Mining Subclassing Directives to Improve Framework Reuse

Marcel Bruch Mira Mezini Martin Monperrus
Darmstadt University of Technology
{bruch,mezini,monperrus}@st.informatik.tu-darmstadt.de

***Abstract*—To help developers in using frameworks, good documentation is crucial. However, it is a challenge to create high quality documentation especially of hotspots in white-box frameworks. This paper presents an approach to documentation of object-oriented white-box frameworks which mines from client code four different kinds of documentation items, which we call subclassing directives. A case study on the Eclipse JFace user-interface framework shows that the approach can improve the state of API documentation w.r.t. subclassing directives.**

I. Introduction

Object-oriented frameworks are a great vehicle in supporting code reuse [11], [5]. White-box frameworks are those frameworks that use inheritance, i.e. application-specific code consists of subclasses of framework classes [18]. White-box frameworks, while very flexible, are difficult to learn and use [6], [15], [14]. For example, instantiating the JFace white-box framework[1] to program a user interface requires that the developer identifies the right classes to extend among 203 available public classes and that she correctly overrides methods among 20 overridable methods per class in average[2].

To help developers in using frameworks good documentation is crucial. However, it is a challenge to create high quality documentation for white-box frameworks [1], especially given the complexity of today's frameworks [8]. As a result, framework users often miss the correct piece of documentation as recently shown by Robillard [14].

The documentation of object-oriented frameworks may contain different kinds of information [5]: high-level description of the architecture (e.g., used design patterns and class diagrams), information about what the code does, code snippets, directives stating how to use framework classes or methods, etc. This paper focuses on directives stating how to use the framework. More specifically, we use the term ***"subclassing directive"*** to designate pieces of documentation related to how to subclass a framework class or how to override a framework method.

We present an approach to improve the quality of *"subclassing directives"* of white-box frameworks. The core idea is that subclassing directives can be reverse-engineered from application-specific code (called in this paper *client code*), i.e. that *how-to-use* documentation of a particular software artifact can be inferred from how it is actually used. Bloch [1] (Item 17, pp. 87-92) urges developers of extensible classes to test them by writing subclasses before delivering them, arguing that it is the actual extensions of a base class that reveal the relevant hotspots and not only those that are exposed and documented. This fits with our intuition that actual usages found in existing clients are a good source for mining subclassing directives.

These mined directives can be used by developers of new clients as a complementary source of information in addition to the API documentation delivered with the framework. They can also be used by framework developers who can peruse the list of inferred subclassing directives to eventually identify incorrect or incomplete documentation, to update the existing documentation, and improve the quality of framework subclassing directives.

An directive may be incorrect if its formulation clashes with actual usages. For instance, a method that is documented as *Subclasses may override* whereas 100% of client code do override it, is probably incorrect. Documentation is incomplete when some directives are not documented at all: for instance our approach finds directives of the form *Subclasses may override, but must call the super implementation* (100% of client code does so) which are not documented in the API documentation.

More specifically, our contributions are as follows:

1) We propose four different kinds of subclassing directives and present arguments for them. We show that they are complementary and that having just one or the other is not sufficient. We argue that when developers do not have this information they lose time in understanding how to use a framework or in solving a bug related to a violation of an undocumented directive.
2) We present an approach to mine framework subclassing directives from client code. We present one mining technique per proposed subclassing directives. Three of these techniques are based on metrics gathered from client code, the fourth one is based on a machine learning clustering algorithm.
3) We present a case-study to validate the proposed approach. The subject of the case study is the Eclipse JFace framework, a powerful, open-source, and industry-proven framework developed by IBM. This case study shows that our approach improves both the correctness and the completeness of subclassing directives present

[1]JFace grounds the Eclipse IDE.
[2]For the source of these numbers see section IV.

In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR'2010)

in API documentation.

4) We present a tool, called Core, that implements the proposed approach. Core presents the mined subclassing directives as an extension to the API documentation in the Eclipse IDE for Java. It is publicly available at http://www.stg.tu-darmstadt.de/research/core/.

The reminder of this paper is structured as follows. Section II presents four kinds of subclassing directives. Section III presents techniques to mine each of them from client code. The case study evaluating the approach is presented in section IV. Then, section V presents the integration of the approach into the Eclipse IDE. Section VI discusses related work and section VII concludes the paper.

II. Four Kinds of Subclassing Directives

A subclassing directive is a piece of documentation stating how to subclass a framework class. They can be of different kinds. In this paper, we claim that the documentation of white-box frameworks requires four types of subclassing directives. In the following, we define them and give rationales for having each of them by discussing possible consequences if they are missing or if developers overlook them.

A. Method Overriding Directives

To instantiate a white-box framework, the developers need to know which framework classes are designed to be subclassed and which methods therein are designed to be overridden in an application-specific manner. A method overriding directive is a piece of documentation stating whether a framework method is designed to be overridden by client code. A class is documented as designed for subclassing if it contains at least one method overriding directive.

Method overriding directives are part of Johnson's patterns to document frameworks [6], as shown by the following excerpt:

> *Each drawing element in a HotDraw application is a subclass of Figure, and must implement displayOn, origin, extent, and translateBy.*

In certain programming languages, some method overriding directives are enforced by the language itself. For instance, in Java, the `abstract` modifier for methods forces subclasses to override it, and the keyword `final` forces subclasses to use the framework implementation of the method.

Method overriding directives can be found in the API documentation of framework (an example is given in figure 1).

B. Method Extension Directives

A method extension directive is a piece of documentation stating whether a method overriding a framework method should call the super-implementation. If the client-specific implementation of a framework method does not call the super-implementation when required, it may violate internal framework protocols resulting in runtime problems.

