FEMIR: A Tool for Recommending Framework Extension Examples

Muhammad Asaduzzaman Chanchal K. Roy Kevin A. Schneider Daqing Hou†
Department of Computer Science, University of Saskatchewan, Canada
†Electrical and Computer Engineering Department, Clarkson University, USA
{md.asad, chanchal.roy, kevin.schneider}@usask.ca, dhou@clarkson.edu

***Abstract*—Software frameworks enable developers to reuse existing well tested functionalities instead of taking the burden of implementing everything from scratch. However, to meet application specific requirements, the frameworks need to be customized via extension points. This is often done by passing a framework related object as an argument to an API call. To enable such customizations, the object can be created by extending a framework class, implementing an interface, or changing the properties of the object via API calls. However, it is both a common and non-trivial task to find all the details related to the customizations. In this paper, we present a tool, called *FEMIR*, that utilizes partial program analysis and graph mining technique to detect, group, and rank framework extension examples. The tool extends existing code completion infrastructure to inform developers about customization choices, enabling them to browse through extension points of a framework, and frequent usages of each point in terms of code examples. A video demo is made available at https://asaduzzamanparvez.wordpress.com/femir.**

***Index Terms*—API, framework, reuse, extension point, extension, partial program analysis, graph mining**

I. Introduction

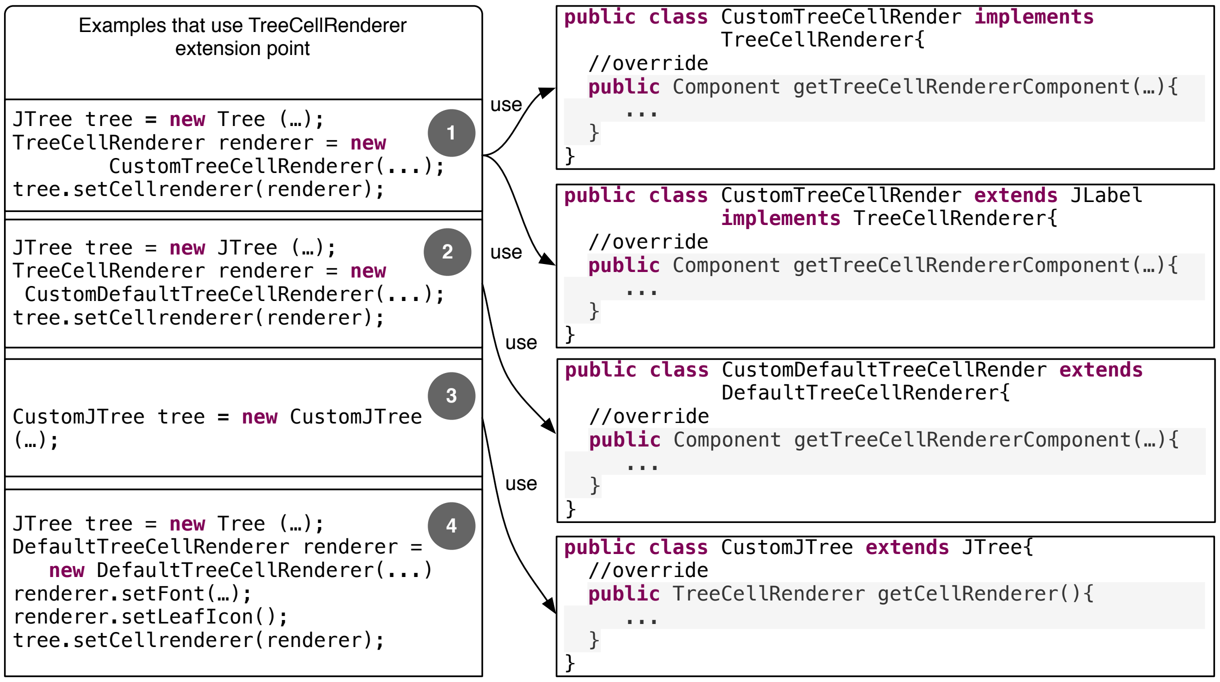
When developing applications, developers extensively rely on software frameworks to save both development time and effort. This is largely due to the fact that frameworks enable developers to reuse existing functionalities instead of working from scratch. Besides reusability, another advantage of using a framework is that the existing implementation can be customized. This is particularly useful when the current framework implementation does not directly meet application specific requirements.

