Once the source code of the example is generated, code summarization techniques (see Sect. 4.4) can then be used to produce the documentation of the example. However, to the best of our knowledge, no such approach has yet been proposed in the literature.

6 Elusive Information

One of the very first challenges encountered by a developer who needs to use a new library is to find its documentation [15, 25, 63]. It can be hard to distinguish between *difficult to find* and *non-existent* documentation. So, a developer might lose a precious amount of time learning that difference.

When the documentation is accessible, the reader needs to find the information he is looking for. Indexes provide a simple access, based on identifier lists. Using an index effectively requires the reader to know the name of what he is searching. One problem is that, sometimes, writers and readers may not use the same lexicon to talk about the same things [30].

Even if the reader knows what to search and where to find it, considering its current location in the index, he may have to follow a long path through the documentation to

[19] `http://nitlanguage.org/tools/nitunit/`

reach his goal [22]. HTML browsers by default have no search capabilities that work across multiple pages.

When the documentation is unclear or incomplete, the reader may instead decide to look at the code, if the source code is indeed available. In that case, the reader will have to locate the appropriate piece of code possibly within hundreds or thousands of source files [7].

Whether it be for searching text from documentation, source code or examples, search engines provide a quick access to the information [38]. Indeed, Freitas *et al.* [26] show that search engines are efficient tools to discover and understand a new library. So the documentation generator needs to help supporting such a feature.

6.1 Client-Side Search Engines

According to the Doxygen documentation,[20] client side search engines are the easiest way to enable searching. These engines, implemented using JavaScript, run entirely in the reader's browser so they can be packaged with the source code. They also provide live searching, i.e., search results are presented and adapted as one types.

Still, according to Doxygen, this method has some drawbacks: It is limited to searching for symbols only, does not provide full text search capabilities, and does not scale well to large projects (searching then becomes slow).

Along with Doxygen, ApiGen, Perldoc, NaturalDocs, Nitdoc, ScalaDoc, Sphinx, and YUIDoc all provide client side search engines.

6.2 Server-Side Search Engines

When the documentation is hosted online, server side search engines can be used. Server side engines are more efficient than JavaScript ones and scale better on medium to large projects. They also provide full text search within documentation pages.

ApiGen provides a connector to the Google Custom Search Engine.[21] Doxygen and Sphinx provide server side search engines with basic features but raw data can be generated to be used by external tools like Xapian,[22] an open source search engine library. Godoc provides a server side search engine capable of full text and code search.[23].

In recent years, search from source code became a popular study field among researchers. For instance, Chen and Rajlich [14] propose an approach that analyzes dependencies between code elements to extract high level concepts from source code; the reader can then use these concepts in search queries. Marcus *et al.* [45] use latent semantic indexing to achieve the same goal, whereas Poshyvanyk *et al.* [58] use formal concept analysis and Kim *et al.* [42] borrow tools from artificial intelligence.

Holmes et al. [38] show that source code search engines are generally based on a complex query language that needs specific inputs from the reader who already needs to know the software. Thus, some studies focus on the possibility to express source code with natural language [31, 26, 45, 12].

[20] `http://www.stack.nl/~dimitri/doxygen/manual/searching.html`
[21] `https://www.google.com/cse/`
[22] `http://xapian.org/`
[23] `http://godoc.org/golang.org/x/tools/cmd/godoc`

Search engines can be used to find examples too. For instance, Krinke [44] analyzes a program's dependencies to suggest similar pieces of code from a search query. In this case, the main issue is how to present extracted code to the reader in an understandable format. Some approaches try to automate code presentation using the search query [33, 49, 48, 13, 37].

7 Stale Documentation

We group outdated, imprecise, incomplete and missing documentation under the stale documentation issue. Even when the documentation is generated from the source code, textual elements written in comments are not checked. Textual documentation can refer to code elements like identifiers or namespaces. They can also contain internal references or external links.

For example, suppose a developer writes about a class X in a comment. And then, several months later, a second developer renames this class as Y. So, what happens to the comment? It is up to the reader to guess the name substitution to apply when reading the documentation. Even if IDEs provide some basic search and replace features to update comments when performing refactoring tasks, there is a risk of outdated documentation [4].

