Using Traceability Links to Recommend Adaptive Changes for Documentation Evolution

Barthélémy Dagenais and Martin P. Robillard

Abstract—Developer documentation helps developers learn frameworks and libraries, yet developing and maintaining accurate documentation requires considerable effort and resources. Contributors who work on developer documentation often need to manually track all changes in the code, determine which changes are significant enough to document, and then, adapt the documentation. We propose AdDoc, a technique that automatically discovers documentation patterns, i.e., coherent sets of code elements that are documented together, and that reports violations of these patterns as the code and the documentation evolves. We evaluated our approach in a retrospective analysis of four Java open source projects and found that at least 50% of all the changes in the documentation were related to existing documentation patterns. Our technique allows contributors to quickly adapt existing documentation, so that they can focus their documentation effort on the new features.

Index Terms—Documentation, Maintainability, Frameworks

1 Introduction

Developers usually rely on libraries or application frameworks[1] when building applications. Frameworks provide standardized and tested solutions to recurring design problems.

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 [1]. Various types of documents are available to help developers learn about frameworks, ranging from Application Programming Interface (API) reference documentation to tutorials and reference manuals.

In a previous study on the documentation process in open source projects, we observed that adapting the developer documentation to new releases of a project can become a significant challenge over time [2]. Members of documentation teams reported that developers released undocumented code and that the documentation team was always struggling to properly cover the new features and adapt the existing documentation. This led to frustration for both documentation team members and users who were told about the changes in the code without an easy way to learn about them. This previous study considered *developer documentation* to include all manually crafted documents and excluded automatically generated documentation and reference documentation extracted from source code (e.g., using the Javadoc tool). We use the same definition in this article.

• *B. Dagenais is CTO at Resulto Inc.*

• *M.P. Robillard is associate professor at McGill University.*

1. Unless otherwise specified, we use the term *framework* to refer to any reusable software artifacts including application frameworks, libraries, development toolkits, etc.

Ideally, documentation should be comprehensive: significant removals, changes, and additions of features should be represented in the documentation. Because there are usually no explicit traceability links between documentation and the code, open source contributors must rely on manual methods to ensure that significant changes are not forgotten.

The Hibernate framework[2] provides an example of clear divergence between an evolving framework and the corresponding reference manual. Between two major releases, developers of Hibernate added 5 661 new *code elements*, and deprecated 166 code elements (see Table 5). A **code element** is a publicly accessible software unit, as defined by the corresponding programming language. For example, in Java, a code element can be a package, a class, a method, or a field. When inspecting the Hibernate manual for these two releases, we found that only six of the 5 661 new code elements were documented. There were also 26 references to deprecated code elements, without any mention of the code elements that replaced the deprecated ones.

In previous work, we introduced RecoDoc, a technique that automatically identifies *code-like terms* in developer documentation. A **code-like term** is a word or a sequence of tokens that looks like a code element because it follows its naming convention or syntactic properties (e.g., CamelCase, parentheses, dot notation). RecoDoc links these code-like terms to code elements [3]. For example, RecoDoc can identify that the expression `save()` in a sentence refers to the method `org.hibernate.Session.save()`. In this article, we report on AdDoc[3], a recommendation system that leverages RecoDoc to *(1)* auto-

2. References to projects are presented in Table 9 at the end of the paper.

3. AdDoc is open source and available at http://cs.mcgill.ca/~swevo/recodoc

matically generate recommendations for recently-added framework elements that needs to be documented, and *(2)* identify references to deprecated or deleted code elements that need to be corrected. We implemented these new techniques in a recommendation system to facilitate their evaluation.

To recommend new code elements that should be documented, we propose to infer *documentation patterns*. A **documentation pattern** is a coherent set of code elements referenced by the documentation. For example, the section "Pessimistic locking" of the Hibernate 3.3.2 manual referred to most constants declared in the class `LockMode`. Knowing this documentation pattern is important because if a new constant in `LockMode` is added in the next release of Hibernate, we could recommend to document this new constant in section "Pessimistic locking", because it matches the documentation pattern.

To identify references to deprecated or deleted code elements, we propose to compare the traceability links recovered between two releases of a manual. For example, Section "Pessimistic locking" of the Hibernate 3.3.2 manual mentions the code-like term `lock()` which refers to the method `Session.lock()`. This method was deprecated in Hibernate 3.5, but the manual still refers to the code-like term `lock()` without mentioning a non-deprecated replacement. We could thus recommend to correct this code reference in the section on locking.

We evaluated our recommendation system in three steps. First, we reviewed the documentation patterns inferred on four Java open source projects and determined that they were representative of real patterns. Then we performed a retrospective analysis on eight consecutive versions of the documentation of four open source projects and we found that our recommendation system could identify 50% of new code elements that were documented in further documentation versions. Finally, the main contributor of one of the analyzed open source projects reviewed our results and confirmed their accuracy while providing insightful comments on their potential use.

The contributions of this article include *(1)* the concept of documentation patterns, a technique to infer documentation patterns, and detailed data on the accuracy of documentation patterns on four open source projects, *(2)* AdDoc, a technique to recommend documentation adaptations with a retrospective analysis on the documentation of these four open source projects to evaluate the accuracy of the recommendations, and *(3)* the analyzed results of a case study with one open source contributor who evaluated both the documentation patterns and the recommendations.

In the remainder of this article, we present a motivating example showing all the steps of our technique: finding traceability links, inferring documentation patterns, and producing recommendations (Section 2). We then describe our strategies and algorithms to infer documentation patterns and produce recommendations (Section 3). We present the evaluation of our recommendation system in Section 4 with a retrospective analysis of the documentation of four Java open source target systems and a case study with an open source contributor. We discuss the related work in Section 5 and conclude in Section 6.

2 General Approach

To illustrate how we can recommend documentation improvements when the code evolves, let us take the example of the Joda Time library. Joda Time is a Java library that provides utility classes and methods to manipulate dates and times. In addition to the API reference documentation, the Joda Time project includes a manual with a user guide covering the main features of the library, as well as a number of specific pages covering a few selected features in depth.

Between versions 1.0 and 1.4, the developers of Joda Time added more than 15 classes and 800 *members* to the code, and they deprecated 20 members (see Table 5 in Section 4.2). We consider that a **member** is a public or protected field or method. Before releasing the code, the developers updated the documentation: at that point, they had to determine *(1)* which new code elements to document, *(2)* where to document these elements, and *(3)* which documentation sections had to be corrected due to the deprecated elements.[4]

Some of the classes introduced in version 1.4 were part of a coherent set of code elements that were already documented in version 1.0. More precisely, the descendant classes of the interface `ReadablePeriod` were documented in the section "Using Periods in Joda-Time" on the page "Period" in version 1.0. In version 1.4, the developers of Joda Time created seven classes implementing `ReadablePeriod`. To keep the documentation consistent, the developers added a reference to these seven new classes in the appropriate section on the page "Period". When there are numerous new code elements added as part of a release, finding which elements should be documented to keep the documentation consistent can be a tedious task.

