ANALYSIS AND RECOMMENDATIONS FOR DEVELOPER LEARNING RESOURCES

by

Barthélémy Dagenais

School of Computer Science
McGill University, Montreal

February 2012

A thesis submitted to McGill University
in partial fulfillment of the requirements of the degree of

Doctor of Philosophy

Abstract

Developer documentation helps developers learn frameworks and libraries, yet developing and maintaining accurate documentation require considerable effort and resources. Contributors who work on developer documentation need to at least take into account the project's code and the support needs of users. Although related, the documentation, the code, and the support needs evolve and are not always synchronized: for example, new features in the code are not always documented and questions repeatedly asked by users on support channels such as mailing lists may not be addressed by the documentation. Our thesis is that by studying how the relationships between documentation, code, and users' support needs are created and maintained, we can identify documentation improvements and automatically recommend some of these improvements to contributors. In this dissertation, we (1) studied the perspective of documentation contributors by interviewing open source contributors and users, (2) developed a technique that automatically generates the model of documentation, code, and users' support needs, (3) devised a technique that recovers fine-grained traceability links between the learning resources and the code, (4) investigated strategies to infer high-level documentation structures based on the traceability links, and (5) devised a recommendation system that uses the traceability links and the high-level documentation structures to suggest adaptive changes to the documentation when the underlying code evolves.

Résumé

La documentation pour les développeurs aide ces derniers à apprendre à utiliser des bibliothèques de fonctions et des cadres d'applications. Pourtant, créer et maintenir cette documentation requiert des efforts et des ressources considérables. Les contributeurs qui travaillent sur la documentation pour les développeurs doivent tenir compte de l'évolution du code et des besoins potentiels des utilisateurs de la documentation. Même s'ils sont reliés, la documentation, le code et les besoins des utilisateurs ne sont pas toujours synchronisés : par exemple, les nouvelles fonctionnalités ajoutées au code ne sont pas toujours documentées et la documentation n'apporte pas nécessairement de réponse aux questions posées à répétition sur des forums de discussion. Notre thèse est qu'en étudiant comment les relations entre la documentation, le code, et les besoins des utilisateurs sont crées et maintenues, nous pouvons identifier des possibilités d'améliorations à la documentation et automatiquement recommander certaines de ces améliorations aux contributeurs de documentation. Dans cette dissertation, nous avons (1) étudié la perspective des contributeurs de documentation en interviewant des contributeurs de projets en code source libre, (2) développé une technique qui génère automatique un modèle de la documentation, du code, et des questions des utilisateurs, (3) développé une technique qui recouvre les liens de traçabilité entre les ressources d'apprentissage et le code, (4) examiné des stratégies pour inférer des structures abstraites de documentation à partir des liens de traçabilité et (5) développé un système de recommandation qui utilise les liens de traçabilités et les structures abstraites de documentation pour suggérer des changements adaptatifs quand le code sous-jacent évolue.

Acknowledgments

I have been fortunate to be surrounded by seasoned researchers, software engineers, and technical writers willing to share their experience, expertise, and encouragement throughout the journey that led to this thesis.

My supervisor, Martin, has always been ready to review my work quickly and to provide insightful advice, even for the 100th revision of a paper when separated by multiple timezones, a sabbatical, and countless attention-seeking tasks. He never stopped at "good enough" and always raised the bar which led to research work that I am particularly proud of. Thank you Martin.

It has been a great pleasure to work with my friend and mentor at IBM Research, Harold. I learned a lot from our research discussions, from his kindness too, and I thank him for his advice in the toughest moments.

Conducting a qualitative study and interviewing real people on the phone for the first time can be scary if you are used to quantitative studies and totally afraid to pick up the phone in general. I thank Rachel for helping me improve my interviewing techniques and analysis skills, and for giving me confidence in the qualitative work I was doing.

I am thankful to the contributors of open source projects, senior software engineers, and technical writers who accepted to squeeze an interview with me in their busy schedule. I never expected to be part of a family barbecue (over the phone), to speak with a groom a day after his wedding or to hear so many war stories from technical writers. What I learned from these interviews will be useful for the rest of my career.

My colleagues, Annie, David, Ekwa, and Tristan, were always available to bounce ideas with me and review my papers. Thank you: it has been fun working with you guys.

My parents were my biggest supporters, always ready to read my papers, cheer me up when I had doubts, and tell me that I would finish with a Ph.D. Merci à vous deux!

Finally, I would like to thank my fiancée, Geneviève, for her continuous encouragements and support during the most difficult times and for showing me how to remain calm no matter what. I am also thankful that Geneviève was always ready to discuss research methodologies and statistics when I needed help.

My doctoral studies were financially supported by a NSERC Alexander Graham Bell Canada Graduate Scholarship, a FQRNT Doctoral research scholarship, and a McGill University Graduate Fellowship.

Barthélémy Dagenais
McGill University
February 2012

Contents

List of Figures

List of Tables

Chapter 1
Introduction

Developers usually rely on libraries or application frameworks[1] when building applications. Frameworks provide standardized and tested solutions to recurring design problems. For example, hundreds of applications like Google Code Search and Twitter use the JQuery framework to provide an interactive user experience with Javascript and AJAX.[2]

To use a framework, developers must learn many things such as the domain and design concepts behind the framework, how the concepts map to the implementation, and how to extend the framework [40]. Various types of documents are available to help developers learn about frameworks, ranging from Application Programming Interface (API) documentation to tutorials and reference manuals. A few studies have been conducted to assess the effectiveness of various documentation types [11, 12, 55] and to find out what documentation properties industrial developers desire [31, 51].

We found during a literature review (see Chapter 6) that the creation and maintenance of developer documentation was barely studied and that the documentation process is thus poorly supported. For example, the Spring Framework manual[3] has approximately 200 000 words (twice the size of an average novel) and has gone through

[1] Unless otherwise specified, we use the term framework to represent any reusable software artifacts such as libraries and toolkits.

[2] http://docs.jquery.com/Sites_Using_jQuery

[3] References to project and documentation tools are presented in Table A.2 in the Appendix

five major revisions. Creating and maintaining this documentation potentially represents a large effort yet we do not know what kind of problems documentation contributors encounter or what factors they consider when working on the documentation.

Contributors who work on developer documentation need to at least take into account the project's code (the documentation teaches how to use the code) and the needs of users (the documentation reader). Although related, these three entities, documentation, code, and users, evolve and are not always synchronized: for example, new features in the code may be left undocumented or questions repeatedly asked by users on support channels (e.g., mailing lists) may not be addressed by the documentation.

The main technical challenge in automatically linking and synchronizing documentation, support channels, and the project's code comes from the inherent ambiguity of unstructured natural language. For example, if a sentence in the documentation mentions the "save" method, it may not be obvious to which method declaration the documentation is referring to (e.g., if there are more than one save method). Automatically disambiguating such reference would enable the development of many kinds of analyses and documentation tools such as the automatic detection of incorrect references to deprecated or deleted code elements in the documentation.

We conducted a qualitative study with core contributors of large and popular open source projects and we analyzed the motivations and the consequences of documentation decisions made by these contributors to devise a model of developer documentation and to design techniques that improve the documentation process and increase the documentation quality. Our thesis is that by studying how the relationships between documentation, code, and users' needs are created and maintained, we can identify documentation improvements and automatically recommend some of these improvements to contributors.

Following our qualitative study, we devised a documentation analysis tool chain that can automatically generate a model of developer learning resources and link this model with a project's codebase. We then created a recommendation system that

suggests corrections and improvements to the documentation when the underlying codebase evolves.

The documentation analysis tool chain that we present in this dissertation opens many research opportunities on developer documentation. In the past, researchers and practitioners have mostly relied on manual inspection of documents [57] or coarse-grained modelization techniques [8]. Our work enables the systematic and automated analysis of documentation and support channels, and the generation of a fine-grained documentation model that can be used to compute quality metrics, suggest recommendations, and test hypotheses (e.g., on the relationship between documentation structure and learnability) at a larger scale than it is possible with manual inspection.

1.1 Application Frameworks and Developer Documentation

The main advantages of application frameworks are modularity, reusability, extensibility, and inversion of control [30]. Frameworks promote modularity and abstraction by encapsulating ever changing implementation behind stable interfaces that developers can use in their applications. Because frameworks are modular, a change in the framework should have a localized impact on the dependent applications [50]. Frameworks are also reusable in the sense that they define generic components that can be customized and reapplied to create applications.

Developers build software applications by extending frameworks from well-defined extension points (also called hotspots or hooks). These extension points require application developers to extend classes, call methods, write configuration files, or use a combination of these strategies. Frameworks are also characterized by inversion of control [44]: frameworks are responsible for the general control flow of the application and they dispatch events to application-specific extensions plugged into the framework's extension points. Inversion of control enables developers to concentrate

on application-specific features instead of having to implement a general dispatching mechanism such as the standard control flow of a web shopping cart.

We recognize two types of framework documentation. The first type of documentation supports framework developers in maintaining the framework or adding new features to the framework itself. This documentation usually describes the coding conventions, the development process, or the architecture and the internal design of the framework. This type of documentation is associated with well-known practices and is supported by international standards [37] and research [14]. Our dissertation focuses on the second type of documentation, which is oriented towards application developers who extend a framework to create new applications. This type of documentation may describe the elements accessible by application developers (reference documentation), how to perform common tasks with the framework (getting started documentation), and the main concepts and terminology used by the framework (conceptual documentation).

1.2 State of the Art on Developer Documentation

Most of the research work on developer documentation has focused on studying how developers use documentation and devising techniques to better document programs. We present a brief overview of the past studies and current techniques related to developer documentation. Chapter 6 provides a more comprehensive survey of the various techniques that can help the creation, analysis, and maintenance of documentation.

1.2.1 How Developers Use Documentation

Most of what we know about how developers use documentation can be traced back to the work of Carroll et al. [11], which in itself is a confirmation of the adult learning theories by Knowles [41]. In controlled experiments, Carroll et al. observed that the step-by-step progress induced by traditional documentation such as tutorials and reference manuals was often interrupted by periods of self-initiated problem solving by users. Indeed, users ignored steps and entire sections that did not seem related to real

tasks, and they often made mistakes during their unsupervised exploration. Because this active way of learning was not what the designer of traditional documentation intended, Carroll et al. proposed four characteristics that are essential to effective documentation:

1. Documentation should be action-oriented by focusing on real tasks and activities.
2. The amount of text should be kept to a minimum and the choice of words should reflect the knowledge of the user and not the underlying software concepts.
3. Common errors should be mentioned along with resolution steps.
4. Each section should be short and self-contained and it should provide exploration starting points.

Researchers confirmed in different studies that these four characteristics indeed made the documentation more effective than traditional, systematic, documentation [11, 12, 53, 58].

1.2.2 Generating Documentation

The classes and methods of frameworks are often documented to explain in natural language what the roles of the classes are and what the methods do. Support for authoring and visualizing API documentation has evolved greatly: at first, these activities were supported by proprietary tools (e.g., VisualWorks), then, languages like Java provided a standard way to tie the API documentation to the source code (Javadoc [42]). Similar tools for other languages followed (e.g., Doxygen supports 11 languages like C++ and Python). Nowadays, these tools automatically generate and update API documentation from the comments inlined in the code. For example, Javadoc automatically generates the list of API elements that use a particular class.

Because framework users must combine the various methods and classes of a framework, researchers have devised many techniques that mine client programs to infer usage information. For example, Acharya et al. devised a technique that extracts

compact partial orders of framework methods from client programs [5]. Michail used association rules taking into account instantiations and inheritance relationships to mine library usage patterns in client programs [46]. For example, a usage pattern could be that classes inheriting from a library class `A` usually create an instance of class `B` from the library.

Other techniques have been devised to suggest relevant code examples for common framework usage tasks. For example, MAPO mines open source repositories and indexes API usage patterns, i.e., sequence of method calls that are frequently invoked together [62]. Then, MAPO recommends code snippets that implement these patterns, based on the programming context of the user.

1.2.3 Modelization of Documentation

Few researchers have attempted to categorize and modelize developer documentation. Kirk et al. conducted case studies on framework usage and identified four general kinds of learning problems that documentation must address: (1) mapping or finding out which concrete class should be used to implement a particular concept, (2) interactions or finding out how classes communicate together in the presence of polymorphism, inversion of control, and subtle dependencies, (3) functionality or finding out what a framework class does, and (4) architecture or how to make modifications that are consistent with the architecture [40].

Butler et al. classified documentation into ten categories such as recipe, framework overview, and reference manuals [10]. The authors identified the type of information provided by each document and their granularity. The categorization was based on the authors' experience and has not been empirically evaluated.

1.2.4 Documentation Evolution

As frameworks evolve, the documentation must be maintained to reflect the changes such as feature addition, refactoring, and code element deprecation and deletion.

Robillard proposed concern graphs as a solution to document scattered concerns that evolve [52]. The idea is to capture a set of related code elements as an intension

(e.g., all callers of method m1()) and an extension (e.g., the actual set of callers). When the concern evolves and new callers are added, the concern intension automatically captures the new extension, which makes the solution robust to evolution. Finding an appropriate intension for a particular concern can be challenging: this is why we devised ISI4J, a technique that automatically infers a set of intensions from a set of code elements manually selected by a software developer [19].

Other techniques try to find replacements for code elements that were deleted or deprecated. These techniques automatically document *migration paths* for client programs using a framework. For example, SemDiff analyzes the method calls evolution in the source code history of a framework to recommend method replacement [24].

1.2.5 Identification of Code in Documentation

As we noted earlier, one of the main challenge in automatically analyzing documentation comes from the inherent ambiguity of unstructured natural language. Two research projects in this field have influenced our work on documentation.

Bacchelli et al. generate a model of email messages from support channels and link the messages to a framework model [8]. Both models are fine-grained: structural relationships are represented in the framework model (e.g., inheritance and member declarations) and code fragments are identified in the email messages (e.g., method call, field reference). The inferred links between the email messages and the framework model are coarse-grained though because only references to class names in emails are matched to actual classes in the model.

In a previous work, we created Partial Program Analysis (PPA) to parse incomplete Java programs such as code snippets [20]. PPA accepts source code that cannot be compiled (e.g., it contains only a method body) and produces type-resolved Abstract Syntax Trees by completing the abstract syntax tree and by inferring the missing declarations. Such incomplete programs are frequent in developer documentation and are encountered in the form of embedded code fragments and code snippets.

1.3 Challenges in Creating and Maintaining Documentation

Although the documentation needs of developers have been extensively studied, we observe that the documentation process from the perspective of the contributors have rarely been studied and we do not know if the current documentation tools and practices are adequate and cost-effective. To complicate matters, there is no standard documentation terminology or format, which makes comparisons across tools and projects difficult. Because developer documentation is written in unstructured natural language, which is fundamentally difficult to parse, automated analysis and modelization of documentation has been limited so far. We review the main challenges encountered by practitioners and researchers when analyzing, creating and maintaining documentation.

Complex factors and consequences of documentation decisions.

While certain types of documentation, such as API documentation, are systematic by their nature, other types of documentation requires more thought and can serve multiple goals.

For instance, how do documentation writers decide which framework elements to cover in a tutorial? What factors do they consider when they allocate their documentation time between writing code examples, tutorials and API documentation? What are the consequences of documenting a change quickly after making it as opposed to waiting just before a full release?

For example, some open source contributors consider that tutorials do not only provide a learning aid, but that they are part of the project's marketing [23]. A good tutorial demonstrates the key features that a framework offers and how easy it is to use these features. This additional consideration, marketing, partly explains why documentation authors will accept the high cost associated with writing and maintaining a tutorial. It also indicates that relying on automatic documentation generation alone would probably not be an appropriate strategy as both the choice

of the elements that are covered by a tutorial and how they are covered are partially related to the project's marketing.

Learning more about the documentation decisions made by contributors can help us identify improvement opportunities, but also avoid documentation tools whose helpfulness would be limited at best.

Evolving Documentation.

A framework's code can change significantly between two releases, but if the changes are mostly related to the implementation, only a small subset of the new code elements will be documented. The problem then becomes: given all the considerations noted so far (marketing, previous documentation choices, new features), how can we identify the new code elements that should be documented in a new release?

For example, between two releases of the Hibernate framework, 5661 new code elements were added in the codebase, but only 6 of these code elements were mentioned in the new release of the documentation. To be consistent with previous documentation choices, deciding which code elements to document in a new release is time-consuming because the documentation is large (70 900 words) and there are many changes to consider. Although a documentation tool cannot take into account all factors (e.g., marketing), it could infer past documentation decisions and indicate which new code elements match these decisions.

Another type of changes are the deletion or deprecation of code elements. Between two releases of the Hibernate framework, 166 code elements were deprecated, but only three references out of 29 in the documentation were corrected.[4]. In other words, the documentation still mentions 26 code elements that have been deprecated in favor of newer elements. Because the documentation and the codebase are not linked, the only way to search for code element references is to use textual search tools such as grep. Unfortunately, if a deprecated code element has a common name, regular expression searching tools can produce a large number of false positives. For example, the following method was deprecated in Hibernate 3.5.5: `Session.get(String, Serializable, Lock)`. Because the method name, *get*, is a common word, a textual search in the

[4]We studied the evolution of the code and the documentation of four open source projects in Chapter 5. The number of deprecated code elements and references come from Table 5.1 and Table 5.4

Hibernate manual returned 296 matches, but only one of these matches actually referred to the deprecated method declaration. Complex regular expressions could disambiguate common words from method calls, but they cannot identify the method parameters' type and in the absence of a detailed code model, each deprecated method must be manually fed to the textual search tool. For large projects with many code changes and extensive documentation, it is easy to forget to document a change.

Recovering Traceability Links.

Deciding when, how, and what to document is complicated by the numerous factors contributors need to consider, but as shown in the previous paragraphs, the difficulty is compounded by the lack of traceability links between the code and the documentation. As opposed to API documentation that is automatically generated and where each documented element can be clearly linked to a code declaration, the code references in manuals and tutorials are ambiguous.

Automatically linking documentation and code implies two main challenges: (1) documentation content must be first categorized (English, XML snippet, Java snippet, method call, etc.), and (2) methods and fields often have common names and are declared in various types. For example, in the four open source systems we extensively studied (see Section 3.3), each method name mentioned in the documentation matches on average 16.8 method declarations in 13.5 types.

Humans can figure out which exact method declaration is mentioned in the documentation because they understand the context in which the method is mentioned. Consider the simplified documentation example presented in Figure 1.1. Although the method `createQuery` is declared in five types in the Hibernate framework, we can precisely find which method declaration is referred to if we know that:

1. `createQuery` is mentioned in Section A.1.
2. Section A.1 is part of Section A.
3. `Session` is mentioned in Section A, so it is in the *context* for `s.createQuery`.
4. `Session` declares `createQuery`.

Section A

You always obtain a Query using the current Session.

Section A.1

Hibernate queries sometimes return tuples of objects:
Iterator kitten = s.createQuery("select kitten from Cat").

Figure 1.1: Documentation Example Loosely Adapted from the Hibernate 3.5 Manual.

1.4 Documentation Analysis Tool Chain

As we discussed in Section 1.3, the main technical limitation to the creation of advanced documentation tools and processes is the lack of fine-grained traceability links between the developer learning resources and the code. In this dissertation, we present RecoDoc, a documentation analysis tool chain that is able to (1) generate a model of the documentation, code, and support channels of a project, (2) link the various models together at a fine level of granularity, and (3) recommend improvements to the documentation based on these models.

1.4.1 Documentation Model

Given the code, the documentation, and the support channel archive (e.g., mailing list archive), our tool chain generates detailed models of these artifacts. Figure 1.2 shows a partial model of the links and artifacts that would be automatically generated for the documentation example presented in Figure 1.1.

At first, a set of parsers would go through the Java code and the HTML code of the documentation to generate the model components (represented by boxes) and the basic relationships (represented by plain lines). As it can be seen from the model, we distinguish actual code elements (the class `Session`) from the references to these code elements (the *code-like term* Session).

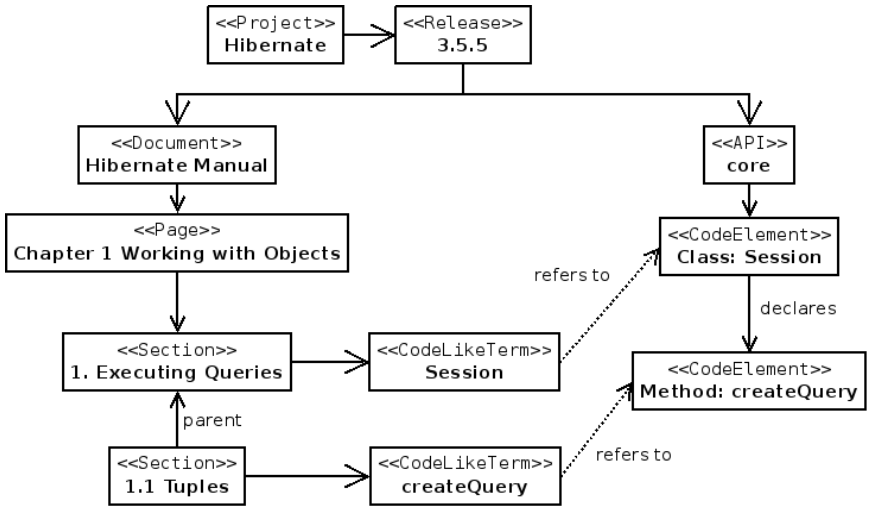

Figure 1.2: Model of the Hibernate Documentation and Code

As we explained in Section 1.3, the relationships in the model are essential to understand the context in which a code-like term such as Session is mentioned. The important relationships for our example are labeled in Figure 1.2.

1.4.2 Recovering Fine-Grained Traceability Links

In the second step, a linker (part of our tool chain) attempts to recover the traceability links between the documentation model and the code model. Our linker is able to link class-level (coarse-grained) and subclass-level elements (fine-grained elements such as methods and fields) to code references in the documentation. These traceability links are represented by dotted arrows in Figure 1.2.

For example, to be able to link the code-like term `createQuery` with the code element `Session.createQuery`, our linker first searches for all code elements whose name is `createQuery`. Then, using a pipeline of filters, the tool tries to identify the correct code element.

In our example, there are five code elements named `createQuery` declared in five types (e.g., `Session`, `AbstractEntityManagerImpl`, ...). One filter in the pipeline called

Global Context Filter would notice that one of the declaring type, `Session`, is mentioned in the parent section of the `createQuery` and thus, it must be the right declaring type.

Our linker can capture and interpret more complex relationships, such as inheritance and various granularity levels of context. **The hypothesis guiding the design of our filtering approach is that elements referenced closer to each other are more likely to be related than elements referenced further part**.

In an evaluation study with four large open source systems (see Section 3.3), we found that our linker could link code-like terms to code elements with a high precision and recall (96%).

1.4.3 Recovering High-Level Documentation Structures

The third step performed by our tool chain is to recover high-level documentation structures. For example, let us assume that section A.2 in the Hibernate documentation describes most of the methods declared by the `Session` class (e.g., `createQuery`, `save`, `load`).

It can be useful for practitioners to know not only that each individual method is mentioned in Section A.2, but also that there is a more abstract relationship being documented. For instance, our tool chain would be able to infer that Section A.2 is linked to a *documentation pattern*, i.e, a coherent set of code elements. This documentation pattern is similar to a concern as it has an intension, "all methods declared by the class `Session`", and an extension, {`createQuery, save, load, ...`}.

Our tool chain finds documentation patterns by computing the set of all possible intensions and extensions and by computing the coverage of each pattern. For example, if 4 of the 5 methods declared in the `Session` class are mentioned in the documentation, the coverage of the documentation pattern is 80%. Patterns with a coverage lower than 50% are filtered out.

We manually inspected the documentation patterns inferred by our tool chain on four open source projects (see Section 4.1.2) and we found that 82% of the inspected

documentation patterns were meaningful, i.e., the inferred intension matched the focus of the documentation section.

1.4.4 Recommending Adaptive Changes

The last step performed by our tool chain is to recommend adaptive changes by using the fine-grained traceability links and the high-level documentation structures recovered in the previous steps.

For example, if the method `lock` is added to the `Session` class in Hibernate 3.6, our recommender will detect that this new method is matched by the documentation pattern "all methods declared by the class `Session`" in Section A.2. The recommender will thus suggest to mention the new method `lock` in Section A.2 of the Hibernate 3.6 documentation release. When we evaluated our recommender on four open source systems, we found that 50% of the new code elements mentioned in a documentation release could be explained by documentation patterns inferred by our tool chain. Considering that multiple factors (e.g., marketing, learnability, and past decisions) are involved in the decision to document a new code element, the high-level documentation structures by our tool chain could reduce the decision time by suggesting code elements that are related to previous documentation decisions.

