FrUiT: IDE Support for Framework Understanding

Marcel Bruch Thorsten Schäfer Mira Mezini
Software Technology Group
Department of Computer Science
Darmstadt University of Technology
{bruch,schaefer,mezini}@st.informatik.tu-darmstadt.de

ABSTRACT

Frameworks provide means to reuse existing design and functionality, but first require developers to understand how to use them. Learning the correct usage of a framework can be difficult due to the large number of rules to obey and the complex collaborations between the classes. We propose the use of data mining techniques to extract reuse patterns from existing framework instantiations. Based on these patterns, suggestions about other relevant parts of the framework are presented to novice users in a context-dependent manner. We have built FrUiT, an Eclipse plug-in that implements this approach and present a first assessment by mining parts of the Eclipse framework.

1. INTRODUCTION

Software reuse is one of the major goals in software engineering. It provides several benefits to developers such as reduced costs, higher quality, and shorter time to market. Several approaches to reuse exist, one of which are object-oriented frameworks. A framework is a “set of cooperating classes that makes up a reusable design for a specific class of software” [2]. By extracting the design into abstract classes and defining their responsibilities and collaborations, frameworks not only enable reusing functionality at code level, but also enable reuse at the design level.

Unfortunately, framework reuse suffers from the disadvantage that it is sometimes difficult to master them. Especially novice users of a framework require a lengthy learning process [7]. To support a wide range of specific applications, frameworks are designed with flexibility in mind. This involves more abstract classes and complicates the understanding process. Further, developers need to know the design of a framework. It is not sufficient to comprehend a single class, but one also needs to understand its collaborations.

Even though several approaches exist to give developers an understanding of how to use a framework correctly, they have some disadvantages. Some techniques (e.g., tutorials or cookbooks) require the creation of additional artifacts which can be very costly. In most cases, framework developers are expert programmers and do not have the time (and are not keen on) writing appropriate documentation. Hence, even when such documentation artifacts exist, they are often outdated [4]. Other approaches focus on using artifacts that do not have to be created additionally. For instance, the framework code itself can be explored [1] or existing instantiations can serve as examples [6]. But, following such approaches, users often need to read large portions of code which is not related to the understanding of the framework's usage, e.g., internal framework code or code specific to a single framework instantiation. Further, developers have to search for appropriate framework or example code on their own. Given the large size of frameworks and the large number of instantiations, this is a difficult task.

Summarizing, techniques are needed that

- do not involve a large additional effort for the framework developer,
- present only those parts needed to understand how to use the framework, and
- show relevant information in a context-dependent manner.

To satisfy these requirements, we propose to combine the use of data mining techniques [5] with a context-dependent presentation [3]. Based on existing framework instantiations, frequently occurring reuse rules are extracted and stored in a database. Framework users can then automatically query those rules that are relevant in the context of their current task. We have build the prototype FrUiT, a **Fr**amework **U**nderstand**i**ng **T**ool integrated into Eclipse[1], which implements this approach. It provides means to a) mine reuse rules and b) present them in a context-dependent manner.

The remainder of this paper is structured as follows. Section 2 shows our tool from a user's point of view. In section 3 we elaborate on its implementation. In section 4 we present first results. Section 5 discusses issues of the current prototype and presents future work. Finally, we summarize the paper in section 6.

[1]http://www.eclipse.org

2. WALKTHROUGH

To illustrate how FrUiT can improve the software development process, we will show its capabilities in a short walkthrough. The walkthrough describes the creation of an Eclipse wizard using the UI framework JFace. It demonstrates which usage patterns were extracted from existing applications that use the classes `WizardDialog`, `Wizard` and `WizardPage`.

We start the implementation by creating a new `WizardDialog` (index 1 in figure 1), which is the top-level dialog for an Eclipse wizard. We analyze the source code with FrUiT and examine the implementation hints shown in the suggestion view (index 2 in figure 1).

The first rule suggests to call method `open()` on our wizard dialog with a confidence of 100%. The confidence of a rule indicates how many percent of the classes in a similar context followed the suggestion – in this case how many classes that instantiated a wizard dialog also invoked the method. We follow the first hint and add the call on the wizard dialog.