7.1 Documentation Checking

Some tools provide warnings about missing documentation at generation. Doxygen[24], Nitdoc[25] and SandCastle[26] warn about missing comments on public entities. YARD, using YardStick[27], also enforces examples on public methods. Javadoc, using the `-Xdoclint` option, validates the comments content by checking missing documentation, tag mismatch with respect to method signature, and improper punctuation.[28] Scaladoc does the same using ScalaStyle[29] and Perldoc with PodChecker[30]. Warnings about missing elements provide a good reminder for forgotten documentation but can produce much noise for entities that do not need to be documented. No tool seems to provide more advanced documentation checking.

In the literature, some approaches have been proposed to automatically check the documentation beyond missing comments. Hens *et al.* [36], for instance, automatically extract questions asked on external support to compose a Frequently Asked Question page directly linked to the documentation. Campbell *et al.* [11] and Wang and Godfrey [80] use StackOverflow and topic analysis to detect poorly documented parts of a library: If a particular method or class engenders a large amount of discussion, then its documentation needs to be improved.

Arnaoudova [4] defines metrics that can detect linguistic anti-patterns in code such as term dispersion in the lexicon used in the code and the documentation, or incoherence

[24] `http://www.stack.nl/~dimitri/doxygen/manual/config.html`

[25] `http://nitlanguage.org/tools/nitdoc/`

[26] `https://www.simple-talk.com/dotnet/.net-tools/taming-sandcastle-a-.net-programmers-guide-to-documenting-your-code`

[27] `https://github.com/dkubb/yardstick/wiki/Rules`

[28] `http://docs.oracle.com/javase/8/docs/technotes/tools/unix/javadoc.html`

[29] `https://github.com/scalastyle/scalastyle/wiki/Scalastyle-proposed-rules-(Scaladoc-Comments)`

[30] `http://perldoc.perl.org/Pod/Checker.html`

between comments and actual code behavior. Those metrics can then be used to check the consistency between the terms used in the documentation and those used in the source code.

Tan and Marinov [68] also check the concordance between code comment and code behavior using Natural Language Processing analysis. More precisely, they analyze English text in Javadoc comments to infer a set of *likely properties* for a method; they then generate a set of random tests to check the inferred properties and detect inconsistencies.

Rubio-Gonzalez and Liblit [64] use static analysis on source code to determine every possible exceptions that can be raised by a piece of code. They then compare this list to the documentation and check whether all error codes are documented and up-to-date. As shown by Arnout and Meyer [5], static analysis can also be used to identify contracts present in the code but not mentioned in the documentation.

7.2 Collaborative Documentation

The life-cycle of documentation does not stop once it has been generated. It is only when the documentation is used by readers that its quality and usefulness for developers become appreciable. This section presents approaches that can be used to improve documentation using readers' knowledge and experience.

Once the documentation is generated, readers can spot poorly documented areas of the library or identify errors. They thus needs to send feedback to writers to help improve the documentation.

Usually, this communication occurs through external supports like forums or mailing lists where readers can discuss their experiences and share problematic issues with other readers or writers [57]. This approach makes the information difficult to find, as readers have to browse through the Internet to find the appropriate discussions.

Some tools, like PhD, RDoc, SandCastle, ScalaDoc, or Sphinx, work with pluggable rating or discussion systems. Rating systems plugins allow readers to rate the quality of a piece of documentation, typically using a five stars system. Dicussion plugins allow readers to exchange additional information about the documentation they are reading, for example, to share concrete examples, signal stale documentation, or ask for more information.

Sometimes, readers can collaborate and correct or improve the documentation themselves, lightening the writers' workload. In many open-source projects, collaborative documentation is supported through the source control system. Readers are expected to edit the code to improve the comments, and documentation edits follow the same validation process as source code—i.e., peer review, unit and integration testing, release. This approach ensures the quality of the proposed edits but does not encourage newcomers to participate, as it requires a high level of commitment only to correct a simple typo.

PhD lets readers improve the documentation by editing the web page in a wiki. Even if this approach is easy to use, it still is prone to desynchronization with source code.

Nitdoc also provides editing from the web page but saves the changes using GitHub, with *commits* and *pull-requests* that are automatically transmitted to the project maintainer.[31] Once reviewed, changes can then be integrated in source code by merging, preserving links between documentation and code.

[31] `https://github.com/privat/nit/commit/fa45624f2ff95641427`

7.3 Measuring Documentation Quality

Writers need to measure the quality of the documentation they produce. For Khamis *et al.* [40], documentation quality is hard to measure automatically because of the use of natural language. Farooq *et al.* [23] propose that peer reviews be used to check the documentation usability. Quality checking a documentation can become tedious depending on size or frequency of update.