Because the developers deprecated 20 members, they also had to correct the documentation by removing reference to these members or by recommending alternatives. For example, the method `Chronology.getBuddhist` was deprecated, but it was referenced four times on two different pages ("User Guide" and "Buddhist calendar system"). The developers updated two of the four references to point to the new method `BuddhistChronology.getInstance()`, but the other two references were left uncorrected. In a large release, tracking which members were deleted and deprecated and whether they were mentioned in the documentation can be a time-consuming task.

4. Documenting before releasing major code changes is one of the three documentation workflows that we observed in our previous qualitative study [2].

We propose a recommendation system that can analyze the documentation and the code of a software project and suggest documentation improvements when new code elements are added, deleted, or deprecated. To produce these recommendations, we complete five distinct tasks in sequence.

1. Retrieving Project Artifacts. We first retrieve the documentation and the code from various releases of a software project. We parse the artifacts and produce an instance of a Documentation Meta-Model. For example, we downloaded the documentation and the code of JodaTime 1.0 and 1.4, and we parsed the documentation to produce a model of pages, sections and code snippets.

2. Inferring Traceability Links between Documentation and Code. We identify expressions that look like code elements in the documentation, and link them to code elements in a release. For example, the term `year()` in the documentation could be linked to the code element `DateTime.year()` in JodaTime 1.4.

3. Inferring Documentation Patterns. Using the fine-grained traceability links inferred in step 2, we find coherent sets of code elements documented in a particular release. For example, the section "Using Periods in Joda-Time" in JodaTime 1.0 mentions the descendant (children) types of the interface `ReadablePeriod`. We call these sets of coherent code elements, *documentation patterns*.

4. Producing Addition Recommendations. We produce addition recommendations by identifying documentation patterns that grow between two releases. For example, in JodaTime 1.4, the number of non-abstract `ReadablePeriod` descendants went from two to seven. Because section "Using Periods in Joda-Time" was referring to the `ReadablePeriod` descendants in JodaTime 1.0, we can recommend for this section to refer to the new descendants in JodaTime 1.4.

5. Producing Deletion Recommendations. Because we have a model of the code, we can also identify which code elements were deleted or deprecated between JodaTime 1.4 and 1.0. For example, we identified 20 deprecated members in the codebase of JodaTime 1.4, and we recommended to correct the 16 references to these deprecated members.

We perform the first two steps of our approach using RecoDoc, a documentation analysis tool that we presented in earlier work [3]. We implemented the last three steps in a new tool called AdDoc. We provide a brief summary of the first two steps in the remainder of this section and we present AdDoc in Section 3.

We devised both RecoDoc and AdDoc by studying the Spring Framework, a large and complex Java project. We wrote an initial prototype that analyzed various releases of the Spring Frameork and once we were satisfied with the results, we wrote the current version of RecoDoc and AdDoc and evaluated them on four target systems.

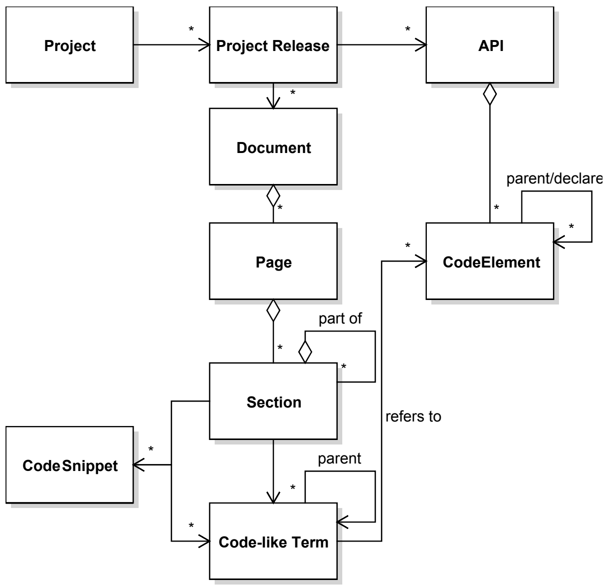

Fig. 1. Documentation Meta-Model. The cardinality of an association is one unless otherwise specified.

2.1 Collecting and Processing Artifacts

We previously proposed a technique called RecoDoc that automatically analyzes the documentation and the support channel archives of an open source project and that precisely links code-like terms such as `year()` to specific code elements in the API of the documented framework or library (e.g., `DateTime.year()`) [3]. RecoDoc 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.

RecoDoc takes as input the source code of a system and the URL of the documentation index such as the table of contents of a reference manual. RecoDoc then crawls the documentation to download the relevant HTML pages (i.e., documentation pages, emails). All documentation tools and archives we are aware of can produce an HTML output.

Once the artifacts are collected, RecoDoc builds an instance of a project artifacts meta-model (see Figure 1) for each release of the code and documentation of a project. We originally designed the meta-model based on our previous study on developer documentation [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).

Codebase. We consider that the *API* of a project consists of all the accessible *code elements* of a project (public class, method, field). 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 Java package declares a Java class and a Java class declares methods and fields).

Document. The documentation of a project consists of one or more *documents*. For example, the Http-

Components project has two main documents: the HTTPClient and HTTPCore tutorials. Each document has a list of *pages* and each page has a list of *sections*. A section may be part of a larger section and the section hierarchy depth is unbounded. RecoDoc considers a documentation page to be equivalent to an HTML page, but this rule can be adapted for long, single-page documents where chapters or main sections would be considered to be pages. We assume that documentation authors make a conscious decision when they divide their documents into pages and sections, i.e., the content from different sections of the same page is more related than the content from sections of different pages.

We process documents with an extensible parser infrastructure: each format has its own parser and if the HTML of a document was written by hand, we create an appropriate parser.

Code-like Terms and Code Snippets. Documentation sections can refer to *code-like terms* 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 methods, camel cases for classes). When multiple code-like terms are joined together, we keep that relationship. For example, in the expression `TimeUnit.SECONDS`, we say that `TimeUnit` is a parent of `SECONDS`. A **code snippet** is a small region of source code that can be further divided into a list of code-like terms. We use a specialized parser that infers missing types to parse code snippets into code-like terms [4].

2.2 Linking Documentation with Code

Given a project artifacts meta-model, the main challenge in linking code elements to code-like terms comes from the inherent ambiguity of unstructured natural language. For example, the user guide of the Joda Time library [5] 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 in Joda Time that declare a `year` method, and not all of them are in the same type hierarchy. In this 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, attempting to find the matching code element based on a textual search of the method name would fail. In fact, in the four open source projects we studied (see Section 4), we found that a text 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.