Rerunning the analysis does not change the results furthermore. We turn to the second rule which suggests to instantiate a subtype of `Wizard` with a confidence of 82% (the caret symbol before the class name stands for "subtype of"). The rationale view (index 3 in figure 1) shows the preconditions of the rule selected in the suggestion view. In our case, we see that 82% of the classes that create a `WizardDialog` (antecedent 1 in the first row) also followed the suggestion and instantiated a subtype of `Wizard`.

We point the mouse above the suggestion to reveal the corresponding JavaDoc comment (index 4 in figure 1). It shows us that a `Wizard` is primary responsible for the creation of a set of `WizardPage`s and it provides the logic executed when the user presses the finish button. Hence, we follow the second rule and create a subclass of `Wizard` in order to instantiate it later. After analyzing the newly created `Wizard` class, FrUiT shows 10 programming hints. The most interesting ones are:

- *override performFinish()* with 100 %,
- *override addPages()* with 87 %,
- *call addPage(IWizardPage)* with 93 %, and
- *instantiate IWizardPage* with 80 % confidence.

After reading the corresponding API documentation we decide to override methods `performFinish()` and `addPages()`. Rerunning the analysis with these two new facts increases the confidence for *call addPage(IWizardPage)* to 100% and for the rule *instantiate IWizardPage* to 88%.

In order to satisfy the *call addPage(IWizardPage)* suggestion we create a new class which implements the `IWizardPage` interface. The `IWizardPage` has to provide the SWT controls that represent the page and determines whether the user has supplied enough information to complete the page.

Again we analyze the empty class file that only contains the class declaration. We get one rule with a confidence of 99 % saying that we should rather extend the class `WizardPage` than implement the `IWizardPage` interface from scratch. We change the class declaration accordingly and a new analysis leads to a large number of suggestions. Most of them concern SWT components like buttons, labels and text fields. One rule of interest is *call setControl(Control)* which has a confidence of 100%. Reading the API documentation of `IDialogPage.setControl(Control)` shows that we must call this method from inside method `createContents()` to get a working wizard page.

In this walkthrough we have shown how FrUiT supports novice framework users in learning the correct usage of a framework. Based on a given context, FrUiT suggests program elements that seem relevant, because other framework developers used them together. The suggestions guide developers to collaborators of a class (e.g., from a `Wizard` to a `WizardPage`) or alternative implementations (e.g., by suggesting to use a `WizardPage` instead of implementing `IWizardPage` from scratch).

3. IMPLEMENTATION

Mining reuse rules

To find frequently occurring reuse rules, we follow a three step approach. First, we extract information from the framework instantiation examples. Second, we use data mining techniques to mine for reuse rules based on this information. Third, we apply filters on the found rules to remove those that are not relevant for framework understanding. In the following, we elaborate on each step.

Extraction phase: The basic idea of finding reuse patterns in code is to search the example instantiations for similarities that might represent a pattern. We transform each example class into a set of *class properties*, which will be used as the input for the mining algorithm. We currently consider 5 kinds of class properties:

extends:A denotes that the example class inherits from class `A`.

implements:A is created if the example class implements interface `A`.