For instance, if a programmer does not call the super implementation of the method `Dialog.close` of JFace when overriding it, she gets a unclosable window which totally hangs the application. Also, she does not get a stack trace to localize the error. This shows the importance of having documented method extension directives.

```
/**
  * Creates the control for the
  * tool bar manager.
  * Subclasses may override this method
  * to customize the tool bar manager.
  */
```

Fig. 1. API Documentation containing a method overriding directive (`ApplicationWindow.createToolBarControl` of Eclipse JFace)

To homogenize these directives, the programmers of Eclipse published a guideline to explain how to document them [4]. Programmers should use one of the following expressions: *subclasses may extend this method* or *subclasses may re-implement this method*. Extending means that subclasses have to call the super-implementation. Re-implementing means that subclasses must not call the super-implementation. Note that both directives are not supported by Java modifiers hence they have to be in the API documentation. Also, by default, a directive of the form *"Subclasses may override this method"* means that subclasses may or may not call the super-implementation.

C. Method Call Directives

Although the hotspot overriding is fully application-specific, frameworks may have expectations in terms of framework methods that should be called by the overriding code. A method call directive is associated to an overridable framework method and states which methods should be called inside client implementations of this framework method.

For illustration, consider an Eclipse JFace wizard page that creates some visual components to display information. The `createContents()` method is the place where to create application-specific components. The framework expects the developer to call `WizardPage.setControl()` somewhere in the body of the overriding `createContents()` in order to register the application-specific control. Omitting this call leads to a cryptic runtime error (*"Assertion failed"*) and no stack trace to localize the error.

Even if it's not explicit in Johnson's description of documentation patterns [6], there are such directives in the real patterns he presents:

> *However, methods that change some attribute of a figure must notify the objects that depend on it. This is done by sending the `willChange` message to itself before changing the attribute, and sending the `changed` message to itself afterwards.*

D. Class Extension Scenarii

The directives discussed so far concern individual methods. However, just giving a novice user of a white-box framework a "flat" set of methods that could be overridden leaves her

with many open questions. Which methods should she actually override? The methods with a computed likelihood higher than a threshold? Or, are there other togetherness criteria determining "typical units of co-overridden methods"? The criteria to choose subsets of methods to override together may depend on the framework and on the class.

This is the rationale for defining a *class extension scenario*[3] as a set of typically co-overridden methods. Class extension scenarii are ready-to-use. Developers who have never extended a particular class, can rely on them to choose the set of methods to override together. Without them, developers lose time in answering the boundary problem aforementioned.

The Eclipse website has a section containing tutorial articles. Some of them are about how to typically subclass framework classes (e.g. the tutorial about `PreferencePage`[4]).

E. Importance Level

There is a concern which crosscuts the four aforementioned directives: whether the directive is a strong requirement (e.g. *Subclass must call the super implementation*) or a weak one (e.g. *This method may be overridden*). We can identify two ways of indicating importance levels.

First, they can be expressed with modal verbs (e.g. may, should, must, etc.). This approach fits well with natural language and fuzzy expert knowledge. Johnson [6] uses them, and the Eclipse guidelines for subclassing directives as well [4]. Second, previous work about the mining of subclassing directives [13], [22], [20] generally uses importance values (e.g. probability).

F. Recapitulation

We have presented four important subclassing directives (i.e. kinds of documentation required to extend a framework). For each of them, to attest their usefulness: a) we explained what may happen if they are missing and b) we showed that real world documentation already contains some of them.

III. Techniques to Mine Subclassing Directives

We now define techniques to mine subclassing directives from client code of white-box frameworks. Mined directives can be used by framework developers to improve the quality of the documentation and by framework users to find the pieces of information they need to correctly use the framework, thus complementing the documentation delivered with the framework.

A. Mining Overriding Directives

To determine the likelihood that a method is designed for being overridden, we define a metric called *ovLikelihood*, which represents the importance of overriding a framework method. A method overriding directive is created for each method whose value of *ovLikelihood* is not null. In the following, we give its definition and illustrate that it is meaningful

[3]In this paper, we will use the italian plural form, *scenarii*, of this italian word.

[4]http://www.eclipse.org/articles/Article-Preferences/preferences.htm

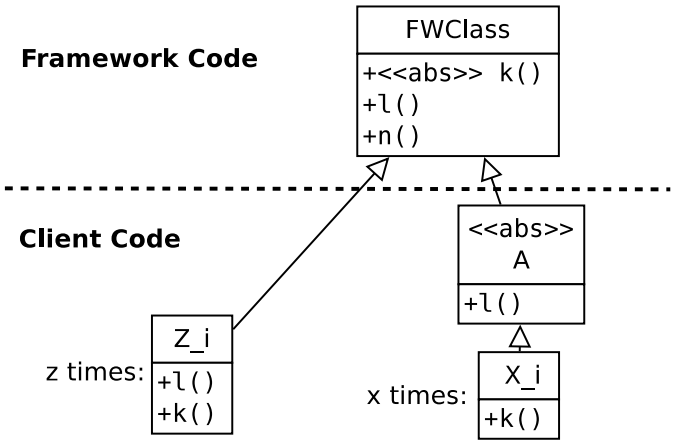

Fig. 2. An Example to Assess the Correctness of the Metric *ovLikelihood*

by considering cases where we know the likelihood value. The definition of *ovLikelihood* is as follows:

$$ovLikelihood(fwMeth) = \frac{\sum_{clCl} ov_{clCl,fwMeth}}{\sum_{clCl} ov_{clCl,fwMeth} + \sum_{clCl} not_{clCl,fwMeth}}$$

Thereby, for any framework class `fwCl`, framework method `fwMeth`, and non-abstract client subclass `clCl`: $ov_{clCl,fwMeth} = 1$, if `clCl` overrides `fwMeth` directly or inherits an overriding implementation from an intermediate client superclass; $not_{clCl,fwMeth} = 1$, if `clCl` does not override `fwMethod` at all.