Our link recovery process takes as input a collection of code-like terms. RecoDoc associates each code-like term with a kind (i.e., package, class, method and field) determined by the syntactic hints of the term (e.g., presence of parentheses). The code-like term is also associated with the other terms present in its context (e.g., the terms attached to it, the terms in the same documentation section, and the terms attached to the same documentation page). The output of the link recovery operation is a ranked list of code elements that are potentially referred to by each code-like term. Because a code-like term may match multiple code elements (e.g., the term `year()` may match multiple methods named `year()`), Figure 1 shows a many-to-many relationships between code-like terms and code elements.

Given a collection of code-like terms, we perform the following steps:

Step 1. Linking Types. Given a code-like term, we find all object types (classes and interfaces) 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.

Step 2. Disambiguating Types. A code-like term that refers to a type may be ambiguous if multiple types share the same simple name. 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 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.

Step 3. 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. Most filters rely on the terms mentioned in the context of the code-like terms to filter out the code elements.

Step 4. Linking Misclassified Terms. Our parser may occasionally misclassify code-like terms. For example, the term `HTTP` found in a tutorial we studied (HttpClient) may be classified as a field (e.g., Java constants are written in uppercase). Although there is no such field in the codebase, there is a type with that name (`org.apache.http.protocol.HTTP`).

In this step, we process all terms that were not linked to any code element 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 previous steps: types, methods, fields. The linking technique used is the same as in the previous steps (e.g., simple name matching for types).

We implemented the artifacts collection and linking techniques in RecoDoc and applied it to 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).

2.3 Limitations

The adaptive maintenance of the documentation when a code element is deprecated and the addition of new code elements matching a pattern are only two examples of documentation modifications. As we found out in a previous 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 [2, Section 4].

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. More over, systematically reviewing code changes and documentation to detect inconsistencies is a tedious, effort-intensive, low-creativity task and as such it is a good target for additional automated support.

3 Recommending Documentation Improvements

We can suggest documentation additions and removals as a framework evolves by leveraging fined-grained traceability links recovered by RecoDoc and by inferring documentation patterns from these links.

We first describe the concept of documentation patterns, then we explain how we infer these patterns, how we compute documentation addition recommendations, and how we compute documentation removal recommendations.

3.1 Documentation Patterns

The documentation of a framework sometimes systematically *covers* the code elements by following 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 mentioned in a sentence or in a code snippet of the documentation.

We can think of a documentation pattern as a concern graph [6], which is a representation of program structures as a redundant extension (discrete set of code elements) and intension [5] (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.

5. We use the term "intension" in the sense of Eden and Kazman, to indicate a structure that can "range over an unbounded domain" [7, p.150]

For example, Section 13.3 of the Spring Framework 3.1 documentation 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`, `...`}.

To be able to capture documentation patterns in a variety of documents for different frameworks, the intensions must be sufficiently general. We observed three 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 as 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. We expect to find these intensions in other projects as well because they were observed in prior code evolution research [8], [9].

3.2 Inferring Documentation Patterns

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

Step 1. Computing Code Patterns. We compute all the possible *code patterns* in a codebase. A **code pattern** is a set of code elements structurally related by an intension. Intuitively, a documentation pattern is a code pattern whose elements are mentioned in the documentation.

For each code element `c`, we can compute code patterns generated from eight *intension templates*. An **intension template** is a parameterized intension formed by combining the three kinds of intensions we mentioned in Section 3.1:

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.

A package can declare 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 abstract classes are sometimes completely ignored by the documentation: in many cases 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 1).

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 2 shows an example of a codebase with five classes. If we assume that `c` is the abstract class `AbstractBean`, then the following code 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.

Step 2. 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 with 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 value in the unit interval. This value indicates the proportion of code elements in a pattern that are mentioned in the documentation. This step also outputs the sections and pages mentioning each code element in the pattern.

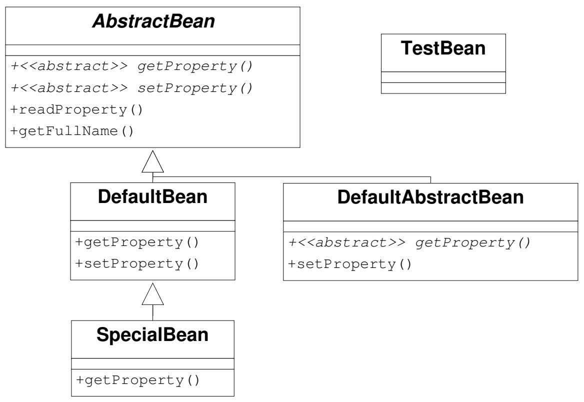

Fig. 2. Example of Code Elements

Step 3. 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. We require the majority of the code elements to be mentioned in the documentation.

Step 4. 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 exactly 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 0.40 (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: (A) "All classes extending `ApplicationContext`" and (B) "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. We present in Section 4.1 a sensitivity analysis demonstrating the linear impact that the threshold has on the number of combined patterns. Figure 3 shows this linear impact on all four systems.

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.

As part of the algorithm, once we have combined redundant code patterns, we select the *most representative* pattern within each group of redundant patterns, which

ALGORITHM 1: Combining Redundant Patterns

```
Input: List of code_patterns
Output: List of doc_patterns
patterns = sort_by_size(code_patterns, reverse=true);
doc_patterns = {};
processed = {};
for i in [0 .. patterns.size-1] 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-1] 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;
    doc_pattern.patterns = combined_patterns;
    add doc_pattern to doc_patterns;
end
return doc_patterns;
```

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.

Step 5. 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 are 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* mainly covers a documentation pattern if the coverage in the documentation unit is more than 0.75 (COVERAGE_THRESHOLD) of the coverage of the extension of the pattern. A **documentation unit** is a section or a page. 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, 0.875 (7/8), is superior to the threshold of 0.75. The relatively high threshold, 0.75, 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. We analyze the impact of this coverage threshold on the inferred locations in Section 4.1. Figure 4 illustrates the impact of the threshold on all four systems.

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
```

3.3 Recommending documentation additions

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. We assume that if the documentation pattern is relevant, the new elements fitting the pattern are associated with a feature worth documenting and omitting these new elements would make the documentation less comprehensive.

In a previous example (see Section 3.1), 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.

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 required its own documentation page 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.

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 3.2. 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 high coverage in the documentation, which is represented by one code pattern.

For the purpose of recommending documentation additions, we combine high-coverage code patterns into documentation patterns only after we have identified patterns for *both* releases of the documentation.

Step 1. Inferring Code Patterns.

We reuse the process presented in Section 3.2 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 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).

Step 2. 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. We match code elements in the intension of code patterns by using their fully qualified name.

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 eight 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.

In this example, we can match the code patterns in both releases because they have the same intension and they point to the same element. RecoDoc has linked the code-like term `DataSource` to the code element `javax.sql.DataSource` in both releases of the documentation.

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 3.4.

We also 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.

Finally, we do not attempt to match intensions whose code elements have been renamed or moved between two releases. If the code element of an intension is renamed, the corresponding code pattern is discarded, but other redundant patterns might still be matched across the two releases. For example, if the code element `javax.sql.DataSource` is renamed to `javax.sql.DataSource2` between two releases, we will not be able to match the code patterns we provided as an example. There might be though another intension that has not changed (e.g., all classes ending with the token Source) and that has a similar extension.

If the code elements in the extension of a code pattern are renamed, the coverage of the code pattern will decrease in the new release.

Step 3. 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 to where the new code elements should be mentioned. We use Algorithm 2 presented in Section 3.2 to find the location of the pattern.

Step 4. Combining Recommendations.

Finally, we combine redundant recommendations that are a subset of a larger recommendation in Algorithm 3. This process is similar to Algorithm 1 with one key difference: we select the recommendation with the most new code elements instead of selecting the recommendation with the highest initial coverage because we want the documentation maintainer to consider all potential code elements.

ALGORITHM 3: Combining Redundant Recommendations