Wingkvist and Ericsson [83] propose to measure documentation quality using the *Goal-Question-Metric* paradigm [77]. The writer chooses a set of objectives—what should be documented—and infers a set of metrics that can be used to check if the objectives are reached.

Khamis *et al.* [40] go further by defining a set of metrics inspired from natural language processing that can be applied on Javadoc comments. Their approach makes it possible for the writer to automatically check the quality of the text contained in the documentation.

8 Comparison Summary and Discussion

Table 2 presents a summary of the tool comparison on features supporting program comprehension. Blocks of columns are the five key issues discussed in the current paper: lack of structure, lack of abstraction levels, lack of examples, elusive information, and stale documentation. Each such block is split in sub-columns linked to feature groups that try to address these issues.

Note that the counts mentioned in this table are not "points" *per se*, so the higher is not necessarily the better. A "•" in the grid solely indicates that a tool offers *some* feature that tries to address a specific issue of code documentation. The quality of the feature, its importance, or its usability for the writer or the reader is not taken in account.

At first sight, the comparison grid is surprisingly scarce: only 55 marks in total (out of a maximum of 247 possibilities, i.e., 1 out of 5). The higher total for a feature group, reified examples, is 10 (out of a maximum of 22 tools, i.e., about 1 out 2). The higher total for a tool, Doxygen, is 7 (out of a maximum of 13 groups, again 1 out 2).

8.1 On the Tools

Surprisingly, most tools only offer basic features and do not try to address the identified issues related to program comprehension.

Only five tools have a total of 5 or above: one multi-language, Doxygen, and four language specific, Godoc (Go), Nitdoc (Nit), ScalaDoc (Scala), and Sphinx (Python). On the other hand, the original and well known Javadoc has only a total count of 1, as it only checks for missing tags and bad references—see Section 7.1.

Unfortunately, the choices of features available to a user are greatly constrained by the programming language, as there are few documentation generators available and most are language-specific. Thus, the users of any given language usually do not know or use the documentation generators of other languages and, so, there are no requests to clone the other tools' best features.

No specific correlation seems to be observed between the programming language family and the coverage of the tools. For instance, Sphinx and Doxx both target a dynamically-

	Lack of structure		Lack of abstraction			Lack of examples			Elusive information		Stale doc.		
Tool	Structure	Suggestions	Lists	Figures	Code	Examples	Checking	Generation	Client-side	Server-side	Validation	Collaborative	Quality
Doxygen	•		•	•		•			•	•	•		
Nitdoc				•		•	•		•		•	•	
ScalaDoc			•	•		•			•		•	•	
Sphinx	•		•			•	•			•		•	
Godoc				•	•	•	•			•			
SandCastle	•					•					•	•	
Perldoc						•			•		•		
YARD				•		•					•		
YUIDoc			•			•			•				
ApiGen									•	•			
Haddock	•		•										
phpDoc.	•			•									
PhD	•											•	
AppleDoc						•							
Docco					•								
Javadoc											•		
JSDoc						•							
Nat. Docs									•				
RDoc												•	
Univ. Rep.					•								
Doxx													
Autodoc													
Codox													

Table 2: Summary of tool comparison. A "•" indicates that a tool (line) provide a solution within a corresponding feature group (column). This means the tool offers something *above* the basic set of features to improve program comprehension in the corresponding feature group. The details about tools' features can be found in the indicated sections (column). Tools without any "•" indicate basic tools that offer only the common minimal set of features. Feature groups without any "•" indicate that some features are proposed only in research papers, but are not currently integrated in any tool.

typed script-like object-oriented language, whereas Scaladoc and Sphinx target quite distinct languages—statically-typed vs. dynamically-type languages.

8.2 On the Issues

Lack of structure

Six tools allow the writer to control the generated documentation's structure. Links between documentation elements must be specified by hand. No tool provides automatic reading suggestions.

Lack of abstraction

The kinds of abstraction most often used are lists and trees. Six tools provide graphs or diagrams that give a higher level view of the software. Content and order of the lists and diagrams cannot be easily managed by the writer. Source code presentation is scarcely supported, and only three tools provide some help to a better code understanding.