To understand the properties of this metric, let us now consider the class diagram depicted in figure 2. `FWClass` is a framework class, `A` is a class in client code that overrides `FWClass` and that it abstract. There is also z concrete classes $(Z_1 \dots Z_n)$ that override `FWClass` and x concrete classes $(X_1 \dots X_n)$ that override `A`.

Method `k` is an abstract framework method that *must* be overridden. Hence, its value has to be 100% ($ovLikelihood_k = \frac{x+z}{x+z} = 100\%$) Method k also illustrates the rationale of discarding abstract client classes from the counting: if we do not discard them, then $ovLikelihood_k = \frac{x+z}{x+z+1} < 100\%$ which is incorrect.

Even if l is overridden in an intermediate abstract client class A, it counts as actually overridden in all concrete subclasses of A; hence the value of $ovLikelihood_l$ must be 100% and this is indeed what our metric calculates ($ovLikelihood_l = \frac{x+z}{x+z} = 100\%$).

Method n is never overridden, hence the value of $ovLikelihood_n$ must be 0% ($ovLikelihood_n = \frac{0}{0+x+z} = 0\%$).

B. Mining Extension Directives

To mine extension directives, we propose a metric that counts the number of methods that override a framework

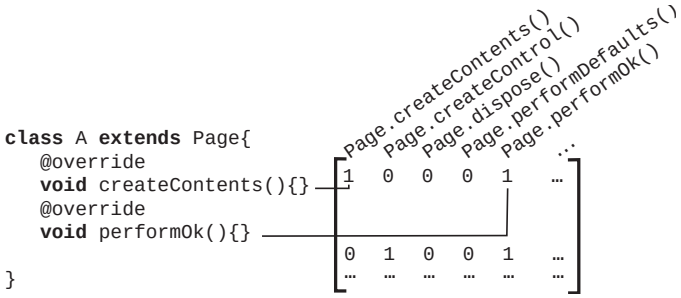

Fig. 3. Mapping Code to a Binary Space to Mine Extension Scenarii with LCA

method and call the super-implementation (i.e., extend the framework method):

$$exLikelihood(fwMeth) = \frac{\sum_{clCl} super_{clMeth,fwMeth}}{\sum_{clCl} ov_{clCl,fwMeth}}$$

where $ov_{clCl,fwMeth} = 1$, if `clCl` contains a method `clMeth` which overrides `fwMeth`; $super_{clCl,fwMeth} = 1$, if `clCl` contains a method `fwMeth` which overrides `fwMeth` and call the super implementation.

A method extension directive is created for each method whose value of *exLikelihood* is not null.

C. Mining Call Directives

To mine a call directive for a framework method `fwMeth`, we propose to collect all self-calls executed within the control flow starting from each method overriding `fwMeth`. The following defines a metric which represents the importance of calling a framework method `fwMeth2` in the control flow of another framework method `fwMeth1`

$$clLikelihood(fwMeth1, fwMeth2) = \frac{\sum_{clCl} call_{clCl,fwMeth1,fwMeth2}}{\sum_{clCl} ov_{clCl,fwMeth}}$$

where $ov_{clCl,fwMeth} = 1$, if `clCl` contains a method which overrides `fwMeth`; $call_{clCl,fwMeth1,fwMeth2} = 1$, if `clCl` contains a method which overrides `fwMeth1` and calls `fwMeth2` in its control flow.

A call directive is created for each pair of framework methods, whose value of *clLikelihood* is not null.

D. Mining Class Extension Scenarii

We propose to use a clustering algorithm on client code to mine the set of methods commonly overridden together. For each framework class `fwCl`, one selects the client classes that subclass `fwCl` and clusters the methods that are often overridden together. As with other subclassing directives, mining existing clients enables to discover extension scenarii that have not been covered by tutorials.

For each framework class `fwCl` a binary matrix is build. Each row of the matrix represents a subclass of `fwCl`; each column represents an overridable method of `fwCl`. Whenever a subclass `i` overrides a framework method `j` the position `(i,j)` of the binary matrix is 1, it is 0 otherwise. For illustration, figure 3 shows the binary matrix of a class `Page` and elaborates on the row for a subclass `A`, whose code is shown on the left-hand side.

Since the data grounding the clustering algorithm is binary, we use a data mining algorithm called *Latent Class Analysis (LCA)* appropriate to such binary data [10]. For each framework class, the algorithm outputs zero or more class extension scenarii. We define a default extension scenario as a scenario which covers at least 5 percent of the data, a heuristic which gives satisfying results according to our experience. If several scenarii satisfy these constraints, the default scenario is simply the one that has the greatest probability (as given by LCA).

E. Defining Importance Levels

As discussed in II-E, the importance level of each directive can be represented by modal verbs or importance values. We propose to give the programmers both, for instance, *Subclass may override this method (32%)*. Our rationales are 1) modal verbs are intuitive and accessible to novice users and 2) importance values are useful for users who know how to interpret them.

We propose the following heuristics to map importance values to modal: 100%→MUST; 80%-100%→ SHOULD, 20%-80%→ MAY; 1%-20%→ RARELY. It seems satisfactory according to our own experience and according to the users of the tool. Validating them by a controlled user study is left out of the scope of this paper and one of the areas for future work.

F. Implementation

We have implemented a static analysis for each technique described above. The analyses target Java bytecode and use the Wala bytecode toolkit[5]. The implementations of *ovLikelihood* and *exLikelihood* are simple countings on a direct representation of the code. The implementation of *clLikelihood* is based on call graphs. To cluster the co-overridden methods (the class extension scenarii), we have reused an implementation of LCA provided by NASA: Autoclass[6].

IV. Case-Study: Improving the Documentation Quality of a Mature Framework

In this section, we evaluate whether our system is able to improve the quality of the API documentation of a real-world framework with respect to subclassing directives.