```
Input: List of $redundant\_recommendations$
Output: List of $final\_recommendations$
$recommendations$ = sort_by_initial_coverage(
$redundant\_recommendations$, reverse=$true$);
$final\_recommendations = \{\}$;
$processed = \{\}$;
for $i$ in [0 .. $recommendations.size$-1] do
    $recommendation = recommendations[i]$;
    if $recommendation$ in $processed$ then
        continue;
    end
    add $recommendation$ to $processed$;
    $combined\_recommendations = \{pattern\}$
    for $j$ in [$i+1$ .. $recommendations.size$-1] do
        $trecommendation = recommendations[j]$;
        if $trecommendation \subseteq recommendation$ then
            add $trecommendation$ to
            $combined\_recommendations$;
            add $trecommendation$ to $processed$;
        end
    end
    $recommendation.sub\_recommendations$ =
    $combined\_recommendations$;
    add $recommendation$ to $final\_recommendations$;
end
return $final\_recommendations$;
```

3.4 Recommending documentation removals

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 as can be seen in Algorithm 4. We first recover the set of links between the codebase at release M and the documentation at release N. Then, for each code element that was deprecated or deleted in the codebase between release M and N, we produce a recommendation if the code element is part of a link.

4 Evaluating Documentation Improvement Recommendations

We evaluated the different steps of AdDoc on four open source projects: Joda Time, HttpComponents, Hibernate, and XStream. We selected the same four projects that we used to evaluate RecoDoc, the tool that recovers the fine-grained traceability links, because we demonstrated that RecoDoc had a high accuracy on these projects and

ALGORITHM 4: Recommendation documentation removals