Finally, our recommender can also use low-level links to suggest adaptive changes. For example, if the method `createQuery` is deprecated in Hibernate 3.6, our recommender will suggest to correct the reference to `createQuery` in Section A.1. In the four open source systems we studied (see Section 5.3), RecoDoc found 103 references to deprecated code elements and 31 of these references had not been corrected by the documentation authors to this date. The precision and recall of RecoDoc (90% and 99% respectively) was superior to the precision and recall of simple textual matching tools like grep (48% and 99% respectively).

1.5 Organization of the Dissertation

This dissertation consists of two main parts: the qualitative study that we conducted to better understand how open source contributors create and maintain documentation, and the presentation of our documentation analysis tool chain motivated and guided by the study.

More precisely, in Chapter 2, we present a qualitative study we conducted to better understand the factors that open source contributors consider when making documentation decisions, and the consequences of these decisions on their project. We interviewed 12 core contributors of large and popular open source projects and 10 experienced developers who had extensively used open source documentation. The description of the decisions, their factors, and their consequences in the context of the three documentation modes we identified forms the first contribution of this dissertation. This chapter was also published in the proceedings of the International Symposium on Foundations of Software Engineering [23].

Following our qualitative study, we devised a meta-model describing fine-grained components and relationships of the documentation, code, and support channels of a software system. In Chapter 3, we present the main components and relationships of the documentation meta-model. Then, we provide the details of our tool chain: how we parse artifacts to generate a model, how we link the models together, and how we evaluated the tool chain on four large open source systems. The description of the documentation meta-model and the presentation of the documentation analysis tool chain constitutes the second and third contributions of this dissertation. This chapter was published in the proceedings of the International Conference on Software Engineering [25].

In Chapter 4, we present an exploratory study that we conducted to infer high-level documentation structures such as (1) documentation patterns, and (2) relationships between documentation sections and support channel messages. We show that we can find meaningful documentation patterns and relationships between messages and sections with a relatively high precision, but that only documentation patterns can currently be used to recommend documentation improvements. The description of

the high-level structures inference strategies and their evaluation on four open source systems forms the fourth contribution of this dissertation.

By relying on the low-level traceability links and high-level structures inferred from the previous chapters, we describe a recommendation system that suggests adaptive changes to documentation in Chapter 5. When a new release of a software system is produced, our recommendation system can (1) precisely identify references to deprecated code elements in the documentation, and (2) recommend which new code elements should be documented and where in the documentation. The presentation of the recommendation strategies and their evaluation on four open source systems form the fifth contribution of this dissertation.

In Chapter 6, we provide an overview of the related work and highlight the key differences and novelties of our research. Finally, in Chapter 7, we discuss future work involving our documentation analysis tool chain and we conclude by summarizing the contributions of the work described in this dissertation.

Chapter 2
Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors

Creating and maintaining developer documentation represents a large effort yet we do not know the kind of problems documentation contributors encounter, the factors they consider when working on the documentation and the impact their documentation-related decisions have on the project. For instance, does documenting a change immediately after making it have different consequences than documenting all changes before a release? Answering these questions provides insights about the techniques that are needed to optimize the resources required to create and maintain developer documentation.

We conducted an exploratory study to learn more about the documentation process of open source projects [23]. Specifically, we were interested in identifying the documentation decisions made by open source contributors, the context in which these decisions were made, and the consequences these decisions had on the project. We performed semi-structured interviews with 22 developers or technical writers who wrote or read the documentation of open source projects. In parallel, we manually inspected more than 1500 revisions of 19 documents selected from 10 open source projects.

Among many findings, we observed how updating the documentation with every change led to a form of embarrassment-driven development, which in turn led to an improvement in the code quality. We also found that all contributors who originally selected a public wiki to host their documentation eventually moved to a more controlled documentation infrastructure because of the high maintenance costs and the decrease of documentation authoritativeness. Such observations could enable practitioners to make informed decisions by analyzing the trade-offs encountered by their peers and enable researchers to build documentation tools that are adapted to the documentation process.

2.1 Method

We based our exploratory study on grounded theory as described by Corbin and Strauss [15]. Grounded theory is a qualitative research methodology that employs *theoretical sampling* and *open coding* to formulate a theory "grounded" in the empirical data. By following grounded theory, we started from general research questions and refined the questions, and the data collection instruments, as the study progressed. As opposed to random sampling, grounded theory involved refining our sampling criteria throughout the course of the study to ensure that the selected participants were able to answer the new questions that have been formulated. For example, after having interviewed two contributors of Perl projects, we filtered out further Perl projects; after having interviewed four contributors from library projects, we sent more invitations to contributors of framework projects.

We analyzed the data, collected through interviews and document revisions, using open coding: we assigned codes to sentences, paragraphs, or revisions and we refined them as the study progressed. We then reviewed the codes several times and linked them to emerging categories, a process called axial coding. Finally, the goal of a study using grounded theory is to produce a coherent set of hypotheses laid in the context of a process, that originates from empirical data.

All reported observations are linked to specific cataloged evidence. For instance, as we present our observations, we mention the codes of the participants from which we derived our conclusions. We also provide a summary of our findings in Table 2.4, which shows the list of participant codes related to contributed to each finding.

Our method follows that of previous software engineering studies based on grounded theory [6, 22, 27]. These references provide an additional discussion on the use of grounded theory in software engineering.

2.1.1 Data Collection

We learned about the documentation process of open source projects by gathering data from three sources. We interviewed developers who contributed to open source projects and their documentation (the *contributors*): these developers were often the founder or the core maintainer of the project.[1] Most of the observations reported in this dissertation come from these interviews. We also interviewed developers who frequently used open source projects and who read their documentation (the *users*). We wanted to determine how developers used documentation and what kind of documentation was the most useful to them. Finally, we analyzed the evolution of 19 documents from 10 open source projects (the *historical analysis*). Because some projects started more than 15 years ago, it was often difficult for the participants to remember the various details of the documentation process. Our systematic analysis of the revisions provided us with a more comprehensive and detailed view of that documentation's evolution.

The projects of the contributors, the users, and the historical analysis were selected in parallel so they are not necessarily the same. We used this strategy to preserve the anonymity of the contributors and to allow us to provide concrete examples by naming real open source projects when discussing observations from the users' interviews and the historical analysis. Additionally, this sampling strategy enabled us to perform data triangulation by evaluating our observations on different projects.

[1] Unless otherwise specified, we assume that the contributors have commit access to their project's repository.

The Contributors. To recruit contributors, we began by making a list of open source projects that were still being used by a community of users and that were large enough to require documentation to be used. We relied on Google and ohloh[2] to search for open source projects and we only selected projects that fulfilled these five criteria:

1. The project offered some reuse facilities for programmers (e.g., frameworks, libraries, toolkits, extensible applications),
2. The project was more than one year old.
3. There was at least one active contributor in the last year (e.g., a contributor answered a question on the mailing list in 2009).
4. The project had more than 10k lines of source code.
5. The project had more than 1000 users (measured by the number of downloads, issue reporters, or mailing list subscribers).

We selected projects from a wide variety of application domains and programming languages to ensure that our findings were not specific to one domain in particular.

After having selected a project, we looked at its web site and at the source repository to identify the main documentation contributors. When in doubt, we contacted one of the founders or core maintainers. We sent 49 invitations to contributors, 12 of which accepted to do an interview.

Each contributor who accepted our invitation participated in a 45-minute semi-structured phone interview in which we asked open-ended questions such as "how did the documentation evolve in your project?" and "what is your workflow when you work on the documentation?". As we explained in Section 2.1, we did not ask the same questions to all contributors because the questions evolved as the study progressed, but we included the main questions we asked during the first and the last interview in Appendix D.

[2] http://www.ohloh.net/

Participant Code	Project Age (years)	Project Domain
C1	> 5	General Purpose Web Library
C2	> 1	General Purpose Library
C3	> 15	Database Library
C4	> 10	General Purpose Library
C5	> 5	Web App. Framework
C6	> 10	Databinding Framework
C7	> 5	Blogging Platform
C8	> 5	Web App. Framework
C9	> 15	Database
C10	> 5	Web App. Framework
C11	> 15	General Purpose Library
C12	> 15	Web Server

Table 2.1: Documentation Writers (Contributors)

A few contributors talked about various projects they worked on or used, but most contributors focused on one project. Table 2.1 shows the profile of the contributors we interviewed: the number of years since the project started and the general domain covered by the project. To preserve the anonymity of the participants and the projects, we did not report the experience of the individual contributors and the main programming language of the projects. All of our participants had more than five years of programming or technical writing experience (up to 25 years) and the programming language of the projects also varied greatly: Perl (2 contributors), Java (2), Javascript (1), C(2), C++ (1), PHP (2), Python (2).

The Users. To recruit developers who used open source projects and read documentation, we relied on the list of users of stackoverflow.com, a popular collaborative web site where programmers can ask and answer questions. We wanted to interview users who had various amounts of expertise in terms of programming languages and years of programming experience. Stackoverflow user profiles indicate how many questions each user has asked and answered and the tags associated with these questions (e.g., a question might be related to *java* and *eclipse*). We filtered out all users who did not have contact information published on their profile and who were primarily answering

Participant Code	(years) Exp.	Project Domain	Programming Language
U1	> 10	Web Applications	Java, PHP
U2	> 10	System Prog., Database	Perl
U3	> 20	System Prog.	C
U4	> 10	System Simulators	C,C++,Java
U5	> 5	Web Applications	Python, Java, C
U6	> 5	Financial Applications	Java
U7	> 5	Web Applications	PHP
U8	> 25	Database	C++
U9	> 25	Web Applications	PHP
U10	> 3	Web Applications	PHP

Table 2.2: Documentation Users

questions related to the .NET platform because we judged that they were less likely to have a rich experience with open source projects.[3]

We sent 38 invitations and recruited 10 participants. We sent each participant an email asking for a list of open source projects that had good or bad documentation. We purposely did not define good or bad documentation because we wanted the participants to elaborate on their definition during the interview. Each developer participated in a 30-minute semi-structured phone interview that focused on their experience with the documentation of the projects they selected, and then, on their experience with documentation in general. Because developers may have to write documentation as part of their work, certain developers provided insights on their documentation process and we took into account these observations in our study.

Table 2.2 shows the profile of the developers we interviewed: the number of years of programming experience, the main field they are professionally working in, and the programming languages they mentioned during the interview. Most participants used many open source projects as part of their work or as part of hobby projects so their documentation needs are not exclusive to their field of work.

[3]The documentation experience of .NET developers is of interest, but not for this particular study on open source projects. We are aware that with the CodePlex project (www.codeplex.com), open source projects in .NET are becoming more mainstream.

Project	Prog. Lang.	Domain	Document	Length Words	Age Yrs	#CS	#C	%CC
Django	Python	Web Fmk.	Tutorial Part 1	3700	4.25	89	11	31%
			Tutorial Part 3	2692	4.25	61	7	57%
			Model API	4140	4.25	191	7	53%
WordPress	PHP	Blogging Platform	Writing a Plug-in	2523	4.00	126	56	wiki
			Plug-in API	2013	4.75	127	56	wiki
KDE Plasma	C++	GUI Fmk.	Getting Started	1521	2.00	51	21	wiki
			Plasma DataEngines	1854	0.75	10	7	wiki
Hibernate 3	Java	Databinding Fmk.	QuickStart	2497	1.00	21	2	9%
			Collections Mapping	3076	1.00	37	4	11%
Spring	Java	Application Fmk.	Beans Framework	30061	4.50	233	15	18%
			Transactions	9584	5.75	87	9	26%
GTK+	C	GUI Fmk.	GTK+ 2.0 Tutorial	56765	9.00	54	10	28%
Firefox	XML	Web Browser	How to build an extension	3163	4.25	316	143	wiki
DBI	Perl	Database Lib.	Module Documentation	34221	5.00	145	3	19%
Shoes	Ruby	GUI Fmk.	Manual	18887	1.00	34	5	6%
Eclipse	Java	Application Fmk.	Creating the plug-in project	559	4.75	16	6	25%
			Application Dialogs	720	7.25	26	8	4%
			Documents and Partitions	841	6.00	24	8	8%
			Resources and the file system	1638	7.75	26	8	8%

Table 2.3: Evolution of Documents

The Historical Analysis.

We systematically analyzed the evolution of documents of open source projects that maintained their documentation in a source repository (e.g., CVS) or in a wiki. We also used the same criteria as for the contributors to select projects for our historical analysis.

For each project, we selected from one to four documents. The first document was a tutorial or a similar document that told users how to get started with the project. The second document was a reference document . We assumed that these two types of documents were distinct enough that they might exhibit different evolution patterns. We had to analyze a different number of documents per project because there is no documentation standard across projects and it was impossible to compare documents of the same size or of the same nature. For example, documents ranged from a complete manual in one file (e.g., the GTK Tutorial) to document sections separated in small files and presented on many pages (e.g., Eclipse help files).

We analyzed the history of the documents by looking at their change comments and by comparing each version of the documents. This was necessary because often the change comment was not clear enough. For example, a commit comment mentioned fixing a "typo", but in fact, the actual change shows a code example being modified. Through several passes of open coding, we assigned a code to each revision to summarize the rationale behind the change. Table 2.3 shows descriptive statistics of the documents we inspected such as the time between the first and last revision *that we could find* (in years), the number of change sets (#CS), the number of different committers who modified the files (#C), and the percentage of revisions that originated from community contributions (%CC). We report the details of the revision classification in Appendix A.

We considered that all revisions that mentioned a bug number, a contributed patch, or a post from a forum or a mailing list originated from the community. It was not always possible to determine the source of the change when the documents were hosted on a wiki, so we indicated "wiki" in the table.[4]

[4]This is only a rough estimate because core contributors sometimes create bug reports themselves and other times, they forget to include the source of the change request.

2.2 Conceptual Framework

Following the analysis of the interviews and the document revisions, we identified three *production modes* in which documentation of open source projects is created. Although we expected documentation to be produced in different modes, the study helped us concretize what these modes were and what they corresponded to in practice. These production modes guided our analysis of the main decisions made by contributors (Section 2.3). Figure 2.1 depicts how the documentation effort was distributed in the lifecycle of the open source projects we studied.

First, contributors create the *initial documentation*, which requires an upfront effort that is higher than the regular maintenance effort. Then, as the software evolves, contributors *incrementally change* the documentation in small chunks of effort (e.g., spending 20 minutes to clarify a paragraph). Sometimes, major documentation tasks such as the writing of a book on the project requires a *burst* of documentation effort.

In addition to the three production modes, we note that documentation writers make important *decisions* at specific *decisions points*. As illustrated in Figure 2.2, decisions are influenced by contextual factors and they have consequences in terms of required effort and impacts for the project. This chapter focuses on the relationships between the decisions, their factors, and their consequences.

For example, for the decision point "When to adapt the documentation to the project's evolution", there are many possible decisions (e.g., updating the documentation shortly after making a change, before an official release, before making a change, etc.).

The decisions related to a decision point are not mutually exclusive, but each decision has some specific effort and impact associated with it. The consequences of a decision can also become a factor over time. For example, four contributors sought to document their changes as quickly as possible after realizing that they often improved their code while documenting. We analyzed the consequences of the documentation decisions from many perspectives (contributors, users, and evolution) to evaluate the trade-offs involved with each decision.

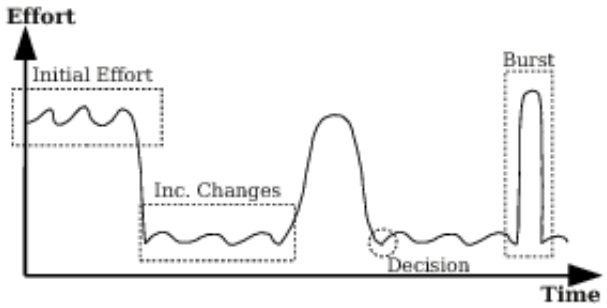

Figure 2.1: Documentation production modes

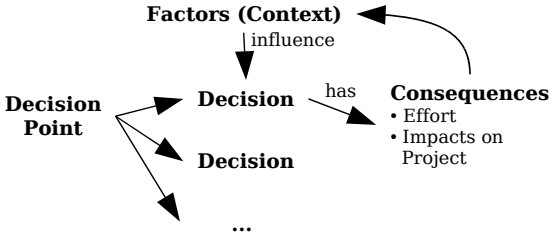

Figure 2.2: Decisions made in a documentation production mode

2.3 Decisions

We provide an overview of the documentation production modes and the decisions points. Then, we discuss in detail the six decisions that had the largest impact on the documentation creation and maintenance of the projects we studied, as determined by our analysis. Underlined sentences represent major observations for each decision. Table 2.4 provides a summary of the consequences of these six decisions on five aspects of open source projects.

Initial Effort.

When a project starts, contributors encounter two main decision points. First, contributors must select tools to create, maintain, and publish the documentation. There are three main types of infrastructure that are used by contributors, sometimes in combination with each other: wikis (see Section 2.3.1), documentation suites (e.g., POD, Sphinx, or Javadoc), and general documents such as HTML.

In our historical analysis, we observed that the editing errors (e.g., forgetting a closing tag) caused by the syntax of any documentation infrastructure were responsible for an important amount of changes and that better tool support could probably mitigate this problem: 55.4% in Eclipse (HTML), 11.4% in Django (Sphinx), 11.1% in GTK (SGML), and 6.7% in WordPress (wiki).

A second decision point that developers encounter early on concerns the type of documentation to create. Contributors typically create one type of documentation initially and the documentation covers only a subset of the code. Then, as the project evolves, contributors create more documents of various kinds. After analyzing the interviews of both contributors and users, we identified three types of documentation based on their focus: a task is the unit of *getting started documentation* (Section 2.3.2), a programming language element (e.g., a function) is the unit of *reference documentation* (Section 2.3.3), and a concept is the unit of *conceptual documentation*. These documentation types are consistent with previous classification attempts [10, 11].

Incremental Changes.

Small and continuous incremental changes are the main force driving the evolution of open source project documentation. We noticed in our historical analysis that all changes except a few structural changes and the first revisions concerned a few words or a few lines of code and that these changes occurred regularly throughout the project history (see Table A.1 in the Appendix). In this production mode, open source contributors encounter two major decision points: how to adapt the documentation to the project's evolution and how to manage the project community's contributions.

We found in our historical analysis that software evolution motivated at least 38% of the revisions to the documents we analyzed (adaptation and addition changes). We encountered five strategies (i.e., decisions) that contributors used to adapt the documentation to the project's evolution: contributors (1) updated the documentation with each change (Section 2.3.4), (2) updated the documentation before each release, (3) relied on a documentation team to document the changes (Section 2.3.5), (4) wrote the documentation before the change and used it as a specification, or (5) did not document their changes.

The second decision point contributors encounter is to determine how to manage the documentation contributions from the community. These contributions come in various forms: (1) documented code patches (Section 2.3.4), (2) documentation patches, (3) documentation hosted outside the official project's web site, (4) comments and questions asked on official support channels (Section 2.3.6), and (5) external support channels such as stackoverflow.com. Managing the documentation contributions represents a large fraction of the documentation effort: in our historical analysis, we found that 28% of the document revisions, excluding documents on wikis, originated from the community.

Bursts.

During a project's lifetime, the documentation occasionally goes through major concerted changes that we call bursts. These changes improve the quality of the documentation, but they require such effort that they are not done regularly.

Publishers sometimes approach contributors of open source projects to write *books* about their projects: six contributors in our study mentioned that they (or their close collaborators) wrote books. One consequence of writing books is that contributors think more about their design decisions: *"it forced me to be more precise, to think carefully about what I wrote"*$_{C3}$.[5] This particular contributor made many small changes to clarify the content of the official documentation while he was writing the book. Because books about open source projects are not always updated, their main advantage lies in the improvement of the quality of the official documentation and the time that the contributors take to reflect on their design decisions.

Contributors also *change the documentation infrastructure* when it becomes too costly to maintain. Maintenance issues either come from custom tool chains, *"it is so complex that our release manager can't build the documentation on his machine"*$_{C4}$, or from a barrier of entry that is not high enough (e.g., wiki).

The last type of burst efforts are the major reviews initiated by the documentation contributors themselves. During these reviews, contributors can end up rewriting the

[5]Identifiers are associated with quotes for traceability and to distinguish between participants. Identifiers of contributors and users begin with a "C" and "U" respectively.

whole documentation (C5) or simply restructuring its table of contents (C8). We observed that major reviews lasted from six weeks (C7) to three years (C9).

2.3.1 Wiki as Documentation Infrastructure

We begin our description of major decisions with the selection of a public wiki to host the documentation infrastructure. Wikis enable contributors to easily create a web site that allows anybody to contribute to the documentation, offers a simple editing syntax, and automatically keeps track of the changes to the documentation.

Context.

Contributors select wikis to host their documentation when the programming language of the project is not associated with any infrastructure (such as CPAN with Perl) or when the project contributors want to rely on crowdsourcing to create documentation, i.e., they hope that users will create and manage the documentation.

Public wikis also offer one of the lowest barriers to entry: the contribution is one click away. According to contributors like C7, it is a powerful strategy to build a community around the project: C7 started to contribute on his project by fixing misspelled words.

Consequences.

Although wikis initially appear to be an interesting choice for contributors, all the projects we surveyed that started on a wiki (4 out of 12: C1, C5, C7, C10) moved to an infrastructure where contributions to the documentation are more controlled. As one contributor mentioned: *"the quality of the contributions... it's been [hesitating] ok... sometimes [it] isn't factual so we had to change that... but the problem has been SPAM"*$_{C1}$. Indeed, we observed in our historical analysis that projects on wikis are often plagued by SPAM (24.1% of the revisions in Firefox) or by the addition of URLs that do not add any valuable content to the documentation (e.g., a link to a tutorial in a list already containing 20 links).

Another problem with wikis is that they lack authoritativeness, an important issue according to our users: *"I don't want to look at a wiki that might be outdated or incorrect"*$_{U3}$

For example, we observed cases such as a revision in a Firefox tutorial where one line of a code example was erroneously modified (possibly in good faith). The change was only discovered and reverted one day later (June 13th 2006).

Finally, because the barrier to entry is low, i.e., there is not much effort required to modify the documentation, the documentation can become less concise and focused over time: *"there's a user-driven desire to make sure that every single possible situation is addressed by the documentation. [these situations] were unhelpful at best and just clutter at worst"*$_{C5}$. According to C7, managing public wikis of large projects is a full-time job.

Alternatives. As users and contributors mentioned, the community is less inclined to contribute documentation than it is to contribute code, so the barrier to contribute documentation must be lower than the barrier to contribute code. Still, there exist mechanisms that encourage user contributions and that do not sacrifice authoritativeness, such as allowing user comments at the bottom of documents. Another strategy is to explicitly ask for feedback within the documents. For example, we observed in our historical analysis that Django provides a series of links to ask a question or to report an issue with the documentation on every page. Hibernate provides a similar link on the first page of the manual only. We could not find such a link in the Eclipse documentation. This strategy could explain in part the number of revisions that were motivated by the community: Django: 48%, Hibernate: 10%, and Eclipse: 10%.

2.3.2 Getting Started as Initial Documentation

Getting started documentation describes *how to use* a particular feature or a set of related features. It can range from a small code snippet (e.g., the synopsis section at the beginning of a Perl module) to a full scale tutorial (e.g., the four-part tutorial of Django).

Context.

Contributors create getting started documentation as the first type of documentation so that users can install and try the project as quickly as possible. Contributor C8 mentioned that for open source projects, getting started documentation is the

best kind of documentation to start with because once a user knows how to use the basic features, it is possible to look at the source code to learn the details of the API.

For seven contributors (C1, C2, C4, C5, C7, C8, and C10), "getting started" documentation has not only a training purpose, but it also serves as a marketing tool, it should *"hook users"*$_{C1}$, specifically when there are many projects competing in the same area. In contrast, the contributors of the five oldest projects (C3, C6, C9, C11, C12) reported that there was no marketing purpose behind the getting started documentation: these projects were the first to be released in their respective field and the contributors wrote the documentation for learning purpose only.

Contributors of libraries that offer atomic functions that do not interact with each other (C1 and C11) felt that getting started documentation was difficult to create because no reasonable code snippet could give an idea of the range of features offered by the libraries. These contributors still tried to create a document that listed the main features or the main differences with similar libraries.

Consequences.

The importance of getting started documentation was confirmed by users who mentioned examples of projects they selected because their documentation enabled them to get started faster and to get a better idea of the provided features. For example, U5 selected Django over Rails because the former had the best getting started documentation, even though the latter looked more *"powerful"*$_{U5}$. C2 confirmed that users evaluate Perl projects by looking at their synopsis.

Writing getting started documentation is challenging though: *"technical writing... I didn't have much exposure... I got used to it to some degree, but it is a challenge... it can take a lot of time"*$_{C6}$. Finding a good example on which to base the getting started documentation, an example that is realistic but not too contrived, is difficult (C11).

2.3.3 Reference Documentation as Initial Documentation

Contributors may decide to initially focus on reference documentation by systematically documenting the API, the properties, the options and the syntax used by a project.

Context.

When a library offers mostly atomic functions, reference documentation is the most appropriate documentation type to begin with because, as contributor C11 mentioned, it can be difficult to create getting started documentation that shows examples calling many functions.

