Automatically Locating Framework Extension Examples

Barthélémy Dagenais*
School of Computer Science
McGill University
Montréal, QC, Canada
bart@cs.mcgill.ca

Harold Ossher
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598
ossher@us.ibm.com

ABSTRACT

Using and extending a framework is a challenging task whose difficulty is exacerbated by the poor documentation that generally comes with the framework. Even in the presence of documentation, developers often desire implementation examples for concrete guidance. We propose an approach that automatically locates implementation examples from a code base given lightweight documentation of a framework. Based on our experience with concern-oriented documentation, we devised an approach that uses the framework documentation as a template and that finds instances of this template in a code base. The concern instances represent self-contained and structured implementation examples: the relationships and the roles of parts composing the examples are uncovered and explained. We implemented our approach in a tool and conducted a study comparing the results of our tool with results provided by Eclipse committers, showing that our approach can locate examples with high precision.

Categories and Subject Descriptors

D.2.6 [**Software Engineering**]: Programming Environments; D.2.7 [**Software Engineering**]: Distribution, Maintenance, and Enhancement

General Terms

Algorithms, Documentation, Experimentation

1. INTRODUCTION

Understanding and extending an application framework is a difficult task. Even when documentation artifacts such as tutorials are available, developers often want to look at real implementation examples for concrete guidance [12]: clear and working examples are an important complement to text.

When developers want to look at examples of framework extensions documented in a tutorial, they can generally find such examples in existing applications. For example, a large number of applications extend the Eclipse Rich Client Platform (RCP)[1], which provides Graphical User Interface facilities to create components such as *editors* and *views*. Finding an extension example such as the implementation of an editor within an existing application requires identifying those pieces of code that make up that extension. This can be difficult because those pieces of code might not be located together and might be intermixed with other code, and some pieces of code might be shared among several extensions. For instance, a typical Eclipse application includes many editors, and it is not always obvious which pieces of editor code belong to which editors. Even if a developer succeeds in identifying the important code elements that implement a framework extension, the responsibilities of those elements and their relationships to the details in the tutorial may not be clear. For example, what is the role of `StructuredTextEditor`? What is the relationship between it and `JSPTagInfoHoverProcessor`?

Two families of approaches have been researched to help developers locate implementation examples, unrelated to documentation. Tools such as Strathcona [7] mine code bases to recommend snippets of code relevant to a development task. These snippets of code are useful to understanding fine-grained framework extensions, but are not adequate for extensions spanning multiple classes, such as a text editor, because the examples are typically related to code found in only one method. Feature-location techniques [4, 9, 10, 20] are another family of approaches that locate a set of functions implementing *one* feature. Unfortunately, these techniques do not leverage the fact that framework extensions exhibit a common structure, and this structure is also not reflected when presenting the implementation to the user.

We propose a technique that enhances framework documentation by automatically locating structured implementation examples of documented framework extensions in a code base. The structure of the examples mirrors the organization of the documentation, and the relationships between the elements comprising the examples are uncovered and explained. We implemented this technique in XFinder (eXample Finder), an extension of Mismar [3], which is a concern-oriented documentation toolset. *Guides* to framework extensions are created and encoded as concerns in Mismar. XFinder basically reuses these concerns as *concern templates* and tries to find instances of these templates in the code base. The two types of documentation artifacts augment each other: the guide provides a template used to structure and locate the examples, and the examples are a form

*This research was conducted while the author was working at the IBM T.J. Watson Research Center.

[1] www.eclipse.org

SIGSOFT 2008/FSE-16, November 9–15, Atlanta, Georgia, USA

of documentation in themselves. For instance, assume that there is a Mismar guide to help developers create Eclipse text editors. Given the entire Eclipse code base, XFinder would find all text editors (Java, XML, HTML, JSP, etc.), and for each of these editors, XFinder would identify the piece of code implementing each step of the guide.

To validate to what extent our approach correctly identifies implementation examples, we performed an *experts study* on the Eclipse text editor framework extension, in which XFinder's results were compared to results provided by the editors' developers. We found that our approach correctly located 93% of the implementation pieces of five text editors, a high level of precision that was confirmed by two case studies performed on two other frameworks. The contributions of this paper include (1) a technique to automatically locate structured implementation examples of a framework extension based on documentation, and (2) a framework documentation toolset that integrates the documentation along with examples.

In the remainder of this paper, we start by describing concern-oriented documentation (Section 2). Then we describe how we can use this type of documentation to find implementation examples (Section 3) and present the results of the preliminary evaluation we performed on our approach (Sections 4 and 5). We cover the related work (Section 6) and conclude in Section 7.

2. CONCERN-ORIENTED DOCUMENTATION

Mismar is a toolset tightly integrated within the Eclipse development environment that allows developers to create documentation simply by pointing out elements in the software system that are important for a particular task [3]. Elements include classes, methods, extension points, files and even web pages. From these elements, a *guide* for the task with an appropriate step for each element is created. For example, if the user selects the XYZ interface as an important element, an "Implement XYZ interface" step is created. Hence, a developer can create a complete guide by simply dragging and dropping elements from a system onto the guide editor. Once a step is created, comments and references to other artifacts can be added to provide contextual information. Moreover, it is possible to reorder the steps or to change their nature (e.g., from "Implement XYZ interface" to "Use XYZ interface"). We refer to this approach as *concern-oriented* because it focuses on software artifacts and their relationships instead of steps or process; a concern [14, 17] captures all elements in a software system that are relevant to a particular point of interest. Mismar guides are saved in an extensible concern model [3].

The resulting guide is also tightly integrated into the development environment, providing interactive support to the user following the guide. For example, when the user performs the "Implement XYZ interface" step by double-clicking on it, the "Create new class" wizard in Eclipse is launched, initialized with the step information. As the user performs the steps, the resulting artifacts, called *outputs*, are recorded for presentation as implementation examples to future users. A collection of outputs in Mismar is called a *result*, and is also a concern modeled in the concern model. A result can be presented to the user in two ways: for each step, a list of outputs from multiple results (see Figure 1), or overall results, each showing a coherent set of outputs for all steps.

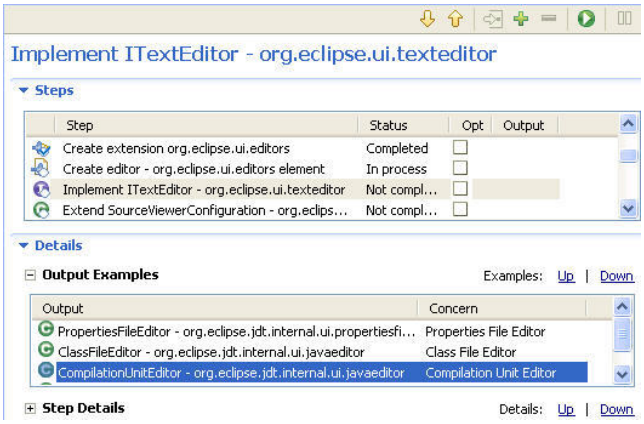

Figure 1: Mismar Result View

As just noted, examples are captured as a guide is used. There are two scenarios, however, in which examples might exist and yet not be tied to the guide. To find such existing examples and associate them with a guide, we devised an approach that is implemented as XFinder, an Eclipse plug-in extending Mismar.