Lack of examples

Examples are generally reified using some kind of annotations. But only three tools automatically check the examples and help to keep them up-to-date. No tool support the automatic extraction of examples from existing code or the generation of new examples.

Elusive information

Surprisingly, search engines are not a feature provided by many tools. Most of the proposed engines are client based and only provide search in an index. Four tools provide server side engines that can perform textual or code search.

Stale documentation

Six tools can check the quality of the generated documentation. Checking consists mainly in detecting missing comments for public methods. Collaborative editing does not appear to be popular among mainstream documentation generators. No concrete solution is given to automatic documentation quality measuring.

As shown in the comparison grid, few tools address all the issues. Overall, coverage is scarce. In fact, only Doxygen and Sphinx provide at least one feature in each column block, although not in each sub-column.

As mentioned above, the data presented in Table 2 does not aim to provide a detailed "quantitative" comparison. Nonetheless, this analysis shows that, although specific features to improve program comprehension do exist and are implemented in some tools or are proposed by some researchers, the best tools only propose a small subset of them. Thus, there is still much room for improvement toward designing a powerful documentation tool.

9 Conclusion

In this paper, we discussed the key issues of auto-documentation with respect to program comprehension : Documentation based on indexes generally lacks structure, abstraction levels and examples; Access to the documentation or to the information within documentation can be difficult; Documentation can contain outdated references or links even when it is generated from source code.

Based on these issues, we analyzed existing documentation generators to examine the solutions they propose and discussed these solutions. We gave a comprehensive comparison of mainstream documentation generator support for program comprehension.

The results of our comparison show that only two tools, Doxygen and Sphinx, propose some kind of solution that address each category of issues, though with only a subset of features. Furthermore, although there are tools that propose original solutions to some of the issues, it appears that few of the competing tools use those ideas and solutions. Finally, even though program comprehension can be eased by the quality of its documentation, the documentation generated by the most popular documentation generator tools—for instance, Javadoc—is not always the most appropriate.

As future work, we want to put together some of the best practices listed in this paper within a tool that will include the most promising solutions to address the presented issues. We also want to evaluate the features from a quality point of view and measure how each feature improves program comprehension.

References

[1] M. Allen, "PerlDoc," 2014.

[2] Apache Fondation, "Ant," 1999.

[3] Apache Fondation, "Maven," 2002.

[4] V. Arnaoudova, *Towards Improving the Code Lexicon and its Consistency*. PhD thesis, École Polytechnique de Montréal, 2014.

[5] K. Arnout and B. Meyer, "Uncovering hidden contracts: The .NET example," *Computer*, vol. 36, no. 11, pp. 48–55, 2003.

[6] J. Ashkenas, "Docco," 2010.

[7] S. Bajracharya, T. Ngo, E. Linstead, P. Rigor, Y. Dou, P. Baldi, and C. Lopes, "Sourcerer : A Search Engine for Open Source Code," in *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'06)*, pp. 681–682, 2006.

[8] K. Beck, *Test-Driven Development—By Example*. Addison-Wesley, 2003.

[9] D. Brown, "PHP.net," 2014.

[10] R. P. L. Buse and W. Weimer, "Synthesizing API usage examples," in *International Conference on Software Engineering (ICSE'12)*, (Zurich, Switzerland), pp. 782–792, IEEE Press, June 2012.

[11] J. C. Campbell, C. Zhang, Z. Xu, A. Hindle, and J. Miller, "Deficient Documentation Detection: A Methodology to Locate Deficient Project Documentation Using Topic Analysis," in *Working Conference on Mining Software Repositories (MSR'13)*, (San Francisco, CA, USA), pp. 57–60, IEEE Press, 2013.

[12] W.-K. Chan, H. Cheng, and D. Lo, "Searching connected API subgraph via text phrases," in *ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE '12)*, (New York, New York, USA), ACM Press, 2012.

[13] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF : A Search Engine for Java Using Free-Form Queries," *Lecture Notes in Computer Science*, vol. 5503, pp. 385–400, 2009.

[14] K. Chen and V. Rajlich, "Case study of feature location using dependence graph," in *International Workshop on Program Comprehension (IWPC '00)*, pp. 241–247, IEEE Comput. Soc, 2000.

[15] S. C. B. de Souza, N. Anquetil, and K. M. de Oliveira, "A study of the documentation essential to software maintenance," in *International Conference on Design of Communication Documenting & Designing for Pervasive Information (SIGDOC '05)*, (New York, New York, USA), p. 68, ACM Press, 2005.