In contrast to libraries with atomic functions, frameworks expecting users to extend and use the framework in some specific ways need more than reference documentation according to the users we interviewed. Frameworks, by their nature, require users to compose many parts together, but reference documentation only focuses on one part at the time: *"interactions between these classes is often very difficult to get a grasp of... you need more information how the overall structure of the framework works"*$_{U4}$.

When contributors initially create the reference documentation, they either systematically document all parts of their projects (C1, C2, C3, C11) or they rely on a more pragmatic approach (C5, C7, C8, C10): *"I try to go for anything that is not obvious"*$_{C10}$. Indeed, programming languages can be self-documenting when the types and members are clearly named. Users repeatedly confirmed that when a function has a clear name and a few well-named parameters, they will just try to call the function and will only seek the documentation if they encounter a problem. In statically typed languages such as Java, developers will use auto-complete to learn about the possible types and members and in dynamic languages such as Python, developers will execute code in an interpreter and call functions such as `help()` (displays API documentation) to learn more about a program. For weakly and statically typed languages such as C, developers cannot rely on type names (because many types are integer pointers) or on facilities provided by an interpreter and reference

documentation becomes more important. User U3 mentioned that the equivalent of an empty API documentation with only the type name, (e.g., Doxygen), could save him hours of source code exploration.

Consequences.

Comprehensive reference documentation, especially for libraries, can contribute to the success of a project. For example, contributor C1 ensured that all functions of his project were documented before releasing the first version. According to C1, even if his project launched a year after a competing project, the user base grew quickly because the competing project had no documentation. Nowadays, although there are at least four other libraries providing similar features, C1's project has the largest user base and C1 attributes this success in large part to the documentation, a claim that was confirmed by four users.

In terms of effort, reference documentation is the easiest type of documentation to create according to the contributors we interviewed. For example, when C4 works with a developer who does not have strong technical writing skills, C4 works on the getting started documentation and let his colleague works on the reference documentation.

2.3.4 Documentation Update with Every Change

One strategy to adapt the documentation to a project's evolution is to document a change quickly after implementing it or requiring external developers to include documentation and tests with the code they contribute.

Context.

Although all projects except two have a policy that all changes must be documented before a new version is released, we observed that seven contributors preferred to document their changes shortly after making them instead of waiting just before the release (C2, C3, C4, C6, C8, C9, C10). These developers considered that documentation was part of their change task and they saw many benefits to this practice (described below).

Contributors C4, C8, and C10 ask external contributors to document their code contribution. These three contributors want to ensure that the coverage of the code by

the documentation stays constant and that the contribution is well thought-out. Contributors sometimes bend this policy to encourage more code contributions. For example, C10 accepts code contributions without documentation for his smaller projects or for experimental features in larger projects.

Consequences.

Documenting changes as they are made ensures that all the *"user-accessible features are documented"*$_{C1}$, a documentation property that users often mentioned.

Another, perhaps more surprising, advantage of updating the documentation with every change is that it leads to a form of *"embarrassment-driven development [EDD]: when you have to demo something, and documentation is almost like having a demo, you'll fix it [usability issue] if it's really annoying"*$_{C10}$. Contributors reported that they modified their code while working on the documentation of their project to attempt to: (1) adopt a clearer terminology, (2) add new tools to reduce the time it takes to use the project, (3) improve the design of their project, or (4) improve the usability of the API.

We observed that EDD happens when contributors are working on getting started documentation and are describing how to accomplish common tasks: this is when contributors take the perspective of the users and must compose many parts of the technology together. EDD is possible when the development process exhibits these properties: contributors who write the documentation have code commit privileges, these contributors can modify the code without going through a lengthy approval process, and the documentation process is not totally separated from the coding process. For example, C4 mentioned that he tried to write documentation as soon as possible in the development process: *"it's common that I discover that when I'm writing [documentation] I need to change the design of the library because I discover that my design isn't explainable"*$_{C4}$.

Nevertheless, two contributors minimized the benefit of embarrassment-driven development. For example, in the project of C9, many contributors review each code change so most usability or design problems are caught during the review phase and not while writing documentation. Contributor C11 added that for libraries that

provide atomic functions, unit tests covering the common scenarios will also enable developers to detect API usability issues (e.g., is it easy to call this method in the unit test?).

One advantage of requiring code contribution to be documented is that it helps project maintainers to evaluate large contributions: *"I start with the documentation: if the documentation is good I have fairly good confidence in the implementation. It's pretty hard to have a well-documented system that is badly implemented"*$_{C4}$.

As users mentioned, one potential issue with requiring that all changes be documented is that developers might write content-free documentation: comprehensive policies established by C1 such as requiring a code example for each function help developers avoid this issue.

2.3.5 Use of a Separate Documentation Team

Three contributors, C5, C7, and C12, were part of a dedicated documentation team in their project.

Context.

We observed that external contributors formed documentation teams and officially joined a project when the original code contributors believed that documentation was important to their project, but lacked motivation (i.e., they preferred to write code) or confidence in their documentation skills.

We observed two types of documentation teams. The first type (C5 and C7) is responsible for documenting everything, from the new features to in-depth tutorials. The other type (C12) is responsible for improving the documentation such as adding examples, polishing the writing style, or completing the documentation, but code contributors are still responsible for documenting their changes.

Consequences.

Relying on a documentation team to document most changes (first type of team) had many disadvantages. First, code contributors outnumbered documentation contributors so the documentation lagged behind the implementation of new features, a situation that led to frustration, both for the users and the documentation team: *"our*

release cycle should include documentation itself, [we need] more than just API reference, [we need] prose that covers kind of usages things. We're releasing tons of code..., but there is no documentation for it and people got frustrated"$_{C5}$. A second disadvantage was that developers who implemented the change did not become aware of usability issues on their own. All contributors who were in a documentation team mentioned that they sometimes acted as testers and reported issues with new features to the developers, but developers were not always receptive to their comments: *"somebody made a decision and it became that 'name', meanwhile someone in the documentation had made [another] decision on what the name would be... It was a very big struggle in naming things: [the developer name] confuses lots of things"*$_{C7}$. In contrast, C12, who is part of the second type of team, mentioned that the development team usually let the documentation team work on the terminology.

Contributors C7 and C12 mentioned that having a documentation team lowers the barrier to entry: users with no advanced knowledge of a programming language can still contribute to the documentation and become an official contributor with commit privileges.

2.3.6 Documentation Updates based on Questions

One strategy used by contributors to leverage the community is to consider questions asked on support channels (e.g., mailing list) to be a bug report on the documentation.

Context.

The best example of this strategy came from Contributor C9 who sent us a list of emails that had been exchanged on the mailing list about an unclear section in the documentation. The exchange started with a question about the difference between two parameters: *"I see six emails... The problem is that the nuance of this particular command was really not clearly spelled out... This is a case where we really aren't*

doing our job"$_{C9}$. C9 then attempted to edit the problematic section in the documentation and submitted a patch for review to the mailing list. After a few email exchanges with other contributors, C9 further edited the section and committed his changes. Overall, the change took at most 20 minutes. Other contributors described a similar experience when managing questions.

Consequences.

Community feedback is essential to write clear documentation: *"when I write documentation, I skip things which need to be documented. But I am not aware of that. It's impossible to get around that problem unless you actually have someone else... who does not understand the details about the system"*$_{U4}$. In our historical analysis of changes, we found that more than half of the clarification changes (102 out of 195) were about explicitly stating something that was implied, such as adding an extra step to a tutorial. Community feedback helps locate these parts of the documentation that needs clarification and that could not have been foreseen by the contributors.

The main effort when continuously improving the documentation does not lie in the changes themselves but in constantly looking for occasions to improve the documentation. As C12 said, only a small percentage of the community contribute through the various channels (IRC, mailing list, bugs). A question raised by one individual on the mailing list might actually be asked by many more users. Many strategies are then used to evaluate if a question should be addressed by the documentation: was the question asked many times, is the answer provided by the community right or wrong, is the question addressed at all by the documentation, and is the question about an English-related issue and asked by a non-native English speaker?

Finally, users mentioned that they look for the presence of a live community when selecting a project: an active support channel gives some assurance to the users that their questions will be answered if they encounter any problem.

Impact on − >	Doc. creation	Doc. maintenance	Adoption	Community contributions	Learnability
Public wiki	Made the creation faster [C1, C7, C10]	Increased maintenance [C4, C5, C7, C10]	Very divergent opinions [C1, C5, C7, C10] [U3, U6, U7, U9]	Lowered barrier to entry, led to low quality [C1, C5, C7, C10]	Led to "corner cases clutter" [C1, C5, C7]
Getting started	Required strong writing skills [C1, C2, C4, C5, C6, C8, C11]		Was used as marketing tool [C1, C2, C6, C8, C10] [U1, U5, U8]		Improved for framework [C4, C5, C8, C10] [U1, U2, U4, U5, U6, U8]
Reference doc.	Was the easiest type to create [C1, C4, C6, C8]	Was more costly to maintain [C1] [U3]	Was mostly important for libraries & Competitive advantage [C1] [U3, U7, U8, U9]		Improved for libraries [C11] [U3, U4, U5, U6, U7, U8, U9, U10]
Doc. update with changes	Required smaller upfront effort [C1, C9]	Led to small but numerous changes more adapted to open source development process [C1, C3, C9, C11, C12]	Led to better coverage, a selection criteria [C4] [U1, U3, U4, U8, U10]	Increased the barrier to entry but improved the quality of the contributions [C4, C8, C10]	EDD led to API usability improvement [C2, C4, C8, C10]
Doc. team	Documentation lagged behind released features[7] [C5, C7]	Documentation effort shifted from dev team to doc. team [C5, C7, C12]		Lowered barrier to entry [C7, C12]	Improved clarity and conciseness of documentation [C5, C7, C12]
Updates based on questions	Lowered upfront effort [C8, C9]	One of the main sources of maintenance [C1, C2, C3, C6, C8, C9, C12]	Was a sign of community activity, a selection criteria [U2, U6]	Questions became a contribution [C1, C2, C3, C4, C6, C9, C12]	Led to many clarifications [C1, C2, C3, C4, C6, C9, C12]

Table 2.4: Decisions and their consequences

2.3.7 Summary

The decisions made by contributors have been presented as part of the documentation production modes and the decision points, which is useful to understand the context of these decisions. Yet, the consequences of the decisions for the contributors and the users are orthogonal to the documentation process. Table 2.4 shows the main consequence of the six decisions presented in Section 2.3 for five aspects of open source projects: the documentation creation effort, the documentation maintenance effort, the project adoption, the number and quality of community contributions, and the learnability of the technology. For each consequence, we indicate the contributors (C) and users (U) who discussed it. Because our questions and selection criteria evolved as the study progressed, these numbers reflect the variety of observations we gathered for each consequence: a quantitative study with a larger sample would form a natural step to evaluate the frequencies of these consequences.

2.4 Quality and Credibility

We evaluated the *quality* (are the findings innovative, thoughtful, useful?) and the *credibility* (are the findings trustworthy and do they reflect the participants', researchers', and readers' experiences with a phenomenon?) of our study by relying on three criteria proposed by Corbin and Strauss [15, p. 305]: fit, applicability, and sensitivity. These evaluation criteria are more relevant for a qualitative study than the usual threats to validity associated with quantitative studies [17, p.202]. The goal of our study was not to generalize a phenomenon observed in a sample to a population: instead we are generating a theory about a complex phenomenon from a set of observations obtained through theoretical sampling.

We produced a four-page summary presenting a subset of the decisions we analyzed and we invited the 12 contributors to review this summary to ensure that our

[7] Only for documentation teams that are responsible for documenting changes

findings resonated with their experience. Six contributors accepted our invitation and responded to a short questionnaire reproduced in Appendix E.

Fit. "Do the findings fit/resonate with the professionals for whom the research was intended and the participants?" The contributors found that the major decisions they made were represented in our summary. They mentioned though that many smaller decisions or factors were missing. For example, C7 remarked that the fact that documentation teams had to always catch up with the development team was not represented well in the summary. We had analyzed most of these details, but for the sake of brevity, we did not include them in the summary. There are only a few decisions that we did not analyze because we thought that they were less relevant to documentation (e.g., how to support users). These comments motivated our choice of presenting only a few important decisions and providing more detailed findings.

Applicability or Usefulness. "Do the findings offer new insights? Can they be used to develop policy or change practice?" To the best of our knowledge, this is the first study on the process taken by contributors to create and maintain developer documentation. We hope that our description of the documentation decisions will help researchers devise documentation techniques that better support documentation decisions. For example, recognizing that the programming language plays an important role in the documentation decisions could lead to the development of solutions for languages that have a less standardized documentation culture.

We believe that this study has many benefits for practitioners. Contributor C11 mentioned that our summary could help other contributors reflect more on their documentation approach. Contributors and users mentioned that there is a general lack of motivation when it comes to contributing documentation. We hope that by uncovering the context and the consequences of documentation decisions, such as how documenting can improve the quality of the code, and how certain types of document contribute to the success of projects, could increase the motivation of contributors and users.

Sensitivity. "Were the questions driving the data collection arrived at through analysis, or were concepts and questions generated before the data were collected?" We

did not enter this study with a blank slate because we have worked on documentation studies and tools in the past. To address this issue, Creswell recommends disclosure of any stance on the issue that researchers had before beginning the study [17, p.217]. For instance, we thought that writing documentation took a large amount of time and effort and we did not think that the community could play such a significant role in the documentation process. We were surprised at first to see the contributors struggle to name a single challenge to documentation. We soon realized how documentation could be seen as a vital and interesting part of open source projects and how the community could help improve the documentation. These early observations forced us to recognize and reconsider our preconceptions and helped us look at the data from a fresher perspective.

2.5 Summary

Developers rely on documentation to learn how to use frameworks and libraries and to help them select the open source technologies that can fulfill their requirements. Following a qualitative study with 22 documentation contributors and users and the analysis of the evolution of 19 documents, we observed the decisions made by open source contributors in the context of three production modes: initial effort, incremental changes, and bursts.

Understanding how these decisions are made and what their consequences are can help researchers devise documentation techniques that are more suited to the documentation process of open source projects and that alleviate the issues we identified. Our findings can also help practitioners make more informed decisions. For example, a better understanding of embarrassment-driven development could motivate developers to document their changes quickly after making them. A better comprehension of the relationship between the type of project (e.g., library or framework) and getting started and reference documentation could help contributors focus their effort on the more appropriate type of documentation.

Chapter 3
Recovering Traceability Links between an API and its Learning Resources

As we observed in Chapter 2, writing, moderating, and maintaining learning resources for frameworks and libraries require a considerable effort. For example, the Hibernate framework forum has more than 180 000 messages and even a smaller libraries such as HttpComponents, which contains 619 classes, has a developer documentation of 28 900 words divided in two manuals, and a mailing list that includes more than 8 500 messages.

Ideally, developer learning resources should be both extensive (covering all parts of the framework's API) and detailed (explaining many low-level programming patterns), while being continually maintained to keep up with feature additions, API usability problems, and community requests. For instance, we observed in Chapter 2 that when a question is repeatedly asked on a mailing list, framework contributors see this as an indication that the documentation needs to be clarified [23]. In cases where there are multiple support channels (chat, mailing lists, forums) and multiple contributors operating in different time zones, the contributors are often unaware that the same code element (e.g., function) is the root cause of several questions. Specifically, because the support channels and the documentation are not explicitly linked to the API, it is difficult for a contributor to determine which code elements cause the most problems and need to be further explained in the documentation.

The main challenge in linking code elements with existing learning resources comes from the inherent ambiguity of unstructured natural language. For example, the user guide of the Joda Time library [3] mentions in the middle of the *Date fields* section: "... such as year() or monthOfYear()". Although it is clear from this sentence fragment that a method named `year` is mentioned, there are 11 classes, not all in the same type hierarchy, that declare a `year` method in Joda Time. The code-like term `year` could also refer to a method declared in an external library frequently used with Joda Time (e.g., Java Standard Library). In this particular case, a human reader would know that the term refers to `DateTime.year()` because the class `DateTime` is mentioned at the beginning of the section, i.e., in the *context* of the method `year()`. However, a simple mechanical match based on the method name and ignoring the context of the term would fail. In fact, in the four open source projects we studied (Section 3.3), we found that a mechanical match **would have failed** to find the correct declaration of 89% of the methods mentioned in the learning resources because these methods were declared in multiple types.

Several techniques have been previously proposed to link project artifacts. However, there is currently no technique that precisely links the documentation, the support channels and the API together at a fine level of granularity. For example, Hipikat links coarse-grained project artifacts such as code commits, emails, and bug reports based on bug numbers [18]. Bacchelli et al. devised a technique that identifies source code (e.g., code snippets) in emails and that can link classes mentioned in the email to classes declared in a codebase [8], but the technique does not work at the sub-class level of granularity.

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

We implemented our technique in a tool called RecoDoc and applied it on four open source systems. We found that our technique identified on average 96% of the code-like terms (recall) and linked these terms to the correct code element 96% of the time (precision). As we show in Chapter 5, the high accuracy of our technique enables the development of reliable approaches that can improve the learning resources based on the relationships between these resources and the API.

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

We begin by presenting a meta-model to represent the various project artifacts (Section 3.1). Then, we describe the linking technique we devised to associate the code-like terms from the learning resources to the code elements of an API (Section 3.2). We present the evaluation we performed on four open source projects in Section 3.3 and we summarize the contributions of RecoDoc in Section 3.4.

3.1 Project Artifacts Meta-Model

A variety of information is needed to understand the context in which a code-like term is mentioned. In the documentation example of Figure 3.1, the method `getParams` is declared in eight types in the HttpClient library. We can precisely find which method declaration is referred to if we know that:

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

[1] http://www.cs.mcgill.ca/~swevo/recodoc

Section 1

HttpGet implements the HTTP GET request in HttpClient.

Section 1.1

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

Figure 3.1: Documentation Example Loosely Adapted from the HttpClient tutorial.

Based on our previous study on developer documentation [23] and on initial prototyping with various releases of the Spring Framework (a large and complex Java project), we designed a meta-model to universally represent the documentation, support channels, and API of any open source project. We use this meta-model to understand the context in which a code-like term is mentioned. We can instantiate this meta-model for any open source project of interest. The main elements of the meta-model are described in the next paragraphs and are represented in Figure 3.2.

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

Codebase. We consider that the *API* of a project consists of all the accessible *code elements* of a project (e.g., public class, method, field, parameter, XML element, but not private method or field). RecoDoc currently parses Java codebases, XML schema files and DTD files. A code element may have one or more parents (e.g., a Java class implements multiple interfaces) and may declare other elements (e.g., a class declares methods). Additionally, each *kind* of code elements is internally represented by a specialized class that keeps track of its specific attributes (e.g., a `MethodCodeElement` has a list of parameters, not shown in Figure 3.2).

Document. The documentation of a project consists of one or more *documents*. For example, the HttpComponents project has two main documents: the HTTPClient and HTTPCore tutorials. Each document has a list of *pages* and each page has

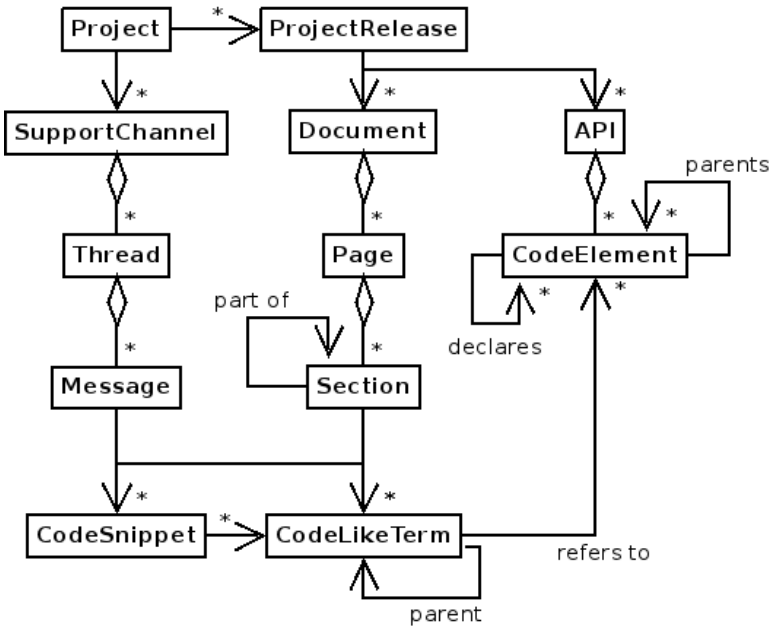

Figure 3.2: Documentation Meta-Model. The cardinality of an association is one unless otherwise specified.

a list of *sections* (e.g., Section 1.1.2. HTTP request). A section may be part of a larger section (e.g., Section 1.1.). As we explain in Section 3.1.1, we consider a documentation page to be equivalent to an HTML page and not to a printed page.

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

Code-like Terms and Code Snippets. Messages and documentation sections can refer to *code-like term* and *code snippets*. A code-like term is a series of characters that matches a pattern associated with a code element kind (e.g., parentheses for functions, camel cases for types, anchors for XML elements). For example, Section 1.1 in Figure 3.1 contains three code-like terms: `getParams`, `RedirectStrategy`, and `getRedirect`. A code-like term list, or term list, is a sequence of code-like terms. We thus

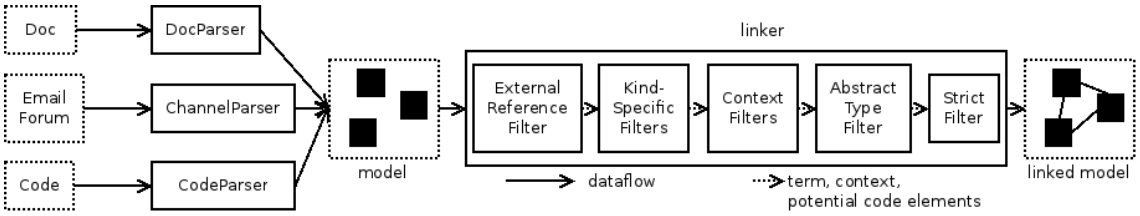

Figure 3.3: Parsing Artifacts and Recovering Traceability Links

consider that the term list `RedirectStrategy.getRedirect` contains two code-like terms and that the first term is the parent of the second.

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

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

Context. We consider that there are three levels of context that can be associated with code-like terms. The *immediate context* contains all the code-like terms in a term list. The *local context* contains all the terms in the same documentation section or the same support message. The *global context* contains all the terms in the same documentation page or in the same support thread. For example, the immediate context, the local context, and the global context for `getRedirect` in Figure 3.1 are respectively: {`RedirectStrategy`}, {`RedirectStrategy, getParams`}, {`RedirectStrategy, getParams, HttpGet`}. We consider that a code-like term `A` is *closer* to a term `B` than to a term `C` if `B` is in a more specific context than `C`. For example, `getRedirect` is closer to `getParams` than to `HttpGet`.

3.1.1 Generating Models

We generate a documentation model from a set of artifacts and recover the links between code-like terms and code elements. Figure 3.3 demonstrates this process.

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

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

Content Classification. Once the model is generated, the parsing infrastructure classifies the content of the documentation and the support channel: it identifies the code-like terms, the code snippets, and their probable kind (e.g., class, method, XML element, Java code snippet, XML code snippet). We relied on existing techniques described in the literature [8] and in our previous work [20] to implement the content classification step. A brief description of the classification process and the evaluation of its accuracy is presented in Appendix B.

Snippet Parsing. We further parse snippets to identify the code-like terms within them. For example, we identify all calls, declarations, and references in Java snippets. We use Partial Program Analysis (PPA) to parse Java snippets [20]. PPA accepts partial Java programs (e.g., method bodies) and produces type-resolved Abstract Syntax Trees by inferring the missing declarations.

Linking. Finally, we attempt to link the code-like terms to code elements in a specific project release. The linking process is described in Section 3.2.

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

3.2 Linking Technique

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

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

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

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

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

Language Ambiguity. We expect learning resources to contain errors made by users and documentation writers. These errors include (1) typographical errors such as `HtttpClient`, (2) case errors such as `basiclineparser`, (3) hierarchy errors (e.g., `Collection.add()`, does not exist and potentially refers to `List.add()`), and (4) parameter errors such as forgetting a parameter in a call.

Given these sources of ambiguity, we make two assumptions that guide our link recovery strategy:

1. Two code-like terms mentioned in close vicinity are more likely to be related than terms mentioned further apart.
2. Members like methods and fields are unlikely to be mentioned without their declaring type also being mentioned in their context (as described in Section 3.1).

3.2.1 Link Recovery Process

Our link recovery process takes as input a collection of code-like terms. Each code-like term is associated with a kind (e.g., method, field, class, annotation) and the other terms present in its context (immediate, local, and global, see Section 3.1). The output of the link recovery operation is a ranked list of code elements that are potentially referred to by each code-like term. Given a collection of code-like terms, we perform the following steps:

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

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

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

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

Linking Members. Given a code-like term referring to a member (method or field), we find all code elements of the same kind that share the name of the term. For example, for the term `add()`, we find all methods named `add` in the API. Then, the potential code elements go through a pipeline of filters that eliminate some elements or re-order the list of potential elements. These filters rely on the types identified in the previous steps and we describe them in Section 3.2.2.