A popular way to customize the behavior of a framework is to pass a framework related object as an argument of an API call. The object can be created by extending framework classes, implementing framework interfaces, or by changing the default properties of the object via setter method calls. We call the formal parameter of the API call an *extension point*. For example, consider the case of the *JTree* class in the Java Swing framework. There can be several ways of extending the functionality of the *JTree* class. For instance, a *TreeModel* can help to define a tree structure displayed by an instance of the *JTree* class. A developer can define her own implementation of the *TreeModel* and call the *setModel* method of the *JTree* with an argument of a new *TreeModel* object to control the tree data structure. A *TreeCellRenderer* defines the way a *JTree* should display its nodes. Furthermore, a developer can provide her own implementation of the *TreeCellRenderer* interface and call the *setCellRenderer* method with an argument of a new *TreeCellRenderer* object to customize the display of nodes (see Figure 1). The formal parameters of the above method calls (*TreeModel* for *setTreeModel* and *TreeCellRenderer* for *setTreeCellRenderer*) are thus examples of extension points of *JTree* that allow a developer to gain finer control over the behavior and presentation of the framework class *JTree*.

In general, there can be several ways to work with an extension point. For example, a developer can create a custom tree data structure by implementing the *TreeModel* interface, by extending the *AbstractTreeModel* class, or by using the *DefaultTreeModel* class. She can then use one of the constructors of the *JTree* class or the *setModel* method to register the custom tree model to an instance of the *JTree* class.

Learning how to correctly customize a framework via extension points is an important and non-trivial task. In this paper, we present a tool for discovering framework extension points and code examples that illustrate how to customize the framework, called *FEMIR* (Framework Extension Point Miner and Recommender). The tool performs static analysis of the source code examples to identify framework extension points, represents their usages as framework extension graphs, and indexes code examples that contain such graphs. To inform developers about related extension points, the tool takes advantages of the code completion infrastructure of the Eclipse integrated development environment (IDE). When a developer types a dot (.) after a receiver object, the tool determines the qualified type name of the receiver object to identify related extension points and integrate the results with code completion proposals. Once a developer selects an extension point from the completion popup menu, *FEMIR* enables the developer to examine different ways of using the extension point, to browse relevant code examples and to learn related extension points. Full details of the technique including a comprehensive evaluation of the tool can be found elsewhere [1].

The rest of the paper is organized as follows. Section II describes the related work. We explain the feature of *FEMIR* in Section III. Section IV presents the architecture of the tool. We present a use case scenario in Section V. Section VI summarizes accuracy and runtime performance of the technique. Section VII presents a user study to understand the usefulness of the tool. Finally, Section VIII concludes the paper.


Examples that use TreeCellRenderer extension point

```
1
JTree tree = new Tree (…);
TreeCellRenderer renderer = new
        CustomTreeCellRenderer(...);
tree.setCellrenderer(renderer);
```

```
2
JTree tree = new JTree (…);
TreeCellRenderer renderer = new
 CustomDefaultTreeCellRenderer(...);
tree.setCellrenderer(renderer);
```

```
3
CustomJTree tree = new CustomJTree
(…);
```

```
4
JTree tree = new Tree (…);
DefaultTreeCellRenderer renderer =
   new DefaultTreeCellRenderer(...)
renderer.setFont(…);
renderer.setLeafIcon();
tree.setCellrenderer(renderer);
```