[16] U. Dekel and J. D. Herbsleb, "Improving API Documentation Usability with Knowledge Pushing," pp. 320–330, 2009.

[17] S. Denier and Y.-G. Guéhéneuc, "Mendel: A Model, Metrics, and Rules to Understand Class Hierarchies," in *IEEE International Conference on Program Comprehension (OCPC'08)*, pp. 143–152, IEEE, June 2008.

[18] E. Duala-Ekoko and M. P. Robillard, "Asking and answering questions about unfamiliar APIs: An exploratory study," in *International Conference on Software Engineering (ICSE '12)*, (Zurich, Switzerland), pp. 266–276, IEEE Press, June 2012.

[19] G. Dubochet, "Scaladoc," 2011.

[20] G. Dubochet and D. Malayeri, "Improving API documentation for Java-like languages," in *Evaluation and Usability of Programming Languages and Tools (PLATEAU '10)*, (New York, New York, USA), pp. 1–1, ACM Press, 2010.

[21] S. Ducasse and M. Lanza, "The class blueprint: visually supporting the understanding of classes," *IEEE Transactions on Software Engineering*, vol. 31, no. 1, pp. 75–90, 2005.

[22] D. S. Eisenberg, J. Stylos, and B. A. Myers, "Apatite : A New Interface for Exploring APIs," in *Conference on Human Factors in Computing Systems (CHI '10)*, pp. 1331–1334, 2010.

[23] U. Farooq, L. Welicki, and D. Zirkler, "API usability peer reviews: a method for evaluating the usability of application programming interfaces," in *Conference on Human Factors in Computing Systems (CHI '10)*, (Atlanta, Georgia, USA), pp. 2327–2336, ACM, 2010.

[24] T. Faulhaber, "Autodoc," 2009.

[25] A. Forward and T. C. Lethbridge, "The relevance of software documentation, tools and technologies," in *Symposium on Document Engineering (DocEng '02)*, (New York, New York, USA), pp. 26–33, ACM Press, 2002.

[26] J. L. Freitas, D. da Cruz, and P. R. Henriques, "A Comment Analysis Approach for Program Comprehension," in *International Conference on Software Engineering (ICSE '13)*, (San Francisco, CA, USA), pp. 11–20, IEEE Press, Oct. 2012.

[27] L. Friendly, "The design of distributed hyperlinked programming documentation," in *International Workshop on Hypermedia Design '95*, pp. 151–173, 1995.

[28] Gentle Bytes, "AppleDoc," 2009.

[29] Georg Brandl, "Sphinx," 2007.

[30] A. Gokhale, V. Ganapathy, and Y. Padmanaban, "Inferring likely mappings between APIs," *International Conference on Software Engineering (ICSE'13)*, pp. 82–91, May 2013.

[31] M. Grechanik, C. Fu, Q. Xie, C. McMillan, D. Poshyvanyk, and C. Cumby, "A Search Engine for Finding Highly Relevant Applications," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering (ICSE '10)*, vol. 1, (New York, New York, USA), pp. 475–484, ACM Press, 2010.

[32] M. Grechanik, Q. Xie, C. Mcmillan, and C. Cumby, "Exemplar: EXEcutable exaMPLes ARchive," *IEEE Transactions on Software Engineering*, vol. 38, no. 5, pp. 259–262, 2009.

[33] S. Haiduc, J. Aponte, L. Moreno, and A. Marcus, "On the Use of Automated Text Summarization Techniques for Summarizing Source Code," in *Working Conference on Reverse Engineering (WCRE '10)*, pp. 35–44, IEEE, Oct. 2010.

[34] Y. Hao, G. Li, L. Mou, L. Zhang, and Z. Jin, "MCT: A tool for commenting programs by multimedia comments," in *International Conference on Software Engineering (ICSE '13)*, (San Francisco, CA, USA), pp. 1339–1342, IEEE Press, May 2013.

[35] W. Heijstek, T. Kuhne, and M. R. Chaudron, "Experimental Analysis of Textual and Graphical Representations for Software Architecture Design," in *International Symposium on Empirical Software Engineering and Measurement (ESEM '11)*, pp. 167–176, IEEE, Sept. 2011.