A. Set Up and Overview of the Results

The subject of the study is JFace, a white-box framework dedicated to user interfaces. It is a representative of heavily used frameworks: it grounds the Eclipse IDE, it is more than 6 years old, and it is used daily by hundreds of developers. So, one can expect its documentation to be already improved several time and hence of good quality. We postulate that if our approach is able to produce directives that complement

[5] http://wala.sf.net

[6] see http://ti.arc.nasa.gov/project/autoclass/.

JFace's documentation or to produce directives that are more precise, it can do so for documentation of less quality as well.

For the study, we compared the subclassing directives in the API documentation of JFace with directives that were automatically extracted by applying our system to client code of JFace. The client code used for the study consisted of 600 MB of mature available Eclipse plugins[7]. Since our codebase is composed of mature code only, we can consider that the extracted directives are correct by construction. Also, note that our implementation was thoroughly tested.

To obtain the list of subclassing directives that are already present in the documentation of JFace, we analyzed the documentation of JFace by performing the following process.

1) We wrote a trivial static analysis which counts the number of public classes and their respective overridable methods (i.e., public/protected and not final methods). We found a total of 203 public classes which have in average 20 overridable methods.
2) We analyzed the collected client code and listed those framework classes that are extended at least ten times. Altogether there were 45 such classes (i.e. approx. one fourth of public classes).
3) We read the Javadoc documentation of each framework method of the collected classes as well as the Javadoc documentation of the containing class and reported the overriding directives, whenever available. Altogether 632 methods were analyzed. We found a total of 237 documented directives.

Since step 3 is manual and error-prone, we performed them twice by two of the authors of this paper and consolidated the results.

Table I presents an overview of the results of comparing subclassing directives of the JFace API documentation with subclassing directives automatically mined by our system. The first part gives an overview of the documentation (of the 45 classes manually analyzed). The second part reports on agreeing documented and mined directives. The third part concerns disagreeing documented and mined directives. The fourth part is dedicated to documented directives that have no correspondence in the mined directives. Finally, the last row in the table concerns mined directives for which we could not find corresponding directives in the documentation.

One finding that is not reported in Table I but which we find interesting to emphasize is that only 30% of overridable methods are actually overridden. This is an empirical proof that the visibility modifiers alone are not sufficient as subclassing directives.

In the following subsections, we first elaborate on the findings reported in parts three, four, and five of Table I. Subsequently, we evaluate the mined extension scenarii. We conclude this section by presenting and discussing the viewpoint of Boris Bokowski, leader of the JFace development team at IBM Canada, further called *the expert*, who kindly agreed to comment a sample of directives mined by our system that we sent to him.

[7]For sake of replicability, the complete list of plugin ids and versions is available upon request.

Directive	#
Documented	**237**
Overriding Directive	153
Extension Directive	69
Call Directive	15
Agreeing Documented and Mined	**181**
Overriding Directive	138
Extension Directive	32
Call Directive	11
Disagreeing Documented and Mined	**45**
Overriding Directive	8
Extension Directive	37
Call Directive	0
Documented and Not Mined	**11**
Overriding Directive	7
Extension Directive	0
Call Directive	4
Not documented and Important in Actual Usages (importance $> 80\%$)	**129**
Overriding Directive	4
Extension Directive	67
Call Directive	58

TABLE I
IMPROVING THE SUBCLASSING DIRECTIVES OF THE JFACE FRAMEWORK

B. Disagreeing Documented and Mined Directives

In this section, we elaborate on the set of documented directives with corresponding mined directives, but with mismatching importance levels. This set breaks down as follows:

- There are 3 methods that are documented as *Subclasses must override* or *Subclasses should override* whereas the overriding frequency in client code is less than 40%. Along the same line, we found 2 *Subclasses must override*, while client code does not always follow them (importance values: 93% and 94%), this indicates a possible change from *must override* to *should override*.
- We found 3 methods that are documented as *Subclasses may override*, while their overriding importance in client code is greater than 80%. Those methods should be documented as *Subclasses should override*. Furthermore, two of them have an importance value of 100% (all client classes overridden them) which would literally mean a *Subclasses must override*. Since we could not state whether it is really a "must" contract or a particularity of our code base, we prefer generating a *Subclasses should override*.
- For extension directives, there are 9 directives of the form *Subclasses must extend* or *Subclasses should extend* in the documentation, while the actual extension frequency in the codebase is less than 80%.
- Further, there are 28 directives of the form *Subclasses may extend* in the documentation, while their mined importance is higher than 80%.

The discussion above reveals non-negligible discrepancy between the importance level of documented directives and importance levels deduced from existing clients. Given that

the codebase that we use consists of production-level client code, the mined directives reflect information that was actually needed at a point in time by the developers of the codebase. Hence, we consider the disagreeing documented directives misleading: Developers probably lose some of their valuable time in investigating and finding out that the misleading directives in the documentation are not useful for them. Hence, we conclude that the results reported in this sub-section show that our system is able to improve the correctness of the API documentation w.r.t. subclassing directives.

C. Documented and Not Mined Directives

There are documented subclassing directives of JFace which are never followed in client code. In the following, we present them and some possible explanations.

- There are 7 methods documented as *Subclasses may override*, which are never overridden in client code. It seems that JFace designers thought that it could be useful to make these methods overridable, but the reality somehow invalidated their assumptions. As emphasized by Bloch [1], it is really difficult to know a priori which hotspots to provide in a base class.
- There are no unused extension directives.
- There are 4 call directives which are never followed in client code.
  - One directive is expressed as *"Use removeAll for clean up references"*. We assume that the scope of this directive is not subclasses but external users of this class.
  - Two directives are expressed as *"Subclasses should call this method at appropriate times"*. The vagueness of the directive indicates that its author did not have a precise contract in mind.
  - One directive is expressed as *"This method is really only useful for subclasses to call in their constructor"*. As for overriding directives, this may be an incorrect guess of the method usages at design time of the framework.