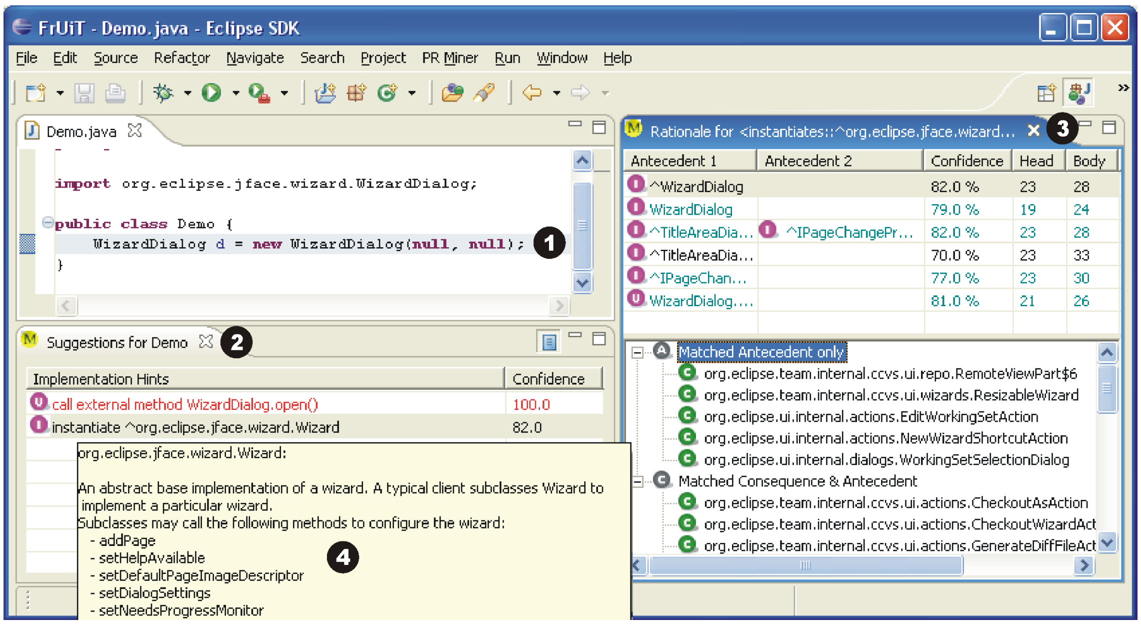
overrides:A.b() expresses that the example class overrides method `b()` from class `A`.

calls:A.b() is used when the example class calls method `b()` from class `A`.

instantiates:A represents a call to a constructor of `A` within the example class.

The first three kinds of class properties occur frequently when inheritance is used to specialize the framework, i.e., in the case of white-box frameworks. The other two kinds of class properties can be found in instantiations of both white-box and black-box frameworks.

Mining phase: After creating a set of class properties for each class in the instantiation examples, we pass them to the data mining algorithm. We use Opus [8], an association rule mining algorithm which has a reasonable performance


FrUiT - Demo.java - Eclipse SDK
File Edit Source Refactor Navigate Search Project PR Miner Run Window Help
Demo.java
Suggestions for Demo
Rationale for <instantiates::^org.eclipse.jface.wizard...
1
2
3
4

```
import org.eclipse.jface.wizard.WizardDialog;

public class Demo {
    WizardDialog d = new WizardDialog(null, null);
}
```

Implementation Hints	Confidence
call external method WizardDialog.open()	100.0
instantiate ^org.eclipse.jface.wizard.Wizard	82.0

org.eclipse.jface.wizard.Wizard:

An abstract base implementation of a wizard. A typical client subclasses Wizard to implement a particular wizard.
Subclasses may call the following methods to configure the wizard:
- addPage
- setHelpAvailable
- setDefaultPageImageDescriptor
- setDialogSettings
- setNeedsProgressMonitor

Antecedent 1	Antecedent 2	Confidence	Head	Body
^WizardDialog		82.0 %	23	28
WizardDialog		79.0 %	19	24
^TitleAreaDia...	^IPageChangePr...	82.0 %	23	28
^TitleAreaDia...		70.0 %	23	33
^IPageChan...		77.0 %	23	30
WizardDialog....		81.0 %	21	26

- Matched Antecedent only
  - org.eclipse.team.internal.ccvs.ui.repo.RemoteViewPart$6
  - org.eclipse.team.internal.ccvs.ui.wizards.ResizableWizard
  - org.eclipse.ui.internal.actions.EditWorkingSetAction
  - org.eclipse.ui.internal.actions.NewWizardShortcutAction
  - org.eclipse.ui.internal.dialogs.WorkingSetSelectionDialog
- Matched Consequence & Antecedent
  - org.eclipse.team.internal.ccvs.ui.actions.CheckoutAsAction
  - org.eclipse.team.internal.ccvs.ui.actions.CheckoutWizardAct
  - org.eclipse.team.internal.ccvs.ui.actions.GenerateDiffFileAct

Figure 1: The FrUiT plug-in in Action

and low memory requirements for in-memory mining. Given a user-specified minimum support s, the algorithm creates rules of the form $x \Rightarrow y$ if at least s example classes that fulfill the precondition x, i.e., contain the elements of x in their class properties, also fulfill y. Often, such rules do not hold in all cases, e.g., due to a slightly different reuse of a component. To indicate the strength of a rule, we use their *confidence* c which is the number of classes that fulfill both x and y divided by the number of classes that fulfill the precondition x only.