[36] S. Hens, M. Monperrus, and M. Mezini, "Semi-automatically extracting FAQs to improve accessibility of software development knowledge," in *International Conference on Software Engineering (ICSE '12)*, (Zurich, Switzerland), pp. 793–803, IEEE Press, June 2012.

[37] R. Holmes and G. C. Murphy, "Using Structural Context to Recommend Source Code Examples Categories and Subject Descriptors," in *International Conference on Software Engineering (ICSE '05)*, pp. 117–125, 2005.

[38] R. Holmes, R. J. Walker, and G. C. Murphy, "Approximate Structural Context Matching : An Approach to Recommend Relevant Examples," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, pp. 952–970, 2006.

[39] JSDoc 3 documentation project, "JSDoc," 2011.

[40] N. Khamis, R. Witte, and J. Rilling, "Automatic quality assessment of source code comments: the JavadocMiner," in *International Conference on Natural Language Processing and Information Systems (NLDB '10)*, pp. 68–79, Springer-Verlag, 2010.

[41] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold, "An Overview of AspectJ," *Lecture Notes in Computer Science*, vol. 2072, pp. 327–353, 2001.

[42] J. Kim, S. Lee, S.-W. Hwang, and S. Kim, "Towards an Intelligent Code Search Engine," in *AAAI Conference on Artificial Intelligence*, pp. 1358–1363, 2010.

[43] D. E. Knuth, *Literate Programming*. Stanford University Center for the Study of Language and Information, 1992.

[44] J. Krinke, "Identifying similar code with program dependence graphs," in *Working Conference on Reverse Engineering (WCRE '01)*, pp. 301–309, IEEE Comput. Soc, 2001.

[45] A. Marcus, A. Sergeyev, V. Rajlich, and J. I. Maletic, "An information retrieval approach to concept location in source code," in *Working Conference on Reverse Engineering (WCRE '04)*, pp. 214–223, IEEE Comput. Soc, 2004.

[46] S. Marlow, "Haddock," 2002.

[47] P. W. McBurney and C. McMillan, "Automatic documentation generation via source code summarization of method context," *Proceedings of the 22nd International Conference on Program Comprehension - ICPC 2014*, pp. 279–290, 2014.

[48] C. McMillan, "Searching, selecting, and synthesizing source code," in *International Conference on Software Engineering (ICSE '11)*, (New York, New York, USA), p. 1124, ACM Press, 2011.

[49] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: A Search Engine for Finding Functions and Their Usages," in *International Conference on Software Engineering (ICSE'11)*, (Honolulu, HI), pp. 1043–1045, 2011.

[50] Microsoft, "MSDN," 2004.

[51] Microsoft, "SandCastle," 2006.

[52] Mike van Riel, "phpDocumentor," 2010.

[53] L. Moreno, J. Aponte, G. Sridhara, A. Marcus, L. Pollock, and K. Vijay-Shanker, "Automatic generation of natural language summaries for Java classes," *IEEE International Conference on Program Comprehension*, pp. 23–32, 2013.

[54] I. Naisan and S. Ibrahim, "Comparison of Two Software Documentation Generators: Universal Report and Doxygen," *Proceedings of International Conferences of Information, Communication, Technology (ICT'09)*, pp. 67–70, 2009.

[55] S. M. Nasehi, J. Sillito, F. Maurer, and C. Burns, "What makes a good code example? A study of programming Q&A in StackOverflow," *IEEE International Conference on Software Maintenance, ICSM*, pp. 25–34, 2012.

[56] S. Oney and J. Brandt, "Codelets: linking interactive documentation and example code in the editor," in *Conference on Human Factors in Computing Systems (CHI '12)*, pp. 2697–2706, 2012.

[57] C. Parnin and C. Treude, "Measuring API documentation on the web," in *International Workshop on Web 2.0 for Software Engineering (Web2SE '11)*, (New York, New York, USA), pp. 25–30, ACM Press, 2011.

[58] D. Poshyvanyk, M. Gethers, and A. Marcus, "Concept location using formal concept analysis and information retrieval," *ACM Transactions on Software Engineering and Methodology*, vol. 21, pp. 1–34, Nov. 2012.

[59] J. Reeves, "Codox," 2014.

[60] F.-G. Ribreau, "Doxx," 2013.

[61] M. P. Robillard, "Automatic generation of suggestions for program investigation," *ACM SIGSOFT Software Engineering Notes*, vol. 30, p. 11, Sept. 2005.