D. Not Documented Important Directives

In this subsection, we discuss in more detail the mined directives with a high importance level that are missing in the API documentation. For each of them, we looked in the corresponding API documentation whether it is documented or not (in both the method-level and the class-level documentation). The results of this study are as follows:

- There are 4 overriding directives extracted from client code with an importance of greater than 80% which are missing in the documentation.
- The system extracts 50 *must* extension directives that are not documented (and 17 undocumented *Subclasses should call the super-implementation*). We assume that this kind of contracts is widely yet implicitly used by framework designers while not being integrated as a documentation best practice. Developers probably lose some of their valuable time in finding out that they are expected to call the super-implementation for these particular methods.
- There are 58 mined call directives with *clLikelihood* $> 80\%$ which are not documented. This clearly indicates that framework designers often use implicit call contracts while they (or framework documenters) often do not document them, or are not aware of their importance for the client code.

The observations just reported show that our system is able to improve the completeness of the API documentation w.r.t. subclassing directives.

E. Relevance of Class Extension Scenarii

We used internet tutorials about JFace to evaluate the relevance of class extension scenarii mined by our system. We carefully read the available tutorials (both text and code snippets) to extract what are the recommended methods to override, which together form a reference extension scenario. We then compared the reference scenarii with the mined ones. For instance, the official Eclipse tutorial *"Preferences in the Eclipse Workbench UI"*[8] explains how to subclass `PreferencePage` by overriding three methods (`createContents`, `performDefaults`, `performOk`). This reference scenario perfectly matches the mined one for `PreferencePage`.

In all we analyzed 31 different tutorials from the IBM developer website[9], the Eclipse website [10] and Javaworld [11]. We found 14 tutorials that contain 25 different reference extension scenarii for JFace.

Table II presents the results of this evaluation. Each row in the table is about a different JFace class. The first column shows the name of the class and the tutorials that were used for finding reference scenarii for that class. The second column shows the mined extension scenarii in italics (there could be several mined and reference scenarii per framework classes) followed by different reference scenarii. The third column indicates whether the reference scenarii match the mined scenarii.

A quantitative summary of the data in the table II is as follows: a) 18 mined scenarii perfectly match the reference. a) 5 mined scenarii partly match the reference counterparts (they are either superset or subset of overridden methods); b) 2 reference scenarii do not have a mined counterpart;

We further made the following qualitative observations:

- We expected to find more informal tutorials from technology guru's blogs or community-based sites. We were surprised to find so many reference scenarii in the Eclipse and IBM websites. This indicates that authoritative sources (Eclipse and IBM) consider extension scenarii as an important documentation artifact.
- However, these authoritative tutorials cover only 10 classes of JFace. There are many more of importance

[8] http://www.eclipse.org/articles/Article-Preferences/preferences.htm
[9] http://www.ibm.com/developerworks
[10] http://www.eclipse.org/articles/
[11] provides "Solutions for Java developers", see http://www.javaworld.com

Context, Tutorial Title and Source	*Mined (in italic)* & Reference Scenario	Evaluation
Class `Wizard`	*addPages, performFinish*	
Extending the Generic Workbench (IBM developerWorks)	addPages,performFinish	OK
Creating JFace Wizards (Eclipse)	addPages, performFinish, canFinish (opt),	OK
Class `WizardPage`	*createControl*	
Extending the Generic Workbench (IBM developerWorks)	createControl	OK
Customizing Eclipse RCP applications (IBM developerWorks)	createControl	OK
Creating JFace Wizards (Eclipse)	createControl, canFlipToNextPage, getNextPage (opt)	x
Class `LabelProvider`	*getImage, getText*	
Using Images in the Eclipse UI (Eclipse)	getText, dispose	x
Using the Jface Image Registry (IBM developerWorks)	getImage, getText	OK
Using the Eclipse GUI outside the Eclipse Workbench, Part 1: Using JFace and SWT in stand-alone mode (IBM developerWorks)	getText	x
Class `ViewerSorter`	*pattern1: category; pattern2: compare*	
Using the Jface Image Registry (IBM developerWorks)	category	OK
How to use the JFace Tree Viewer (Eclipse)	category	OK
How to use the JFace Tree Viewer (Eclipse)	compare	OK
Building and delivering a table editor with SWT/JFace (Eclipse)	compare	OK
Class `ViewerFilter`	*select*	
Using the Jface Image Registry (IBM developerWorks)	select	OK
Customizing Eclipse RCP applications (IBM developerWorks)	select	OK
How to use the JFace Tree Viewer (Eclipse)	select	OK
Class `Action`	*run*	
Rich clients with the SWT and JFace (JavaWorld)	run	OK
Creating an Eclipse View (Eclipse)	run	OK
Class `DialogCellEditor`	-	
Take Control of Your Properties (Eclipse)	openDialogBox	not enough subclasses to mine a pattern
Extending The Visual Editor: Enabling support for a custom widget (Eclipse)	openDialogBox, doSetValue, updateContents	not enough subclasses to mine a pattern
Class `PreferencePage`	*createContents, performOk, performDefaults*	
Preferences in the Eclipse Workbench UI	createContents, performDefaults, performOk	OK
Simplifying Preference Pages with Field Editors (Eclipse)	createContents, performOk, performDefaults	OK
Class `FieldEditorPreferencePage`	*createFieldEditors*	
Simplifying Preference Pages with Field Editors (Eclipse)	createFieldEditors	OK
Mutatis mutandis - Using Preference Pages as Property Pages (Eclipse)	addField, performOK, createContent	x
Mutatis mutandis - Using Preference Pages as Property Pages (Eclipse)	createFieldEditors	OK
Class `TitleAreaDialog`	*createDialogArea, okPressed, configureShell, createButtonsForButton, createContents*	
Extending The Visual Editor: Enabling support for a custom widget (Eclipse)	createContents, createDialogArea	x

TABLE II
REFERENCE EXTENSION SCENARII TO EVALUATE THE VALIDITY OF MINED ONES (IN ITALIC)

which are not documented by an extension scenario. This is exactly where our approach makes its contribution, by providing default extension scenarii when no others are available.