In the first scenario, a developer might obtain a guide that does not have examples provided, but know that his/her workspace contains potential examples. Assume that a developer wants to create a text editor for the Groovy Server Page language (GSP)[2], which combines the Groovy language with HTML. A colleague provides a short tutorial on creating a text editor in Eclipse: the tutorial can already be in the form of a Mismar guide, but without examples, or it can easily be converted to one by selecting the important elements in the tutorial, as described earlier. Because the functionality of the GSP editor is similar to that found in three Eclipse text editors (Java, HTML, and JSP), the developer wants to know how these editors were implemented.

S/he thus loads the Mismar guide in an Eclipse workspace and starts XFinder to find examples. XFinder locates all text editors in the workspace[3] and lists them in the XFinder view (Figure 2). The developer can browse the results to examine any editor implementations of interest. Each is presented as the list of steps from the guide, with a ranked list of potential outputs for each step. The ranking is according to the likelihood that a potential output is the right output for that step of the particular implementation being examined. For example, XFinder indicates that the class `JavaOutlinePage`, which displays the members declared by a Java class, belongs to the Java class file editor. By expanding the `JavaOutlinePage` entry further, the developer could see that, indeed, `JavaOutlinePage` is referenced by the class `ClassFileEditor`, a key element of the editor. Finally, a context menu action allows the user to convert any editor shown into standard Mismar results. Finding examples for a given guide might be useful even if examples were provided with the guide: the user might have more relevant examples in his/her workspace than the guide developer had.

In the second scenario, a developer creating a guide will often do so after having performed the task being described, perhaps several times. During these activities, examples will have been created, but before the guide existed. It would

[2]groovy.codehaus.org/GSP

[3]The source code of the editors can be either in a project or attached to the binaries.

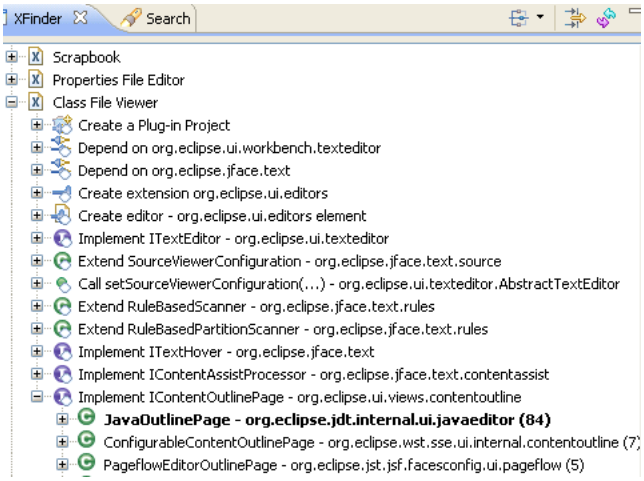

Figure 2: XFinder view

add to the value of the guide if the developer could add these examples to the guide without manually linking them. Assume that a developer is creating a guide for how to create an Eclipse text editor. S/he already has several Eclipse text editors in his/her workspace, and would like to provide these as examples. The developer launches XFinder on the existing guide and obtains a list of all text editors. The developer reviews the outputs selected by XFinder for the most relevant editors and converts them into results, automatically linking them with the guide. The result of this operation is a richer, "illustrated" guide that will be of greater value to users, at low cost to the guide developer. There is the added advantage that the process of finding and examining the examples might reveal flaws in the guide.

3. LOCATING STRUCTURED EXAMPLES

To support the previous scenarios, we designed an algorithm that locates coherent implementation examples. For each example, the algorithm identifies an output, or the fact that there is none, for each step in a given guide.

In Mismar, a guide is a structured concern that contains an ordered list of steps. Each step has a type (e.g., "Extend a Java class") and most of the steps refer to a *main element* (e.g., a Java class, an extension point, etc.). The idea is to use the structure of the guide concern as a template to be matched when searching for examples in a code base.

Given a Mismar guide and a code base, XFinder first searches for all possible outputs for each step. For example, if the type of a step is "Implement an interface" and its main element is the Java interface `ITextEditor`, XFinder finds all classes that implement this interface directly or indirectly. XFinder is not limited to Java-related steps: it can also handle steps involving Eclipse artifacts such as plug-ins, and XFinder plug-ins can provide support for more.

The result of this search is usually to find multiple outputs for each step, belonging to different examples. A key challenge of this work is to cluster those outputs into coherent *example sets*, one for each example containing all the elements of that example. This is done by means of an *example-aggregation algorithm*. It starts with a user-designated *seed step*, whose outputs are deemed to be in one-to-one correspondence with the examples. For example, if the seed step is "Create extension org.eclipse.ui.editor" and there are five such extensions in the code base, XFinder will try to locate five editor implementations. Such extension steps are typically good seed steps for Eclipse extensions. In other contexts, a step asking to implement a major interface or to extend a class is often a good seed step. Ideally, the seed step should not be an optional step or one whose outputs might be shared by multiple extensions. Although, in our experience, one can usually find such seed steps easily, we found during our evaluation (Section 4) that our algorithm was relatively robust to the choice of seed step.

Each output of the seed step designates an example. With this starting point, the aggregation algorithm greedily selects appropriate outputs for other steps based on a variety of relationships between outputs, and adds them to the example set.[4] The intuition is that those outputs making up a single, coherent example will be more closely related to one another than to outputs that belong to different examples. All previously-selected outputs contribute to the determination of the next output: potential outputs that are closely related to multiple selected outputs are selected first by the algorithm. Details of relationship handling and the aggregation algorithm are illustrated in Figure 3. It shows a pruned example consisting of three guide steps, a few outputs for each step, and two types of relationships. The snapshots shown will be described below.

3.1 Relationship Types

XFinder uses different types of relationships to determine whether two outputs are part of the same example. Given two possible outputs, a confidence value between 0% and 100% is computed for each type of relationship. The higher the confidence, the more likely it is that two outputs are indeed related. The confidence value of a relationship falls into four classes: Related, Not Related, Possibly Related (confidence between 0% and 100%), and Inapplicable (confidence is not taken into account). The last class of confidence value, *Inapplicable*, is used when the types of outputs preclude a certain relationship. For example, a Java class and a plug-in extension cannot be related by a Java-to-Java relationship. Snapshot #1 in Figure 3 illustrates the case where two types of relationship are computed between `JavaEditor` and the outputs of the other steps. Based on previous work on program investigation [13] and semi-automated relationship inferencing [6], we currently use four types of relationships.

Name similarity. If two potential outputs have similar names, it is probable that they are related to the same example. Given two possible output names (e.g., the short names of two Java classes), the confidence of the name similarity relationship is the number of common pairs of characters divided by the total number of possible pairs. This metric was previously shown to be robust for assessing the similarity of code-related strings [19, p.4].

Location similarity. If the locations of two potential outputs are similar, it is probable that they are related to the same example. XFinder computes a 3-part location for each output: the *container* (e.g., the project), the *module* (e.g., Java package) and the *unit* (e.g., the Java source file). Given two locations, the confidence of the location similarity relationship is the number of matching location parts divided by the total number of location parts (e.g., 66% or 2/3 if the containers and modules are the same, but not the units).