```
Input: $version\_m$, $version\_n$, $code\_releases$, $doc\_releases$
Output: List of $recommendations$
$links\_n$ = get_links( $code\_releases[version\_m]$,
$doc\_releases[version\_n]$ );
$rem\_elements$ = get_removed_elements(
$code\_releases[version\_m]$, $code\_releases[version\_n]$ );
$recommendations = \{\}$;
for $element$ in $rem\_elements$ do
    for $link$ in $links\_n$ do
        if $element$ in $link$ then
            add $link$ to $recommendations$;
        end
    end
end
return $recommendations$;
```

therefore, it should not heavily influence the results of our recommendation strategies.

The four open source systems are written in Java and they vary in size, domain, and documentation style. Of the four target systems, only Joda Time can be considered as involving exclusively the Java programming language, i.e., the documentation only contains references to the Java API. The other systems include reference to other languages such as XML or SQL.

We first evaluate the accuracy of our documentation pattern inference strategy. Then we evaluate the accuracy of the documentation addition recommendations and the documentation removal additions. Finally, we evaluate the documentation patterns and the recommender system by asking a contributor from one of the target systems to review our recommendations.

4.1 Evaluation of Documentation Patterns Inference

We investigated whether the documentation of the four open source systems 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 titled "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 provide background context for the qualitative assessment of the relevance of the detected documentation patterns, we answered four research questions:

1) How many documentation patterns can AdDoc 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 sensitive is our approach to the thresholds we selected?

We then focused on the more crucial question:

5) How meaningful are the patterns? Are the elements accidentally covered or are the patterns the real focus of the documentation units they are in?

Question #1 - Generation of Patterns. We executed all the steps presented in Section 3.2 on a release of the four open source projets. Table 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 most representative high coverage code pattern has been selected, and *(4)* the average, standard deviation, and median of the coverage of the most representative pattern in each documentation pattern. For example, for the JodaTime documentation, AdDoc generated 3 120 code patterns, 103 of these patterns had a coverage higher than 50%, and after having combined the code patterns, AdDoc found 47 documentation patterns (1.5% of the code patterns). On average, the most representative pattern of the documentation patterns had a coverage of 81.9%.

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).

According to the coverage distribution, high coverage patterns exhibit a wide range of coverage, therefore accepting patterns with a coverage above a low threshold (50%) seemed necessary to yield enough patterns to study.

Question #2 - Pattern Intensions. Table 2 shows for each of the eight kinds of intensions (see Section 3.2) how many documentation patterns AdDoc detected in the documentation of the four target systems. For example, in HttpClient, AdDoc detected 32 documentation patterns that described code elements declared in another code element. The table only considers 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 all capture relations that actually exist in the documentation.

Question #3 - Relationship between Patterns and Sections. Table 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 Table 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. Again, only the most representative pattern of each documentation pattern was considered.

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

The documentation patterns were linked to different documentation units, which indicates that patterns can match topics at 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.

Question #4 - Sensitivity to thresholds. AdDoc used two threshold values that could not be analytically justified and that were derived from early experimentation with the approach on the Spring Framework (THRESHOLD=0.40 in Algorithm 1 and COVERAGE_-THRESHOLD=0.75 in Algorithm 2).

The first threshold, 0.40, was used to combine redundant code patterns. In summary, for two patterns extensions A_e and B_e where $|A| > |B|$, we consider

TABLE 1
Generation of Patterns

System	Gen. Patterns	High Coverage	Doc. Patterns	Average Coverage	Std. Dev. Coverage	Median Coverage
JodaTime 1.6	3 120	103 (3.3%)	47 (1.5%)	81.9%	19.2%	83.3%
HttpComponents 4.1	4 762	232 (4.9%)	139 (2.9%)	74.7%	20.0%	66.6%
Hibernate 3.5	17 619	149 (0.8%)	92 (0.5%)	77.0%	20.1%	75.0%
XStream 1.3.1	2 133	143 (6.7%)	64 (3.0%)	77.7%	20.5%	75.0%

TABLE 2
Types of Intensions

System	Decl.	Concrete Decl.	Child of.	Descendant 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 3
Patterns and Sections Linking

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

that the corresponding patterns A and B are sufficiently different and can be kept distinct if $1.0 - |B|/|A| > 0.4$. Analytically, we can determine that a threshold of 0 would result in many, highly similar, patterns and a threshold of 1.00 would result in fewer documentation patterns.

We computed the number of documentation patterns produced when using different thresholds. As Figure 3 shows, there is an almost linear decrease in the number of documentation patterns as the threshold increases. Because we want to avoid combining patterns that are not related, but we also want to avoid generating too many similar recommendations, a threshold of 0.40 represents a good compromise, but could be modified by the users of AdDoc.

The second threshold we used, 0.75, was used to determine the location of a documentation pattern (section, page, multi-page). For example, a threshold of 0.75 requires that more than 75% of the elements in a documentation pattern must be in a section to consider that this pattern is located in this particular section.

Analytically, we can determine that a threshold of 0 will result in all patterns being located in many sections, and a threshold of 1.00 will require all code elements of a pattern to be in a section.

We computed the location of the documentation patterns when using thresholds from 0.50 to 1.00. Figure 4 shows the number of patterns that were located in a section when using different thresholds. The number of sections quickly stabilize when the threshold is between 0.70 and 0.80. The number of documentation patterns located on multiple pages follows an inverse relationship.

Although the four target systems were unrelated, the two thresholds had a similar impact on the results and we expect these results to hold on other systems. The goal of this sensitivity analysis was not to find the best thresholds for all systems, because the exact impact may vary slightly, but to make sure that the thresholds did not have unexpected effect on a particular system and that their impact was clearly understood.

Question #5 - 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 important elements. We randomly selected 25 documentation patterns in each project and we manually inspected the sections and pages that covered their most representative pattern. We only inspected the most representative pattern because we consider all code patterns in a documentation pattern to be redundant: they cover the same subset of code elements and if we consider one pattern to be relevant, the other patterns should be relevant too. During our inspection, we assessed 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

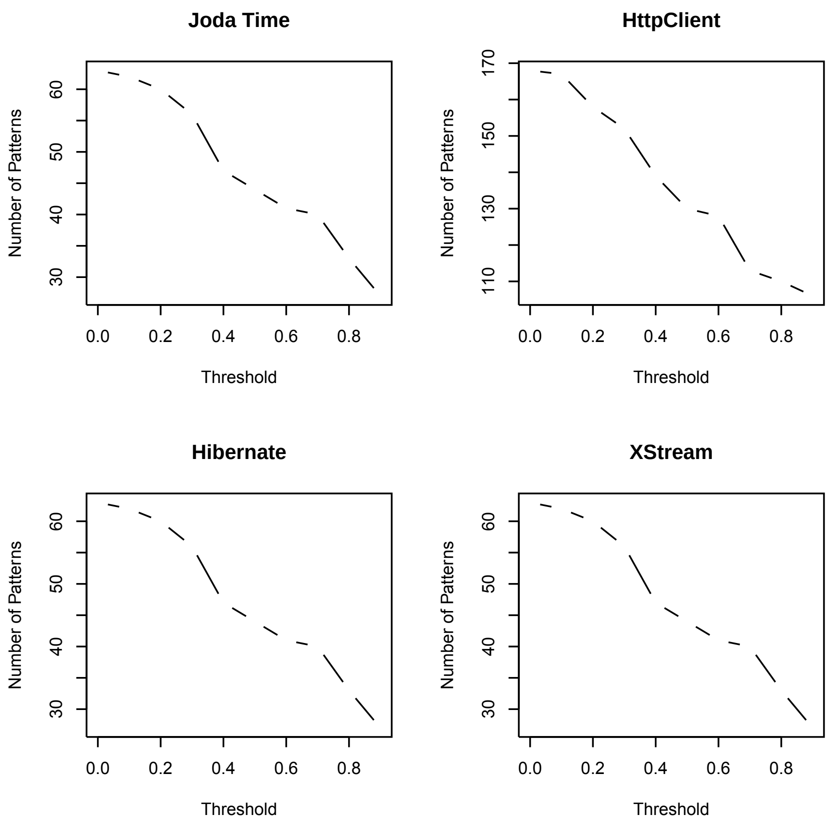

Fig. 3. Sensitivity analysis on the effect of the threshold on the number of patterns after merging.

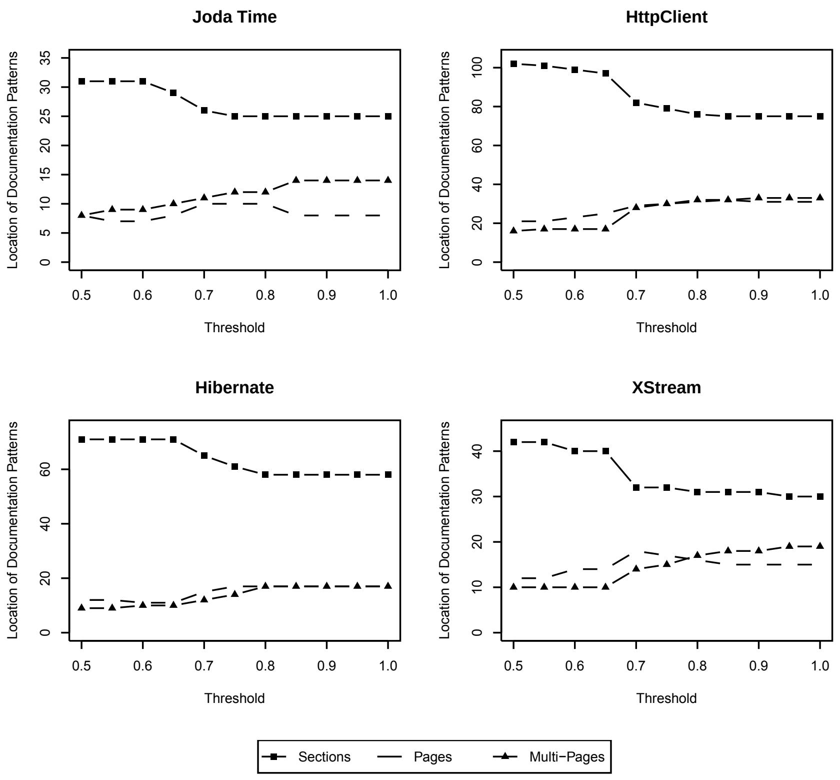

Fig. 4. Sensitivity analysis on the effect of the threshold on the localisation of documentation patterns.

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 shows for each document the number of sections, pages, and sets of pages that were covering the 25 patterns we randomly selected. Note that a pattern can be covered by more than one section (e.g., a tutorial section and a reference section can both discuss the code elements of a pattern), and a pattern can be covered in a set of pages (e.g., each subclass of a pattern can be discussed on its own page). The table then shows the categorization of the units. In the last column, the table shows the lowest coverage of all meaningful (shared or exclusive) patterns.

For example, in Joda Time, the 25 patterns that we randomly selected were discussed in 16 sections, 6 pages, and 7 sets of pages (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 documentation 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. Some of the meaningful patterns had a coverage of 50% which provide evidence that a higher threshold to classify high coverage patterns would have eliminated these meaningful patterns.