F. The Expert Viewpoint

To get an unbiased view on the observed differences between mined and documented directives, we asked Boris Bokowski, leader of the JFace development team at IBM Canada, to comment on a sample of mined directives, which we formulated as suggestions of change to the API documentation. The size of the sample (9 items) was chosen so that the questionnaire can be answered in less than 20 minutes. The directives we included in the sample were selected according to the following characteristics: 1) they are related to an important and known class of JFace so as to be sure that the expert is fluent with this class 2) they have a very high importance value so as to reflect the added value of using our approach to find important incorrect or missing directives. In

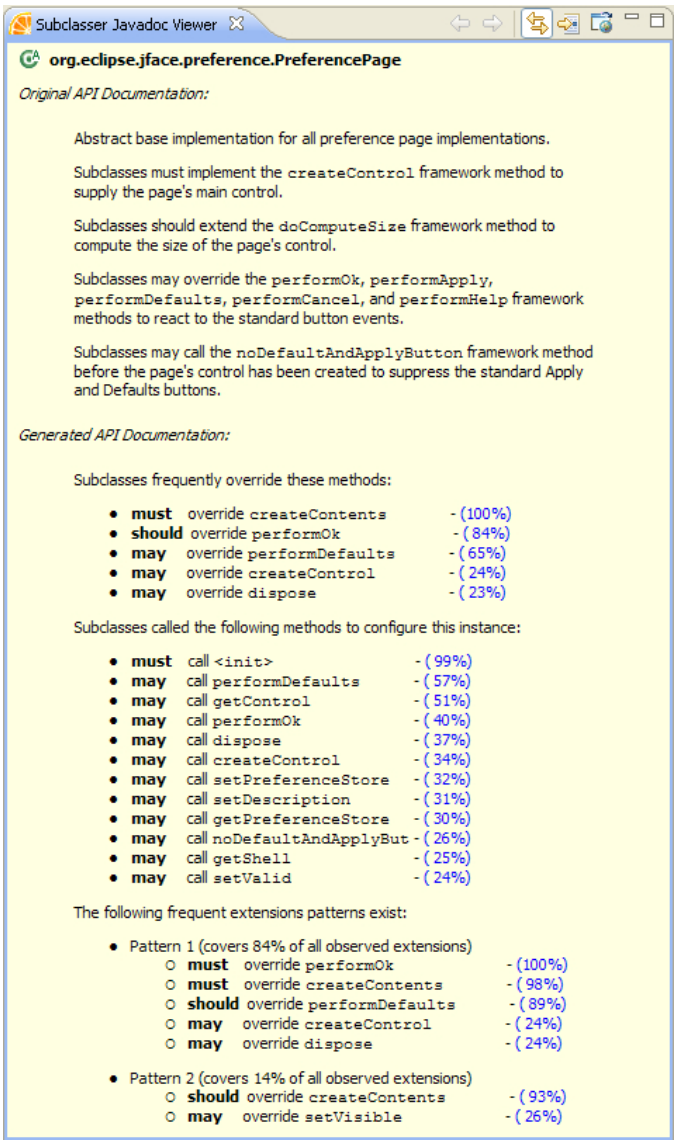

Fig. 4. Class-level API Documentation Extended with Automatically Mined Subclassing Directives.

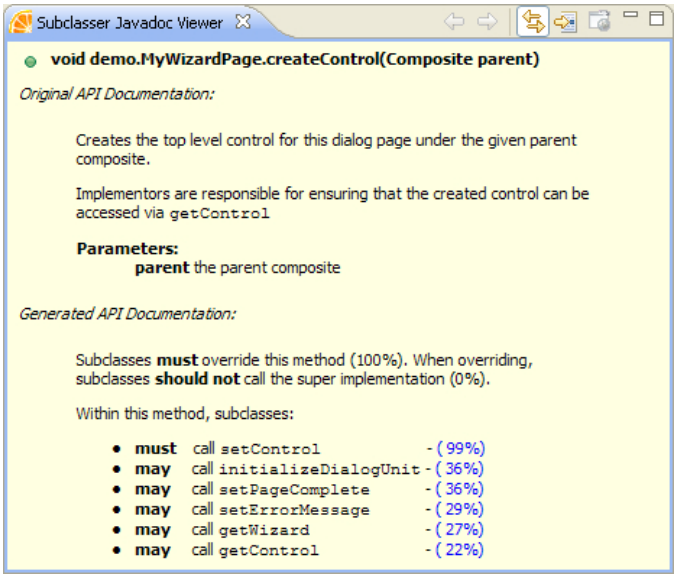

Fig. 5. Method-level API Documentation Extended with Automatically Mined Subclassing Directives.

the following, we summarize the feedback that we got from the expert.

The expert agreed on two suggestions to change two *"Subclasses may override"* directives in the API documentation to *"Subclasses should override"* directives, as mined by our system. He asked us to fill respective entries in the bug repository of Eclipse, which we did[12]. For illustration, we elaborate on one of these changes. The API documentation states for `PreferencePage.performOK` that *Subclasses may override*. However, since our algorithm found out that the method was actually overridden in 84% of the client subclasses, we suggested to change the directive to *Subclasses should override*.