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

In this step, we take as input all terms that were not linked to any code elements in the previous steps. Then, we search for code elements of any kind that have the same name as the term. We group the potential code elements by their kind and we attempt to link them in the order they were processed in steps one through three: types, methods, fields. The linking technique used is the same as in the previous steps (e.g., simple name matching for types, name matching and filtering heuristics for members).

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

the linking errors could have been prevented by repeating steps one through four, but a more accurate parser and better filtering heuristics would have improved the results (Section 3.3).

3.2.2 Filtering Heuristics

Because of the four sources of ambiguity mentioned in Section 3.2, it often happens that a code-like term may be linked to many potential code elements. For example, in the evaluation of our technique, we found that on average, each term classified as a method could be linked to 16.8 methods declared in 13.5 different types.

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

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

We describe each category of filters in the order they are executed in the pipeline. Each category is represented in Figure 3.3. We indicate the type of ambiguity these filters address at the end of each filter.

External Reference Filter

This filter identifies code-like terms that are likely referring to elements outside the system's codebase or concepts with names similar to code elements. This filter considers that all types of the Java Standard Library are external references. Then, the filter tries to match the terms to a list of words that is specific to the system being analyzed. For example, the term `HttpClient` is both the name of a type and of a system. In the HttpClient mailing list, this term is almost exclusively used to refer to the system, unless the term appears in a code snippet. When this filter identifies an

external reference, it eliminates all the potential code elements and the subsequent filters are never activated. Although the list of system-specific words need to be provided by the user, the number of words per system is generally low (e.g., 5 words for HttpClient) and it does not significantly impact the accuracy of our approach (see Table 3.5 in Section 3.3.2) (External Reference Ambiguity)

Kind-Specific Filters

These filters are activated only for certain kinds of terms. We implemented two such filters and they are activated for terms representing a method.

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

Parameter Types. If the code-like term includes the types of the parameters, this filter eliminates the potential code elements whose parameter types do not match. When the parameters are given as arguments instead of types (e.g., `put(obj,obj)` vs. `put(Object,Object)`), we match the parameter types based on the name similarity: if the name of a parameter in the term matches 80% of the name of the parameter type and if more than half of the parameters match, we consider that the term matches the code element. The name similarity of two parameters is obtained by computing the number of common pairs of characters divided by the number of possible pairs. This is a metric that has been found to be robust for assessing the similarity of code-related strings [60, p.4]. We determined the thresholds during initial prototyping of the approach. (Overload Ambiguity).

Context Filters

These filters look at the context for the code-like term to determine which potential code element is most likely being referred to. All of these filters try to find the declaring type of a term in the term's context.

Context. Types that declare a member are often mentioned in the vicinity of the member. Given a term classified as a member (e.g., method, field), this filter tries to

find in the immediate context a type declaring the member. If it fails, it tries to find such type in the local context. Finally, it tries to find a type declaring the member in the global context. When the filter finds one or more type that declares the member, the filter eliminates all potential code elements that are not declared by these types. (Declaration Ambiguity)

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

As an example of this filter, consider the section "Using a MutableDateTime" of the JodaTime user guide [3]. This section contains a snippet with the following code-like term: `toMutableDateTime()`. There are three potential code elements whose name match the term: `{Instant.toMutableDateTime, ReadableDateTime.toMutableDateTime, AbstractInstant.toMutableDateTime}`. The filter thus looks in the context for the term and finds that the local context contains the following types: `{MutableDateTime, DateTime}`. None of these types declares the `toMutableDateTime` method, but one of their ancestors, `ReadableDateTime` does, so the filter eliminates all potential code elements except `ReadableDateTime.toMutableDateTime`

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

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

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

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

Abstract Type Filter

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

Strict Filter

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

If there are more than one potential code element, we select the first element from the ranked list if at least one of the context filter was activated. The context filters may not be activated if the declaring type of a code element was not found. This condition is based on our second assumption that a member is rarely mentioned without its declaring type. The context filters are thus highly important in our

filtering pipeline: they eliminate potential code elements based on the context for a code-like term, and they also determine whether a term will be linked at all.

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

3.3 Evaluation

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

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

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

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

3.3.1 Study Design

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

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

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

HttpComponents is a Java library that simplifies the communication with a web server. HttpComponents is split in two main components, HttpCore and HttpClient. It has more than 85 KLOC. Its documentation is written in DocBook and it is split in two documents. The document we studied, HttpClient Tutorial, has 15 275 words. The main support channel for HttpComponents is a mailing list hosted on Apache. This library does not need any configuration file, but the documentation and the support channel often mention various protocols (e.g., HTTP). The library is used by calling methods, instantiating classes and, in some advanced scenarios, by implementing interfaces.

Hibernate is an Object-Relational Mapping framework (ORM) written in Java: it enables clients to persist objects to a relational database. It has more than 905

KLOC. Its documentation is split in three main documents (it was merged into two documents when we finished the study) and the document we analyzed, the main reference manual, has 70 900 words. The support channel is a forum. This framework usually requires a configuration file written in XML or a property file. Hibernate is used by calling methods, instantiating classes, making queries written in a custom language (HQL), and optionally using Java annotations. The documentation and the support channel mention the Java API, the SQL language, the HQL language and the configuration files.

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

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

Unit of Analysis. We randomly selected 100 support messages and 100 documentation sections for each target system: we will refer to these as *units*. We inspected each unit and we manually identified the code snippets and the code-like terms that might refer to a code element in the system's codebase. We tried to link each code-like term to the most specific code element in the API, unless the code-like term was referring to a set of code elements (e.g., all implementations of the `save()` method),

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

Table 3.1: Target Systems Version

in which case, we selected the most general code element such as a method declared in an interface. We read the entire page or support thread of each unit to select the code element that was the most likely being referred to.

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

Following our manual inspection, we launched RecoDoc, which analyzed all support messages and documentation sections of the four projects. It is necessary to at least analyze all the sections and support messages in the same page or support thread as the units in our random sample because RecoDoc may need to analyze the global context of a code-like term (see Section 3.2.2).

Table 3.2 shows the characteristics of the units for each target system: the average, the minimum, and the maximum number of words, per selected unit (sample), the average number of words and standard deviation for all units (population), the average number of code-like terms for the sample and the population, and the average number of code-like terms that RecoDoc linked to a Java code element for the sample and the population. The length (in words) of the units in our random sample was always within 0.2 of the standard deviation of the population. The wide range of units RecoDoc analyzed provide an evidence that our approach can be used in practice, for small or large units.

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

Table 3.2: Units of Analysis: Random Sample (S) and Population (P) Characteristics

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

Table 3.3: Results of Link Recovery Evaluation

3.3.2 Results

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

positive), (5) RecoDoc linked a term that we linked to another term (false negative and false positive).

Table 3.3 shows the results of our evaluation. The second column, *Insp.*, gives the number of code-like terms that we linked to a code element during our inspection. The third column, *Recodoc*, gives the number of terms that RecoDoc correctly linked to a code element (exact matches). The number in parentheses adds the similar matches to the number of exact matches. The fourth column, *Prec.* gives the precision of RecoDoc (exact matches divided by number of links found by RecoDoc). Finally, the fifth column, *Recall*, gives the recall of RecoDoc (exact matches divided by number of links found by our inspection).

RecoDoc Accuracy. The results from Table 3.3 clearly indicate that our technique can correctly link code-like terms to code elements with a high precision and a high recall (all over 90%, except the Hibernate documentation). RecoDoc practically always linked code-like terms in snippets to a correct code element because Partial Program Analysis recovered most of the necessary type information (type of parameters, type of declaring type, etc.).

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

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

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

Table 3.4: Context Filters Activation Profile

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

Filtering Heuristics. Of the 300 228 code-like terms that RecoDoc linked in all documentation sections and support messages (not just the sample), 160 970 were type members such as a method. On average, each of these code-like terms could potentially be linked to 16.8 members declared in 13.5 different types. This is evidence that linking members is technically challenging. After going through all the filtering heuristics introduced in Section 3.2.2, each code-like term could potentially be linked to only 0.7 member on average. This is evidence that our filtering heuristics are effective at reducing the number of potential matches.

As we mentioned in Section 3.2.2, the context filters are the most important filters of our pipeline because (1) they eliminate potential code elements based on the context for a term, and (2) at least one contextual filter must be activated to link a term. Only one context filter can be activated for each term. Table 3.4 shows how often each context filter was activated in the target systems (the numbers are only for the 160 970 type members). For example, in the Joda Time documentation, RecoDoc found the declaring class in the immediate context of 40.1% of the code-like terms representing methods or fields.

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

Table 3.5: Causes of Code-Like Terms not Being Linked.

The sum of the usage profile of the context filters does not reach 100% because our technique can link a code-like term to a member without using any context filter. For example, in the Joda Time Channel, RecoDoc found the declaring class of 95.4% of the code-like terms in their context. RecoDoc did not find the declaring class of the other 4.6% code-like terms, but because these code-like terms could refer to only one code element in the codebase, RecoDoc linked them nonetheless (see Strict Filtering in Section 3.2.2).

Our technique found the declaring class of 86.1% (51.7 + 1.5 + 25.8 + 5.9 + 1.2) of the terms in their immediate or local context. This indicates that most documentation sections and support messages are self-contained and can be understood by readers without scanning the entire documentation page or support thread. The immediate and local context filters were not sufficient to reach a high linking accuracy and RecoDoc linked 5.9% of the terms by using the types mentioned in the term's global context.

Unlinked Code-Like Terms. RecoDoc chose to not link 971 577 code-like terms that looked like a method or a field, but that did not refer to any code element. Table 3.5 shows the reasons why RecoDoc did not link these code-like terms: (1) we did not find a code element whose name matched the code-like term, (2) an external reference filter was activated and eliminated all potential code elements of a term, or (3) the strict filter was activated. For example, for the Joda Time documentation, 89.1% of the 386 code-like terms that RecoDoc correctly did not link did not match any code

element in the Joda Time codebase. 4.7% of these unlinked terms were eliminated by the external reference filters, and 6.2% were eliminated by the strict filtering pass.

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

The external reference filters were particularly useful in the Hibernate documentation because the API declares many methods whose name are the same as the methods in the Java standard library. For example, in section 20.5.4 of the Hibernate documentation, the term `list.clear()` refers to the interface method `java.util.List.clear()` and the Standard Library Classes filter correctly eliminated this term even though Hibernate declared methods with a similar name such as `PersistentList.clear()`. Most of the external reference filters were automatically generated (i.e., they came from the list of types declared in the Java Standard Library), but we also provided a manually selected list of terms (between 5 and 20 terms per project). The impact of these external reference filters was minimal because they were used to reject 7.9% of terms.

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

3.3.3 Threats to Validity

The accuracy of the results is subject to the investigators' assessment of each benchmark code-like term. In some cases, the exact target of a code-like term in the learning resources was not perfectly clear. It is thus possible that we erroneously

linked some benchmark terms during our manual inspection. However, every time our manual inspection and RecoDoc results diverged on these unclear terms, we conservatively assumed that RecoDoc was wrong. Hence, the accuracy of the reported results should represent a lower-bound of the accuracy of RecoDoc. In addition, our detailed classification is publicly available for inspection.[2]

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

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

We executed RecoDoc on the four target systems with the same parameters. Only their parser extension and a small subset of the external reference filter differed. As we showed in the previous section, the external reference filters eliminated only 7.9% of the code-like terms so in any case, external reference filters did not significantly impact the results. Moreover, it would be possible to reduce the usage of these custom filters by considering only the API that is targeted towards regular users. For example, in Hibernate, most classes and methods are never used by users, but RecoDoc still tried to link code-like terms to these elements that were named after technical concepts (e.g., Select). We chose to consider the entire codebase because there was not an objective way to determine whether a type was part of the public or internal API, but a Hibernate developer could probably do this easily.

[2] http://www.cs.mcgill.ca/~swevo/recodoc

3.4 Summary

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

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

Chapter 4
Inferring High-Level Documentation Structures

So far, we showed that we can precisely recover links between fine-grained code elements and code-like terms in developer learning resources. To reason about and improve these learning resources, we wish to to infer more abstract relationships from the fine-grained links we recovered.

In our study of the Spring Framework documentation, we noticed that individual sections often mentioned a set of code elements that were structurally related. For example, Section 13.3 in the Spring Framework 3.1 documentation covers all the subclasses of the `DataSource` interface. This *documentation pattern* can be helpful for both documentation maintainers and developers using the Spring Framework. If a new subclass of DataSource is added, we could automatically warn the documentation maintainers that this new subclass needs to be mentioned in Section 13.3. If a developer is trying to create a subclass of `DataSource`, we could automatically point the developer to Section 13.3.

Similarly, knowing that a support message is related to a documentation section may be useful for developers and documentation maintainers. We could automatically inform the developers that a section may answer the question asked in their message and we could automatically find the sections that need to be clarified based on the number of messages related to these sections.

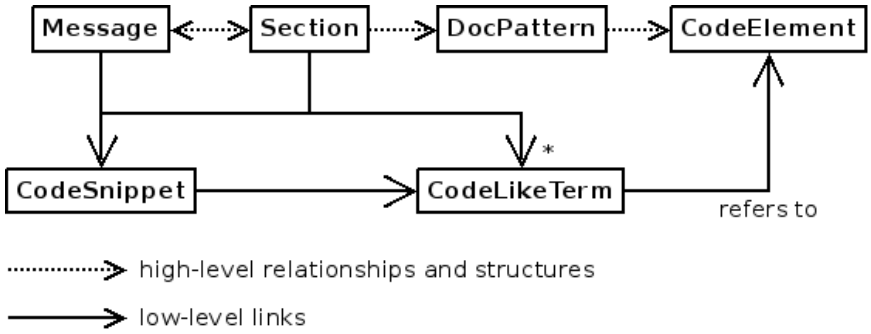

Figure 4.1: Low-level links and High-level Documentation Structures

We investigated how to recover two high-level documentation structures in order to improve the learning resources of frameworks. As we show in Chapter 5, we used the first relationship, documentation patterns, to build an efficient documentation improvement recommender. Figure 4.1 shows our documentation meta-model with the low-level traceability links and high-level documentation structures.

As for the second relationship, we were able to precisely link support messages and documentation sections discussing the same topic, but these links turned out to be unhelpful for improving the documentation. We identified promising strategies for future work to improve the links' usefulness.

4.1 Documentation Patterns

The documentation of a framework sometimes systematically *covers* the code elements of a *documentation pattern*, i.e., a coherent set of code elements that are mentioned in the documentation of a framework. We consider that a code element is covered when it is explicitly named in a sentence or in a code snippet of the documentation.

We can think of a documentation pattern as a concern graph [52], which is a representation of program structures as a redundant extension (discrete set of code elements) and intension (set of relations between the elements). Concern graphs provide a representation of a concern (e.g., a feature, a non-functional requirement) that

is robust to the evolution of the underlying codebase because the relations captured by a concern intension can be used to compute a new extension when the code changes.

For example, Section 13.3 in the Spring Framework 3.1 documentation formally defines all the subclasses of the `DataSource` interface. The intension of this documentation pattern is thus "all concrete subclasses of DataSource", and the extension would be {`SmartDataSource, SingleConnectionDataSource, ...`}.

Documentation Patterns Intensions. To be able to capture documentation patterns in different frameworks and documents, the intensions must be general enough. We observed three such general kinds of intensions in the Spring Framework documentation. Some sections and pages of the documentation cover (1) code elements *declared* in another code element such as the classes declared in a package or the methods declared in a class, (2) code elements in the same hierarchy such all the classes extending another class, and (3) code elements with similar names such as all the code elements starting, ending, or containing a similar token.

4.1.1 Inferring Documentation Patterns

Given a codebase and a documentation release, we perform six steps to find documentation patterns:

1. Compute *code patterns*.
2. Compute *code pattern coverage*.
3. Filter spurious code patterns.
4. Combine redundant patterns into documentation patterns.
5. Select most representative pattern.
6. Link documentation patterns to documentation sections and pages.

Computing Code Patterns.

Given a codebase, we compute the set of structurally related code elements, i.e., code patterns, by using a combination of the three kinds of intensions mentioned in Section 4.1. For each code element `c`, we compute:

1. The set of code elements declared by `c`.
2. The set of concrete code elements declared by `c`.
3. The set of code elements whose immediate parent is `c` (i.e., elements extending `c`).
4. The set of concrete code elements whose immediate parent is `c`.
5. The set of code elements whose ancestor is `c`.
6. The set of concrete code elements whose ancestor is `c`.
7. The sets of code elements starting, ending, or containing a token in `c`'s name.
8. The sets of code elements declared by `c` that start, end, or contain the same token.

We consider that only packages and classes declare other code elements. A package can declare multiple classes (e.g., package `java.util` declares the class `java.util.ArrayList`) and a class can declare methods and fields (e.g., the class `java.util.ArrayList` declares the method `add(Object)`). A class Y is a parent of a class X if X inherits from Y. A class Z is an ancestor of a class X if Y is a parent of X and Z is a parent or an ancestor of Y.

A concrete code element is a class that is not abstract (interface and annotations are considered abstract). Methods in interfaces and abstract methods are abstract. This "concrete" subcategory is important because we observed that intermediate abstract classes are sometimes completely ignored by the documentation and that only concrete classes are mentioned.

When we compute the sets of code elements sharing a token (intensions 7 and 8), we group the elements by their kind to avoid mixing elements of different granularity. For example, if two classes and two methods end with the same token, we compute two code patterns: one for the classes, and one for the methods.

Computing the code patterns is straightforward because all the necessary relationships are already encoded in our model (see Figure 3.2).

Although there are many code patterns, we expect that only a few of these patterns will be actually documented and that some of these patterns will overlap greatly.

Figure 4.2 shows an example of a codebase with four classes. If we assume that `c` is the abstract class `AbstractBean`, then the following patterns are computed with respect to `c`.

1. Code elements declared by AbstractBean: {*getProperty, setProperty, readProperty, getFullName*}.
2. Concrete code elements declared by AbstractBean: {*readProperty, getFullName*}.
3. Children of AbstractBean: {*DefaultBean, DefaultAbstractBean*}.
4. Concrete children of AbstractBean: {*DefaultBean*}.
5. Descendants of AbstractBean: {*DefaultBean, DefaultAbstractBean, SpecialBean*}.
6. Concrete descendants of AbstractBean: {*DefaultBean, SpecialBean*}.
7. Code elements ending with Bean: {*DefaultBean, DefaultAbstractBean, SpecialBean, TestBean*}.
8. Code elements declared by AbstractBean ending with Property: {*getProperty, setProperty, readProperty*}.

Code patterns that contain only one element are no longer considered by our algorithm. For example, the pattern "Concrete children of AbstractBean" is filtered out.

Computing Code Pattern Coverage.

Once we have determined the set of code patterns, we compute the coverage of each pattern in a documentation release. Given the links between code-like terms and code elements recovered by RecoDoc, we can compute how many elements in a pattern have been mentioned in a documentation release. At this point, we are not concerned by the localisation of the coverage: the elements of a pattern may be covered in different sections.

For each pattern, the output of this step is a number in the unit interval. This number indicates the proportion of code elements in a pattern that are mentioned in

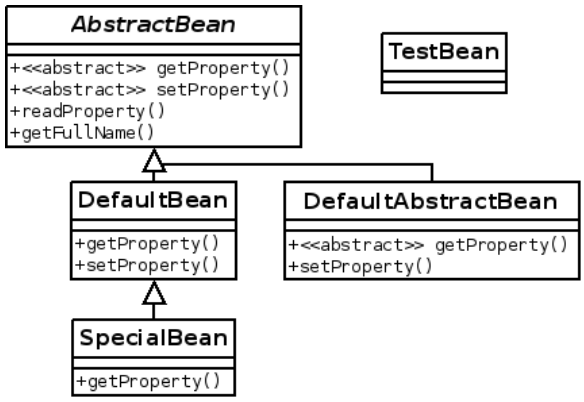

Figure 4.2: Example of Code Elements

the documentation. This step also outputs the sections and pages mentioning each code element in the pattern.

Filtering Patterns with Low Coverage.

We expect that the documentation will only refer to a small subset of all the potential patterns. We eliminate any pattern whose coverage is below 50% because the intension of these patterns clearly does not match the intent of the documentation.

Combining Patterns.

After we have computed and filtered code patterns based on their coverage, we combine the redundant ones. We consider that two code patterns are redundant if one is a subset of the other and the relative difference in the size of their extension is within a certain percentage threshold. For example, in the Spring Framework, the code pattern "All classes extending `ApplicationContext`" and the pattern "All classes ending with the token `Context`" describe the same code elements.

Algorithm 1 presents the main steps performed to combine patterns. We determined during early experimentation with the approach that a difference of 40% in the size of the extension of two code patterns enabled the combination of code patterns that have a similar number of identical code elements while preventing very general and very specific code patterns from being combined. For example, consider the two following patterns: (1) "All classes extending `ApplicationContext`" and (2) "All classes extending `ApplicationContext` and starting with the prefix `Bean`". Although the second

pattern is a subset of the first one, the second pattern is a lot more specific than the first pattern (smaller extension), so they describe different concepts and they should not be combined.

As we show in Algorithm 1, we start by sorting the patterns by the size of their extension in decreasing order. The initial sorting ensures that the groups of patterns are deterministic.

ALGORITHM 1: Combining Redundant Patterns

```
Input: List of code_patterns
Output: List of doc_patterns
patterns = sort(code_patterns, criteria=pattern size, reverse=true);
doc_patterns = {};
processed = {};
for i in [0 .. patterns.size] do
    pattern = patterns[i];
    if pattern in processed then
        continue;
    end
    add pattern to processed;
    combined_patterns = {pattern}
    for j in [i + 1 .. patterns.size] do
        tpattern = patterns[j];
        if 1.0 - (tpattern.size / pattern.size) > THRESHOLD then
            break;
        end
        if tpattern ⊆ pattern then
            add tpattern to combined_patterns;
            add tpattern to processed;
        end
    end
    doc_pattern = select most representative pattern in combined_patterns;
    add doc_pattern to doc_patterns;
end
return doc_patterns;
```

Selecting Most Representative Patterns.

Once we have combined redundant code patterns, we select the *most representative* pattern within each group of redundant patterns, which becomes a documentation pattern:

1. Among the redundant code patterns, we select the pattern with the highest coverage.
2. If more than one pattern has the highest coverage, we select from these patterns the one with the highest number of code elements in its extension.
3. Finally, if more than one pattern has the highest coverage and the highest number of code elements in its extension, we select from these patterns the one with the most general intension, i.e., with the intension declared first in the list of intensions presented in the section "Computing Code Patterns".

In summary, we select in order of importance the pattern (1) with an intension whose extension is well covered in the documentation, (2) that represents more code elements, and (3) that is the most general.

Linking Documentation Patterns to Sections and Pages.

We link each documentation pattern to the most fined-grained documentation unit that covered the code elements in the pattern. In this step, we determine whether the elements of a pattern were mainly covered in a single section, in many sections of the same page, or in the sections of many pages.

Algorithm 2 shows the main steps required to link a documentation pattern to a specific location in a documentation release. As shown in the algorithm, we consider that a documentation unit (section or page) mainly covered a documentation pattern if the coverage of the documentation unit is more than 75% (COVERAGE_THRESHOLD) of the coverage of the pattern. For example, if the eight code elements of a documentation pattern were covered in all sections of the documentation, but a section `x` covered seven of these elements, we would consider that section `x` mainly covered the pattern because its coverage, 87.5% (7/8), is superior to the threshold of 75%. The relatively high threshold (75%) enables the selection of documentation units that cover a large proportion of the documentation pattern while allowing these units to ignore uninteresting or redundant code elements in the pattern.

As the algorithm shows, a pattern may also be mainly covered in multiple locations. For example, if two sections in different pages each present most of the elements

of the documentation pattern, our algorithm will link the pattern with these two distinct sections.

ALGORITHM 2: Linking Patterns to Sections and Pages

Input: *doc_pattern*, *map_of_elements_per_section*, *map_of_elements_per_page*
Output: *locations*
locations = list();
coverage = number of elements covered by *doc_pattern*;
for *(elements, section) in map_of_elements_per_section* **do**
    relative_coverage = number of *elements* / *coverage*;
    if *relative_coverage* > $COVERAGE\_THRESHOLD$ **then**
        add *section* to *locations*;
    end
end
if *locations is not empty* **then**
    return *locations*;
end
multi_pages = list();
for *(elements, page) in map_of_elements_per_page* **do**
    add *page* to *multi_pages*;
    relative_coverage = number of *elements* / *coverage*;
    if *relative_coverage* > $COVERAGE\_THRESHOLD$ **then**
        add *page* to *locations*;
    end
end
if *locations is not empty* **then**
    return *locations*;
else
    add *multi_pages* to *locations*;
    return *locations*
end

4.1.2 Evaluation

We investigated whether the documentation of the four open source systems we studied in Section 3.3 contained documentation patterns that matched the topics of documentation units (sections and pages). Intuitively, for each documentation pattern, we assessed whether each documentation unit linked to a documentation pattern was genuinely describing the code elements in the pattern, or whether the code elements

were present only by accident, e.g., because they were needed to instantiate a more important code element.