Eighty-two percent (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 (intensions) 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, AdDoc 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 Joda Time, AdDoc 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 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 3.2).

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 the XStream manual, AdDoc 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 contexts and they had different meaning: `CacheMode.IGNORE` is about query caching while `ReplicationMode.IGNORE` is about replicating data between databases.

4.2 Recommender Evaluation

To objectively estimate the usefulness and accuracy of our recommendations, we performed a retrospective

TABLE 4
Relevance of Documentation Patterns

System	Single Section	Single Page	Multi Page	Total	Mngfl. Excl.	Mngfl. Shared	Supportive	Accidental	Min. Coverage
Joda Doc.	16	6	7	29	6	20	1	2	50%
HC Doc.	11	7	7	25	6	13	1	5	53%
Hib. Doc.	21	8	2	31	4	22	0	5	50%
XSt. Doc.	23	4	5	32	5	20	4	3	50%
Total	48.2%	21.4%	30.4%	100.0%	17.9%	64.1%	5.2%	12.8%	-

analysis on the documentation of four open source projects. We computed recommendations for an old documentation release from each project and then 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 recommender, 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 all the minor releases (second digit of the release number) for which we could build the documentation. Although the meaning of major and minor releases vary between projects, we avoided major releases because they often introduce 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. We could still apply AdDoc to major releases if there is no significant changes in the structure of the documentation and the code.

Tables 5 and 6 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 change logs, because they provide information that is not integrated with the main documentation, and these pages do not need to be corrected between releases.

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 4.5 the contributor's evaluation on the correctness, the usefulness, and the cost of false positives.

4.3 Addition recommendations evaluation

AdDoc 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, AdDoc 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 AdDoc were accurate. We then manually inspected the documentation at version 1.4 to check if it mentioned the code elements that AdDoc 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 documentation releases (1.5, 1.6.2, and 2.0, the release on the web at the time of writing).

AdDoc 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 7 shows the results of our inspection. The first part of the table, "Precision", indicates how many recommendations were actually implemented in the new release of the documentation. Specifically, the column "Rec. Doc Patterns" shows the number of documentation

TABLE 5
Evolution of codebase

System	Release Old	Release New	Types Old	Types New	Members Old	Members New	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
HttpClient	4.0.1	4.1.1	512	618	3 276	4 066	33	68
Hibernate	3.3.2	3.5.5	1 327	2 124	12 860	17 724	40	126
XStream	1.0.2	1.1.3	117	192	439	1 069	1	23
XStream	1.1.3	1.2.2	192	273	1 069	1 558	7	55
XStream	1.2.2	1.3.1	273	309	1 558	1 779	20	98

TABLE 6
Evolution of documentation

System	Release Old	Release New	Pages Old	Pages New	Sections Old	Sections New	Links Old	Links New
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
HttpClient	4.0.1	4.1.1	8	9	84	100	1 099	1 302
Hibernate	3.3.2	3.5.5	29	30	320	338	1 788	1 879
XStream	1.0.2	1.1.3	12	17	55	80	69	124
XStream	1.1.3	1.2.2	17	25	80	146	124	511
XStream	1.2.2	1.3.1	25	24	146	203	511	659

TABLE 7
Evaluation of Documentation Patterns Recommendations.

	Precision				Recall			
System	Rec. Doc. Patterns	Patterns Correct	Code Elem.	Elem. 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
HttpClient 4.0.1-4.1.1	14	9	27	11	10	11	6	5
Hibernate* 3.3.2-3.5.5	13	8	52	14	0	5	0	1
XStream* 1.0.2-1.1.3	1	1	10	10	1	4	0	0
XStream* 1.1.3-1.2.2	6	5	32	13	13	12	8	2
XStream 1.2.2-1.3.1	7	3	19	9	9	6	8	0
Total	46	29	162	72	46	44	35	10

patterns that generated at least one recommendation for each release. The next column shows the number of documentation patterns for which at least one recommendation was implemented in the next release of the documentation. The column “Code Elem.” 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, “Recall” 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 that between releases 1.0 and 1.4 of Joda Time 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 (annotated with a star in Table 7), 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, AdDoc recommended to document the class `XMLArrayList`, but it was documented only in 1.3.1 instead of 1.2.2. In Table 7, we report that 9 recommended code elements were correct (precision over all future releases), but that we only recommended

8 of the 9 new types in release 1.2.2 (recall on the release 1.2.2 only). In total, we found 5 recommendations in future documentation releases of Hibernate and 2 recommendations in future releases of XStream.

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 AdDoc 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, AdDoc 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, AdDoc 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.

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

AdDoc correctly recommended methods associated with a token. For example, in Joda Time 1.0, AdDoc 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, AdDoc found nine documentation patterns whose extension in future releases were related to internal implementation. For example, in Hibernate 3.3.2, AdDoc 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. AdDoc 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 AdDoc that were spurious (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 inferred documentation patterns and their resulting recommendations made sense but were not implemented by the documentation maintainers. For example, in HttpClient, AdDoc 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 [10].

Examination of False Negatives. 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 AdDoc covered 50% (35 + 10 / 90) of these new types and members. In other words, 50% of the new types and members documented in the new release were not recommended by AdDoc (false negatives).

We manually inspected the documentation of these false negatives and we found that five types (out of 11 not recommended by AdDoc) and three methods (out of 34) were added together in a section. They were part of a new documentation pattern that did not exist in the previous release. The other types and members were added in isolation of each other and were not part of any documentation pattern we could find.

Because there are many factors influencing the documentation decisions of a contributor, AdDoc focused on ensuring that new code elements matching an existing documentation pattern would be recommended. We did not find evidence that we missed such a recommendation in the seven documentation releases we studied.

4.4 Removal recommendations evaluation

For each project release, AdDoc 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 conservatively considered that the documentation had addressed the deletion or deprecation of a code element if it: (1) referred to a new replacing element, (2) it mentioned that the element had been deprecated, or (3) it no longer mentioned the element.

We also compared AdDoc with a traditional textual search tool (grep). For each deprecated class, we performed a case-sensitive search of the class name (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 first search (without the opening parenthesis) yielded hundreds of false positives, we only kept the second search for comparison.

Finally, we performed a case insensitive search and we manually inspected the results to determine the number of references to deprecated code elements.

Table 8 shows the results of our inspection for each release. The "Deprecated Code Elem." column shows the number of deprecated elements for which AdDoc found at least one reference in the documentation. The "Found References" 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 precision (true positives divided by found references), and the recall computed from a manual inspection (true positives divided by references to deprecated elements). The numbers in parentheses represent the results for the textual search with grep.

Finally, at the end of the table, we show the number of references that had been corrected in one of the next documentation releases (Corrected Ref) and the number of references that had been left unchanged (Not Corrected)

For example, for HttpClient, we found that 13 deprecated elements were referenced in the documentation. AdDoc found 32 references to these 13 elements and the textual search found 38 references. Thirty-three of these references were pointing to a deprecated elements: AdDoc 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 incorrect reference was identified by both tools.

It is clear from these results that AdDoc 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 27 results.