Filtering phase: In the last step, we filter out rules that deem not interesting for our application. Three different classes of rules are filtered:

Misleading rules Assume that we found a rule $y \Rightarrow z$ with a confidence of 80%. We refer to a rule $x \wedge y \Rightarrow z$ which has a confidence below 80% as misleading, because the presence of x actually decreases the likelihood of finding the item z.

Overfitting rules Suppose that we have a rule $y \Rightarrow z$ with a confidence of 80%. A second rule $x \wedge y \Rightarrow z$ with a confidence of 80.5% is rejected as an overfitting rule, because the introduction of an additional precondition does not yield to a significantly higher confidence.

Specific rules Imagine that we have two rules $extends : A \Rightarrow overrides : A.m()$ and $overrides : A.n() \Rightarrow overrides : A.m()$, both with the same confidence. If a class overrides `A.n()`, it always also extends `A`; thus, we remove the second rule because it is already covered by the first one.

Presenting reuse rules

After creating a database with the reuse rules, we present the "relevant" rules to novice users of the framework in a context-dependent manner. Therefore, we extract the context, i.e., all class properties of the currently edited file in the Eclipse editor. This works similar to the extraction phase described above. We then execute a query on the rules database to find all rules that match the given context. The results are grouped by the suggestion (the right-hand side of a rule) and presented in the suggestion view together with their confidence. By selecting a suggestion, users can see the rationale, which is the left-hand side of the rules. They can also see all examples and counter-examples of a rule.

4. EVALUATION

To evaluate our approach, we mined plug-ins containing user interface code for rules in the SWT/JFace framework. We searched for rules with at most two class properties in their context. Further, we required a minimum support of 10 and a minimum confidence of 50%. The example basis was comprised of all plug-ins shipped with the Eclipse IDE that have the term `ui` in their name[2]. We present the application of FrUiT by using three different parts of the framework and discuss the resulting rules in the following.

Layout

Alice wants to create a user interface comprised of several widgets. She knows that a composite widget is needed, but has no experience with layout management in SWT. She starts by creating a field holding the composite:

```
private Composite c = new Composite(null, SWT.NONE);
```

After analyzing the code, FrUiT presents the following ten rules with suggestions to other program element:

[2]This is the Eclipse naming convention for plug-ins containing user interface code.

#	Rule	Conf.
1	call method Composite.setLayout(Layout)	94%
2	instantiate ˆorg.eclipse.swt.widgets.Layout	92%
3	instantiate org.eclipse.swt.layout.GridLayout	88%
4	call method Layout.Layout()	84%
5	instantiate org.eclipse.swt.layout.GridData	83%
6	call method Control.setLayoutData(Object)	83%
7	call method GridLayout.GridLayout()	80%
8	instantiate org.eclipse.swt.layout.Label	61%
9	call method Label.Label(Composite, int)	61%
10	call method Label.setText(String)	55%

The three bottom-most rules concern the usage of labels, which seem to be used often together with composites. More interesting, the top-most rules all concern layout management. Alice takes a look at the first rules and sees that most users call method `Composite.setLayout` (rule 1) and instantiate an object of type `Layout` or one of its subtypes (rule 2). Further, rule 3 indicates that a `GridLayout` is used in most cases.

After creating a `GridLayout` instance and reanalyzing the code, rules 2 to 4 vanish because they are fulfilled now. New rules do not appear, but the existing ones get a higher confidence. For instance, rule 1 now has a confidence of 100% and the confidence of rules 5 and 6 raise to 93%. Alice reads the documentation for the suggested program elements and recognizes that she needs a `GridData` object which should also be passed to the `Composite`. After writing corresponding code, she successfully handeled the layout management and FrUiT only provides further suggestions to widgets often used within a composite, such as labels or buttons.