[62] M. P. Robillard and R. DeLine, "A field study of API learning obstacles," *Empirical Software Engineering*, vol. 16, pp. 703–732, Dec. 2010.

[63] T. Roehm, R. Tiarks, R. Koschke, and W. Maalej, "How do professional developers comprehend software?," in *International Conference on Software Engineering (ICSE '12)*, (Zurich, Switzerland), pp. 255–265, IEEE Press, June 2012.

[64] C. Rubio-González and B. Liblit, "Expect the unexpected: error code mismatches between documentation and the real world," in *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '10)*, pp. 73–80, 2010.

[65] L. Segal, "YARD," 2007.

[66] G. Sridhara, L. Pollock, and K. Vijay-Shanker, "Automatically detecting and describing high level actions within methods," *Procedings of the 33rd International Conference on Software Engineering - ICSE 2011*, pp. 101–110, 2011.

[67] J. Stylos, B. A. Myers, and Z. Yang, "Jadeite: improving API documentation using usage information," in *Extended Abstracts on Human Factors in Computing Systems (CHI EA '09)*, pp. 4429–4434, 2009.

[68] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens, "@ tcomment: Testing Javadoc comments to detect comment-code inconsistencies," in *IEEE International Conference on Software Testing, Verification and Validation (ICST'12)*, pp. 260 – 269, 2012.

[69] A. Terrasa, "Nitdoc," 2011.

[70] The Go Authors, "Godoc," 2009.

[71] The PHP Documentation Group, "PhD DockBook," 2007.

[72] D. Thomas, "RDoc," 2001.

[73] Universal Software, "Universal Report," 2001.

[74] G. Valure, "Natural Docs," 2003.

[75] A. van Deursen and T. Kuipers, "Building documentation generators," in *IEEE International Conference on Software Maintenance (ICSM '99)*, pp. 40–49, IEEE, 1999.

[76] D. van Heesch, "Doxygen: Source code documentation generator tool.," 2008.

[77] R. van Solingen, V. Basili, G. Caldiera, and H. D. Rombach, *Goal Question Metric (GQM) Approach.* John Wiley & Sons, Inc., 2002.

[78] T. Vestdam and K. Nørmark, "Maintaining Program Understanding–Issues, Tools, and Future Directions.," *Nordic Journal of Computing*, vol. 11, no. 3, pp. 303–320, 2004.

[79] T. Votruba, J. Hanslík, and O. Nešpor, "ApiGen," 2010.

[80] W. Wang and M. W. Godfrey, "Detecting API usage obstacles: A study of iOS and Android developer questions," in *Working Conference on Mining Software Repositories (MSR '13)*, (San Francisco, CA, USA), pp. 61–64, IEEE Press, May 2013.

[81] X. Wang, L. Pollock, and K. Vijay-Shanker, "Automatic Segmentation of Method Code into Meaningful Blocks to Improve Readability," in *Working Conference on Reverse Engineering (WCRE '11)*, pp. 35–44, IEEE, Oct. 2011.

[82] F. W. Warr and M. P. Robillard, "Suade: Topology-Based Searches for Software Investigation," in *International Conference on Software Engineering (ICSE '07)*, pp. 780–783, IEEE, May 2007.

[83] A. Wingkvist and M. Ericsson, "A metrics-based approach to technical documentation quality," in *International Conference on the Quality of Information and Communications Technology (QUATIC '10)*, pp. 476–481, IEEE, Sept. 2010.

[84] Yahoo! Inc., "YUIDoc," 2011.

[85] A. T. Ying and M. P. Robillard, "Code Fragment Summarization," *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, pp. 655–658, 2013.

[86] A. T. Ying and M. P. Robillard, "Selection and presentation practices for code example summarization," *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, pp. 460–471, 2014.

[87] C. Zhang, J. Yang, Y. Zhang, J. Fan, X. Zhang, J. Zhao, and P. Ou, "Automatic parameter recommendation for practical API usage," in *International Conference on Software Engineering (ICSE '12)*, pp. 826–836, IEEE, June 2012.

[88] Q. Zhang, W. Zheng, and M. Lyu, "Flow-augmented call graph: A new foundation for taming api complexity," *Lecture Notes in Computer Science*, vol. 6603, pp. 386–400, 2011.

[89] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO : Mining and Recommending API Usage Patterns," in *European Conference on Object-Oriented Programming (ECOOP '09)*, pp. 318–343, 2009.