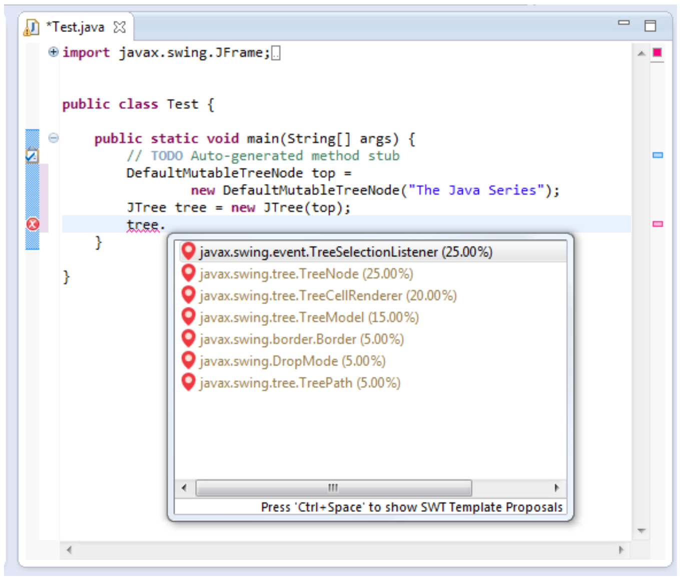
```
public class CustomTreeCellRender implements
             TreeCellRenderer{
  //override
  public Component getTreeCellRendererComponent(…){
     ...
  }
}
```

```
public class CustomTreeCellRender extends JLabel
             implements TreeCellRenderer{
  //override
  public Component getTreeCellRendererComponent(…){
     ...
  }
}
```

```
public class CustomDefaultTreeCellRender extends
             DefaultTreeCellRenderer{
  //override
  public Component getTreeCellRendererComponent(…){
     ...
  }
}
```

```
public class CustomJTree extends JTree{
  //override
  public TreeCellRenderer getCellRenderer(){
     ...
  }
}
```

Fig. 1. Using an extension point (TreeCellRenderer) of the *JTree* class to gain finer control over cell rendering. Code examples on the left-hand side show four different ways of using the TreeCellRenderer extension point.

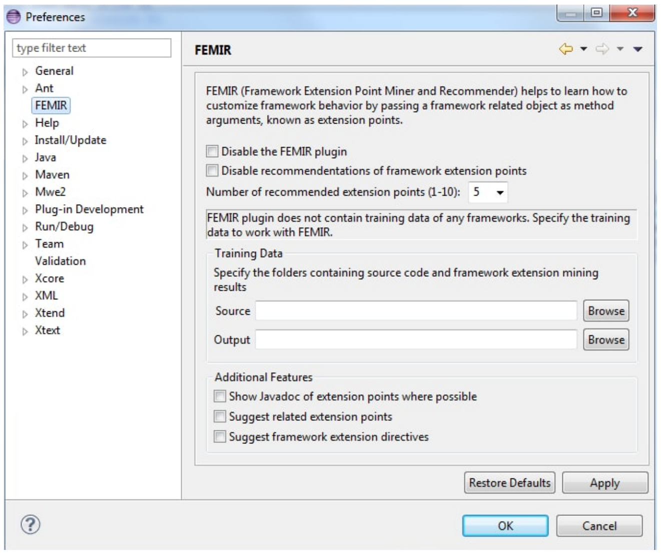


```
import javax.swing.JFrame;

public class Test {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        DefaultMutableTreeNode top =
                new DefaultMutableTreeNode("The Java Series");
        JTree tree = new JTree(top);
        tree.
    }

}
```

```
javax.swing.event.TreeSelectionListener (25.00%)
javax.swing.tree.TreeNode (25.00%)
javax.swing.tree.TreeCellRenderer (20.00%)
javax.swing.tree.TreeModel (15.00%)
javax.swing.border.Border (5.00%)
javax.swing.DropMode (5.00%)
javax.swing.tree.TreePath (5.00%)
```

Press 'Ctrl+Space' to show SWT Template Proposals

Fig. 2. An example of FEMIR suggesting framework extension points for the javax.swing.JTree receiver type. Extension points are sorted based on the frequencies they are used in the code repository.



FEMIR

FEMIR (Framework Extension Point Miner and Recommender) helps to learn how to customize framework behavior by passing a framework related object as method arguments, known as extension points.

☐ Disable the FEMIR plugin
☐ Disable recommendentations of framework extension points
Number of recommended extension points (1-10): 5

FEMIR plugin does not contain training data of any frameworks. Specify the training data to work with FEMIR.