Toolbar

Bob wants to implement a toolbar for his new research project. A search for the corresponding term brings him to the class `ToolBar` and he creates an instance of it:

```
ToolBar bar = new ToolBar (null, SWT.BORDER);
```

Next, he uses FrUiT and 18 rules with a confidence ranging from 52% to 79% are presented in the suggestions view. 10 rules concern layout management (a `ToolBar` is a subtype of `Composite`), 4 rules concern tool bar management, 3 rules suggest to create other widgets and one rule proposes the use of a `ContributionManager`. The toolbar related rules are:

#	Rule	Conf.
1	instantiate org.eclipse.swt.widgets.ToolItem	75%
2	call method ToolItem.setImage(Image)	69%
3	call method ToolItem.setTooltipText(String)	62%
4	call method ToolItem.ToolItem(Toolbar,int)	62%

These four rules indicate that Bob should consider to create and initialize a `ToolItem`. In fact, the documentation states that "instances of this class represent a selectable user interface object that represents a button in a tool bar." Hence, Bob adds another line of code to his class:

```
ToolItem item = new ToolItem(toolBar, SWT.PUSH);
```

Again, the analysis with FrUiT brings up 19 suggestions. However, rules 3 and 4 are removed (we already created a `ToolItem`) and the confidence of rules 2 and 3 changed to 100%. Further, two new rules came up. They suggest to call `ToolItem.addSelectionListener(SelectionListener)` (100%) and `Item.setText(String)` (71%).

Bob takes a look at the suggested methods, initializes the `ToolItem` with a text, an image, and a tooltip text and adds a `SelectionListener` to it. He gets a working toolbar containing a single button.

TableViewer

Carol was asked to implement a table viewer. After creating a field of type `TableViewer` and analyzing the code, FrUiT presents more than 40 rules. Carol is not willing to investigate all suggestions, but only considers the top 10 rules:

#	Rule	Conf.
1	instantiate ˆorg.eclipse.swt.widgets.Layout	100%
2	instantiate ˆorg.eclipse.swt.widgets.Widget	100%
3	call method Layout.Layout()	100%
4	call method TableViewer.setContentProvider(...)	100%
5	call method TableViewer.setLabelProvider(...)	100%
6	call method Composite.setLayout(Layout)	97%
7	instantiate org.eclipse.swt.widgets.Table	97%
8	call method Table.Table(Composite, int)	97%
9	call method TableViewer.setInput(Object)	97%
10	instantiate org.eclipse.swt.layout.GridData	96%

Half of the rules concern tables. All other framework users pass a content provider and a label provider to the table viewer. Further, most of them also pass an input to the viewer and create a `Table`. Carol follows the suggestions and reanalyzes her code. In addition to many layout related hints, FrUiT suggests to take a look at the viewer's methods `addSelectionChangedListener`, `getSelection` and `getTable`. In addition, a rule suggests to create a `TableColumn`, and two others indicate that the `Table`'s methods `setHeaderVisible` and `setLayout` could be important.

Summarizing, our tool shows Carol many collaborators of the `TableViewer`, which are needed to implement a visual table in the SWT/JFace framework.

5. DISCUSSION AND FUTURE WORK

The first results of using FrUiT for the SWT/JFace framework are promising. We have shown that, given a starting point to the desired framework functionality, our tool presents relevant program elements to the user. Further, as more context is available, the confidence of the relevant suggestions tends to increase.

However, our first experiences with using the FrUiT prototype revealed the following issues that will be considered in our future work.

Scalability: FrUiT "learns" how to use a framework by analyzing existing instantiations. We assume that the more examples are used in the mining process, the better the quantity and quality of the resulting reuse rules will be. Unfortunately, simple mining algorithms as used in [5] and also our Opus algorithm are not scalable enough to mine a large amount of example code. We are currently implementing an efficient algorithm based on frequent closed itemsets [9].