We sent the expert three suggestions for changes concerning three different *"Subclasses may extend"* directives found in the documentation. According to the Eclipse guidelines this actually means *"may override, should extend"*. Our system mined three different directives: (1) *"may override, must extend"*, (2) *"should override, may extend"*, and (3) *"may override, should extend"* for the respective three methods. The expert basically agreed that the first two suggestions were meaningful. Yet, he argued for not performing the first change we suggested as "it would render existing code incompatible with the specification". He disagreed on the third suggestion arguing that our suggestion was merely making explicit the actual meaning of *"Subclasses may extend"* according to the Eclipse guidelines. While this is somehow a matter of taste, we still find that dissociating overriding and extension directives makes the directives more clear. The fact that we mined three different interpretations of the *"Subclasses may extend"* directive, which the expert principally agreed on, indicates that simply stating *"Subclasses may extend"* is very misleading.

The expert reviewed an extension directive with a very high importance (i.e. *Subclasses should call super*), for which he answers *"I don't see why we would require subclasses to call the super implementation"*. We analyzed the client classes that support this mined directive. It turned out that 1) the support of this directive is low (11) and 2) the clients override this method just to add some application-specific logic, which makes sense to be called at this point in the framework control-flow, but which is not related to the initial intent of this method. This motivated us to set up a higher filter on the support of each directive in newer versions of the tool.

The expert reviewed 2 mined method call directives. Unfortunately, due to the specific directives selected for review and some unclarity in the expert's comments, we can not derive any generalizable conclusions.

The expert reviewed 1 mined default extension scenario.

[12]cf. bugs # 288461 and # 288462, https://bugs.eclipse.org/bugs/show_bug.cgi?id=288461

It is about extending the class `TrayDialog` which extends `Dialog`. While the documentation of `Dialog` related to subclassing is quite comprehensive, the subclassing documentation of `TrayDialog` is really scarce. The mined default extension scenario for `TrayDialog` consists of 11 methods to override. The expert answered that *"'documentation items like this are useful*. In addition, he noted that they are *"probably better suited for a tutorial rather than the API specification."*.

The second part of the expert comment raises questions about where to put subclassing directives. In API documentation? In tutorials? Should we duplicate documentation at the class level documentation and at the method level? How to handle documentation that targets different kinds of users (novice versus experts), etc.? A thorough investigation of these questions is out of scope of this paper. For now, our answer to these questions is that subclassing directives should be integral part of the API documentation in a way that we elaborate on in the next section. Further investigations involving user studies are out of the scope of this paper.

V. Integrating Subclassing Directives in IDEs

Subclassing directives are of primary importance for framework users and hence should be tightly integrated into the development environments so that developers find and use them easily. In the following, we present an integration proposal of our mined subclassing directives into the Eclipse IDE.

First, the initial documentation written by the framework developers is the primary source of information. It contains a lot of valuable information , e.g., the functional goal of a method, its design rationales, etc. The mined directives complement the original API documentation. Second, since there is already a Javadoc viewer in every default Eclipse installation and that developers may already use it, we propose to extend this viewer by enriching the initial API documentation with the mined subclassing directives (and not to create a new view). Finally, since mined directives are relevant at both the class level and the method level API documentation, the extended viewer supports both.

When developers browse the class level documentation, they are shown the list of overriding directives, the list of method call directives, and the extension scenarii that have a high support. Directives are shown in natural language (e.g., *Subclasses may override*) together with the underlying importance value. Figure 4 shows a screenshot of our viewer displaying the API documentation (both hand-written and generated directives) for the class `PreferencePage` of the JFace framework.

When developers browse the method level documentation, the documentation view acts differently. It gives the initial documentation, and in addition adds the mined overriding directive, the extension directive, and the method call directives, if any. Figure 5 shows an example of the method level documentation. Note that both screenshots show directives for real classes of JFace and that real data underlies them.

This integration is seamless with respect to the IDE and the usual way of programming with Eclipse. Developers have new information at the place they naturally would look at: the Javadoc viewer. If no information is available for certain classes or certain frameworks, the viewer is simply the default one and shows only the initial API documentation.

VI. Related work

For Pree, *hotspot mining* is a manual activity that consists of examining maintenance data, investigating framework use cases and questioning the framework experts. On the contrary, our approach to hotspot mining is fully automated.

Schauer et al. [17] presented a method and a tool to automatically recover hotspots from C++ code. In particular, their approach is able to differentiate between *inheritance hotspots* (IHS - based on the design pattern *Inheritance Template Method*) and *composition hotspots* (CHS - based on the design pattern *Composition Template Method*). Their approach is based on the source code analysis of the framework. On the contrary, our approach is based on instantiation code, which enables us to have concrete information about how the framework under study is actually used. From a functional viewpoint, our approach provides more information to the user, namely the extension and the call directives.

Tourwe and Mens [21] address the problem of framework reuse by making explicit framework contracts by so-called *metapatterns*. This approach requires (a) a development environment that supports metapatterns, and (b) that framework designers enrich their code with the formal description of the metapatterns. Along the same lines, previous approaches [7], [9], [12], [19] address the framework evolution problem and propose different techniques for specifying, checking, and enforcing explicit specialization interfaces for frameworks. On the contrary, our approach does not require additional work from framework designers, who can use the tool to improve the current state of documentation.

Schaefer et al. [16] address the problem of framework evolution, i.e., how client code should change in response to a new version of the framework. Like with our approach, they mine instantiation code to mine knowledge. However, the problem scope is different: Their tool produces a list of changes to do in order to adapt existing client code to changes in the framework API, our tool produces a list of subclassing directives to support the development of new framework clients.

Michail [13] proposes an approach to mining subclassing directives based on association rules mining. There are two important differences between this proposal and ours. First, the proposal by Michail [13] does not address extension directives. Further, it provides coarse-grained call directives, at the class level, i.e., of the form *"call a framework method when extending a framework class"*. In previous work [2], we also presented an approach that provides the same coarse-grained call directives. On the contrary, our call directives are context-dependent. We tell the user that she has to call method X inside the body of method Y: this is very important to create code that respects the framework internal control flow.