Training Data
Specify the folders containing source code and framework extension mining results
Source
Output

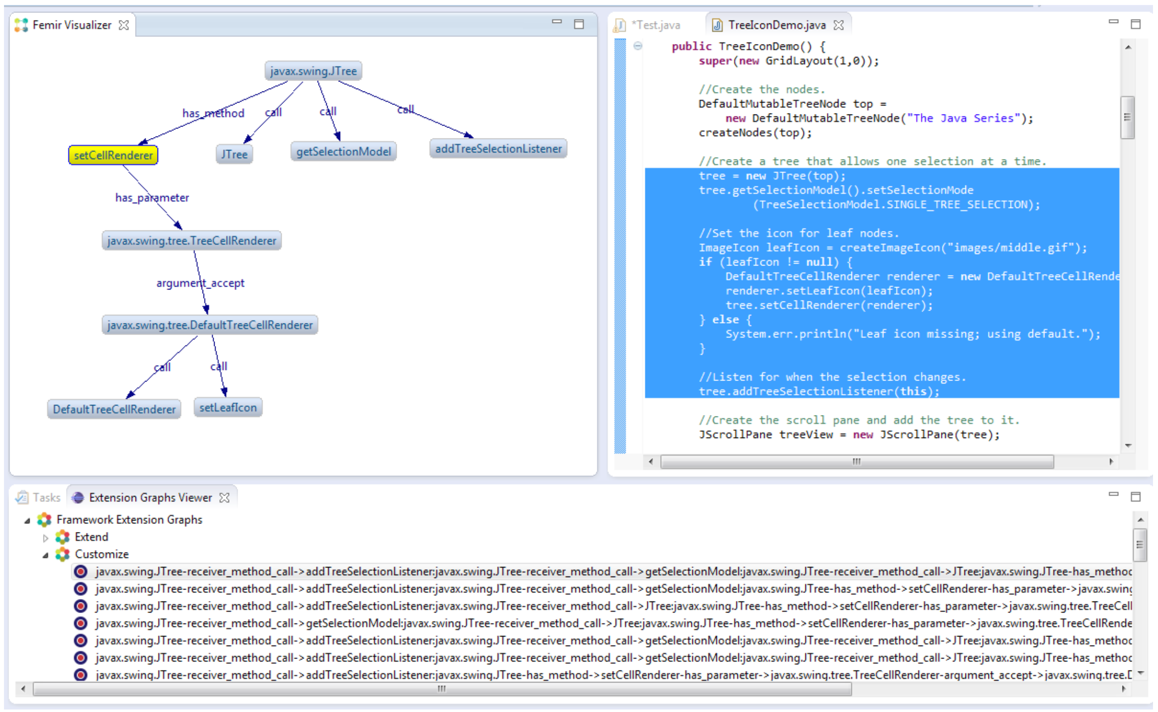
Additional Features
☐ Show Javadoc of extension points where possible
☐ Suggest related extension points
☐ Suggest framework extension directives

Fig. 3. Preference page to configure various options of the FEMIR plugin.

II. Related Work

Dagenais and Ossher [5] focus on finding framework usage examples because examples are a form of documentation in themselves. The technique has been implemented in a tool called *XFinder*. It requires developers to create guides as a sequence of steps for using framework extensions where the steps of a guide are expressed as concerns using *Mismar*, a concern oriented documentation toolset. Given a code base and a framework, *XFinder* can locate examples implementing each step of the guide. While their technique focuses on finding framework extension examples, we focus on finding framework extension points and their usages. Another difference is that our technique does not require to provide the set of steps required to use framework extension points, rather *FEMIR* uses source code analysis together with graph mining technique to identify these steps automatically.