[4]We experimented with a fuzzy, relational clustering algorithm, but found that it was not as effective as a greedy algorithm, for reasons too complex to explain here.

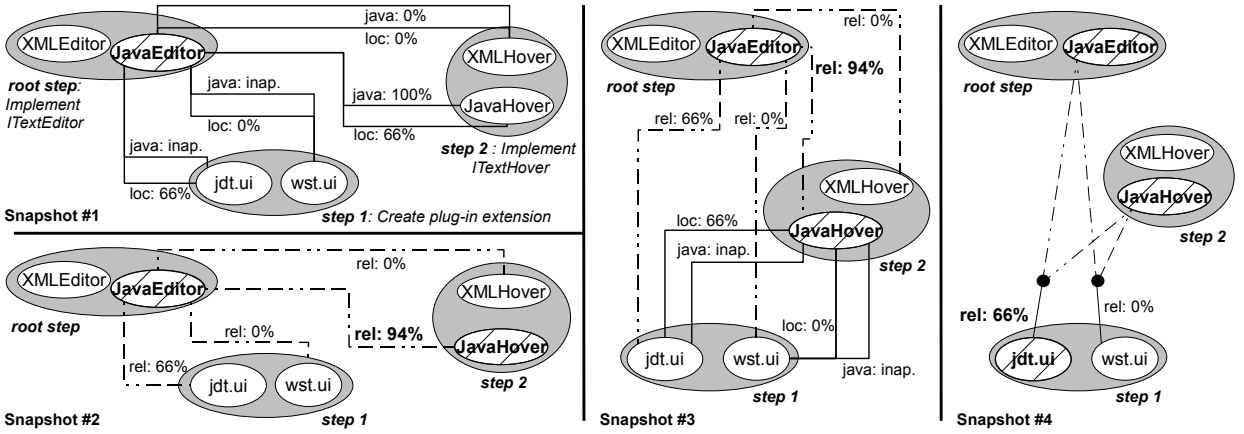

Figure 3: XFinder Algorithm with strong combination strategy

Java-to-Java relationship. Two Java elements (e.g., a class and a method) are probably in the same example if they are related. XFinder currently computes three Java relationships: *calls*, *contains*, and *refers to*. The confidence of a Java-to-Java relationship is 100% if the two Java outputs are related. If an ancestor of one output has a Java-to-Java relationship with the other output, the confidence is 100% - 5 * n, where n is the hierarchy distance between the ancestor and the original output. Otherwise, the confidence is 0%.

Plug-in-To-Java relationship. A plug-in extension is probably related to a Java class if the former explicitly refers to the latter (e.g., the editor extension element indicates which Java class realizes the editor functionality). The confidence of this relationship is 100% if the plug-in extension explicitely refers to the Java class and 0% otherwise.

3.2 Combining Relationships

The separate relationships between two outputs, each with its own confidence level, need to be aggregated into a single relationship whose confidence level expresses the likelihood that the two outputs belong in the same example. For example, in Snapshot #1 of Figure 3, two relationships, *loc* (location similarity) and *java* (Java-to-Java), are shown between `JavaEditor` and each output of each other step. In Snapshot #2 they are combined into a single relationship, *rel*, showing that `JavaHover` is the most closely-related element, and therefore the one most likely to belong in the same example as `JavaEditor`.

During our initial experimentation, we tried several combination strategies: (1) *average:* averaging the confidence value of all relationships, (2) *bonus:* giving different weights to relationships based on their confidence values, and (3) *strong:* like bonus, but favoring domain-specific relationships (Java-To-Java and Plug-in-To-Java), which we refer to as *strong relationships*. Because we use a greedy aggregation algorithm, we found that a naive combination strategy such as *average* could throw off our approach by selecting a wrong output and then sending our algorithm down an erroneous path. Indeed, the average strategy gives the same weight to coincidental relationships like name and important relationships like Java-To-Java: we found that this often favors outputs that are similarly named and located at the expense of strongly-related outputs. Consequently, we explored the two other strategies in order to prevent the selection of wrong outputs. Both the bonus and strong approaches operate on a list of relationships ordered by confidence value, but the strong approach moves any strong relationship(s) before all others. The combination function weights relationships based on their position in the list using the formula

$$\frac{\sum_{i=0}^{n-1} r_i \alpha (1-\alpha)^i}{\sum_{i=0}^{n-1} \alpha (1-\alpha)^i}$$

where $R = r_0, ...r_{n-1}$ is the ordered list of relationships between two outputs, and α is a decay constant. The higher the value of α, the greater the weight we give to the favored relationships. In the case of the strong strategy, it sometimes happens that no strong relationship is found between two outputs: we then only compute the average of the relationships' confidence values to ensure that we will not promote a less relevant relationship. We found the strong combination strategy with an α value set to 0.8 to be the most successful strategy during initial prototyping; this was later confirmed in our evaluation (see Section 4.4).

3.3 Example-Aggregation Algorithm

The objective of the example-aggregation algorithm is to compute the example set containing all the step outputs making up a single, coherent example. Some steps might be found not to have outputs, because the steps were optional, or, perhaps, because the example is imperfect.

The algorithm starts with the example set containing a single output for the seed step (e.g., `JavaEditor` in snapshot #1). Each iteration adds one more output to the set, from a step not yet represented in the set (called a *remaining step*), or determines that all remaining steps have no outputs, which terminates the algorithm. The algorithm also remembers all computed relationships as it proceeds, for its own use in later iterations and for presentation to the user.

The first iteration starts with the single seed output in the example set. It computes all relationships between this output and all outputs in all remaining steps (snapshot #1 of Figure 3). It then aggregates the relationships, as described earlier (snapshot #2), and selects the new output with the largest confidence (`JavaHover` in snapshot #2). If the confidence is less than a threshold value of 50%, the selected output is dropped, all remaining steps are deemed to have no outputs, and the algorithm terminates. Otherwise, the selected output is greedily placed in the example set, and the algorithm proceeds to the next iteration. Note that all related outputs are eventually listed for the user in confidence order, but the algorithm greedily selects just the highest one for use in its subsequent iterations.

Each subsequent iteration evaluates the relationships between the new output added to the example set and the outputs of all the remaining steps (snapshot #3 in Figure 3, which shows both the aggregated relationships computed during the first iteration (dashed) and the new ones, between `JavaHover` and the outputs of step 1 (solid)). These new relationships are aggregated, as before. Now we have a situation where each output in each remaining step is related to multiple outputs in the example set. These relationships are also aggregated, using the *strong* combination approach, as before (snapshot #4). Finally, the new output referred to by the aggregated relationship with the largest confidence (`jdt.ui` in snapshot #4) is selected, and the algorithm proceeds as above. The step ordering is irrelevant for this algorithm: in Figure 3, an output for step 2 is selected before step 1, because of the confidence levels.

The algorithm uses a threshold confidence value of 50% to detect cases where steps do not have outputs in a particular example. A confidence value below 50% means that the output does not have a strong relationship with any other previously-selected output, and that its name and location are significantly different from the other outputs'. If no output has a combined confidence over 50%, XFinder presumes that no output is similar enough to belong in the example set.