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 AdDoc takes a few seconds.

4.5 Assessment by a Core Contributor

To provide a basic validation of the format and potential usefulness of the recommendations we compute with AdDoc, we contacted the primary author of each of the four projects involved in our evaluation to ask them to assess the recommendations. For our purpose, we estimate that only an expert with a complete and authoritative knowledge of the documentation would be suited to make reliable comments on the value of documentation adaptation recommendations. Specifically, peripheral developers without a deep tie to the documentation could not be expected to be aware of the challenges involved in its creation and maintenance.

Only the core contributor of Joda Time positively replied to our invitation and accepted to review 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?
2) Did the recommendations match the contributor's intent?
3) What is the cost of 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, 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). We also provided the intension that had ranked second as the most representative pattern of the documentation pattern: we wanted to elicit feedback about the accuracy of the intension and not just about the recommended elements. Because the contributor was not rewarded for his evaluation, we found that providing one alternative intension struck the balance between evaluation efficiency and completeness. Figure 5 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.

Because the contributor reviewed the raw results, the evaluation is conservative and provides a lower bound on the accuracy and usefulness of the recommendations.

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 (16+5) deletion and deprecation recommendations that we suggested (see Table 8), the contributor judged that they were all correct. The contributor noted that two recommendations identified old documentation

TABLE 8
Removed and Deprecated Elements Recommendations

System	Deprecated Code Elem.	References Found	True Positive	Precision	Recall	Corrected Ref.	Not Corrected
Joda 1.0-1.4	6	16 (16)	16 (16)	100% (100%)	100% (100%)	14	2
Joda 1.4-1.5	2	5 (5)	5 (5)	100% (100%)	100% (100%)	4	1
Joda 1.5-1.6.2	0	0	0	100% (100%)	100% (100%)	0	0
HttpClient	13	32 (38)	32 (33)	100% (87%)	97% (100%)	32	1
Hibernate	18	38 (123)	29 (29)	76% (24%)	100% (100%)	3	26
XStream 1.0.2-1.1.3	0	0	0	100% (100%)	100% (100%)	0	0
XStream 1.1.3-1.2.2	2	2 (5)	1 (1)	50% (20%)	100% (100%)	0	1
XStream 1.2.2-1.3.1	8	21 (24)	20 (19)	95% (79%)	100% (95%)	20	0
Total	49	114 (211)	103 (103)	90% (49%)	99% (99%)	72	31

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 Assembled-Chronology (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 match your documentation intent? If not, does the similar documentation pattern match 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?

Fig. 5. Example of an Addition recommendation sent to the Joda Time Contributor

bugs that still needed to be fixed in the current release of Joda Time. The contributor thought that the third documentation bug we identified was technically an issue, but that it did not need to be fixed because the sentence about the deprecated element was still true.

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). Nevertheless, if a new descendant of `ReadablePeriod` is created in a new release, we believe that the contributor would still consider referring to it in both sections because most of the descendants are already mentioned and the sections were adapted in previous releases. It is thus more likely that the problem with the addition recommendation was the disagreement on what "documentation intension" means.

Cost of False Positives. The contributor reported that he 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.

4.6 Discussion

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

A total of 8 981 code elements were introduced in the target 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 AdDoc 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 AdDoc made, 103 were correct and AdDoc 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 AdDoc was clearly superior: AdDoc 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 [11]. SemDiff analyzes the source history of a framework and recommends method replacements when a method is deleted or deprecated between two releases. 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 [2] and confirmed in this section, 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.

4.7 Threats to Validity

The target systems we evaluated were different than the one we used to devise RecoDoc and AdDoc. Specifically, we manually inspected the Spring Framework code and documentation and we wrote an initial prototype to link code-like terms to code elements and to make recommendations on a few of its releases. We then implemented and evaluated RecoDoc on four target systems and published a paper reporting the results [3]. Finally, we completed the implementation of the current version of AdDoc and evaluated it on the same four target systems as RecoDoc.

Choice of target systems. We selected the same target systems used to evaluate RecoDoc because their documentation format, application domain, size, and usage scenarios vary widely. These differences provide evidence that AdDoc is not constrained to a single type of framework.

The accuracy of the traceability links recovered by RecoDoc directly impacts the ability of AdDoc to infer documentation patterns: incorrect traceability links may result in false positives and missed traceability links may result in false negatives. Having access to a corpus of validated traceability links was essential to independently evaluate the accuracy of AdDoc. RecoDoc makes two assumptions about the documentation: (1) two code-like terms mentioned in close vicinity are more likely to be related than terms mentioned further apart, and (2) methods and fields are unlikely to be mentioned without their declaring type in their context. The second assumption made by RecoDoc matches the Java type system, but would not be directly applicable to other programming languages such as Go and JavaScript that use duck typing or prototypes. The population of systems to which our results are applicable is therefore limited to those built using statically-typed languages with a declarative structure like Java. Within this target population both of the assumptions above are general and can be expressed independently of any project-specific attributes; As such they impose no major constraint on the type of projects for which RecoDoc and AdDoc could function as expected.

Finally, we used the same eight generic intension templates across the four projects, but the intensions were different. For example, an intension template may be "All code elements declared by class `c`", but class `c` will be different for each project. The generalizability of our approach is thus not limited to specific intensions, but as we mentioned in Section 4.1, we cannot infer documentation patterns that do not match our eight intension templates.

Evaluation of Documentation Pattern Inference. Determining whether a section 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). Our assessment of each link is publicly available for inspection along with the artifacts we collected for each project.[6] Given the difficulty in recruiting open source contributors [2] and the time required to manually inspect these results, we believe that inspection of the results by the authors represented an appropriate trade-off. Additionally, given that the documentation was intended for users, there is no reason to believe that contributors would be more rigorous or systematic than researchers for this task.

The quantitative results (e.g., number of code patterns detected) are dependent 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.

In our evaluation of the documentation patterns, we did not study recall, i.e., the number of detected links out of the total number of links in a target system release. We wanted to focus our effort on the links we could uncover instead of the links we missed. Additionally, because of the large number of potential links, the cost of computing an oracle would have outweighed its value. In Section 4.3, we studied one aspect of recall by investigating documented changes in a codebase that were not captured by documentation patterns.

Recommender Evaluation. 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.

Similarly to the evaluation of the documentation pattern inference, the accuracy of the links recovered by RecoDoc impacts the accuracy of AdDoc's recommendations. The impact is direct for deletion recmmendations: a link to a deprecated element missed by RecoDoc would result in a missed deletion recommendation. Addition recommendations are more resilient to RecoDoc accuracy because a documentation pattern involving many code elements would still be inferred if RecoDoc missed a few links.

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.

5 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.