Bruch et al. [2] proposed a technique that mines four sub-classing directives of object-oriented white box frameworks [2]. These are method overriding directives, method extension directives, method call directives and class extension scenario. While the technique can answer how to extend a class, it does not focus on identifying how to use that extended class. On the contrary, *FEMIR* can answer both questions.



```
    public TreeIconDemo() {
        super(new GridLayout(1,0));

        //Create the nodes.
        DefaultMutableTreeNode top =
            new DefaultMutableTreeNode("The Java Series");
        createNodes(top);

        //Create a tree that allows one selection at a time.
        tree = new JTree(top);
        tree.getSelectionModel().setSelectionMode
                (TreeSelectionModel.SINGLE_TREE_SELECTION);

        //Set the icon for leaf nodes.
        ImageIcon leafIcon = createImageIcon("images/middle.gif");
        if (leafIcon != null) {
            DefaultTreeCellRenderer renderer = new DefaultTreeCellRende
            renderer.setLeafIcon(leafIcon);
            tree.setCellRenderer(renderer);
        } else {
            System.err.println("Leaf icon missing; using default.");
        }

        //Listen for when the selection changes.
        tree.addTreeSelectionListener(this);

        //Create the scroll pane and add the tree to it.
        JScrollPane treeView = new JScrollPane(tree);
```

```
Framework Extension Graphs
  Extend
  Customize
    javax.swing.JTree-receiver_method_call->addTreeSelectionListener:javax.swing.JTree-receiver_method_call->getSelectionModel:javax.swing.JTree-receiver_method_call->JTree:javax.swing.JTree-has_methoc
    javax.swing.JTree-receiver_method_call->addTreeSelectionListener:javax.swing.JTree-receiver_method_call->getSelectionModel:javax.swing.JTree-has_method->setCellRenderer-has_parameter->javax.swinc
    javax.swing.JTree-receiver_method_call->addTreeSelectionListener:javax.swing.JTree-receiver_method_call->JTree:javax.swing.JTree-has_method->setCellRenderer-has_parameter->javax.swing.tree.TreeCell
    javax.swing.JTree-receiver_method_call->getSelectionModel:javax.swing.JTree-receiver_method_call->JTree:javax.swing.JTree-has_method->setCellRenderer-has_parameter->javax.swing.tree.TreeCellRende
    javax.swing.JTree-receiver_method_call->addTreeSelectionListener:javax.swing.JTree-receiver_method_call->getSelectionModel:javax.swing.JTree-receiver_method_call->JTree:javax.swing.JTree-has_methoc
    javax.swing.JTree-receiver_method_call->addTreeSelectionListener:javax.swing.JTree-receiver_method_call->getSelectionModel:javax.swing.JTree-receiver_method_call->JTree:javax.swing.JTree-has_methoc
    javax.swing.JTree-receiver_method_call->addTreeSelectionListener:javax.swing.JTree-has_method->setCellRenderer-has_parameter->javax.swing.tree.TreeCellRenderer-argument_accept->javax.swing.tree.D
```

Fig. 4. Once the *TreeCellRenderer* extension point is selected (third entry in Figure 2), *FEMIR* shows all of its framework extension graphs grouped by categories (bottom view), as well as visualization of a selected graph from the Customize category (top left) and a framework extension that is an example of the graph (top right). The selected example corresponds to case number 4 in Figure 1.

In a different study, Bruch et al. [3] developed a technique that mines framework usage examples to collect reuse rules [3]. While they use association rule mining to determine reuse rules, we apply graph mining technique. Furthermore, our reuse rules tend to be larger than theirs.

Thummalapenta and Xie [7] developed a technique, called *SpotWeb*, that given a framework determines both hotspots and coldspots leveraging code search engines. However, SpotWeb does not focus on identifying how to extend the functionality of framework classes.

Techniques that mine frequent patterns in source code examples are also related to our study. Zhong et al. [8] developed a technique, called *MAPO*, that uses sequential pattern mining technique to discover frequently used method call sequences. Nguyen et al. [6] proposed *GrouMiner*, a graph-based approach that can mine frequent usage patterns involving multiple objects from source code. The above techniques do not focus on detecting API usages spread across multiple source files. However, framework extensions may spread across multiple files and the same extension point can be used in multiple ways. *FEMIR* handles both.