The complexity of the algorithm is bounded by n, the number of steps in the guide, and m, the maximum number of possible outputs per step, which is typically higher than n. Computing the relationships between two outputs and aggregating them is assumed to take constant time k, because the number of relationships is fixed (at 4). Since we compute the relationships between the last selected output and the potential outputs of all the remaining steps, the complexity of our example-aggregation algorithm can be expressed as

$$\sum_{i=1}^{n} kim = km\frac{n(n+1)}{2} = O(mn^2)$$

3.4 Seed Step Heuristic

During early experimentation with some small projects, we observed that, when an example contains artifacts shared with other examples, the greedy aggregation algorithm could become confused: once it selects a shared artifact, it starts evaluating other shared artifacts, sometimes at the expense of specific artifacts. We thus added an optional heuristic that gives a 15% bonus to the name and location relationships between the seed step output and any other potential output. Given two artifacts with equivalent strong relationships, one shared and one specific to the example, this heuristic favors the specific artifact. If most artifacts have a similar location and name, this heuristic can be harmful, however, as discussed in Section 5.1. XFinder therefore automatically recommends that this heuristic be disabled in one case of such co-location: when XFinder runs on a project that does not reference any other projects.

4. EXPERTS STUDY

To validate to what extent XFinder produces correct results, we designed two studies. The first, presented in this section, quantitatively evaluates the various factors impacting the effectiveness of our approach by comparing XFinder's results on one framework extension with results obtained from experts on this extension. Case studies, presented in Section 5, evaluate the effectiveness of XFinder with respect to two other frameworks. For the experts study, we were interested in evaluating the following criteria:

1. The overall quality of the results provided by XFinder as measured by the number of correct outputs selected for each framework extension example.
2. The stability of our algorithm with respect to external factors not under the control of XFinder: the quality of the guide and the choice of the seed step.
3. The impact of various XFinder parameters: α, the combination strategy, and the seed step heuristic.

4.1 Experimental Design

To evaluate the above criteria, we selected the text editor extension of the Eclipse platform,[5] which enables developers to create text editors that provide standard features such as code completion, text hovering, and syntax highlighting. Based on our own experience with text editors, we created a guide in Mismar consisting of 13 steps (shown under the Class File editor in Figure 2). More precisely, the guide contains (1) mandatory configuration steps such as creating a project and a plug-in extension, (2) mandatory Java steps such as implementing the `ITextEditor` interface, and (3) optional Java steps related to optional functionality such as syntax highlighting. The idea was to use this guide as an input to XFinder and to find text editor implementations in a standard Eclipse distribution.

To evaluate the correctness of XFinder results, we contacted ten Eclipse committers within IBM who were responsible for the development or maintenance of Eclipse text editors, and asked them to complete a questionnaire on a specific editor they were familiar with. Eight of the ten committers replied positively and six of them completed the questionnaire. The questionnaire consisted of four sections: the first section asked demographic questions, the second asked questions on evaluating XFinder results, the third section covered the quality of our 13-step guide and the fourth section asked questions about framework documentation in general. Only the first and second sections were mandatory. Since all responders agreed to detailed reporting of their answers, the results of the second section are available online.[6] On average, the responders had 7.1 years of experience in Java development and 5.4 years of experience specific to Eclipse development. Four of them were the main developers of the editors they were asked to evaluate; the two others evaluated editors they were familiar with.

The second section of the questionnaire presented the 13-step guide to creating a text editor. For each step, we provided a list of potential outputs for the step, and asked the responder to identify the correct one(s) by: (1) selecting one, (2) selecting multiple and ranking them according to their importance for this step, (3) indicating that this step was not implemented, or (4) indicating an output not listed. The list of outputs was created by selecting all outputs that resided in the same project as the editor under evaluation, adding a random list of outputs from other projects (not exceeding 15 outputs in total) and then randomizing the order of the outputs. This particular design was selected for practical reasons: for certain steps, there were more than 30 possible artifacts and listing them all (or offering no choice at

[5] www.eclipse.org

[6] bart.prologique.com/projects/mismar/xfinder-eval

Editor	All Relationships	Name & Location	Strong only
Ant	92 (100)	83 (92)	58 (66)
Java source	92 (92)	58 (58)	58 (58)
Java class	92 (92)	42 (75)	58 (58)
HTML	100 (100)	50 (50)	42 (42)
JSP	92 (92)	58 (58)	42 (42)
Feature	33 (92)	33 (92)	0 (0)
Average	83 (94)	54 (71)	43 (44)
w/out outlier	93 (95)	58 (67)	52 (53)

Table 1: XFinder results

all) would have put an unacceptable burden on the Eclipse committers.

The answers from the Eclipse committers gave us an oracle to validate the results of XFinder for the Ant Editor, Java Source Editor, Java Class Editor, HTML Editor, JSP Editor, and Feature Editor (an editor for an Eclipse extension artifact). These editors represent an interesting range of implementation examples because they exhibit characteristics that typically make locating code examples difficult. For example, the two Java editors (class and source) share many classes, and the HTML and JSP editors are scattered among multiple projects and share one main class. The Feature, HTML and JSP editors also implement some parts of the editor in a non-standard way, using mechanisms not covered by Eclipse tutorials. The Ant editor is the simplest editor in our sample because it is the only editor defined in its project and it provides a single output for each step.

We executed XFinder using the text editor guide as input on an Eclipse workspace referencing the following plug-in families: Web Tool Platform, Plug-in Development Environment, Java Development Tools, Platform Text, and Ant. We then compared the results obtained by various configurations of XFinder with the answers given by the six Eclipse committers. We analyzed the results in the light of the three criteria mentioned at the beginning of Section 4. The results of each execution are also available online.

4.2 Quality

To evaluate the quality of the results provided by our approach, we executed XFinder by selecting the step "Create `editor` extension element" as the seed step and using the following parameters: $\alpha = 0.8$, combination strategy = strong, seed step heuristic = enabled. The seed step we chose is the only step in the guide that cannot potentially refer to a shared output: each editor in Eclipse must be declared by this extension element, and only one editor can be declared by each element. All of the other steps can produce an output that is shared by multiple editors (e.g., the HTML and JSP editors both share the same implementation of `ITextEditor`). The parameters we used were found to be the most successful during early validation of the approach.

After 128 seconds, XFinder returned a list of 49 editors, 21 of which were text editors (the others were form or graphical editors). For the six editors evaluated by the Eclipse committers, we looked at the 12 steps (the 13th being the seed step) and compared XFinder's recommendations with the answers from Eclipse committers. We considered that a recommendation for a step was correct if either (1) XFinder's top recommendation matched one of the outputs identified by the Eclipse committer or if (2) XFinder and the Eclipse committer both indicated that there was no output for that step. We also relaxed the metric by considering for each step the top three recommendations when XFinder had selected an output: if one of the three recommendations matched an output selected by the Eclipse committer, we considered the recommendation to be correct according to the relaxed metric. These two metrics accommodate the scenarios where a tool would automatically use the results of XFinder and where a user would browse the results: in the former case, only the top recommendation is used, but in the latter case, the user is expected to look at least at the top three recommendations while reviewing the results.