Rule pruning: While the three pruning techniques presented in section 3 significantly decrease the number of rules that do not contribute to the understanding of a framework, we identified the need also to remove "transitive rules". For instance, consider you have two rules: $A \Rightarrow B$ (confidence 100%) and $B \Rightarrow C$ (confidence 95%). Often, you will also get a transitive rule $A \Rightarrow C$ with a high confidence. However, in our domain such transitive rules clutter the suggestion view. For instance, consider the layout example in section 4. Rule #10 suggests to initialize a label by calling its `setText(String)` method. However, this transitive rule is based on the observation that most composites first instantiate a label (rule #9) and then initialize it. Thus, we should present the rule #9 only and postpone the suggestion to call `setText(String)` after a label is instantiated.

Rule quality: Currently we only provide suggestions at the class level, e.g., "if you implement a `Wizard`, you should also instantiate an `IWizardPage` and call `addPage(IWizardPage)`". We will improve the detail of a suggestion by incorporating information from static analysis. This will enable us to provide instance based suggestion, i.e., that they should instantiate an `IWizardPage` and pass *this* page as a parameter to `addPage(IWizardPage)`" declared in the *current* `Wizard`.

Rule presentation: Another observation from the prototype is, that suggestions often can be categorized. For instance, the tool bar example from section 4 showed suggestions for the layout concern and the tool bar management. Having these groups explicit would enable users to hide all suggestions that are currently not relevant for them. We will investigate along which properties the suggestions should be grouped and provide a tree-based presentation that enables to collapse or expand single suggestion categories.

Context definition: FrUiT's suggestions are presented in a context-sensitive manner. Currently, we use the whole Java class in the active editor as the context. However, we experienced that it is sometimes useful to have a single line of code as a context. For instance, consider you are implementing a view and want to add a new table to it (cf. section 4). If your view implementation already contains other code, e.g., for initializing the other widgets, you will also see suggestions related to those widgets, and not only suggestions concerning the creation of a table viewer. We will provide a feature that enables users to select a specific range of the source code, which will serve as the context for the suggestions.

We will implement the features discussed above in our future work. Further, we will create rule databases by mining instantiations of several parts of the Eclipse API. These will be made available to the Eclipse community and we plan to perform a field study to investigate the benefits and weaknesses of our approach in detail.

6. SUMMARY

We presented FrUiT, a tool that combines data mining techniques with a context-dependent presentation. FrUiT supports novice programmers when using a framework. Our approach has several benefits. First, by using existing framework instantiations written by experts to learn from, we do not require additional effort to create special artifacts such as documentation or code snippets. Second, by using data mining we extract significant reuse rules. Hence, only knowledge that concerns *using* the *framework* is presented to the user. Third, we relieve developers from searching for rules of interest explicitly. Instead, the tool uses the current context and presents relevant rules automatically.

More information about our research on framework understanding can be found at the following website:

http://www.st.informatik.tu-darmstadt.de/Frameworks

Acknowledgments

We would like to thank Ivica Aracic, Christoph Bockisch, and Johannes Fürnkranz for their fruitful comments on our work. This research is supported by a 2006 IBM Eclipse Innovation Award.

7. REFERENCES

[1] M. Eichberg, M. Haupt, M. Mezini, and T. Schäfer. Comprehensive software understanding with SEXTANT. In *Proceedings of the International Conference on Software Maintenance*, pages 315–324. IEEE Computer Society, 2005.

[2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley, 1995.

[3] R. Holmes and G. C. Murphy. Using structural context to recommend source code examples. In *Proceedings of the International Conference on Software Engineering*, pages 117–125. ACM Press, 2005.

[4] T. C. Lethbridge, J. Singer, and A. Forward. How software engineers use documentation: The state of the practice. *IEEE Software*, 20(6):35–39, 2003.

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

[6] L. R. Neal. A system for example-based programming. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 63–68. ACM Press, 1989.

[7] F. Shull, F. Lanubile, and V. R. Basili. Investigating reading techniques for object-oriented framework learning. *IEEE Transactions on Software Engineering*, 26(11):1101–1118, 2000.

[8] G. I. Webb and S. Zhang. Beyond association rules: Generalized rule discovery. `http://www.csse.monash.edu.au/~webb/Files/WebbZhang03.pdf`.

[9] M. J. Zaki. Generating non-redundant association rules. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 34–43. ACM Press, 2000.