III. Features

FEMIR is available as an Eclipse plugin. It can be activated to detect framework extension points and their usages by mining source code examples. This requires to select appropriate commands from the menu. When a developer types a dot (.) after a receiver expression to suggest completion proposals, *FEMIR* also activates to recommend relevant extension points (Figure 2). However, this requires that extension point detection results from a previous run of *FEMIR* and the corresponding source code are available to the plugin. To avoid interrupting other types of completion proposals (such as method completion proposals), we put the recommendations of *FEMIR* in a separate completion proposal category and the tool recommends a fixed number of extension points. However, this number can be configured by the user. The plugin can be enabled or disabled from its own preference page in the Eclipse IDE (Figure 3). When a developer selects the completion proposal made by *FEMIR*, the plugin shows extension point usage patterns in a view of the Eclipse workbench window using a *JFace TreeViewer* component (Figure 4). Developers can see the framework extension graph of the usage pattern and other related extension points by selecting a leaf node of the tree. When a developer selects a node in the graph, *FEMIR* shows the corresponding code example. For example, a developer can select a node representing overridden method and *FEMIR* shows code examples to help her learning how to override that method. Depending on the selection of nodes in graph the code example also changes. For example, if a developer selects a node representing a client class that implements a framework interface, *FEMIR* shows examples of implementing that interface. This includes all the methods that are overridden to implement the interface. Developers can also learn related extension points and several framework extension directives that goes together. For example, if a developer selects the *TreeModel* extension point, *FEMIR* also suggests implementing a *TreeCellRenderer* extension point. This is done based on the association property of extension points. Similarly, by selecting an overridden method developers can

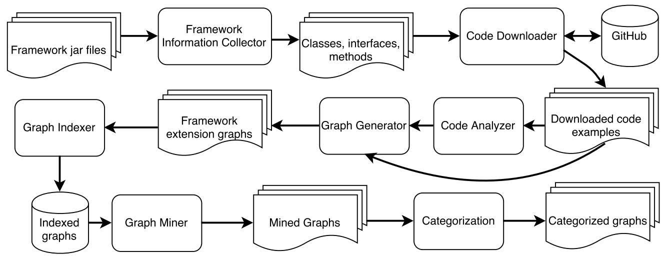


Fig. 5. Working process of the graph miner of FEMIR

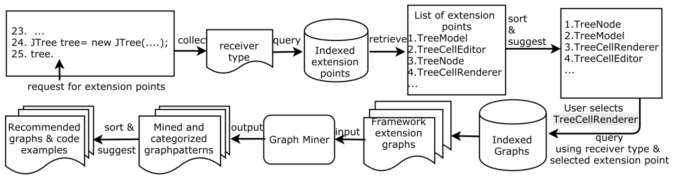


Fig. 6. Overview of the recommender component of FEMIR

learn which other methods are frequently overridden with the selected one.

IV. Architecture

The *FEMIR* tool is written entirely in Java and implemented as an Eclipse plugin. It consists of two main components. These are the miner and the recommender. Figures 5 and 6 summarize the working process of these components. We briefly describe both components below. However, further information on both components can be found elsewhere [1].

A. Miner

The Miner component is responsible for detecting framework extension points. It identifies their usages and mines frequent patterns of using framework extension points. It works in the following steps.

1) Collecting framework information: *FEMIR* requires that the framework jar file is included in the target project. It collects the fully qualified name of classes, interfaces, super classes, and implemented interfaces. For each method, *FEMIR* collects the method name, return type and types of parameters.

2) Downloading code examples: *FEMIR* can collect relevant code examples of the input framework from GitHub[1] code repository. Our selection is based on the fact that a large number of open source repositories are available and GitHub provides public APIs to search and download code examples. Alternatively, developers can also instruct *FEMIR* to use previously downloaded code examples. For example, to identify repositories that use the *Swing* API, we use the following query: *import AND javax AND swing*. The tool selects those repositories that are written in Java language and have forked at least once. After collecting repository information, *FEMIR* downloads source code examples.

3) Analyzing code examples: The goal of this step is to identify relevant extension points and generate framework extension graphs. There are two issues that need to be dealt with. First, the code examples may be incomplete or syntactically incorrect. Second, type bindings of API method calls are usually missing that are essential to identify usages of framework extension points. *FEMIR* uses the Eclipse JDT parser to parse the source files and also uses the PPA (partial program analysis) for Eclipse plugin, to resolve type bindings of source code elements [4]. The JDT parser can handle source files that are incomplete or contain syntactic errors. The PPA plugin is currently only available for the Eclipse 3.6 (Helios) and supports Java 5 constructs[2]. *FEMIR* inherits this limitation.