Table 1 shows the results of this study. The first column gives the name of the editor and the second column contains the percentage of correct recommendations. The numbers in parentheses represent the percentages of correct recommendations if we consider the top three outputs suggested by XFinder.[7] For example, for the Ant editor, XFinder recommended the correct output for 11 of the steps (11/12 or 92%), and the correct output was in the top three recommendations for all steps (12/12 or 100%).

In 83% of the cases, XFinder correctly identified the outputs belonging to a particular editor and the steps where no output had been produced. When looking at the top three recommendations, XFinder achieved a success rate of 94%. This is evidence that our approach can automatically locate framework extension examples with considerable accuracy. In 13 of the 18 cases where an Eclipse committer had selected multiple outputs for a step, XFinder gave a high confidence value to multiple correct outputs. For example, in the Java source editor, all four classes implementing the syntax highlighting feature were recommended with the same confidence value (78) and one of them was selected by XFinder: our approach selects one output because the current version of Mismar only supports one output per step.

The only exception is the Feature editor, for which XFinder performed poorly by only identifying the correct output for four steps. Looking more closely at the Feature editor, we realized that it did not match exactly the definition of a text editor as provided by the Eclipse platform: this editor is a multi-page editor, i.e., a composite editor that offers multiple views of the same data. It provides nine views, seven of them being form-based and two of them being text editors. Moreover, the extension element defining the Feature editor (the seed step) refers to the class `FeatureEditor` which belongs to the hierarchy of `MultiPageEditor` and does not implement the interface `ITextEditor`. This means that strong relationships from the main class implementing the Feature editor are useless for finding outputs of other text-editor steps. The algorithm must thus rely on weaker relationships such as location and name similarity, which are less accurate. This is a limitation of our approach: XFinder struggles to locate an example whose structure greatly differs from the documentation, rather than identifying the problem. In this particular case, a guide on creating multi-page editors would probably give better results. Because we consider this editor to be an outlier, we did not include its results when assessing the impact of other factors in the next sections: there was no factor that improved the results for this editor and the presence of an outlier might have hindered our evaluation.

Finally, we also executed XFinder using only Name and Location relationships (third column of Table 1), and then

[7]Correctness will always be reported in this format from now on.

Seed	Precision	Editors
Editor extension element	93 (95)	5/5
Project	96 (100)	2/5
SourceViewerConfiguration	86 (94)	3/5
ITextEditor	92 (94)	3/5

Table 2: Choice of seed step

Steps	# of steps	Precision
Complete	13	93 (95)
w/o optional	8	94 (100)
w/o configuration	9	85 (93)
w/o SourceViewerConfig.	11	74 (74)

Table 3: Guide quality

only Java-To-Java and Plug-in-To-Java relationships (fourth column). XFinder performed poorly and selected correct outputs in 54% and 43% of the cases, respectively. These results provide evidence that all relationships are necessary and that a more naive approach, considering only weak or strong relationships, would probably fail in identifying most of the correct outputs.

4.3 Stability

When locating examples, there are two main factors that are outside the control of XFinder: the choice of the seed step and the quality of the guide. To assess the algorithm's robustness with respect to the choice of the seed step, we executed XFinder by selecting different seed steps that, based on the guidelines given in Section 3, could likely be chosen by a user because they are mandatory and usually representative of an editor. Table 2 contains the results of this analysis. The first column indicates the seed step we used, the second column presents the percentage of correct recommendations, and the third column indicates the number of editors that could be found using this seed step. For example, for the seed step "Create a plug-in project" (seed = Project), there were only two editors out of the five that could be found, the outlier not included. For those two editors, XFinder correctly identified an output for 96%(100%) of the steps. The editors value, 2/5, is obtained by counting the number of editors that have, for the seed step, an output that represents their implementation only. For the project seed step, we cannot include the Java source, Java class or JSP editors because their respective projects define other editors also: XFinder is then confused and may select arbitrary outputs from those multiple editors. Unfortunately, XFinder does not detect the presence of multiple examples, a limitation that remains an area for future work. Unfortunately, XFinder does not detect the presence of multiple examples, a limitation that remains an area for future work. Overall, these results suggest that our approach can produce accurate results even with suboptimal seed steps, for the cases it can handle. For example, with the project seed, XFinder could not compute strong relationships (e.g., Java-to-Java) between the seed step output and other potential outputs. To circumvent this problem, XFinder greatly favored outputs that were in the same project using the seed step heuristic until stronger relationships could be computed. These stronger relationships were eventually necessary for editors like the HTML editor, whose implementation was scattered in multiple projects.

Another factor that might impact XFinder's effectiveness is the quality of the guide used to locate an example. Indeed, framework documentation is often incomplete or out of date. To assess the impact of the documentation's quality, we executed XFinder with various subsets of the steps in the original guide. Table 3 reports the results. The first column indicates the steps that we removed from the original guide, the second gives the number of steps, including the seed step, remaining in the guide, and the third shows the percentage of correct recommendations for the five editors. The first row reports the results of executing XFinder with the complete guide (13 steps). The second row represents the guide without the optional steps, such as providing syntax highlighting. For the third execution, we simulated a tutorial dealing only with code elements, because many tutorials focus exclusively on these: we removed the configuration steps such as the creation of a project and the addition of dependencies. In the fourth execution, we evaluated a particularly bad scenario by removing two critical steps: those asking to extend and call a method from the `SourceViewerConfiguration` class, an essential element of any text editor.

In general, our approach is robust with respect to the quality of the guide. Not surprisingly, removing optional steps increased the success rate: outputs of optional steps are typically more difficult to locate because they are sometimes accessed indirectly (no strong relationships) or they are not present at all (it is harder to be sure that an output is not provided). Removing the configuration steps slightly decreased the success rate because the few outputs that depended only on the name and location similarity did not reach a high-enough confidence level. Finally, the removal of the two key steps related to the `SourceViewerConfiguration` class had a major impact on XFinder effectiveness. Because this class is usually the hub connecting all the other classes in a text editor, XFinder was unable to compute strong relationships and missed several correct outputs. These results suggest that XFinder will probably perform well with incomplete or slightly out-of-date guides, but its effectiveness will degrade significantly if key steps are missing.

Parameter	Precision	Parameter	Precision
heuristic	*93 (95)*	α=0.5	85 (87)
w/o heuristic	85 (88)	α=0.6	93 (97)
average	52 (53)	α=0.7	93 (97)
bonus	83 (93)	*α=0.8*	*93 (95)*
strong	*93 (95)*	α=0.9	93 (95)

Table 4: XFinder parameters

4.4 XFinder Parameters

Thus far, we executed XFinder by fixing values to three main parameters: the usage of the seed step heuristic (Section 3.4), the combination strategy (Section 3.2), and the α value used in some combination strategies. To validate the optimality of these values, we report the results of executing XFinder with different parameter values in Table 4. The first and third columns indicate which values were given to the parameters and the second and fourth columns show the percentages of correct recommendations. Default values used in the previous sections are *in italics*.

The first parameter we studied was the usage of the seed step heuristic. This heuristic was essential in selecting correct outputs for the two editors that were scattered among multiple projects, i.e., the HTML and JSP editors. Indeed, the class that implements the interface `ITextEditor` for both

editors, `StructuredTextEditor`, is shared by multiple other editors and refers to default implementations of optional features. By increasing the confidence value of implementations that had a name or location similar to the seed of these editors, the heuristic helped XFinder in selecting the outputs that were specific to them. While enabled, the heuristic did not hinder the location of outputs for other examples, but disabling the heuristic slightly reduced XFinder's success rate: our approach missed the correct output for 3 steps in the HTML editor and 2 steps in the JSP editor. This indicates that, generally, the seed step heuristic should be enabled, but we show in Section 5.1 when it should not.