API Documentation. Magyar described an early attempt to maintain the links between API documentation and code [12]. 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. Zhong and Su proposed an API documentation checker that locates code elements that are deprecated or that do not exist in a codebase [13]. Their approach does not perform any type inference and does no try to link the documentation between two releases of the codebase so they discard (and potentially miss) code elements that their approach cannot resolve.

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 [14]. MAPO mines open source repositories and indexes API usage patterns, i.e., sequence of method calls that are frequently invoked together [15]. Then, MAPO recommends code snippets that implement these patterns, based on the programming context of the user. A complete review of such techniques can be found in the survey published by Robillard et al. on API property inference techniques [16].

6. http://cs.mcgill.ca/~swevo/recodoc

Augmenting Existing Documentation. 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 [17]. 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. Many researchers have experimented with the use of information retrieval techniques to recover the links between source code elements and free-text documents. Early attempts include the work of Antoniol et al., who applied two information retrieval techniques, the probabilistic model and the vector space model, to find the page in a reference manual that were related to a class in a target system [18], and the work of Marcus and Maletic, who experimented with latent semantic indexing (LSI) for similar purposes [19]. Information retrieval techniques work best when the entities to be linked can be expressed by several words (e.g., entire documents and complete class definitions), but they cannot be successfully employed to resolve links to elements described by only a few tokens (e.g., method signatures).

Recently, Bacchelli et al., compared both the vector space- and LSI-based information retrieval techniques with a simple pattern-matching approach [20]. They concluded that, for the purpose of linking emails with type-level source code entities, the lightweight approach was consistently superior. Although the pattern-based approach does not suffer from the imprecision of IR-based techniques, it too cannot easily handle fine-grained code elements; these must be scoped within a declaring element, and often have a common name (such as "add"). In contrast, RecoDoc was designed specifically to link single code-like terms to fine-grained elements as accurately as possible. Rigby and Robillard provide a systematic comparison of the three approaches described above with RecoDoc [21, Section 3].

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 [22]. 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 titles, 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) [23]. 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 [20]. 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 [9]. 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) [8]. 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 [24]. The presence of a term in the context of another term is called a *discourse feature*. As opposed to our technique,

TABLE 9
Open source projects mentioned in this paper

Projects	
Hibernate	`hibernate.org`
HttpComponents	`hc.apache.org`
Joda Time	`www.joda.org`
XStream	`xstream.codehaus.org`

users of general NLP techniques typically need to train the techniques on a corpus first to develop a reliable classifier for a specialized task.

6 Conclusion

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.

We devised a technique that can help documentation maintainers and users by making adaptation recommendations based on the links between the code base and the learning resources of a software development project. We built on our previous work to recover these implicit links, and we proposed algorithms to infer documentation patterns and detect violations of these patterns.

When we executed our approach on the documentation of four open source projects, we found that 82% of the inferred documentation patterns were meaningful, i.e., the intension of the patterns matched the topic of the documentation unit. We then conducted a retrospective analysis on the documentation history of four open source projects and found that at least 50% of all the additions in the documentation could be predicted with documentation patterns. The other additions were either related to documentation patterns that did not exist in previous releases or unrelated to any pattern we could think of.

Finally, our recommendation technique also detected 99% of references to deleted or deprecated elements and found 31 incorrect references in the documentation of four open source projects.

Acknowledgments

The authors would like to thank the open source contributor who accepted to review our results and the readers who commented on the first paper. This project was supported by NSERC.

References

[1] D. Kirk, M. Roper, and M. Wood, "Identifying and addressing problems in object-oriented framework reuse," *Journal of Empirical Software Engineering*, vol. 12, no. 3, pp. 243–274, 2007.

[2] B. Dagenais and M. 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*, 2010, pp. 127–136.

[3] ——, "Recovering Traceability Links between an API and its Learning Resources," in *Proceedings of the IEEE/ACM International Conference on Software Engineering*, 2012, pp. 47–57.

[4] B. Dagenais and L. Hendren, "Enabling Static Analysis for Partial Java Programs," in *Proceedings of the ACM SIGPLAN Conference on Object-Oriented Programming Systems Languages and Applications*, 2008, pp. 313–328.

[5] "Joda Time User Guide," http://joda-time.sourceforge.net/userguide.html, accessed 31-Aug-2011.

[6] M. P. Robillard and G. C. Murphy, "Representing concerns in source code," *ACM Transactions on Software Engineering and Methodology*, vol. 16, no. 1, pp. 1–38, February 2007.

[7] A. H. Eden and R. Kazman, "Architecture, design, implementation," in *Proceedings of the IEEE/ACM International Conference on Software Engineering*, 2003, pp. 149–159.

[8] B. Dagenais, S. Breu, F. W. Warr, and M. P. Robillard, "Inferring structural patterns for concern traceability in evolving software," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 254–263.

[9] M. Kim and D. Notkin, "Discovering and representing systematic code changes," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 309–319.

[10] J. M. Carroll, P. L. Smith-Kerker, J. R. Ford, and S. A. Mazur-Rimetz, "The minimal manual," *Journal of Human-Computer Interaction*, vol. 3, no. 2, pp. 123–153, 1987, erlbaum Associates.

[11] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," *ACM Transactions on Software Engineering and Methodology*, vol. 20, no. 4, pp. 19:1–19:35, 2011.

[12] M. Magyar, "Automating software documentation: a case study," in *Proceedings of the ACM SIGDOC International Conference on Computer Documentation*, 2000, pp. 549–558.

[13] H. Zhong and Z. Su, "Detecting API documentation errors," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems, Languages, and Applications*, 2013, pp. 803–816.

[14] S. Thummalapenta and T. 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*, 2008, pp. 327–336.

[15] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and recommending API usage patterns," in *Proceedings of the European Conference on Object-Oriented Programming*, 2009, pp. 318–343.

[16] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, "Automated API property inference techniques," *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, May 2013.

[17] U. Dekel and J. D. Herbsleb, "Improving API Documentation Usability with Knowledge Pushing," in *Proceedings of the IEEE/ACM International Conference on Software Engineering*, 2009, pp. 320–330.

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

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

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

[21] P. C. Rigby and M. P. Robillard, "Discovering essential code elements in informal documentation," in *Proceedings of the 35th ACM/IEEE International Conference on Software Engineering*, 2013, pp. 832–841.

[22] D. Cubranic, G. C. Murphy, J. Singer, and K. S. Booth, "Hipikat: A Project Memory for Software Development," *IEEE Transactions on Software Engineering*, vol. 31, no. 6, pp. 446–465, 2005.

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

[24] M.-F. Moens, *Information Extraction: Algorithms and Prospects in a Retrieval Context*. Springer, 2006.

Barthélémy Dagenais is CTO at Resulto Inc. a web application development startup. He received his Ph.D. and M.Sc. in Computer Science from McGill University and a B.App.Sc. from Université du Québec à Montréal. http://infobart.com/

Martin P. Robillard is an Associate Professor of Computer Science at McGill University. His research focuses on problems related to API usability, information discovery, and knowledge management in software engineering. He received his Ph.D. and M.Sc. in Computer Science from the University of British Columbia and a B.Eng. from École Polytechnique de Montréal. http://www.cs.mcgill.ca/~martin