For example, the documentation pattern "all descendants of `DataSource`" matches the topic of Section 13.3 in the Spring Framework (the title of this section is "Controlling database connections"). In contrast, the documentation pattern "all classes starting with URL" covered in the Page "Chapter 1. Fundamentals" of the HttpClient manual, does not match the topic of the page, which is about HTTP protocol concepts such as requests and responses. The code elements in this documentation pattern are used throughout the page to support the construction of the more important objects (e.g., requests).

To support our qualitative assessment of relevance of the detected documentation patterns, we answered these research questions:

1. How many documentation patterns can we find in documents? How representative are these patterns (coverage)?
2. What kinds of intensions are the most frequent? Are they all useful to find documentation patterns?
3. How are the patterns usually covered (sections, pages, multi-pages)?
4. How meaningful are the patterns? Are the elements accidentally covered or are the patterns the real focus of the documentation units they are in?

Generation of Patterns. We executed all the steps presented in Section 4.1.1 on the same documentation releases that we analyzed in Section 3.3. Table 4.1 shows for each project's documentation (1) the number of code patterns generated, (2) the number of code patterns with a high coverage ($> 50\%$), (3) the number of documentation patterns once the redundant code patterns with high coverage are combined, and (4) the average coverage of the patterns with high coverage. For example, for the JodaTime documentation, we generated 3 120 code patterns, 103 of these patterns had a coverage higher than 50%, and after having combined the code patterns, we found 47 documentation patterns (1.5% of the code patterns). On average, the high coverage code patterns had a coverage of 84.3%.

System	Gen. Pat.	High Cov.	Doc. Patterns	Average Cov. %
Joda Doc.	3 120	103 (3.3%)	47 (1.5%)	84.3%
HC Doc.	4 762	232 (4.9%)	139 (2.9%)	80.0%
Hib. Doc.	17 619	149 (0.8%)	92 (0.5%)	74.0%
XSt. Doc.	2 133	143 (6.7%)	64 (3.0%)	80.9%

Table 4.1: Generation of Patterns

As we expected, the number of documentation patterns is much lower than the number of code patterns and the code patterns that are mentioned in the documentation usually overlap with at least another code pattern. For example, in the JodaTime documentation, each documentation pattern came from the combination of 2.2 code patterns on average (103 divided by 47).

Pattern Intensions. Table 4.2 shows for each of the 8 kinds of intensions (see Section 4.1.1) how many documentation patterns we detected in the documentation of the four target systems. For example, in HttpClient, we detected 32 documentation patterns that described code elements declared in another code element. The table only shows the intension of the most representative pattern for each documentation pattern.

The distribution of the documentation patterns is mostly consistent across the target systems. For example, the intension with the most documentation patterns in all target systems are code elements declared by another code element and sharing a common token. This particular intension is useful in identifying small sets of methods (e.g., all methods declared by `HttpClientConnection` and starting with the token "receive"). All intensions were used to identify at least one documentation pattern, which provide evidence that they are all useful.

Relationship between Patterns and Sections. Table 4.3 shows for each document (1) the number of documentation patterns located in a single section with the number of sections with at least one pattern in parentheses, (2) the number of patterns located on a single page with the number of pages with at least one pattern in parentheses, and (3) the number of patterns located in multiple pages. In

System	Decl.	Concrete Decl.	Child of.	Desc. of.	Concrete Desc.	Shared Token	Decl. & Token	Total
Joda Doc.	1	1	1	3	5	2	34	47
HC Doc.	32	4	4	5	8	27	59	139
Hib. Doc.	10	1	6	8	5	3	59	92
XSt. Doc.	10	2	5	5	4	13	25	64
Total	15.5%	2.3%	4.7%	6.1%	6.4%	13.2%	51.8%	100.0%

Table 4.2: Types of Intensions

Table 4.3, the single-page patterns **add** to the single-section patterns (which are by default single-page patterns), while the multi-page patterns consist of the rest of the patterns.

For example, in the JodaTime documentation, we found 25 patterns that were mainly covered in a single section. Sixteen sections out of 125 in Joda documentation covered such documentation patterns (one section can cover more than one pattern).

The documentation patterns were linked to different documentation units, which indicate that our patterns can match topics of different levels of granularity. For example, the documentation pattern "All fields declared in `ConnRoutePNames`" was entirely covered by Section 2.4 "HTTP route parameters" in the HttpClient manual. In contrast, the documentation pattern "All classes declared in package hbm2ddl and starting with token `Schema`" represents multiple tools that are explained in a full page in the Hibernate Documentation (Chapter 21. Toolset Guide).

If we exclude the multi-page patterns, less than half of the sections and pages were linked to a documentation pattern. Although some sections do not refer to Java code elements and could not potentially be linked to a documentation pattern, we believe that more intensions, such as those taking into account call relationships, should be investigated in the future to cover more sections in the documentation.

Relevance of Documentation Patterns. Although we found that documentation patterns exist in the documentation of the four target systems, we wanted to evaluate qualitatively whether the extension of these patterns were described in the documentation units or whether they were mentioned together by accident or to support more

System	Single Section	Single Page	Multi-Page	Total
Joda Doc.	25 (16/125)	10 (1/25)	12	47
HC Doc.	79 (40/100)	30 (5/9)	30	139
Hib. Doc.	61 (47/338)	17 (10/30)	14	92
XSt. Doc.	32 (18/203)	17 (3/24)	15	64
Total	57.6%	21.6%	20.8%	100.0%

Table 4.3: Patterns and Sections Linking

important elements. We randomly selected 25 documentation patterns in each project and we manually inspected the sections and pages that covered these patterns. We tried to assess whether the coverage of each pattern was:

1. Meaningful and exclusive: the section or page covered the pattern and it was the main focus of the documentation unit. There was a sentence or a group of words that matched the intension of the pattern.
2. Meaningful, but shared: the section or page covered the pattern, but there were also other patterns or elements that were covered by the documentation unit and that were as important as the pattern. There was a sentence or a group of words that matched the intension of the pattern (or a more general intension).
3. Supportive: the elements in the pattern were all related, but they were not the focus of the section and they appeared only to instantiate or contextualize the elements that were the focus of the documentation unit.
4. Accidental: there was nothing in the documentation unit that matched the intension of the pattern. The code elements were mentioned together by accident.

Table 4.4 shows for each document the number of sections and pages that were covering the 25 patterns we randomly selected (more than one section can represent a pattern). Then, the table shows the categorization of the units. For example, in Joda Time, the 25 patterns that we randomly selected were located in 16 sections, 6 pages, and 7 multi-pages for a total of 29 documentation units. Out of these 29 units, we judged that 6 had a meaningful coverage of the pattern and that the pattern was the

sole focus of the section. 20 had a meaningful coverage of the pattern but these units also focused on code elements that were not covered by a pattern. One unit covered the elements of a documentation pattern to support more important elements. Two units accidentally covered the elements of a documentation pattern.

82% (17.9% + 64.1%) of the documentation units covered a documentation pattern that matched the topic of the unit. This result provides evidence that our technique can identify with a relatively high precision the structural relationships (intension) discussed in a documentation unit.

Exclusive and Meaningful Coverage.

We found that when a documentation pattern involved all the constants in a class, the pattern was always the main focus of a section. For example, we identified the pattern "all fields in `ExecutionContext`" that was covered in Section 1.2 "HTTP Execution Context" of the HttpClient manual.

Although in the previous example, the section's title and the intension shared a common name (`ExecutionContext`), this was not always the case. For instance, in JodaTime, we identified the documentation pattern "All methods declared in `AbstractDateTime` that starts with the prefix *to*" in section "JDK Interoperability". Indeed, the methods `toCalendar` and `toGregorianCalendar` are the main interoperability points between Joda Time and the Java Standard Library.

A documentation pattern was rarely meaningfully covered by multiple pages (three exclusive and meaningful multi-page patterns out of 21), except when each page described a single element of the pattern. This was the case of the pattern "All descendants of `AssembledChronology`" in Joda Time. Each class of this pattern is presented in a single page (e.g., Islamic calendar system, Julian calendar system, etc.).

Shared and Meaningful Coverage. As shown in Table 4.4 most of the documentation patterns were not the sole focus of a documentation unit. In 18 documentation units (out of 75 shared and meaningful), we found that the documentation pattern was a proper subset of a larger documentation pattern, hence the incomplete coverage. For example, Section "Converters" of the XStream manual presented all the classes implementing the `Converter` interface, but our technique generated many subpatterns

such as "all descendants of `AbstractReflectionConverter`". These patterns are usually combined together unless the size of their extension differs greatly (see Section 4.1).

The other reason for shared coverage was when more than one distinct documentation pattern was mentioned in a documentation unit. For example, section 16.4 "Associations" in the Hibernate manual covered both the pattern "all methods of `Restrictions` that starts with the prefix *eq*" and "all methods of `Criteria` that ends with suffix *alias*". Each documentation pattern, taken individually, incompletely covered the section.

Supportive.

Only a few documentation patterns contained *related code* elements that were not the focus of the documentation units. For example, in XStream manual, we identified the documentation pattern "all descendants of `AbstractFilePersistenceStrategy`". Although the classes in this pattern were clearly related (`FileStreamStrategy` and `FilePersistenceStrategy`), they were covered by multiple pages for different reasons. The former was mentioned while discussing performance strategies and the latter was mentioned while discussing object conversion strategies.

Accidental.

12.8% of the documentation units we inspected accidentally covered a documentation pattern. This was the case of the pattern "All fields starting with prefix `ignore`" in the Hibernate manual. Although the two fields that matched this intension were covered by multiple pages, they were mentioned in different context and they had different meaning: `CacheMode.IGNORE` is about query caching while `ReplicationMode.IGNORE` is about replicating data between databases.

4.2 Support Channels and Documentation

In our analysis of the Spring Framework support channel, we found that messages often discussed topics presented in a documentation section, but the messages rarely explicitly mentioned the section. Automatically finding messages and sections that discuss the same topic can help both developers asking questions and documentation

System	Single Section	Single Page	Multi Page	Total	Mean. Excl.	Mean. Shared	Supportive	Accidental
Joda Doc.	16	6	7	29	6	20	1	2
HC Doc.	11	7	7	25	6	13	1	5
Hib. Doc.	21	8	2	31	4	22	0	5
XSt. Doc.	23	4	5	32	5	20	4	3
Total	48.2%	21.4%	30.4%	100.0%	17.9%	64.1%	5.2%	12.8%

Table 4.4: Relevance of Documentation Patterns

maintainers: the documentation section may answer the question asked in the message and multiple questions on the same topic may indicate that the documentation needs to be clarified.

Researchers have used information retrieval techniques in the past to link documents together with mitigated success [7]. Because RecoDoc provides the list of fine-grained elements mentioned in both messages and documentation sections, we can now precisely match documents that mention the same code elements and that are likely to discuss the same topic. Although it would make sense to combine both techniques (information retrieval and heuristics based on the presence of code elements), we first want to investigate whether we can find useful relationships with our heuristics and what kind of refinement would be needed.

Matching messages and documentation sections based on a single common code element results in a high number of spurious correspondences. For example, as we show in Section 4.2.1, there are 36 908 messages in the Hibernate forum that refer to a code element discussed in a documentation section. On average, each of these messages is related to 39 documentation sections making it doubtful that these messages are really related to so many different topics at once.

We thus need relationship criteria that are more selective. Intuitively, messages sharing a high number of code elements with a section are more likely to discuss a common topic than messages sharing a lower number of code elements. Additionally, if the common code elements are mentioned in the text of the messages as opposed to a code snippet, these code elements are likely to be more significant because code

snippets often require extra code elements unrelated to the topic to provide proper context.

Finally, even if a message and a documentation section are discussing the same topic, this relationship might not be helpful at all to the developers and the documentation maintainers. For example, a message might be about reporting a bug or requesting an extension to a feature presented in a documentation section. In these instances, the documentation does not need to be clarified and the documentation will not help the message's author.

4.2.1 Evaluation

We investigated whether using the number of common code elements between messages and documentation sections could lead to the identification of relationships that can help documentation maintainers and developers asking questions.

For each of the four target systems we studied in the previous sections, we computed the list of support messages and documentation sections that shared a specified number of code elements. In Table 4.5, we report the number of messages that match at least one documentation section. The second column of the table shows the total number of messages in the support channel for each system. The third, fourth, and fifth column show the number of messages that shared **at least** one, three, and five code elements with a documentation section. The sixth, seventh, and eight column show the number of messages that mentioned in their text (as opposed to in a code snippet) at least one, three, and five code elements also mentioned in a documentation section. The average number of sections that were matched with each message is given in parentheses. For example, in Joda Time, there were 265 messages that shared at least three code elements with a documentation section, and on average, each of these messages was related to 2.4 sections.

Table 4.6 shows the same numbers, but from the perspective of documentation sections. For example, in Joda Time, there were 36 sections out of 125 that shared at least three code elements with a support message. On average, each of these sections was related to 17.5 messages.

System	Msgs	Any-1	Any-3	Any-5	Text-1	Text-3	Text-5
Joda Mail.	2 951	1 311 (21.1)	265 (2.4)	29 (1.7)	1142 (13.0)	103 (2.2)	15 (1.9)
HC Mail.	9 393	1 876 (13.0)	523 (8.7)	238 (8.5)	997 (2.3)	23 (1.1)	0 (0.0)
Hib Forum	185 447	36 908 (38.7)	13 642 (14.8)	6 235 (7.9)	27 252 (29.0)	5 213 (2.4)	142 (1.5)
XSt Mail.	6 357	2 376 (16.7)	744 (5.4)	282 (2.9)	1 509 (5.5)	44 (1.0)	6 (1.0)

Table 4.5: Number of Messages matching a Documentation Section

System	Sections	Any-1	Any-3	Any-5	Text-1	Text-3	Text-5
Joda Doc.	125	65 (425.9)	36 (17.5)	13 (3.8)	63 (235.6)	15 (15.1)	5 (5.7)
HC Doc.	100	72 (338.3)	51 (89.6)	45 (45.1)	58 (38.8)	13 (1.9)	0 (0.0)
Hib Doc.	338	172 (8 307.0)	107 (1 889.1)	64 (769.7)	150 (5 267.7)	69 (182.2)	19 (11.1)
XSt Doc.	203	80 (495.2)	40 (100.0)	27 (30.6)	59 (141.8)	9 (5.0)	2 (3.0)

Table 4.6: Number of Sections matching a Message

As it could be expected, increasing the number of common code elements dramatically decreases the number of related messages and sections. Moreover, when we consider only the code elements that are mentioned in the text of the support messages, the number of related sections and messages decreases even further.

From a quantitative perspective, this second criteria seems to be crucial in the identification of meaningful relationships. For example, when we selected messages and sections sharing at least five code elements (mentioned anywhere in the message), each of the selected messages in HttpClient and Hibernate were related on average to 8.5 and 7.9 sections respectively. It is highly unlikely that so many messages are related to so many different topics. In contrast, when we selected messages and sections sharing at least three code elements, *mentioned in the text of a message*, the average number of sections matched to each selected message did not go over 2.4.

Qualitative Evaluation. We showed that matching messages and sections based on (1) how a code element is mentioned (text vs. code snippet) and on (2) a higher number of common code elements yield a small number of matches, but these matches seem more focused.

We evaluated qualitatively to what extent relationships found with these two criteria were *meaningful* and *helpful* to documentation maintainers and developers asking questions. For each target system, we manually inspected a random sample of 25 messages that had been matched to sections sharing at least three code elements mentioned in the text of the messages. We used this threshold because it was very selective: it identified a manageable number of messages and sections that were likely to be related (i.e., small number of matched sections per message). A less selective threshold would result in a higher number of meaningless relationships (and thus unhelpful) and a more selective threshold would not identify enough relationships. Even at this threshold, the HttpClient target system had only 23 matched messages (two below our random sample).

For our manual inspection, we read the sampled messages, their respective support threads (to understand the context), and the matched sections. We then classified how meaningful the relationship between the message and each matched section was:

1. Meaningful and exclusive: the section entirely covered the topic discussed in the support message. There was a sentence or a group of words in the message that summarized a topic discussed in the documentation section.
2. Meaningful and shared: the section covered a subset of the topics discussed in the support message. Again, there was a sentence or a group of words in the message that summarized a topic discussed in the documentation section.
3. Supportive: the common code elements mentioned in the message and the section were not related to the topic discussed in the message. These elements appeared only to instantiate or contextualize the code elements that were the focus of the message.
4. Accidental: the message and the section were unrelated and the code elements were mentioned together by accident.

When we determined that a message and at least one matched section had a meaningful (exclusive or shared) relationship, we then tried to assess whether:

System	Mean. Excl.	Mean. Shared	Supportive	Accidental	Total Mean.	Help Doc.	Help Msg.	No Help
Joda	11	31	0	7	21	4	0	17
HC	20	4	1	0	23	7	2	13
Hib	8	4	33	0	11	6	1	4
XSt	11	4	6	5	15	1	4	10

Table 4.7: Type of relationship between messages and sections sharing at least three code elements mentioned in the text of a message

1. the message could help clarify the documentation section. This was the case when someone on the support thread clarified a topic, linked two topics that appeared to be unrelated in the documentation, or stated that the problem encountered on the support thread was common and provided a solution.
2. the documentation could answer the questions asked in the message or in the support thread.
3. the relationship was unhelpful to documentation maintainers and developers asking questions.

Table 4.7 shows the results of our classification. For each target system, we first report how many matched sections were meaningful (columns Mean Excl., Mean Shared., Supportive, Accidental). Because a message may be matched to more than one section, the total number of sections per target system can be above 25. Then, we show the number of messages that had a meaningful relationship (complete or incomplete) with at least one section (Total Mean.). The theoretical maximum for this column is 25 for all target systems except HttpClient (23). Finally, we show the number of helpful relationships that we found (Help Doc., Help Msg., No Help). In our classification, helpful messages are a subset of meaningful messages.

For example, for Joda Time, we inspected 25 messages that matched a total of 49 documentation sections (11 + 31 + 7). We found that 7 of these sections had been accidentally matched, i.e., they shared at least three code elements with a message, but these code elements were not related at all to the topic discussed in the message

and the section. 21 messages had a meaningful relationship with a section. We found that four of these messages could have helped clarified the documentation, but we could not determine how the rest of the messages (17) could help the documentation or could be helped by the documentation.

Except for Hibernate, we found that most messages and sections sharing at least three common code elements discussed the same topic. For example, the message "RE: xml file on one single level" in the XStream mailing list matched the section "Context" because they shared four code elements. After reading the message and the support thread, we concluded that both the section and the message discussed the same topic (serialization and deserialization). This sentence from the message summarized well the topic: "However, the code that processes the object graph as tree is part of the MarshallingStrategy implementation (resp. the MarshallingContext)".

The messages on the Hibernate forum often referred to three code elements that are almost always needed when using this framework: `SessionFactory`, `SessionFactory.openSession`, and `Session`. These three code elements were mentioned in two sections (3.4 Optional configuration properties, and 12.2.3. Exception handling) and 14 messages out of 25 were wrongly matched to these sections. For example, the message "Core 3.2.4. native id generator no longer works" was about primary key generation, which had nothing to do with sections 3.4 and 12.2.3.

Unfortunately, even when a message was discussing the same topic as a matched section, the message could rarely help clarify the documentation or vice versa. For example, the message "Multiplication operations" in JodaTime was correctly matched to section "Intervals and time periods", but the message contained a feature request: "basically floating point multiplication operations on Durations an Intervals". The documentation did not need any clarification and it could not help answer the message, so we classified this message as unhelpful.

We found a few messages that could have helped clarify the documentation. For example, in the message "Some Period Questions" from the JodaTime mailing list, a developer asked a question about the difference between two concepts (Period and Duration) and one of the author of JodaTime explained the subtle difference in a reply. This explanation was not present in the documentation and would likely clarify it.

In other cases, the documentation could have helped answer the question asked in the message. For instance, in the message "Units for connection timeouts", from the HttpClient mailing list, a developer asked a question about the time unit associated with a parameter. This question was answered in the matched section, "2.1 Connection parameters".

Clearly, automatically linking messages to documentation sections has the potential to help both developers and documentation maintainers. Seeing the list of documentation sections related to a message being written could help the message's author at a very low cost: if the author is requesting a feature or reporting a bug, he can safely ignore the linked sections. In contrast, listing all the messages that are related to a documentation section may not help a documentation maintainer for now because there are too many false positives, i.e., messages that are unhelpful even if they are related to a section. Based on our observations, we found two criteria that could further help a maintainer determine in advance whether a message indicates that the documentation needs to be clarified.

First, classifying a support thread with broad categories (e.g., feature request/bug report, design discussion, or question), either with an automated classifier or with manual tagging, would greatly improve the selection of helpful messages. Second, we observed that even in our small sample, a few developers frequently answered the questions. Some of these developers were not even official contributors of the target system. We found that the explanations given by these developers often clarified the documentation. Because these developers also participated in design, bug, and feature request discussions, we could not automatically categorize their answers as helpful.

4.3 Discussion

We showed in this chapter that, even at the exploratory stage, our technique can precisely identify meaningful high-level relationships between documentation units and support messages, and documentation structures in the form of documentation

patterns. For instance, 82% of documentation patterns we computed were the focus (exclusive or shared) of a documentation unit while 71% of the links between support messages and documentation sections were meaningful (81% if we exclude Hibernate).

Documentation patterns may help improve developer learning resources. For example, because the documentation patterns are based on general intensions, code elements introduced in a new release may match these intensions, indicating that the elements need to be documented. We also found a few examples where the support messages helped clarified the documentation, but in general, the links between messages and documentation sections were not helpful and more work is needed to discriminate real questions from design discussions, feature requests, and bug reports.

In our evaluation of the high-level documentation structures, we did not study recall, i.e., the number of detected links out of the total number of links in a target system release. Given the exploratory nature of our study, we wanted to focus our effort on the links we could uncover instead of the links we missed. Moreover, because of the large number of potential links, the cost of computing an oracle would have outweighed its value. In Chapter 5, we study one aspect of recall by investigating documented changes in a codebase that were not captured by documentation patterns.

Threats to validity.

The evaluation of our high-level structures detection technique is subject to the same threats to validity as discussed in Section 3.3.3.

Specifically, we evaluated our technique on four different target systems and the technique was devised while studying the Spring Framework learning resources to prevent overfitting.

Determining whether a section, a support message, and a documentation pattern discuss the same topic is inherently a subjective, but highly informative assessment. To mitigate investigator bias, we based this assessment on explicit and verifiable criteria (e.g., the presence of a sentence summarizing the topic, explicit mention that a question was asked several times, etc.) and our assessment of each link is publicly available for inspection.[1] Given the exploratory nature of our investigation and the

[1] http://www.cs.mcgill.ca/~swevo/recodoc

difficulty in recruiting open source contributors [23], we believe that inspection of the results by the authors represented an appropriate trade-off.

Finally, the quantitative results (e.g., number of code patterns detected) are dependant on the accuracy of RecoDoc. We demonstrated that for these four target systems, RecoDoc was highly accurate in linking code-like terms to code elements, but there were still a few errors that likely impacted our results: we may have missed links or inferred erroneous links. During our qualitative assessment of the links, we never encountered a missing or erroneous link.

Chapter 5
Recommending Adaptive Changes for Documentation Evolution

We demonstrated that RecoDoc can precisely recover fine-grained links between documentation and code as well as high-level documentation structures. We now show how we can leverage these two types of links to build a recommendation system that suggests adaptive changes to the documentation when the underlying codebase evolves. For example, if a documented method is removed from the codebase, the documentation should be adapted to take into account this change. We make the assumption that the code evolves first and that the documentation is then adapted: this is one of the three main workflows that we observed in Chapter 2.

As we found out in our qualitative study on developer documentation, there are many factors that motivate the decisions to document certain code elements and to ignore others. These factors are complex and subjective, and cannot all be systematically considered by a given tool. For example, open source contributors consider learnability, marketing, their own experience, writing style guidelines, and feedback from users when creating and maintaining the documentation (see Section 2.3.2).

We chose to focus our effort on adapting the documentation to code evolution because code changes can be precisely detected, and must be reflected in the documentation. This kind of recommendations, adaptive changes, is only a first step

toward building a more comprehensive recommender that can take into account the multiple sources of improvement in documentation.

5.1 Documentation Patterns Evolution

When new features are added to an API, documentation maintainers need to update the documentation to cover these new features. To be useful, the documentation of large frameworks has to stay concise and cannot cover every single code element. The problem then becomes: given the documentation choices in the previous documentation release, which new code elements should be documented?