	[3] AAQ'1998	[13] ICSE'2000	[22] FSE'2003	[2] ETX'2006	[20] ASE'2008	Our approach MSR'2010
Data	Smalltalk	C++	Java source code	Java bytecode	Google CodeSearch	Java Bytecode
IDE Integration Proposal						+
Overriding Directive	+	+	+	+	+	+
Extension Directive						+
Call Directive		-		-		+
Extension Scenario		+				+

TABLE III
PREVIOUS WORK DOES NOT ADDRESS FULL WHITE-BOX REUSE (+ STANDS FOR "SUPPORTS", - FOR "PARTIALLY SUPPORTS")

Demeyer [3], Viljamaa [22] and Thummalapenta et al. [20] also describe mining-based approaches to hotspot detection. Table III compares our contributions compared to these close papers including [13] and [2]. Programming with white-box frameworks requires all subclassing directives together since having just one or the other is not sufficient. We contribute with a comprehensive and unified approach on mining subclassing directives and especially we provide extension directives and well-scoped call directives: both document essential parts of the logic of today's white-box frameworks.

VII. CONCLUSION

We presented a new approach to improve the quality of white-box framework documentation. For each framework class, our system mines from client code a set of subclassing directives that are all required to quickly and correctly extend the framework: 1) what methods to override 2) should the overriding method call the super implementation 3) is the overriding method expected to call certain framework methods, and 4) what are the methods usually overridden together. This approach is complementary to manually written API documentation. Framework developers can update the documentation accordingly and framework users can access to the mined directives as an add-on to the original documentation in the IDE.

A case study evaluates the approach. We mined subclassing directives for the Eclipse's JFace framework for user-interfaces. We found that the existing API documentation is both incorrect (45 incorrect pieces of documentation) and incomplete (129 missing high-importance directives) compared to current usages of JFace.

Our current work goes in two directions. First we try to automate the analysis of existing documentation with natural language processing techniques. Then, it will be possible to automatically detect erroneous and missing documentation related to subclassing. Second, when developers are given a particular directive, they still need the intent behind the directive (e.g. *Subclasses may override to change the color of the widget*). We are studying how to mine not only the directive, but also the rationale behind each subclassing directive, based on e.g. framework code, symbol names and existing documentation (at least to the extent possible); we imagine illustrating such mined intents with generated code snippets.

REFERENCES

[1] J. Bloch. *Effective Java (second edition)*. Addisson-Wesley, 2008.
[2] M. Bruch, T. Schäfer, and M. Mezini. FrUiT: IDE support for framework understanding. In *OOPSLA Workshop Eclipse Technology Exchange*, pages 55–59. ACM, 2006.
[3] S. Demeyer. Analysis of overriden methods to infer hot spots. In *Proceedings of the Workshop on Techniques, Tools and Formalisms for Capturing and Assessing the Architectural Quality in Object-Oriented Software colocated with ECOOP'98*, 1998.
[4] J. des Rivières. How to use the eclipse api. Technical report, Eclipse Foundation, 2001.
[5] M. Fayad, D. Schmidt, and R. Johnson. *Building application frameworks: object-oriented foundations of framework design*. Wiley, 1999.
[6] R. Johnson. Documenting frameworks using patterns. In *Proceedings of Object-oriented programming systems, languages, and applications (OOPSLA'1992)*, page 76. ACM, 1992.
[7] G. Kiczales and J. Lamping. Issues in the design and specification of class libraries. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1992.
[8] D. Kirk, M. Roper, and M. Wood. Identifying and addressing problems in object-oriented framework reuse. *Empirical Software Engineering*, 12(3):243–274, 2007.
[9] J. Lamping. Typing the specialization interface. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1993.
[10] P. Lazarsfeld and N. Henry. *Latent structure analysis*. Houghton, Mifflin, 1968.
[11] T. Lewis. *Object-oriented application frameworks*. Manning Publications Co. Greenwich, CT, USA, 1995.
[12] M. Mezini. Maintaining the consistency of class libraries during their evolution. In *Proceedings of the Conference on Object-oriented Programming, Systems, Languages, and Applications*, 1997.
[13] A. Michail. Data mining library reuse patterns using generalized association rules. In *ICSE*, pages 167–176. ACM, 2000.
[14] M. Robillard. What makes apis hard to learn? answers from developers. *IEEE Software*, 26(6):27–34, 2009.
[15] C. Scaffidi. Why are apis difficult to learn and use? *Crossroads*, 12(4):4, 2006.
[16] T. Schaefer, J. Jonas, and M. Mezini. Mining framework usage changes from instantiation code. In *Proceedings of the 30th international conference on software engineering*, pages 471–480, 2008.
[17] R. Schauer, S. Robitaille, F. Martel, and R. Keller. Hot spot recovery in object-oriented software with inheritance and composition template methods. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'99)*, pages 220–229, 1999.
[18] H.-A. Schmid. Systematic framework design by generalization. *Commun. ACM*, 40(10):48–51, 1997.
[19] P. Steyaert, C. Lucas, K. Mens, and T. D'Hondt. Reuse contracts: managing the evolution of reusable assets. In *Proceedings of OOPSLA*, 1996.
[20] S. Thummalapenta and T. Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Procedings of ASE'2008*, pages 327–336, 2008.
[21] T. Tourwe and T. Mens. Automated support for framework-based software evolution. In *Proceedings of the International Conference on Software Maintenance (ICSM'2003)*, pages 148–157, 2003.
[22] J. Viljamaa. Reverse engineering framework reuse interfaces. In *Proceedings of ESEC/FSE*, pages 217–226, 2003.

In: Proceedings of the 7th IEEE Working Conference on Mining Software Repositories (MSR'2010)
This preprint is provided by the contributing authors to ensure timely dissemination of scholarly and technical work.