The second parameter we studied was the selection of one of our three combination strategies. As can be seen in Table 4, the naive *average* strategy was unsuccessful and XFinder recommended the correct output in only 52% of the cases. The *bonus* strategy performed better, with a success rate of 83%, approaching the success rate of the *strong* strategy, 93%. The variation between the last two combination strategies can be explained by the fact that, for six steps, *bonus* favored location similarity at the expense of a Java relationship, resulting in the selection of an incorrect output.

Finally, because it appeared that the *strong* relationship combination strategy was best, we ran XFinder with different values for the α parameter, effectively varying the degree to which we favor strong relationships. Except for a value of 0.5, the variation in the α parameter did not significantly impact our results. The only difference is that one correct output is ranked third for a value of 0.6 and 0.7 and ranked fourth for a value of 0.8 and 0.9, hence the variation in the top three recommendations' correctness. We conclude that between 0.6 and 0.9, our algorithm is not sensitive to the value of α, which suggests that it should not need to be tailored to a particular framework extension.

4.5 Threats to Validity

Although we analyzed the results of six editors, the external validity of our study is limited by the fact that we studied only one framework extension and one guide, which means that it might not generalize to other frameworks and tutorials. This is mitigated by the fact that the framework extensions we studied exhibited a wide variety of structures, and we evaluated the impact of the documentation quality. The analysis of various parameters also suggests that some strategies are clearly more efficient than others, which should limit the need to tailor those parameters to a particular framework. Furthermore, to reduce this threat, we present in the next section two case studies that we performed on other frameworks.

The choices that we offered in the second section of the questionnaire to the Eclipse committers could be overly leading because some of the choices were taken from the project where most parts of the editor under evaluation resided. This threat was mitigated by the fact that all projects, except the ant project, defined many editors and that two editors were scattered across multiple projects. Additionnally, the Eclipse committers indicated that there was no implementation for seven steps and they specified an artifact not mentioned in the choices for two steps, which provide evidence that the committers did not feel compelled to select only the choices that we offered.

Investigator bias was mostly limited to the choice of framework extension and the creation of the documentation. We argue that Eclipse text editors are common enough to be of interest. As for the documentation, three Eclipse committers reviewed our guide in the third section of the questionnaire and concluded that it was mostly complete, with one committer suggesting one extra step. Though we used experts to validate the correctness of our results, human errors remain; in fact, we needed to contact two responders to correct mistakes in their answers. Finally, even if our approach usually provides correct results, we can only make hypotheses about how developers would use our tool. Validating those hypotheses is a natural next step in the evaluation of XFinder.

```
1- Create a Java project
2- Extend soot.toolkits.scalar.AbstractFlowAnalysis
3- Call soot.toolkits.scalar.AbstractFlowAnalysis:doAnalysis
4- Extend soot.Transformer
5- Implement soot.tagkit.Tag
6- Call soot.tagkit.Host:addTag
7- Call soot.Transform:Transform
8- Call soot.Pack:add
```

Figure 4: Soot Guide

5. CASE STUDIES

To gather preliminary evidence that our approach produces similarly high-quality results when applied to other frameworks and guides, we performed two case studies on framework extensions outside of the Eclipse platform. Although investigator bias is inevitable with such studies, we tried to reduce its impact by selecting framework extensions that (1) were simple enough for us and external reviewers to manually review the results, (2) had publicly-available documentation so we could create a Mismar guide as objectively as possible, and (3) had a decent number of implementations. We found two such framework extensions, Soot data flow analysis [16], and Swing JTable.[8] The results of both case studies are available online.

5.1 Soot Static Analysis

Our first target system, Soot, is a static analysis framework commonly used by researchers to create various kinds of static analyses and improve the performance of Java and AspectJ programs. The version we studied was 2.2.4 and it comprised 180739 lines of code. We selected the ability to create a custom data flow analysis as the framework extension. We used two Soot tutorials related to this extension[9,10] to write a guide: we basically dragged the main classes and methods in the Soot source code that were referenced by the tutorials and dropped them into our guide editor, a process that took less than 5 minutes. The final guide consisted of 8 steps helping a developer to create a data flow analysis, annotate classes using this flow analysis and register the analysis within the Soot framework. Figure 4 lists the steps of the guide without the textual description accompanying them. The tutorials indicate that the last five steps are optional.

Because Soot already defines a significant number of flow analyses (some are optional, others are provided with the framework), we used Soot as a client program. We selected the second step, "Extend AbstractFlowAnalysis", as the seed step and we executed XFinder on a workspace containing the source code of the Soot project. We used the same parameters as in Section 4.2 except that the seed step heuristic was

[8] java.sun.com/products/jfc/tsc/articles/architecture/
[9] www.sable.mcgill.ca/soot/tutorial/analysis/index.html
[10] www.sable.mcgill.ca/soot/tutorial/tagclass/index.html

System	Version	LOC
Abacus GUI Builder	1.8	57171
BNF for Java	alpha 1	9058
JDecompiler	1.2	85607
Fredy's SQL Tool	2.4.2	41809
Class Editor	2.23	10027

Table 5: Target systems

disabled on XFinder's recommendation (see Section 3.4). After 93 seconds, XFinder returned a list of 43 data flow analyses, 4 of them being abstract analyses not interesting for this case study. We then randomly selected 15 of these flow analyses. For each class extending the `AbstractFlowAnalysis` abstract class in our random sample, we manually identified the output for each step. We then compared our findings with the results provided by XFinder: 97.8% of the time, XFinder's recommendation matched our finding. By computing a 95% confidence interval, we estimated the precision of XFinder on this framework extension to be 97.8% $\pm$ 4%, or 6.8 $\pm$ 0.3 correct recommendations out of 7 steps.

Generally, we found the artifacts comprising Soot data flow analyses to be strongly related to one another, and often to be similarly named, which matched the premises of our approach. For example, `ParityAnalysis` is called by `ParityTagger`, which annotates classes by creating instances of `StringTag`, a common class used by many analyses. XFinder missed two outputs because the usage of a factory method introduced an indirection, inhibiting the computation of strong relationships. The presence of such methods is not uncommon, and XFinder is usually able to rely on weaker relationships to identify the correct output, but in this case the name and location of the correct outputs were not similar enough to previously-selected outputs. Additionally, in this case, dynamic analysis would be required to identify the correct output because the factory method uses a condition on the value of a program argument.

Finally, we went against the recommendation of XFinder by running the analysis with the seed step heuristic enabled and we observed that the precision fell to 81%. This drop is due to two factors that reinforce each other: (1) the presence of many optional steps and (2) the similar names and locations of potential outputs (e.g., almost all analysis classes contain the word "Analysis" in their names and many analyses are in the same package). The seed step heuristic increased the confidence value of similarly-named and located outputs for optional steps, which resulted in the selection of wrong outputs for steps that should have had none. Our greedy aggregation algorithm then computed relationships from these wrong outputs, resulting in the selection of wrong outputs for the remaining optional steps. With fewer optional steps, the absence of relationships between correct outputs and wrong outputs would prevail over the presence of relationships between wrong outputs, as in the case of the Eclipse editor guide.