For example, between the 4.0.1 and 4.1.1 releases of HttpClient, 896 code elements (classes, methods, fields) were added, but only 1.8% of these new elements were mentioned in the new release of the documentation.

Given the models of the API and the documentation generated by RecoDoc for each release of a framework, we propose to identify all the new code elements that fit an existing documentation pattern and that should be documented in the new release of the documentation.

In a previous example (see Section 4.1), we identified in Section 13.3 of the Spring manual a documentation pattern that covered all concrete subclasses of `DataSource`. If a new subclass of `DataSource` is added in the next release of the Spring Framework, we should recommend to mention this subclass in section 13.3. Such recommendations would help documentation maintainers by (1) ensuring that a new code element matching a previous documentation decision is not forgotten, and (2) speeding up the process of deciding whether a new code element should be documented.

Limitation. The main limitation of recommending new code elements that fit existing documentation patterns is that we cannot recommend code elements that are part of a new documentation pattern. For example, a new category of features (cache abstraction) was added in Spring Framework 3.0. This new set of features necessitated a page of documentation on its own and it did not fit an existing documentation pattern

because a new high-level package was introduced with many classes and annotations that did not inherit from existing classes.

5.1.1 Computing Documentation Pattern Recommendations

We perform four steps to identify the new code elements in a release that should be mentioned in the documentation. These steps are based on the inference of documentation patterns as explained in Section 4.1.1. Recall that a code pattern is a coherent set of code elements with an intension and an extension, and that a documentation pattern is a set of redundant code patterns with a high coverage in the documentation, which is represented by one code pattern. In contrast with the previous chapter, we only combine code patterns into documentation patterns at the end of the process because we first have to match and compare the code patterns between two releases and the code patterns may have changed due to the code evolution. For instance, the representative pattern of a documentation pattern may have been deleted in a new release, but one of the redundant code patterns may still exist.

Steps required to identify new code elements that should be mentioned in a documentation release:

1. Infer code patterns with high coverage in releases N and N+1 of the code.
2. Compare the coverage of the code patterns between the two releases.
3. Compute the addition recommendations.
4. Combine the redundant addition recommendations into a single addition recommendation and a documentation pattern.

Inferring Code Patterns.

We reuse the process presented in Section 4.1.1 to detect code patterns with high coverage in two releases. Given two releases of a codebase, N and N+1, and the initial release of the documentation, N, we compute two collections of code patterns: one for

codebase N with documentation N, and one for codebase N+1 with documentation N.

Code patterns in release N that have a coverage less than to 50% are discarded because they are not considered to match the intent of the documentation.

We do not combine the code patterns into documentation patterns at this step to ease the matching of code patterns between the two releases (see next step).

Comparing Pattern Coverage.

In this step, we match the code patterns from the releases N and N+1 based on their intension and we compare their coverage.

For example, the code pattern "all classes extending `DataSource`" contained 5 code elements in Spring Framework 2.0. Three of these elements were mentioned in the documentation of 2.0 (coverage = 60%). In the Spring Framework 3.0, the same pattern now contains 8 elements and three of these elements are mentioned in the documentation of 2.0 (coverage = 37.5%).

From these numbers, we can infer that the coverage of this code pattern decreased and the documentation maintainer should probably document the new code elements.

We discard patterns whose coverage stays constant or increases (this can happen if the number of code elements in the pattern decreases in the new version) because they are not interesting for addition recommendations: we address removed code elements in Section 5.2.

Finally, we discard code patterns that do not have a matching pattern in the previous or current version. For example, if the `DataSource` hierarchy had been deleted in release 3.0 of the Spring Framework, the initial code pattern would have been discarded.

Computing Recommendations.

For each code pattern whose coverage decreased between two releases, we produce a recommendation. Each recommendation contains three components:

1. The initial and new coverage. This indicates how representative the code pattern was and how much changes occurred between the two releases.

2. The new code elements that are part of the code pattern and that are not mentioned in the documentation.
3. The location of the pattern, which provides an indication where the new code elements should be mentioned. We reuse Algorithm 2 presented in Section 4.1.1 to find the location of the pattern.

Combining Recommendations.

Finally, we combine redundant recommendations that are a subset of a larger recommendation. This process is similar to Algorithm 1 because we group all recommendations that are a subset of larger recommendations. The main difference with Algorithm 1 is that instead of selecting the pattern with the highest coverage, we select the code pattern with the most elements as the most representative pattern because we want the documentation maintainer to consider all potential code elements in the pattern and not just a subset of it.

5.2 API Elements Deletion and Deprecation

Code elements may be removed, deprecated, or refactored between releases and the documentation needs to be updated accordingly. For example, a tutorial that mentions a class that has been deprecated in the new release could be updated by removing the reference to the class and mentioning a more appropriate class.

Finding references to removed or deprecated code elements in a document is straightforward with RecoDoc:

1. We recover the set of links between the codebase at release N and the documentation at release N.
2. For each code element that was deprecated or deleted in the codebase between release N and N+1, we produce a recommendation if the code element was mentioned in the documentation.

This type of recommendation demonstrates that we can use fine-grained links to improve documentation.

5.3 Recommender Evaluation

To objectively estimate the usefulness and accuracy of our recommendations, we performed a retrospective analysis on the documentation of four open source projects. We computed recommendations for an old documentation release from each project and then, we compared our recommendations with the newer documentation releases. This comparison provided a baseline to evaluate our recommendations: if one of the subsequent documentation releases contains the changes proposed by our recommendations, it is evidence that the recommendations could have been useful. In contrast, if the documentation release does not contain the change proposed by our recommendations, we will conservatively judge that the recommendations would not have been useful, even though it may just be that the documentation maintainer forgot to document the recommended code elements.

We selected the same four projects that we used to evaluate RecoDoc's linking process because we demonstrated that RecoDoc has a high accuracy and therefore, it should not heavily influence the results of our recommendation strategies.

We selected all the minor releases (second digit of the release number) for which we could build the documentation, but we avoided releases that were not backward compatible or that introduced significant structural changes in the code or in the documentation. For example, between releases 3 and 4, HttpClient was split into two projects, most classes and packages were renamed and moved and the documentation was rewritten from scratch.

Tables 5.1 and 5.2 show the main changes that occurred in the code and in the documentation of the selected project releases. In the first table, we show the number of public or protected types and members before and after the code release, and the total number of deprecated types and members after the code release.

The second table shows the number of pages, sections, and links to code elements before and after the documentation release. We removed pages that were related to project news and changelogs, because they provide information that is not integrated with the main documentation, and these pages do not need to be corrected between releases.

System	Release Src	Release Dst	Types Src	Types Dst	Members Src	Members Dst	Types Deprec.	Members Deprec.
Joda	1.0	1.4	200	219	3 120	3 937	0	20
Joda	1.4	1.5	219	221	3 937	3 974	4	25
Joda	1.5	1.6.2	221	221	3 974	3 991	4	26
HC	4.0.1	4.1.1	512	618	3 276	4 066	33	68
Hib	3.3.2	3.5.5	1 327	2 124	12 860	17 724	40	126
XSt	1.0.2	1.1.3	117	192	439	1 069	1	23
XSt	1.1.3	1.2.2	192	273	1 069	1 558	7	55
XSt	1.2.2	1.3.1	273	309	1 558	1 779	20	98

Table 5.1: Evolution of codebase

System	Release Src	Release Dst	Pages Src	Pages Dst	Sections Src	Sections Dst	Links Src	Links Dst
Joda	1.0	1.4	20	19	113	114	496	564
Joda	1.4	1.5	19	24	114	124	564	604
Joda	1.5	1.6.2	24	25	124	125	604	607
HC	4.0.1	4.1.1	8	9	84	100	1 099	1 302
Hib	3.3.2	3.5.5	29	30	320	338	1 788	1 879
XSt	1.0.2	1.1.3	12	17	55	80	69	124
XSt	1.1.3	1.2.2	17	25	80	146	124	511
XSt	1.2.2	1.3.1	25	24	146	203	511	659

Table 5.2: Evolution of documentation

As a second step to our evaluation, we contacted the contributors of these four open source projects to ask them to evaluate our recommendations. One contributor positively replied to our invitation and we report in Section 5.3.3 the contributor's evaluation on the correctness, the usefulness, and the cost of false positives.

5.3.1 Addition Recommendations

RecoDoc generated addition recommendations for each documentation release: our recommendation system computed a list of code patterns, compared their coverage, and indicated the patterns whose coverage had decreased. To evaluate the usefulness and

accuracy of these recommendations, we were interested in answering these research questions:

1. How precise are the recommendations? Do the recommendations correctly identify new code elements that should be documented given the previous documentation choices?
2. How much of the documentation additions can be explained by documentation patterns? Why did our approach miss some documentation additions?

To answer the first research question, we evaluated our recommendations by manually inspecting the documentation releases that were published after the documentation release for which we generated the recommendations.

For example, RecoDoc generated addition recommendations for the Joda Time 1.0 documentation based on the changes in the code between 1.0 and 1.4. We first inspected the documentation at version 1.0 to ensure that the links and documentation patterns inferred by RecoDoc were accurate. We then manually inspected the documentation at version 1.4 to check if it mentioned the code elements that we had recommended. We looked at each section that had referred to an existing code element in the pattern. If we could not find the references to the new code elements, we inspected the next releases (1.5, 1.6.2, and 2.0, the current release on the web). This evaluation strategy is conservative because it assumes that non-implemented recommendations were explicitly judged irrelevant as opposed to being overlooked by documentation maintainers.

RecoDoc also computed a list of links to new code elements that were introduced in each documentation release to address the second research question. We used this list to determine how many links to new code elements were explained by a documentation pattern and how many links we missed. We also used this list to make sure that our manual inspection did not miss any new links.

Table 5.3 shows the results of our inspection. The column "Doc Patterns" shows the number of documentation patterns that generated at least one recommendation for each release. The next column shows the number of documentation patterns

System	Precision				Recall			
	Rec. Doc. Patterns	Patterns Correct	Single Rec	Single Correct	New Types	New Members	Types Found	Members Found
Joda 1.0-1.4	4	3	21	15	13	6	13	2
Joda 1.4-1.5	1	0	1	0	0	0	0	0
Joda 1.5-1.6.2	0	0	0	0	0	0	0	0
HC 4.0.1-4.1.1	14	9	27	11	10	11	6	5
Hib* 3.3.2-3.5.5	13	8	52	14	0	5	0	1
XSt* 1.0.2-1.1.3	1	1	10	10	1	4	0	0
XSt* 1.1.3-1.2.2	6	5	32	13	13	12	8	2
XSt* 1.2.2-1.3.1	7	3	19	9	9	6	8	0
Total	46	29	162	72	46	44	35	10

Table 5.3: Evaluation of Documentation Patterns Recommendations.

for which at least one recommendation was implemented in the next release of the documentation. The column "Rec" shows the number of new code elements that we recommended and the next column shows the number of these code elements that were actually mentioned in the next release. The second part of the table shows the number of new types and members (in existing types) that were mentioned in the newer documentation release and the number of these types and members that our recommendations covered.

For example, we found between the releases 1.0 and 1.4 of Joda Time that four documentation patterns had a coverage that decreased. New code elements from three of these documentation patterns were mentioned in 1.4. In total, these four documentation patterns contained 21 new code elements and 15 of these code elements were mentioned in the 1.4 release. Between, 1.0 and 1.4, the documentation added one or more reference to 13 types and 6 members: 13 of these types and two of these members were covered by our recommendations.

In Hibernate and XStream, we found documented code elements from our recommendations in a documentation release that was not immediately following the code release. For example, in XStream 1.1.3-1.2.2, we recommended to document the class `XMLArrayList`, but it was documented only in 1.3.1 instead of 1.2.2. This is why the

number of correct recommendations is not always equal to the number of types and members found. As we explained earlier in this section, we only looked at further documentation releases when we could not find an implementation of our recommendations in the documentation release immediately following the code release.

Precision of Recommendations. Considering that the releases we studied introduced 7 865 new members and 1 116 new types, the 46 recommendations (for a total of 162 recommended code elements) of RecoDoc clearly represents an improvement over manually reviewing each code addition. Moreover, because our evaluation strategy was conservative, the low precision of our recommendations (29 / 46 = 63%) represents a lower bound on the accuracy of our technique.

Our recommendations were particularly accurate when they concerned types, and when the intension of the documentation pattern was not related to a common token. For example, for Joda Time 1.0, RecoDoc found the multi-page documentation pattern "all descendants of `BaseChronology`" and it correctly recommended to document the new members of this pattern (`IslamicChronology`, `EthiopicChronology`, etc.) in release 1.4: when manually reviewing release 1.4, we found that the documentation maintainer had created a new page for these new classes.

In HttpClient 4.0.1, RecoDoc found the single-section documentation pattern "All classes declared in the `http.conn.scheme` package" and it correctly recommended to document the new classes in this package (e.g., `LayeredSchemeSocketFactory`) in release 4.1.1.

RecoDoc was also accurate when it detected a documentation pattern related to constants. For example, in XStream 1.1.3, RecoDoc found the pattern "All fields (constants) declared in the `XStream` class" and it correctly recommended to document the new constants in 1.2.2.

RecoDoc correctly recommended methods associated with a token. For example, in Joda Time 1.0, RecoDoc found the single-page documentation pattern "All methods declared in `DateTime` and ending with the token Year" and correctly recommended to document the three new members in release 1.4.

Regarding the false positives, RecoDoc found nine documentation patterns whose extension in future releases were related to internal implementation. For example, in Hibernate 3.3.2, RecoDoc found the pattern "All non-abstract classes in package `org.hibernate.stat`". These classes were refactored in 3.5.5 and an interface was extracted for each of these classes. RecoDoc recommended to document all the new non-abstract classes in 3.5.5 (e.g., `QueryStatisticsImpl`), but because these classes were now part of the internal implementation, they were not documented and the recommendation was incorrect.

We observed only two documentation patterns inferred by RecoDoc that were accidental (e.g., "All classes starting with X" in XStream 1.2.2). Unsurprisingly, the recommendations from these patterns were incorrect.

Finally, we found that six documentation patterns inferred by RecoDoc and the resulting recommendations made sense but were not implemented by the documentation maintainers. For example, in HttpClient, RecoDoc recommended to document the class `CookieRestrictionViolationException` because it was part of the documentation pattern "All classes declared in `org.apache.http.cookie` and starting with Cookie". Although the documentation discusses policies and cookie validation, it never mentions that HttpClient can throw exceptions, which is not a good documentation practice [11].

Documentation Additions vs. Documentation Patterns.

We found that 90 (46 + 44) new types and members had been documented in the documentation release following a code release. Documentation patterns inferred by RecoDoc covered 50% (35 + 10 / 90) of these new types and members.

11 types out of 46 were not part of an existing documentation pattern. For example, in HttpClient, a new section (5.5 Compressed response content) was added in release 4.1.1 to discuss a new set of classes related to character encoding (e.g., `ContentEncodingHttpClient`, `RequestAcceptEncoding`, etc.). These classes would form a new documentation pattern, but since it did not exist in the previous release, RecoDoc did not recommend to document the classes.

The documentation patterns inferred by RecoDoc did not cover most of the methods (34 out of 44). This is because our intensions are not suited to capture small sets

of methods that do not follow a regular structure. For example, in XStream 1.3.1, the documentation authors added a reference to `XStream.autodetectAnnotations()` in a section. The `XStream` class declares more than 50 methods and yet, the section only referred to one method.

Overall, five types (out of 11 not recommended by RecoDoc) and three methods (out of 34) were part of new documentation patterns and were added together in a section. The other types and members were added in isolation of each other.

5.3.2 Deletion Recommendations

For each project release, RecoDoc computed the list of code elements that had been deprecated or deleted and automatically produced a recommendation when these elements were mentioned in the documentation. For each recommendation, we inspected the next documentation releases to check if these references to deprecated or removed elements had been corrected. We considered that the documentation had addressed the deletion or deprecation of a code element if it (1) referred to the new element, (2) it mentioned that the element had been deprecated, or (3) it no longer mentioned the element.

We also compared RecoDoc with a traditional textual search tool (grep). For each deprecated class, we performed a case sensitive search (e.g., "TimeOfDay") in the documentation. For each deprecated method, we performed two case sensitive search: one with the name of the method, and one with the opening parenthesis (e.g., "getISO" and "getISO("). Because the textual search tool is not linked to the code model, we had to provide the tool with a list of deprecated code elements and run the tool for each code element.

Table 5.4 shows the results of our inspection for each release. The "Global Rec" column shows the number of deprecated elements for which RecoDoc found at least one reference. The "Single Rec" column shows the number of references to deprecated or deleted elements that we found between each release (one deprecated element can be mentioned multiple times). Then, the table shows the number of these references that were indeed pointing to a deprecated element (True Positive), the number of references

System	Global Rec	Single Rec	True Positive	Corrected Ref.	Not Corrected	False Positive	False Negative
Joda 1.0-1.4	6	16 (16)	16 (16)	14	2	0 (0)	0 (0)
Joda 1.4-1.5	2	5 (5)	5 (5)	4	1	0 (0)	0 (0)
Joda 1.5-1.6.2	0	0	0	0	0	0	0
HC	13	32 (38)	32 (33)	32	1	0 (5)	1 (0)
Hib	18	38 (123)	29 (29)	3	26	9 (94)	0 (0)
XSt 1.0.2-1.1.3	0	0	0	0	0	0	0
XSt 1.1.3-1.2.2	2	2 (5)	1 (1)	0	1	1 (4)	0 (0)
XSt 1.2.2-1.3.1	8	21 (24)	20 (19)	20	0	1 (4)	0 (1)
Total	49	114 (211)	103 (103)	72	31	11 (107)	1 (1)

Table 5.4: Removed and Deprecated Elements Recommendations

that had been corrected in one of the next documentation releases (Corrected Ref), the number of references that had been left unchanged (Not Corrected), the number of false positives, and the number of missed references computed from a general textual search and manually verified (false negative). The numbers in parentheses represent the results for the textual search.

For example, for HttpClient, we found that 13 deprecated elements were referenced in the documentation. RecoDoc found 32 references to these 13 elements and the textual search found 38 references. 33 of these references were pointing to a deprecated elements: RecoDoc thus missed one reference (1 false negative) and the textual search produced five false positives. Out of these 33 references, 32 had been corrected in the next release of the HttpClient documentation, but one reference had not been corrected. The true positives identified by the textual search were a superset of the true positives found by RecoDoc and the incorrect reference was identified by both tools.

The number of single recommendations reported for the textual search is for the most precise search only (case sensitive search for classes and case sensitive search with first parenthesis for methods). When we performed a textual search for methods without the first parenthesis, we found all the references to deprecated code elements (false negative = 0), but with an unacceptable number of false positives (910).

It is clear from these results that RecoDoc can automate the process of finding references to deprecated code elements with a higher accuracy than a simple textual search. Textual search was particularly imprecise when we searched for short and common method names. For example, in Hibernate, searching for "get" yielded 296 results (27 when searching for "get(")

Additionally, textual search tools lack the relationships with the codebase, so the user has to manually identify all the deprecated elements first, and then execute the textual search tool for each of the deprecated element.

Finally, we found that the identification of references to deprecated elements can uncover documentation errors that are misleading. For example, in Hibernate, the documentation is still telling readers to call `Session.lock()` instead of the new `Session.buildLockRequest()`. Producing these recommendations with RecoDoc takes a few seconds, so we believe that using RecoDoc between each release is valuable.

5.3.3 Expert Evaluation

We contacted one core contributor from each of the four open source projects we studied to ask them to evaluate our results and comment on their potential usefulness. These contributors had not participated to our qualitative study (Chapter 2). A core contributor of Joda Time positively replied to our invitation and reviewed the addition and deletion recommendations. We asked several questions that aimed at answering these three research questions:

1. Would the contributor have followed the recommendations? In other words, are the recommendations correct and pertinent?
2. Do the recommendations match the contributor's intent? For example, does the inferred documentation pattern make sense? Does addressing a deletion or deprecation recommendation is appropriate in all the reported sections?
3. What is the cost of a false positive? Is it easy and quick for the contributor to recognize a false positive?

We sent to the Joda Time contributor a list of addition and deletion recommendations. The addition recommendations presented the inferred documentation pattern and one of the alternative pattern, the coverage difference, the new code elements to document, and the location where the code elements should be added (more than one location could be displayed if our algorithm returned multiple locations). Figure 5.1 shows an example of the addition recommendations that we provided to the contributor.

The deletion and deprecation recommendations that we provided presented the code elements that had been deprecated and the locations of these code elements in the documentation.

Correctness and Usefulness.

Out of the five addition recommendations that we made for the three releases of Joda Time, the contributor judged that one was correct, two were partially correct, and the last two were false positives, which match our own evaluation. For the two partially correct recommendations, (1) the contributor judged that the pattern was too inclusive (all descendants of `BaseChronology`), but that some of the pattern elements needed to be documented, and (2) the token inferred by the pattern was the wrong one, but the code elements had to be documented (the pattern was "All methods declared in `DateTime` ending with the token `year`", but the right token according to the contributor was `with`).

For the 21 deletion and deprecation recommendations that we suggested, the contributor judged that they were all correct. The contributor noted that two of these recommendations identified old documentation bugs that still needed to be fixed in the current release of Joda Time. The contributor thought that the third documentation bug we identified for Joda Time (see Table 5.4) was technically an issue, but that it did not need to be fixed because the sentence about the deprecated element was still true.

Recommendation #2

Pattern: All descendants of BaseChronology

Change: Coverage dropped by 13%: 6 classes were covered (out of 12).
Now there are 17 classes.

New classes to document in 1.4:
org.joda.time.chrono.BasicGJChronology
org.joda.time.chrono.IslamicChronology
org.joda.time.chrono.BasicFixedMonthChronology
org.joda.time.chrono.BasicChronology
org.joda.time.chrono.EthiopicChronology

Where to document:
Each class should be documented in its own page.

Similar pattern: All descendants of AssembledChronology (16 classes, 5 new)

1. If you were about to release a new version of Joda Time, would you follow this recommendation and document most of the suggested code elements?

2. Does the documentation pattern matches your documentation intent? If not, does the similar documentation pattern matches your documentation intent?

3. If this recommendation is incorrect, how much time would it take you to dismiss it? In other words, what is the cost of this false positive?

Figure 5.1: Example of an Addition recommendation sent to the Joda Time Contributor

Documentation Intent.

The contributor found that the three correct documentation patterns that we identified in the addition recommendations partially matched the documentation intent.

The main issue with the addition recommendations was that most patterns were associated with more than one location and each location had a different intent: the contributor did not think that a single pattern should be reported for different locations. For example, the pattern "All descendants of `ReadablePeriod`" was matched to the two following pages and sections: Period/Using Periods in Joda Time and User Guide/Periods. The Period page presents the period concept in details and is appropriate for this pattern. The user guide is a general overview of all the features in Joda Time and the contributor thought that the guide was already long and was not the appropriate place to discuss all the descendants of `ReadablePeriod` (only a manually selected subset were mentioned).

We understand that a documentation pattern has a terse definition compared to the richer documentation intent of a documentation maintainer. Nonetheless, in the light of the contributor's evaluation and our own evaluation, we believe that the inferred documentation patterns were useful as a first step in automatically analyzing the documentation.

Cost of False Positives.

The contributor instantaneously identified the false positives in one partially correct and one incorrect recommendation because the false positives were related to internal classes. For example the last recommendation suggested to recommend the class `BaseLocal`, which was an internal class.

For the other false positives, the contributor had to quickly read the related documentation sections, which took less than five minutes for a partially correct recommendation and less than a minute for an incorrect recommendation.

We consider that the cost of the false positives is acceptable, given the low number of recommendations and the time it takes to read them and discard the false positives.

Additional Observations.

Overall, because all the correct code elements in the addition recommendations had been documented at the time of the release, the contributor did not think that this type of recommendation would have been particularly useful for the Joda Time project. The contributor also explained that he had been fortunate that somebody initially wrote good documentation that had stood the test of time and that the project had not evolved much. We believe that projects with more modifications at each release would likely benefit more from these recommendations.

The deletion and deprecation recommendations would have clearly been useful to the contributor because the next release will include a fix for the two documentation bugs that we identified. The contributor noted though that he would not use our recommendation tools as intended because the release process is already complex: such recommendations would have to be integrated to existing tools.

The Joda Time contributor told us that a recommendation system that could identify the common problem areas mentioned in the mailing list could be useful. This is another motivation for the future work needed to improve this type of recommendation (see the end of Section 4.2.1).

5.4 Discussion

We showed in this chapter how we could use high-level documentation structures and low-level links to produce documentation improvement recommendations when the underlying codebase evolves.

8 981 code elements were introduced in the codebase between the releases we studied, but only 90 of these code elements were mentioned in the documentation release following the code release. The 46 documentation patterns we inferred recommended to document 45 of the 90 code elements mentioned in the documentation. Eight of the code elements not covered by our recommendations were part of new documentation patterns while the 37 others were added in isolation and did not seem to be related to any kind of pattern.

It is clear from these numbers that the documentation does not change much compared to the underlying code base and that larger projects are more likely to benefit from these recommendations than smaller ones that have little new code elements and documentation pages to consider.

Our addition recommendations achieved our goal to detect new code elements that were part of existing documentation patterns, but as we found in the evaluation, there will always be code elements that are documented for other reasons that may not lend themselves to be automatically recommended. Moreover, we need to improve these recommendations by putting them into context. For example, some pages and sections are more focussed than others and are more appropriate locations for addition recommendations (e.g., the Period page vs. the User Guide page in Joda Time).

Out of the 114 deletion and deprecation recommendations we made, 103 were correct and we only missed one reference to a deprecated code element. Additionally, our recommendations found 31 references to deprecated code elements that were still not corrected in the current documentation releases of the four open source projects. When we compared our recommendations with those from a textual search tool (grep), the precision of our system was clearly superior: we produced 11 false positives against 107 by the textual search tool.

We could complement our deletion and deprecation recommendations with additional recommendations from change detection tools such as SemDiff [24]. For example, when we identify the location of a deprecated code element, we could recommend to replace this reference with the replacement element identified by SemDiff.

Adaptive changes are only one type of recommendations that can improve the quality of documentation. As we found out in our qualitative study (Chapter 2) and confirmed in this chapter, recommending adaptive changes can be useful in identifying documentation issues, but other strategies are required as well to cover the full spectrum of potential documentation improvements. For example, other recommendations based on what we know about developers needs and learning theory (e.g., presence of examples, task-oriented) would identify other types of issues and would require a different approach to evaluation.

Threats to Validity.

To evaluate the precision of our recommendations, we analyzed the evolution of the documentation. Because we used historical data, we can only speculate on why the code elements we recommended were referenced or modified by documentation maintainers and we cannot assess how the documentation maintainers would have used our recommendations. To mitigate this threat, one core contributor of an open source project we studied reviewed our results and confirmed most of our observations.

We evaluated the recall of our recommendations by computing a list of links to code elements that were added between each documentation release and by performing a textual search (grep) to find references to deprecated code elements. The former metric is dependant on the precision of RecoDoc and the latter may miss references with typos (e.g., a deprecated class name starting with a lower case). Given the high precision and recall of RecoDoc and the low number of typos in the documentation of these four projects, we are confident that these were not significant limitation to our evaluation of recall.

As it was the case with our previous evaluation studies, the external validity is limited by the documentation standards and practices of the systems we studied. Documentation without regular documentation patterns or systems that do not deprecate or remove code elements between releases would not benefit from our recommendations.

Chapter 6
Related Work

Most of the related work on developer documentation has focused on studying how developers use documentation and on devising techniques to document programs.

How Developers Learn Frameworks and Libraries.

Carroll et al. observed users reading documentation and found that the step-by-step progress induced by traditional documentation such as detailed tutorials and reference manuals was often interrupted by periods of self-initiated problem solving by users [11]. Indeed, users ignored steps and entire sections that did not seem related to real tasks, and they often made mistakes during their unsupervised exploration. Because this active way of learning was not what the designer of traditional documentation intended, Carroll et al. designed a new type of documentation, the minimal manual, that is task-oriented and that helps the users resolve errors [11, 12, 53, 58].

Shull et al., compared the effectiveness of example applications with hierarchy-based documentation [55]. 43 participants used one of these two types of documentation to learn the ET++ framework, which supports developers in building graphical applications in C++. The Hierarchy-based documentation presented the concepts and the classes from abstract to concrete. The Example-based documentation was created by assembling the example applications that came with the ET++ framework. The investigators found that all the participants who had been taught with the hierarchy-based documentation abandoned it after a few weeks because it took

too much time to start writing the program. All participants ended up using example applications.

Kirk et al. conducted three case studies to study the problems encountered by software developers when using a framework [40]. They identified general kinds of questions such as finding out what features are provided by the framework and understanding how classes communicate together in the presence of inversion of control and subtle dependencies. The authors observed that different types of documentation provided answers to a subset of the questions.

Robillard conducted a survey and qualitative interviews in a study of how Microsoft developers learn APIs [51]. The study identified obstacles to API learnability in documentation such as the lack of code examples and the absence of task-oriented documentation. Forward and Lethbridge conducted a survey with developers and managers, and asked questions regarding the use and the characteristics or various software documents [31]. According to the participants, the following properties of software documentation were the most important: content (information in the document), recency, availability, use of examples, and organization (sections, subsections, index).

In another survey, De Souza et al. found that that the two most important documents for developers were the source code and the comments it contained [16]. Documents presenting the data model of the product were also very important.

Nykaza et al. performed a needs assessment on the desired and required content of the documentation of a framework developed by a software organization [49]. The authors observed that junior programmers with deep knowledge of the domain and senior programmers with no knowledge of the domain had similar documentation needs about the framework. The programmers preferred simple code examples that they could copy and execute right away (as opposed to complex examples showing many features at once) and a manual that had self-contained sections so users could refer to it during their exploration (as opposed to manual that must be read from start to finish).

We can conclude from these studies that traditional documentation that is comprehensive and systematic and that focuses on the general concepts is less effective than

concise documentation that provides small and simple examples, concrete details, short task-oriented procedures, and that has an organization supporting exploration. Interestingly, no single type of documentation can resolve all the issues encountered by application developers when using a framework, and the documentation needs of developers with various backgrounds and roles seem highly similar.

We complement these studies by investigating the decisions made by the producers of documentation and by identifying the effort required by these decisions and their impact on the project and the users.

API Documentation.

Magyar described an early attempt to maintain the links between API documentation and code [43]. The tool alerted documentation writers when the documentation of a function was no longer representative of the code (e.g., a parameter was added) and could update the documentation (e.g., by adding a parameter). Nowadays, these functionalities are provided by standard documentation tools such as Javadoc and Doxygen.

In parallel to the development of API documentation, Meyer suggested the use of design by contract with the Eiffel object-oriented programming language [45]. The idea was to document the acceptable states throughout all the lifecycle of the class (invariants), and to document each method with the expected state of the program before invoking the method (preconditions) and after invoking the method (postconditions). The documentation of framework based on contracts can be very precise. In practice though, postconditions can be difficult to fully express [33], so there are research projects such as Spec# that try to balance completeness of specifications and usability [9].

Mining Code Examples.

Many documentation techniques rely on mining code examples to infer usage information about libraries and frameworks. For example, SpotWeb mines code examples found on the web to recommend framework hotspots, i.e., classes and methods that are frequently reused [56]. MAPO mines open source repositories and indexes API usage patterns, i.e., sequence of method calls that are frequently invoked together [62].

Then, MAPO recommends code snippets that implement these patterns, based on the programming context of the user.

Schäfer et al. used a clustering technique to recover the main building blocks of a framework from client programs to build a representation of the framework that is easy to understand by users [54]. Similar classes are grouped together to help users understand the framework.

Augmenting Existing Documentation.

XFinder is a tool that matches the steps of a tutorial to the code elements that implement each step [21]. For example, a tutorial might describe how to implement a text editor using the Eclipse platform: implement interface `x` and call method `y`. Given a codebase implementing several text editors, XFinder will find all the text editors and will map the code elements implementing each editor to the steps of the tutorial. As opposed to RecoDoc which accepts plain HTML and plain text, XFinder expects the tutorial to be encoded in a specific format that identifies the kind of the step and the type artifacts.

Dekel and Herbsleb devised eMoose, a tool that enables framework developers to annotate the API documentation of a framework to highlight "directives" such as preconditions [28]. When a developer writes code that calls a method with an annotated directive, eMoose highlights the method call in the code editor. Contrary to our technique, the links between the documentation and the API must be encoded manually by the framework developers.

We believe that tools can be useful to complement the documentation, but they cannot replace human-written documentation. As we observed in our qualitative study, some documents are used for marketing purposes so they cannot be generated, and writing documentation introduces a feedback loop that is beneficial for the program's usability.

Information Retrieval.

Antoniol et al. applied two information retrieval techniques, the probabilistic model and the vector space model, to find the pages in a reference manual that were related to a class in a target system [7]. The authors found that the accuracy

of both approaches was limited. Information retrieval techniques work best when the entities to be linked can be expressed by several words. Information retrieval technique are thus usually used to link coarse-grained artifacts like entire documents and classes [13, 26, 36] whereas our technique attempts to link single code-like terms to fine-grained code elements.

Hipikat is a tool that generates a project memory from a set of coarse-grained artifacts: bug reports, support messages, source code commits, and documents [18]. The tool stores and indexes the artifacts and then determines whether the artifacts are related. Hipikat uses several strategies to recover the links between the artifacts: presence of bug numbers, textual similarity (using a vector space model), similarity of support messages title, etc. The tool enables developers to query the project memory by returning a set of artifacts related to the query.

Identifying Code Snippets.

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

Island Grammars is a general technique that enables the identification of structured constructs such as code elements in arbitrary content (e.g., an email message) [48]. The main idea is to separate the content into small recognizable constructs of interest (islands) and everything else (water). Our parser implements a similar approach by first identifying the code snippets (big islands) and then, by identifying the smaller code elements (small islands) in the English paragraphs (water).

Bacchelli et al. compared various techniques to identify code elements and code snippets in email messages and found that lightweight techniques based on regular expressions performed better than information retrieval techniques such as latent semantic indexing and the vector space model [8]. We implemented our documentation and support channel parsers with regular expressions based on the observations of this study.

Inferring Intensions.

The addition recommendations that we generate is based on a commonly-used strategy: from a set of discrete elements, an approach tries to infer a structural pattern and reports violations of this pattern.

Examples of such approaches include LSdiff, a tool devised by Kim and Notkin that analyzes change sets to infer structural differences as logic rules [39]. The goal of LSdiff is to produce a logical summary that is easier to understand for software engineers than a textual difference (such as the one produced by the GNU diff tool) or a list of changed code elements. Once a logic rule is inferred, LSdiff can report all violations of this rule. For instance, LSdiff could detect that in a changeset, methods starting with the token "delete" were replaced by methods starting with the token "remove": all methods starting with "delete" that were not renamed would be reported as an error.

ISIS4J automatically infers a set of intensions from a set of code elements manually selected by a software developer (i.e., a concern's extension) [19]. As the underlying software system evolves, ISIS4J uses the inferred intensions to augment the concern's extension with relevant code elements. The intensions supported by ISIS4J are similar to the ones inferred by our documentation analysis tool chain: all descendants of a type, all members declared by a type, etc. As opposed to ISIS4J, we compute intensions based on tokens, but we do not support yet intensions based on callers and accessors.

Natural Language Processing.

Natural Language Processing (NLP) and information extraction techniques frequently rely on the context of a term or the distance between two terms to extract relevant relationships [47]. The presence of a term in the context of another term is called a *discourse feature*. As opposed to our technique, users of general NLP techniques typically need to train the techniques on a corpus first to develop a reliable classifier for a specialized task.

Hill et al. built a technique that links query terms to a set of matching methods in a codebase [32]. The NLP-based technique analyzes the methods and their parameters

by tokenizing their identifiers and determining their part-of-speech (POS) tags to compute multiple propositional phrases (e.g., `addItem(BookItem)` becomes "add item" and "add book"). Our technique could potentially try to match sentence fragments in the learning resources with these propositional phrases.

Chapter 7
Conclusions

Reusing libraries and frameworks is a complex task that requires intimate knowledge about the various features offered by a framework. As frameworks grow in size and complexity, the need for concise but comprehensive documentation increases as well.

The motivation for the work described in this dissertation is to help documentation maintainers and users by making improvement recommendations based on the links between the code base and the learning resources of a software development project. To recover these implicit links, we propose to analyze the context in which a code element was mentioned in a learning resource.

The thesis of this dissertation is that by analyzing how the relationships between documentation, code, and users' needs are created and maintained, we can identify documentation improvements and automatically recommend some of these improvements to documentation contributors.

We begin our investigation by conducting a qualitative study on the documentation decisions made by open source contributors and by identifying the main factors motivating these decisions and their consequences on software development projects. At the end of this study, we identify several recurring documentation problems that could be automatically supported by a recommendation system.

To build this recommendation system, we create a documentation meta-model that represents all the relevant learning resources in a project and their relationships. We build a parsing infrastructure that automatically generates a documentation model from the artifacts of a project. Then, we devise a technique that links the various model elements together by relying on the context in which an element is mentioned.

We extend our documentation meta-model by representing high-level documentation structures, i.e., relationships between high-level concepts such as messages and documentation sections, and structures such as documentation patterns.

Finally, we build a recommendation system that uses the inferred links to recommend adaptive changes to the documentation when the underlying codebase evolves.

7.1 Future Work

We discussed potential extensions of our work in the previous sections and we identified several ideas of documentation tools during our interviews with documentation users and maintainers. We briefly discuss four new directions involving our documentation analysis tool chain. The first two directions address documentation maintainers' needs, while the last two directions are related to documentation users.

Quality Metrics.

One request that was constantly made by open source contributors was the creation of a documentation tool that could compute quality metrics on the documentation. Because our documentation analysis tool chain generates fine-grained models and links, it would be possible to compute and report many kinds of metrics.

For example, contributor C11 in our qualitative study mentioned that a tool that would point out the documentation sections that lack code examples or that have a low code-to-word ratio would be useful.

We could also attempt to measure if a manual is task-oriented, a strong indicator of quality according to Carroll et al. [11]. For example, if we assume that a code example is a representation of a common task in a framework, we could estimate

the minimal number of documentation sections that are required to understand the code example (finding these related sections is trivial given the links recovered by our tool chain). Our hypothesis is that the closer the documentation sections are to each other, the more "task-oriented" the manual is.

Interface for other Recommendation Tools.

As we briefly mentioned in Section 5.4, other recommendation tools can complement our own recommendations. For example, change analysis techniques recommend how to replace deleted or deprecated code elements [24]. These recommendations could complement our own recommendations that find references to deleted and deprecated code elements in the documentation.

Conversely, recommendations from other tools could seed new documentation improvement recommendations. For instance, POPCON computes the popularity of API elements [34] and popular elements should probably be documented. Our tool chain could cross-check the list of popular API elements with the list of code elements mentioned in the documentation and report undocumented popular elements.

Better Integration with Integrated Development Environment.

Integrated Development Environments (IDE) usually provide close integration between the code and the API documentation. For example, Eclipse displays the Javadoc in a tooltip when the cursor moves over a method call.

Other researchers have attempted in the past to provide more information based on the context of the current development task. For example, Strathcona analyzes the context of the current method body the developer is writing to recommend code examples that share a similar structure (e.g., a code example that calls the same methods and inherits the same type as the current code) [35].

We could use the strategies we developed to infer high-level documentation structures (Chapter 4) to provide similar recommendations. For instance, we could present to the developer all the documentation sections sharing at least N code elements with the code being edited. Alternatively, if a developer is using an element present in a documentation pattern, we could present to the developer the sections covering this documentation pattern.

Automated Hyperlinks Generation.

Finally, we could improve existing documentation and support channels by generating hyperlinks in the sections and support messages. Because our tool chain can precisely identify and link code elements mentioned in a sentence or in a code snippet, we could generate a hyperlink to the API documentation for each code element.

Readers of the documentation or support messages would then be able to quickly jump to the reference documentation of any referenced code element. Most documentation tools such as DocBook and Sphinx allow documentation writers to generate such link, but the writer must use a special markup and must provide the fully qualified name of the code element. Because this is a tedious task, usage of hyperlinks is inconsistent in the documentation and a hyperlink generator would improve the quality of the documentation while reducing the documentation effort.

7.2 Contributions

The research described in this dissertation makes five contributions to the software engineering research field.

First, we provide a grounded model of documentation decisions, motivation factors, and consequences that resulted from a qualitative study with open source contributors and developers. This is the first collection of observations on the documentation process in open source projects that we are aware of: as noted by the contributors we interviewed, practitioners can rely on these observations to review and improve their documentation practices and researchers can use these observations to guide their effort in the construction of better documentation tools.

Second, we provide a fine-grained documentation meta-model that describes the main elements composing learning resources and their relationships with the codebase of a project. This meta-model was validated on the artifacts of four open source projects and we relied on the relationships of the meta-model to produce documentation recommendations. The meta-model can be used and extended by other researchers to analyze documentation.

Third, we describe a documentation analysis tool chain that automatically generates documentation models from a project's artifacts and that links the elements of these models. This tool chain enables the automated analysis of documentation by researchers and the development of recommendation systems.

Fourth, we describe two strategies to infer high-level documentation structures such as documentation patterns. These structures serve as an example of the kind of abstract information that can be inferred from the low-level links recovered by our tool chain and they form the basis of one kind of recommendation.

Fifth, we provide a recommendation system that recommends adaptive changes to the documentation when the underlying codebase evolves. Documentation maintainers can directly benefit from the recommendations because they identify documentation inconsistencies.

Finally, as one contributor said in our qualitative study, the main effort in the documentation process is to continuously look for opportunities to improve the documentation. Like integrated development environments that are a collection of tools, we believe that documentation systems will eventually consist of inter-related tools addressing a variety of documentation needs and issues. Our documentation analysis tool chain is a first but sound step toward the development of many kinds of documentation tools and the automated analysis of documentation from various angles.

Bibliography

[1] Hibernate Reference Manual. `http://docs.jboss.org/hibernate/core/3.6/reference/en-US/html/`. Accessed 31-Aug-2011.

[2] HttpClient Tutorial. `http://hc.apache.org/httpcomponents-client-ga/tutorial/html/index.html`. Accessed 31-Aug-2011.

[3] Joda Time User Guide. `http://joda-time.sourceforge.net/userguide.html`. Accessed 31-Aug-2011.

[4] Spring Framework Reference Manual. `http://static.springsource.org/spring/docs/3.1.x/spring-framework-reference/html/`. Accessed 31-Aug-2011.

[5] Mithun Acharya, Tao Xie, Jian Pei, and Jun Xu. Mining API patterns as partial orders from source code: from usage scenarios to specifications. In *ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 25–34, 2007.

[6] Steve Adolph, Wendy Hall, and Philippe Kruchten. A methodological leg to stand on: lessons learned using grounded theory to study software development. In *Proceedings of the Conference of the Center for Advanced Studies on Collaborative Research*, pages 166–178, 2008.

[7] Giuliano Antoniol, Gerardo Canfora, Gerardo Casazza, Andrea De Lucia, and Ettore Merlo. Recovering traceability links between code and documentation. *IEEE Transactions of Software Engineering*, 28(10):970–983, 2002.

[8] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 375–384, 2010.

[9] Mike Barnett, Robert Deline, Manuel Fähndrich, Bart Jacobs, K. Rustan Leino, Wolfram Schulte, and Herman Venter. Verified software: Theories, tools, experiments. chapter The Spec# Programming System: Challenges and Directions, pages 144–152. Springer-Verlag, 2008.

[10] Greg Butler, Peter Grogono, and Ferhat Khendek. A reuse case perspective on documenting frameworks. In *Proceedings of the IEEE Asia Pacific Software Engineering Conference*, pages 94–101, 1998.

[11] John M. Carroll, Penny L. Smith-Kerker, James R. Ford, and Sandra A. Mazur-Rimetz. The minimal manual. *Journal of Human-Computer Interaction*, 3(2):123–153. Erlbaum Associates, 1987.

[12] Ian Chai. *Framework Documentation: How to document object-oriented frameworks. An empirical study.* PhD in Computer Sscience, University of Illinois at Urbana-Champaign, 2000.

[13] Xiaofan Chen. Extraction and visualization of traceability relationships between documents and source code. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 505–510, 2010.

[14] Paul Clements, David Garlan, Len Bass, Judith Stafford, Robert Nord, James Ivers, and Reed Little. *Documenting Software Architectures: Views and Beyond.* Pearson Education, 2002.

[15] Juliet Corbin and Anselm C. Strauss. *Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. Sage Publications, 3rd edition, 2007.

[16] Sergio Cozzetti, B. de Souza, Nicolas Anquetil, and Káthia M. de Oliveira. A study of the documentation essential to software maintenance. In *Proceedings of the ACM SIGDOC International Conference on Design of Communication*, pages 68–75, 2005.

[17] John W. Creswell. *Qualitative Inquiry and Research Design*. Sage Publications, 2nd edition, 2007.

[18] Davor Cubranic, Gail C. Murphy, Janice Singer, and Kellogg S. Booth. Hipikat: A Project Memory for Software Development. *IEEE Transactions on Software Engineering*, 31(6):446–465, 2005.

[19] Barthélémy Dagenais, Silvia Breu, Frédéric Weigand Warr, and Martin P. Robillard. Inferring structural patterns for concern traceability in evolving software. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 254–263, 2007.

[20] Barthélémy Dagenais and Laurie Hendren. Enabling Static Analysis for Partial Java Programs. In *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, pages 313–328, 2008.

[21] Barthélémy Dagenais and Harold Ossher. Automatically locating framework extension examples. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 203–213, 2008.

[22] Barthélémy Dagenais, Harold Ossher, Rachel K.E. Bellamy, Martin P. Robillard, and Jaqueline P. de Vries. Moving into a New Software Project Landscape. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 275–284, 2010.

[23] Barthélémy Dagenais and Martin P. Robillard. Creating and Evolving Developer Documentation: Understanding the Decisions of Open Source Contributors. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 127–136, 2010.

[24] Barthélémy Dagenais and Martin P. Robillard. Recommending adaptive changes for framework evolution. *ACM Transactions on Software Engineering and Methodology*, 20(4):19:1–19:35, 2011.

[25] Barthélémy Dagenais and Martin P. Robillard. Recovering Traceability Links between an API and its Learning Resources. In *To appear in Proceedings of the IEEE/ACM International Conference on Software Engineering*, 2012.

[26] Andrea De Lucia, Rocco Oliveto, and Genoveffa Tortora. Adams re-trace: traceability link recovery via latent semantic indexing. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 839–842, 2008.

[27] Cleidson R. B. de Souza and David F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 241–250, 2008.

[28] Uri Dekel and James D. Herbsleb. Improving API Documentation Usability with Knowledge Pushing. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, pages 320–330, 2009.

[29] Michael D. Ernst, Jake Cockrell, William G. Griswold, and David Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, 2001.

[30] Mohamed Fayad and Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[31] Andrew Forward and Timothy C. Lethbridge. The relevance of software documentation, tools and technologies: a survey. In *Proceedings of the ACM Symposium on Document Engineering*, pages 26–33, 2002.

[32] Emily Hill, Lori Pollock, and K. Vijay-Shanker. Automatically capturing source code context of nl-queries for software maintenance and reuse. In *Proceedings of the IEEE/ACM International Conference on Software Engineering*, pages 232–242, 2009.

[33] Daniel Hoffman and Paul Strooper. API documentation with executable examples. *Journal of Systems and Software*, 66(2):143–156.

[34] Reid Holmes and Robert J. Walker. Informing Eclipse API production and consumption. In *Proceedings of the OOPSLA workshop on eclipse technology eXchange*, pages 70–74, 2007.

[35] Reid Holmes, Robert J. Walker, and Gail C. Murphy. Approximate structural context matching: An approach to recommend relevant examples. *IEEE Transactions of Software Engineering*, 32(12):952–970, 2006.

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

[37] IEEE Society. IEEE Recommended Practice for Software Design Descriptions. IEEE Std 1016-1998, 1998.

[38] Ralph E. Johnson. Documenting frameworks using patterns. In *Proceedings of the ACM SIGPLAN Conference on Object-oriented programming systems, languages, and applications*, pages 63–76, 1992.

[39] Miryung Kim and David Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering*, pages 309–319, 2009.

[40] Douglas Kirk, Marc Roper, and Murray Wood. Identifying and addressing problems in object-oriented framework reuse. *Journal of Empirical Software Engineering*, 12(3):243–274, 2007.

[41] Malcolm S. Knowles and Richard A. Swanson Elwood F. Holton III. *The Adult Learner*. Elsevier, 6th edition, 2005.

[42] Douglas Kramer. API documentation from source code comments: A case study of Javadoc. In *Proceedings of the conference of the ACM Special Interest Group for Design of Communication*, pages 147–153, 1999.

[43] Miki Magyar. Automating software documentation: a case study. In *Proceedings of the ACM SIGDOC International Conference on Computer Documentation*, pages 549–558, 2000.

[44] Martin Fowler. Inversion of Control Containers and the Dependency Injection Pattern. `http://www.martinfowler.com/articles/injection.html`. Accessed 31-Aug-2011.

[45] Bertrand Meyer. Applying "design by contract". *IEEE Computer*, 25(10):40–51, 1992.

[46] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the ACM/IEEE International Conference on Software Engineering*, pages 167–176, 2000.

[47] Marie-France Moens. *Information Extraction: Algorithms and Prospects in a Retrieval Context*. Springer, 2006.

[48] Leon Moonen. Generating robust parsers using island grammars. In *Proceedings of the Working Conference on Reverse Engineering*, pages 13–22, 2001.