The Soot case study indicates that XFinder can locate implementation examples with high precision even if there are many optional steps and the artifacts comprising the various implementation examples have similar names and locations.

5.2 Swing JTable

Our second case study involves the Swing framework, which provides graphical user-interface capabilities on the Java platform. We chose to study the JTable extension, which

```
1- Create a Java Project
2- Implement javax.swing.table.TableModel
3- Use javax.swing.JTable
4- Use javax.swing.JScrollPane
5- Call javax.swing.ListSelectionModel:addListSelectionListener
6- Implement javax.swing.table.TableCellRenderer
7- Call javax.swing.table.TableColumn:setCellRederer
8- Implement javax.swing.table.TableCellEditor
9- Call javax.swing.table.TableColumn:setCellEditor
```

Figure 5: Swing Guide

enables a developer to create a table with rows, columns and cells. We created a guide consisting of 9 steps taken from the Swing JTable tutorial.[11] Creating the guide was a challenging task for three main reasons. First, the Swing API offers many ways of doing the same thing: for example, there are many constructors and setters with different parameters that can be used to bind a `JTable` instance with a `TableModel` instance. We therefore kept the number of method-call steps to a minimum, and only used a subset of the methods mentioned in the tutorial. Second, the Swing platform offers default implementations of virtually everything, making any step optional. Third, it is possible to extend the Swing framework in two ways: by composing objects (e.g., you bind a custom `TableModel` with a `JTable`) or by extending classes (e.g., you extend `JTable`) which makes it doubtful that one tutorial will be representative of all extensions. Figure 5 lists the steps of the guide without the accompanying textual description.

After we created the guide, we looked on SourceForge[12] to find five open source programs that used JTable and that were small enough to be inspected manually. Table 5 shows the client programs we chose for this case study.

We selected the second step in our guide, "Implement TableModel", as the seed step and we executed XFinder on a workspace containing the source code of the five projects. We used the same parameters as in Section 4.2, except that the seed step heuristic was again disabled on the recommendation of XFinder. After 85 seconds, XFinder returned a list of 49 table models, 4 of them being abstract tables not interesting for this case study. We then randomly selected 15 of these tables. For each class implementing the `TableModel` interface in our sample, we manually identified the output for each remaining step. We then compared our findings with the results provided by XFinder: in 88.5% of the cases, XFinder's recommendation matched our finding. For this case study, we considered that if a guide step described some alternative not used in a particular JTable (e.g., extending a JTable instead of composing it), XFinder should not select any output for that step. By computing a 95% confidence interval, we estimated the precision of XFinder on this framework extension to be 88.5% $\pm$ 8.1%, or 7.1 $\pm$ 1.1 correct recommendations out of 8 steps.

Because all the steps were optional, no table in our sample implemented the full guide. XFinder was still able to detect most of the cases where no output was provided by a table implementation, as illustrated by the high precision. Most of the wrong outputs were selected because of the unusually large number of optional steps and the many explicit references to framework default implementations. Figure 6 shows an example of the problem: if a particular JTable implementation explicitly uses framework default implementa-

[11] java.sun.com/docs/books/tutorial/uiswing/components/table.html

[12] www.sourceforge.net

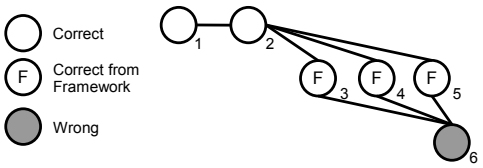

Figure 6: Framework defaults problem

tions for multiple steps, the number of outputs belonging to the framework might outnumber the outputs belonging to the specific JTable. Even if this JTable does not provide implementations for the remaining optional steps, XFinder might select wrong outputs if they also explicitly refer to the framework default implementations: the presence of relationships between framework outputs and wrong outputs prevails over the absence of relationships between specific correct outputs and wrong outputs. This problem did not occur in the Eclipse text editor study because there were fewer optional steps and the editors provided specific implementations for most steps.

The results of the JTable case study indicate that XFinder can locate framework extension examples with reasonable precision even when some examples use alternatives not documented in the guide. It also suggests that future work is required to reduce the problems introduced by default framework implementations.

6. RELATED WORK

Automatically locating framework extension examples intersects with three main research areas: code example location, feature location and framework documentation.

Example location. A number of approaches have been proposed to locate code examples and leverage them to document framework usage. Strathcona is a tool that mines the source code of several example programs and recommends snippets of code found in a method of an example program that is similar to the local programming context [7]. Code examples proposed by Strathcona are first presented as a small UML diagram showing a main class with its most relevant dependencies. Then, the user can see example code taken from a method in the main class or move on to the next recommendation. Strathcona is limited to examples whose code comes from a single method: the tool does not recommend examples that are scattered across several artifacts, which limits its applicability for coarse-grained framework extensions such as text editors. Another limitation of Strathcona with respect to finding implementations of framework extensions is that the structure and the rationale behind the returned code snippets are inconsistent across recommendations: as stated on the tool's website, "some of the heuristics in the backend make different types of guesses to generate the examples and some of them may not be useful to you"[13]. Because XFinder uses the documentation structure to present the code examples, the user always reviews the results in a familiar setting, i.e., as an implementation example of a tutorial.

FrUIT is a tool that solves the opposite problem addressed by XFinder: by applying data mining to a set of framework extension examples, it infers the framework documentation in the form of usage rules such as "Extend class `ABC`", and "Override method `m1`" [2]. These usage rules, displayed with code examples implementing them, can greatly reduce the time required to learn a framework, but it is difficult at this stage to assess the cost of this approach because a full validation study has yet to be published. For example, the authors mentioned in their paper that their data-mining algorithm was not yet scalable enough.

Code examples automatically retrieved from a local repository or from the web[14] are used by Propespector [11] and PARSEWeb [18], respectively, to recommend Method-Invocation Sequences (MIS), each of which is a list of method calls that produce an instance of a destination class, given a source class. For example, a developer who wants to obtain an instance of `ICompilationUnit` from an instance of `IEditorPart` could query PARSEWeb, which would then recommend a sequence of three methods to call. XSnippet is another tool that mines examples from a local repository and can recommend a code snippet showing how to instantiate a particular class [15].

One important advantage these tools have over XFinder is that they do not require any kind of documentation to locate examples of a framework extension and infer usage rules. On the other hand, they do not address the problem of finding examples to illustrate documentation, and they do expect the user to have sufficient knowledge of the framework extension to pose a query. Moreover, code examples and usage rules produced by these tools are usually only related to one main class or method (e.g., getting an instance of `ICompilationUnit`) and are thus more suitable for finding examples of fine-grained framework extensions such as syntax highlighting. As opposed to XFinder, these example-location tools only deal with Java code elements and do not take into account other artifacts such as plug-in extensions and configuration files. Clearly, if a tutorial is unavailable or if the framework extension can be expressed in the body of a method, tools such as Strathcona should be preferred to XFinder.