[49] Janet Nykaza, Rhonda Messinger, Fran Boehme, Cherie L. Norman, Matthew Mace, and Manuel Gordon. What programmers really want: results of a needs assessment for sdk documentation. In *Proceedings of the ACM SIGDOC International Conference on Computer Documentation*, pages 133–141, 2002.

[50] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[51] Martin P. Robillard and Robert DeLine. A Field Study of API Learning Obstacles. *Journal of Empirical Software Engineering*, 16(6):703–732, 2011.

[52] Martin P. Robillard and Gail C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):1–38, February 2007.

[53] Mary Beth Rosson, John M. Carrol, and Rachel K.E. Bellamy. Smalltalk scaffolding: a case study of minimalist instruction. In *Proceedings of the ACM SIGCHI Conference on Human Factors in Computing Systems*, pages 423–430, 1990.

[54] Thorsten Schäfer, Ivica Aracic, Matthias Merz, Mira Mezini, and Klaus Ostermann. Clustering for generating framework top-level views. In *Proceedings of the Working Conference on Reverse Engineering*, pages 239–248, 2007.

[55] Forrest Shull, Filippo Lanubile, and Victor R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE Transactions of Software Engineering*, 26(11):1101–1118, 2000.

[56] Suresh Thummalapenta and Tao Xie. SpotWeb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336, 2008.

[57] Christoph Treude and Margaret-Anne Storey. Effective communication of software development knowledge through community portals. In *Proceedings of the ACM SIGSOFT Symposium and the European conference on Foundations of Software Engineering*, pages 91–101, 2011.

[58] Hans van der Meij. A critical assessment of the minimalist approach to documentation. In *Proceedings of the ACM SIGDOC International Conference on Systems Documentation*, pages 7–17, 1992.

[59] Jukka Viljamaa. Reverse engineering framework reuse interfaces. In *Proceedings of the ACM SIGSOFT symposium and the European conference on Foundations of Software Engineering*, pages 217–226, 2003.

[60] Zhenchang Xing and Eleni Stroulia. Umldiff: an algorithm for object-oriented design differencing. In *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, pages 54–65, 2005.

[61] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: a tool for using software libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.

[62] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and recommending API usage patterns. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 318–343, 2009.

Glossary

code element is a logical unit in a programming language such as a package, a module, a class, a method, a field, etc. A code element can declare another code element and it can also inherit from another code element. 2

code pattern is a set of code elements that share at least one common structural or syntactical property, e.g., all classes that extend a particular interface. When a code pattern describes a subset of the elements of another code pattern, it is said to be redundant. 69

code snippet is a small region of source code that can be further divided into a list of code-like terms. 46

code-like term is a word or a sequence of tokens that looks like a code element such as a class, a method, a field, or an XML tag. Such terms are identified by regular expressions detecting common naming conventions such as CamelCase or underscores. 11

conceptual documentation explains how a concept relate to the usage of a framework or a library. Concepts can come from the problem domain (e.g., Gregorian calendar), or from the solution domain (e.g., session-level caching). 27

context is the set of all code-like terms, and by extension, all code elements, that are mentioned in the vicinity of a term. 47

documentation infrastructure is the set of tools used by documentation writers to organize, format, and publish the documentation. 28

documentation meta-model represents a decomposition of the code, the documentation, and the support channels of a project into elements and their possible relationships. An instance of the documentation meta-model for a particular project is called a documentation model. 67

documentation pattern is a code pattern or a set of code patterns whose code elements are present in the documentation. To qualify as a documentation pattern, 50% of the elements of a code pattern must be mentioned by the documentation. A documentation pattern can represent multiple redundant code patterns. 67

embarrassment-driven development is the process in which developers improve the quality of their code to avoid being embarrassed. The embarrassment can come from a bad demonstration (e.g., the product does not work properly) or from the documentation (e.g., if it takes too many complicated steps to perform a simple task). 34

extension is an enumerated set of code elements. For example, the set containing classes `A`, `B`, `C` is an extension. 68

getting started documentation explains how to perform a task or a series of related tasks using a library or a framework. 27

global context contains all the terms mentioned in the same documentation page or support thread of another term. 47

helpful The relationship between a support message and a documentation is helpful if reading the documentation section could have helped the user answer the question asked in the support message. Alternatively, the relationship can be helpful if the question asked in the support message could have helped the documentation writer identify content that needs to be added or clarified. 84

immediate context contains all the terms in the same term list. For example the terms `b` and `c` are part of the immediate context of `a` in `a.b.c`. 47

intension is a set of common properties between code elements. For example, the set of all classes that extend a particular interface and that start with a particular token is an intension. 68

linking code elements is the process of determining which code element declaration a code-like term is referring to. 42

local context contains all the terms mentioned in the same documentation section or support message of another term. 47

meaningful A documentation pattern is meaningful if it is the focus of a documentation section or support message or if a sentence in the section or message describes the intension of the documentation pattern. 76

reference document systematically covers the logical units (properties, members, types, etc.) of a project artifact. 24

strict filtering is a set of heuristics used to determine whether a code-like term that could potentially refer to a code element is in fact a false positive. Strict filtering heuristics are based on the assumption that a member is unlikely to be mentioned without its declaring type in its context. 56

term list is a list of consecutive code-like terms belonging to the same logical unit. For example a fully qualified name is represented by a term list (parts of the package and the name of the type) and a call chain is also represented by a term list (the target of the first method, the consecutive methods being called, and their parameters). 46

Appendix A
Results of the Historical Analysis

We classified each documentation revision by associating a category summarizing the rationale behind the change. When there were multiple types of change, we identified the change that had caused the largest number of lines in the document to be modified. Ten categories of change emerged from our analysis:

Clarification. Addition of a note or the modification of words to clarify existing content.

Adaptation. Modification of the text to reflect the new state of the project. An adaptive change can range from the update of copyright date to the recommendation of a new feature over and old one.

Addition. Text or examples that are added to a document. For example, when a new feature is released, a section describing the feature is often added in a reference manual.

Structure. When sections are moved inside or outside documents, e.g., when a large document is split in multiple smaller documents.

Format. Modifications of the file syntax, e.g., the addition of an HTML closing tag that had been forgotten in the previous revision.

Links. Addition of a URL to the documentation.

Correction. Modification of a code example because it was broken or the behavior was not the one intended. Because it was not always possible to determine if

Code	Django	WP	Plasma	Hib.	Spring	GTK	Firefox	DBI	Shoes	Eclipse	Avg.
Clarification	**18.2**	**7.1**	**13.1**	6.9	**14.1**	1.9	**9.5**	**16.6**	**5.9**	1.1	**9.4**
Adaptation	**15.8**	**8.7**	**24.6**	**17.2**	**8.8**	**33.3**	7.3	**32.4**	0	**18.5**	**16.7**
Addition	**16.7**	**19**	**18.0**	**24.1**	**20.9**	**9.3**	7.9	**27.6**	**64.7**	**5.4**	**21.4**
Structure	3.2	6.3	3.3	**15.5**	4.4	1.9	1.3	2.07	**8.8**	**6.5**	5.3
Format	**11.4**	6.7	3.3	3.5	6.6	**11.1**	3.2	1.4	**5.9**	**55.4**	**10.8**
Links	6.5	**27.7**	6.6	3.5	1.9	5.6	**14.2**	**4.8**	0	1.1	7.2
Correction	7.6	2.4	**11.5**	**15.5**	**10.6**	**13**	3.8	3.5	**5.9**	0.0	7.4
Polish	**20.5**	**17.4**	**13.1**	**13.8**	**32.8**	**18.5**	**11.7**	**11.7**	**5.9**	**10.9**	**15.6**
SPAM	0.0	2.4	1.6	0.0	0.0	0.0	**24.1**	0.0	0.0	0.0	2.8
Revert	0.0	2.4	4.9	0.0	0.0	5.6	**17.1**	0.0	2.9	1.1	3.4

Table A.1: Classification of document revisions (in %). Top-5 codes for each document are in italic.

a correction was due to refactoring, the modifications of a code example following a refactoring are included in this category.

Polish. Words or sentences that are copy edited, e.g., spelling error. When new sentences or domain-specific words were added to clarify an existing sentence, we considered the change to be part of the clarification category.

SPAM. Unsolicited advertisement or vandalism.

Revert. When the current version of the document is reverted to a previous version. This is often caused by SPAM, but incorrect or unclear addition by contributors can also cause a revert.

Table A.1 shows the distribution of the change categories across the document revisions for each project. The five most popular categories in each project are in bold. The last column, Avg., presents an unweighted average of each category across the 10 projects: because we did not analyze the same number of documents and revisions for each project, a weighted average would be heavily biased toward the documents with the most revisions. We found that the top five category in weighted and unweighted averages were the same.

Documentation Tools and Infrastructures	
DocBook	`www.docbook.org`
CPAN	`www.cpan.org`
POD	`perldoc.perl.org/perlpod.html`
Sphinx	`sphinx.pocoo.org`
Javadoc	`java.sun.com/j2se/javadoc`
Doxygen	`www.doxygen.org`
Maven	`maven.apache.org`
Projects	
Django	`www.djangoproject.com`
WordPress	`wordpress.org`
KDE Plasma	`plasma.kde.org`
Hibernate	`www.hibernate.org`
Spring	`www.springsource.org`
GTK+	`www.gtk.org`
Firefox	`www.mozilla.com/firefox`
DBI	`dbi.perl.org`
Shoes	`github.com/shoes/shoes`
Eclipse	`www.eclipse.org`
Rails	`rubyonrails.org`
Hibernate	`hibernate.org`
HttpComponents	`hc.apache.org`
Joda Time	`joda-time.sourceforge.net`
XStream	`xstream.codehaus.org`

Table A.2: Documentation tools and open source projects mentioned in this dissertation

Appendix B
Parsing Infrastructure

The parsing infrastructure of RecoDoc is responsible for analyzing the artifacts of a project and generating a model (see Figure 3.2). For each project, we create a set of parser extensions that take into account the unique characteristics of a project. These extensions are usually small because the parsing infrastructure already contains most of the parsing logic. For example, the extension responsible for parsing the HttpClient tutorial has only 17 lines of code.

Parsing Code. The code parser takes as input an Eclipse Java project, an XML schema file (.xsd), or a DTD file and generates a list of corresponding code elements and their relationships. For example, the parser generates a code element for each method in a Java project, and it associates each method with its declaring type through the *declare* relationship. New programming languages or configuration file formats can be supported by adding new parsers.

Parsing Documentation. The documentation parser takes as input the URL of the table of contents of a document and a set of URL prefixes. The parser first *crawls* the table of content and transitively downloads all pages, if the pages' URL matches one of the prefixes. Then, the parser uses a set of XPath expressions to identify the various parts of a page (e.g., title, sections, code snippets, emphasized code-like terms). This process is often referred to as *screen scraping*. The set of XPath expressions are provided by a parser extension. Once the various parts of

each page have been identified, the parser generates a corresponding documentation model.

Parsing Support Channels. The support channel parser takes as input the URL of the first page of a forum or the table of contents of a mailing list archive. Like the documentation parser, the support channel parser crawls the support channel to download all pages, uses a set of XPath expressions to identify the various parts of each page (e.g., thread title, message author, etc.) and generates the corresponding support channel model.

Classifying Content. The documentation and support channel parsers are also responsible for identifying the code snippets and the code-like terms. When markup is available (e.g., documentation and forum pages), the parser relies on a list of markup elements provided by the parser extension (e.g., `<div class=‘‘code’’>`) to classify the content. Otherwise, the parser first divides the textual content into paragraphs and then tries to classify the paragraphs as being either a Java Snippet, a Java Exception Trace, an XML snippet, or an English paragraph. The classification is performed by searching for hints (e.g., presence of curly braces and semicolons at the end of a line, presence of XML-like tags, etc.). Paragraphs of the same nature are then merged together. English paragraphs are further analyzed to identify code-like terms: the parser applies a set of regular expressions on the paragraphs to identify terms that look like classes, methods, fields, and XML tags. Even in the presence of markup, English paragraphs are analyzed to identify code-like terms that are not surrounded by markups.

Parser Accuracy. As explained in Section 3.1.1, the content classification step favors recall (number of missed code-like terms) over precision (number of false positives) because our linker only considers code-like terms identified by the parser. To evaluate the accuracy of the content classification step, we randomly selected 20 sections and 20 messages (called *units*) from the three target systems we used for the linker evaluation, for a total of n=3*(20+20)=120.

We inspected each unit and identified code snippets and the code-like terms that referred to Java and XML code elements. We then executed RecoDoc on these units and

System	Found	Real	Prec.	Recall
Joda Doc.	86	50	57%	98%
HC. Doc.	78	45	55%	96%
Hib. Doc.	227	164	72%	100%
Joda Channel	157	68	40%	93%
HC. Channel	75	25	33%	100%
Hib. Channel	140	50	35%	98%
Total	763	402	52%	99%

Table B.1: Parser Accuracy

compared the RecoDoc classification with our manual inspection. Table B.1 shows the results of this comparison: the *Found* column indicates the number of code-like terms found by RecoDoc in the English paragraphs, the *Real* column indicates the number of terms we identified in our inspection, and the last two columns give the recall and the precision. As we expected, the average recall of the parser was very high (99%) at the expense of precision (52%). The low precision is mainly due to the fact that all terms referring to technologies (e.g., JTA) and other programming languages (e.g., SELECT FROM...) look like Java code elements. Although the precision seems low, the parser only selected 763 words from a total of 30794 words (the length of the 120 units), so the aggressive selection weeded out many false positives.

Regarding the classification of paragraphs, RecoDoc found a total of 52 snippets and exception traces and achieved a 100% recall and precision.

As we demonstrated in Section 3.3.2, the low precision of the parser did not hinder the accuracy of the linker.

Appendix C
Research Ethics Board Approval of Qualitative Studies

McGill University Research Ethics Board Approval of User Studies REB FILE #: 102-1009 – Framework Documentation, From Creation to Dissemination (see end of the thesis).

Appendix D
Interview Guide for the First and Last Open Source Contributors

---

The following list contains the main questions that were part of our first interview guide with an open source contributor. Because we performed semi-structured interview, most questions led to follow-up questions.

1. Can you describe your role in [PROJECT]? Can you describe your role in the creation and maintenance of the documentation of [PROJECT]?
2. How are documentation tasks distributed in the team?
3. Can you walk me through the main steps you performed when you created the tutorial [TUTORIAL]?
4. How did the documentation evolve? Which part of [PROJECT] was documented first?
5. When you started to work on the [PROJECT] documentation, how did you decide what part of the project to document?
6. How did you end up with the current format of the documentation [DESCRIBE THE FORMAT]?
7. Can you give me examples of feedback you received from users on the documentation of [PROJECT]?

8. When you create or maintain the documentation of [PROJECT], what are the main challenges or problems you encounter, if any?

The following list contains the main questions that were part of the last interview guide with an open source contributor. As part of our methodology (grounded theory), our research questions and interview questions evolved as the study progressed.

1. Can you tell me more about your development experience and how you got involved in open source projects?
2. How is the documentation created and maintained in [SUBPROJECT]? Is the process different than the other [SUBPROJECTS]?
3. How do you manage the contributions from the community?
4. When you work on the documentation of [PROJECT], what kind of documentation do you create first? What does come next?
5. Typically, what is your personal workflow when you work on documentation?
6. How or where do you receive feedback about the documentation? Is there any particular feedback that struck you over these years?
7. What is the impact of the documentation on the various aspects of your project?
8. Do you think that your documentation supports both newcomers and experienced users? How?
9. How do you see the role of alternative documentation like tutorials found on blogs?
10. What parts of the documentation take the most effort to create?
11. According to you, does documentation have an impact on the quality of the code? How?

Appendix E
Evaluation Questionnaire for the Qualitative Study on Developer Documentation

---

We sent this questionnaire to the open source collaborators who contributed to our qualitative study. We provided a summary reproduced on the next page.

1. Are the three documentation production modes an appropriate classification of your documentation effort?
2. Is there anything that does not fit in these three categories?
3. Are the main decisions you took while documenting your open source project represented in this summary?
4. Is there any important decision that is not represented?
5. Do the factors and consequences for each decision resonate with your experience? Can you give an example (or a counter-example) of one decision with its factors and consequences that you experienced?
6. Are there any major factors or consequences missing from the decisions?
7. (Optional) Do you have any other comments or questions about the summary?

Cost-Effectiveness Factors on Developer Documentation

Developer documentation of open source projects helps potential developers to select a technology and it teaches developers how to use the technology. To better understand the effort involved in documenting open source projects and to find cost-effective documentation strategies, we interviewed 12 open source contributors and 10 developers using open source projects. We also inspected more than 1500 document revisions from 10 open source projects.

We observed the decisions that developers take in three documentation production modes: initial documentation effort, incremental changes, and burst changes. For each decision, we identified the factors motivating the decision and their consequences on the project.

1. Initial Effort

When a project starts, there are many decisions to make with respect to the documentation process.

1.1 Documentation Infrastructure

When starting to work on the documentation, contributors[1] must select the tools they will use to create and maintain the documentation and the infrastructure they will use to build and publish the documentation.

Wiki

Contributors select wikis to create the project documentation when they strongly believe in crowdsourcing or when the programming language of their project does not have a standardized documentation infrastructure.

Advantages of wiki:

- Easy to get started with (setup, configuration, syntax).
- Promote community building.

Disadvantages of wiki:

- Costly to maintain (curating a popular wiki is a full-time job).
- Less authoritative than documents hosted on other types of infrastructure (users don't know what is official, what is current).

General Documents (e.g., HTML, Plain Text, Office Suites)

Contributors choose to use general document formats when they do not want to invest time to learn a documentation suite or when there is no such suite associated to the programming language of their project.

Advantages of general documents:

- Almost no learning curve.

Disadvantages of general documents:

- No support for programming languages (e.g., syntax highlighting, code formatting).
- Limited support for document structure (e.g., generation of table of contents and back links) and multiple output formats.

Documentation Suites (e.g., POD, Sphinx, DocBook, kernel-doc)

Contributors choose to write documents using documentation suites when there is an official or popular documentation suite associated to the main programming language of their project.

Advantages of documentation suites:

- Support programming languages (e.g., syntax highlighting).
- Suggested structure (e.g., POD files or Maven web sites have a similar structure).

Disadvantages of documentation suites:

- Steeper learning curve than previous infrastructures.
- High maintenance effort if the documentation suite is heavily customized.

1.2 Type of documentation

Contributors need to decide what type of documentation to create when documenting a project: they often create one type of documentation initially and the documentation covers only a subset of the code. Then, as the project evolves, more documents are created and different documentation types are mixed in the same document.

---

1 Members of a project who have commit privileges.

Getting Started

This type of documentation describes *how to use* a particular feature or a set of related features. It is particularly developed in projects where contributors believe that the documentation is part of the marketing of the project.

Effort:

- Finding a good example (interesting, not too general or too specific) is difficult.
- Getting started documents require more writing skills than other types of documentation.

Functions – Getting started documentation:

- Generate interest, and influence the selection of the project.
- Provide an overview of the features (scope) and of the quality of the code (API usability).

Reference Documentation

This type of documentation is composed of the systematic documentation of the API, the properties, the options, and the syntax used by a project.

Effort:

- Easy to write this type of documentation because it is very systematic.
- Reference documentation is well supported by documentation suites (e.g., Javadoc).

Functions – Reference documentation:

- Needed for advanced usage (not all users will look at the source code).
- Can be sufficient for libraries with atomic operations but not for frameworks.

Conceptual Documentation

This type of documentation describes the underlying concepts of a project and the rationale behind certain design decisions. It is created to ensure that users understand the project and to prevent users from making mistakes.

Effort:

- Difficult to stay focused and concise: this type of documentation can become confusing for the users (is this a walkthrough, is this a reference, should I bother reading this?).

Functions – Conceptual documentation:

- Useful to understand the cause of an error.
- Should be created when other types of documentation have been created: users want to know how to get moving, not why they move.

2. Incremental Changes

Open source contributors must decide how to adapt the documentation as the project evolves and how to manage the project community's contributions.

2.1 Documenting Software Evolution

One decision point in the lifecycle of a project is how changes (e.g., new feature, refactoring, redesign) are documented. Contributors rely on several strategies that they can combine.

Specification-Based Documentation

Contributors write the documentation before writing the code when they want to use the documentation and the code examples as a specification.

Advantages of spec-based documentation:

- Ensure that the code is well covered by the documentation.

Disadvantages of spec-based documentation:

- If the documentation is published too soon, the implemented behaviour might be different than the desired behaviour.
- The produced API might not adequately respond to user needs.

Change-Based Documentation

Contributors write the documentation while or just after they make a change.

Advantages of change-based documentation:

- Ensure that documentation coverage is always on par with the code.
- Lead to a form a “embarrassment-driven development”: while writing documentation, developers identify design issues with their project and perfect them.

Release-Based Documentation

Contributors write the documentation of new features just before a release: this practice may be driven by a strict release policy.

Advantages of release-based documentation:

- Ensures that no feature is released undocumented.

Disadvantages of release-based documentation:

- Comprehensive policies are necessary to ensure that the documentation does not become generic (e.g., writing content-free sentences to beat the documentation metric).

Documentation Team

Contributors rely on a documentation team to document most or all of the changes.

Advantages of a documentation team:

- The technical writing and the style may be of higher quality.
- Possible to create more in-depth tutorial.

Disadvantages of documentation teams:

- There are more code contributors than documentation contributors: new features are frequently released without documentation.
- Developers do not notice usability/terminology issues themselves and may argue with the documentation team recommendations.

No Documentation

Contributors may barely document the changes when they perceive documentation as a low-value activity (e.g., compared to bug fixing or answering questions) and lack motivation or confidence.

Advantages of not documenting changes:

- Exceptionally, an experimental feature without documentation forces users to read and code and understand the caveats before using the feature.

Disadvantages of not documenting changes:

- Some users can learn a few undocumented features, but they eventually become frustrated when there are too many of these features.

2.2 Managing Community

Project contributors need to decide how to manage the contributions from the community: code patches, questions, comments, and documentation patches.

Barrier to Entry

Contributors establish different kinds of barrier to documentation contributions:

- Wikis offer the lowest barrier to entry: anybody can contribute to the documentation.
- Comments at the bottom of documentation pages allow users to contribute quickly without modifying the official documentation.
- Documentation teams make contributing to the project easier: documentation and code patches are processed by different contributors and there is no need to be a good programmer to obtain commit privileges.

Trade-offs:

- Low barriers favour numerous contributions of questionable quality.
- Higher barriers favour fewer contributions of higher quality.
- The challenge lies in finding a balance between encouraging contributions and preserving authoritativeness.

Code Contribution

Contributors can establish a policy that code contributions must be accompanied by tests and documentation.

Advantages:

- Ensure that code contributors do not outnumber documentation contributors.
- Give confidence that the contribution is well thought-out.

Disadvantages:

- For smaller or less popular projects, this policy could discourage new contributions.

External Documentation

Users sometimes contribute documentation on their own blog. Contributors can manage these contributions in a variety of ways such as placing a list of links on a web page dedicated to the community or allowing certain sections of the documentation to be written by users (e.g., user pages).

Advantages of external documentation:

- Form of evangelism.
- Can document specialized case.

Disadvantages:

- Not updated and it does not always promote the best practices.
- Divide the documentation effort: these documents could be inlined within the official documentation.

Questions on Support Channel

Contributors consider questions asked on support channels to be equivalent to a bug report on the documentation.

Consequences of considering questions to be bug reports:

- The support effort is reduced as the documentation gets clearer.
- Documentation efforts are targeted toward the sections of the documentation that need the most attention according to the users.
- The challenge lies in constantly looking for opportunities to improve the documentation.

3. Bursts

During a project lifetime, the documentation goes through single, major changes that we call bursts. These changes improve the quality of the documentation, but they require such effort that they are not done regularly.

3.1 Books

Publishers may approach contributors of open source projects to write books about their projects.

Advantages of writing books:

- Writing a book encourages contributors to be more precise and to reflect on their design decisions.
- Books are the place to provide complementary code examples and explanations.

Disadvantages of writing books:

- Depending on the license of the book, the large effort required to write a book might not be translated to the official documentation.
- Books are rarely updated.

3.2 Documentation Infrastructure Change

Contributors change the documentation infrastructure when it becomes too costly to maintain.

Advantages of infrastructure change:

- The new infrastructure generally increases the barrier to entry and makes the documentation more authoritative.

Disadvantages of infrastructure change:

- Old documentation must be ported to a new format.

3.3 Major Documentation Review

Contributors initiate major reviews where they systematically review each page of the documentation to ensure that the content is accurate, still relevant, and that it fits the generally style guidelines.

Advantages of major reviews:

- The organization of the content is better: users find what they need more quickly.
- Some parts of the documentation are completely rewritten.
- The quality of the documentation improves.

Disadvantages of major reviews:

- Very time-consuming: this may explain why despite the wish to make these efforts on a regular basis, few projects perform major reviews regularly.