Feature location. Numerous techniques have been devised to automatically locate implementations of features in source code. For example, SNIAFL is a static feature-location tool that combines static analysis and information retrieval to identify the functions that comprise a particular feature [20]. Other approaches use dynamic analysis to find functions that were used during one or multiple program executions [4, 9, 10] and that are specific to a feature. Feature-location techniques are targeted toward identifying a *specific* feature implementation (e.g., the HTML editor) whereas we want to identify *multiple* implementations of the *same* framework extension. To achieve these related goals, feature location tools have requirements similar to XFinder: they need feature descriptions (often taken from specification or design documents) or execution traces and we need lightweight documentation of the framework extension.

Results returned by feature-location tools often take the form of lists of functions along with confidence numbers indicating to what extent each function is relevant to a feature. One could thus run a feature-location tool on a target system for each desired framework extension implementation (e.g., the Java, JSP and HTML editors). As opposed to XFinder, though, the results for each implementation would probably not be presented with the same structure (e.g., the functions would not be presented in the same order, not all parts of the editor would be selected, etc.).

[13]lsmr.cpsc.ucalgary.ca/projects/heuristic/strathcona/using

[14]www.google.com/codesearch

Framework documentation. Various approaches have been proposed to document framework usage and framework extension examples. Fairbanks et al. devised a technique that enables a developer to document patterns of engagement with a framework, called Design Fragments [5] (e.g., call `methodA()` before `methodB()`). When extending a framework, developers can annotate the extension to map the code entities to a design fragment; a tool then statically checks for conformance to the design. Users of design fragments have access to a catalog of fragments along with a list of instances (i.e., implementation examples) for each fragment. Our approach does not require a manual mapping between the documentation and the implementation examples, but we could use design fragments as templates to locate examples and then check their full conformance with the fragments.

Framework-specific modeling languages (FSML) offer another way of describing framework extensions and enable semi-automatic location of implementation examples [1]. A framework extension (called a *concept*) can be represented as a hierarchy of features, which are themselves composed of structural and behavioral patterns defined with a pointcut-like language [8]. For each of these patterns, the authors offer several queries that can locate instances in the code of *one project*. The authors give an example where they are interested in a feature that displays messages in an applet: the query they use returns the string values of all messages that could be determined statically in the code. The patterns are more fine-grained than Mismar steps and can be used to identify very specific behavior. Because the user must select an adequate query for each pattern, we consider this approach to be semi-automatic. Although a Mismar guide could be encoded using some of the more coarse-grained patterns offered by FSML, it is not clear how this approach could be used to cluster the results at the granularity of the framework extension instead of the project.

7. CONCLUSION

We presented a technique for automatically locating structured examples of framework extensions given guides that document the framework. Our approach uncovers the relationships and the roles of the elements in each example. The use of a generic concern model guides the example-location process and allows the inclusion of different artifact types. Our evaluation provides evidence that our approach is efficient and reasonably accurate: an experts study performed on the Eclipse text editor framework extension showed that our approach could correctly locate 93% of the implementations of five text editors. We conclude that given concern-oriented documentation, it is possible to use the underlying concern model as a concern template and then transform the problem of finding implementation examples into a simpler problem: finding instances of this concern template. Documentation not specifically concern-oriented, such as tutorials and cheat sheets, nonetheless has underlying concerns—the elements referenced—so we believe our approach to be more generally applicable. Deriving the concern models automatically remains an area for future research.

Acknowledgments

The authors thank Martin Robillard for his valuable comments on the paper and all the Eclipse committers who contributed to the validation study.

8. REFERENCES

[1] M. Antkiewicz, T. T. Bartolomei, and K. Czarnecki. Automatic extraction of framework-specific models from framework-based application code. In *Proc. of the 22nd Int'l Conf. on Automated Software Engineering*, pages 214–223, 2007.

[2] M. Bruch, T. Schäfer, and M. Mezini. FrUiT: IDE support for framework understanding. In *Proc. of the OOPSLA Workshop on Eclipse Technology eXchange*, pages 55–59, 2006.

[3] B. Dagenais and H. Ossher. Aiding evolution with concern-oriented guides. In *Proc. of the 3rd AOSD Workshop on Linking Aspect Technology and Evolution*, page 4, 2007.

[4] T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Transactions on Software Engineering*, 29(3):210–224, 2003.

[5] G. Fairbanks, D. Garlan, and W. Scherlis. Design fragments make using frameworks easier. In *Proc. of the 21st Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 75–88, 2006.

[6] L. Gong, T. Klinger, P. Matchen, P. Tarr, R. Uceda-Sosa, A. Ying, J. Xu, and X. Zhou. Integrated solution engineering. In *Companion to the 21st Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 726–727, 2006.

[7] R. Holmes, R. J. Walker, and G. C. Murphy. Approximate structural context matching: An approach for recommending relevant examples. *IEEE Transactions on Software Engineering*, 32(12):952–970, 2006.

[8] G. Kiczales, J. Lamping, A. Menhdhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proc. of the 11th European Conf. on Object-Oriented Programming*, pages 220–242, 1997.

[9] R. Koschke and J. Quante. On dynamic feature location. In *Proc. of the 20th Int'l Conf. on Automated Software Engineering*, pages 420–432, 2005.

[10] D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In *Proc. of the 22nd Int'l Conf. on Automated Software Engineering*, pages 234–243, 2007.

[11] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman. Jungloid mining: helping to navigate the api jungle. In *Proc. of the Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 48–61, 2005.

[12] J. Nykaza, R. Messinger, F. Boehme, C. L. Norman, M. Mace, and M. Gordon. What programmers really want: results of a needs assessment for sdk documentation. In *Proc. of the 20th Int'l Conf. on Computer Documentation*, pages 133–141, 2002.

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

[14] M. P. Robillard and G. C. Murphy. Representing concerns in source code. *ACM Transactions on Software Engineering and Methodology*, 16(1):3, 2007.

[15] N. Sahavechaphan and K. Claypool. Xsnippet: mining for sample code. In *Proc. of the 21st Conf. on Object-Oriented Programming Systems, Languages, and Applications*, pages 413–430, 2006.

[16] V. Sundaresan, P. Lam, E. Gagnon, R. Vallée-Rai, L. Hendren, and P. Co. Soot - a java optimization framework. In *Proc. of CASCON*, pages 125–135, 1999.

[17] P. Tarr, H. Ossher, W. Harrison, and J. Stanley M. Sutton. N degrees of separation: multi-dimensional separation of concerns. In *Proc. of the 21st Int'l Conf. on Software Engineering*, pages 107–119, 1999.

[18] S. Thummalapenta and T. Xie. PARSEWeb: A programmer assistant for reusing open source code on the web. In *Proc. of the 22nd Int'l Conf. on Automated Software Engineering*, pages 204–213, 2007.

[19] Z. Xing and E. Stroulia. UMLDiff: an algorithm for object-oriented design differencing. In *Proc. of the Int'l Conf. on Automated Software Engineering*, pages 54–65, 2005.

[20] W. Zhao, L. Zhang, Y. Liu, J. Sun, and F. Yang. SNIAFL: Towards a static non-interactive approach to feature location. In *Proc. of the 26th Int'l Conf. on Software Engineering*, pages 293–303, 2004.