4) Generating framework extension graphs: A framework extension graph describes how a framework extension point is used. Figure 4 shows an example of a framework extension graph where a developer uses the *TreeCellRenderer* extension point. Here, the developer creates a client class by implementing the *TreeCellRenderer* interface and registers an instance of the class using the *setCellRenderer* method argument of an instance of the framework class *JTree*. To determine framework extension graphs, the first step is to generate framework extension points. *FEMIR* analyzes API method calls that has at least one parameter and is related to the target framework type. Thus, the argument to the method call can be a framework class object. Alternatively, it can be created by extending a framework class, implementing framework interfaces or a combination of both. *FEMIR* then generates the graph structure by considering the receiver variable, the method call, the parameter, the argument, and those methods that are called on the receiver or argument variable. If the receiver or the argument is created by extending framework classes or implementing framework interfaces, *FEMIR* considers the extended classes, implemented interfaces, overridden methods and methods that are called in overridden method bodies to generate the framework extension graph.

5) Indexing framework extension graphs: The graphs are indexed based on the receiver type of the method call and the formal parameter type. The indexing is done for effective retrieval of source code. In addition, we also index lists of extension points (i.e., the formal parameters of method calls) by their receiver type. This enables *FEMIR* to find all different extension points given a receiver type.

6) Mining graphs: Given a receiver type and an extension point, the goal of this step is to mine frequent patterns of using that extension point. Given a set of *n* graphs (also known as base graphs), *FEMIR* first generates all one node graphs. In the subsequent steps, *FEMIR* increases the size of the graphs from the previous step by one node. The process continues until all nodes of the graphs are enumerated. To avoid generating the same subgraph structure, *FEMIR* uses a canonical form representation that enables to accurately determine isomorphic graph structures.

The miner also categorizes the graphs into extension pattern categories. Each category represents a common way of using extension points. There are four different extension pattern categories. The *simple* extension pattern does not require

[1]https://github.com/

[2]http://www.sable.mcgill.ca/ppa/

extending a framework class, implementing a framework interface or calling methods on the argument object. In a *customize* extension pattern, developers call a set of methods on the argument object to customize the behavior of the class. The *extend* extension pattern involves extending a framework class and the *implement* extension pattern requires implementing a framework interface. More detailed explanation and examples of extension pattern categories can be found elsewhere [1].

B. Recommender

The recommender component is responsible for recommending possible extension points, their usage patterns and relevant code examples. *FEMIR* integrates the recommender component to the Eclipse code completion engine. Given a receiver type, the recommender thus suggests related extension points as completion proposals. We use the *Complemention-ProposalComputer* extension point of the Eclipse to implement the code completion component of *FEMIR*. However, the extension point only allows to contribute to the list of completion proposals. The same triggering mechanism (typing the dot after the receiver variable) is also used by other code completion components, such as those that contribute to method or field completion proposals. To avoid interrupting other completion proposals we position the proposals contributed by *FEMIR* to a separate proposal category. When a developer selects a completion proposal of type *JavaCompletionProposal* by pressing the enter key, the apply method is called by the underlying framework. To detect such selection the completion proposals contributed by *FEMIR* are created by extending the *JavaCompletionProposal* and we override the apply method of it to customize the implementation.

When a developer selects an extension point, *FEMIR* utilizes the *miner* component to generate frequent patterns of using the extension point and group them into four different categories of extension points. We use *SWT* and *JFace* frameworks for creating graphical user interfaces and presenting results. The results are presented using a *JFace TreeViewer* component. Upon selection of a pattern, the recommender visualizes the graph structure. We use *Zest*[3], a visualization toolkit for Eclipse, for displaying graphs. Developers can interactively browse code examples by selecting different nodes of framework extension graphs. This enables developers to learn usages of extension points and facilitates adapting the code examples to their own implementation.

V. A Use Case Scenario

We describe the use of our tool (see Figure 4 for the *FEMIR* tool) using a scenario that involves the use of the Swing framework for Java. The framework consists of graphical control elements that are used to create graphical user interfaces.

Consider that a developer is using the *JTable* class of the *Java Swing* framework to display data collected from the employees of an office. She wants to mark invalid input of employees in a different color in the *JTable*. However, she does not know how to do that. She searches the internet using the following query: "mark invalid input *JTable*" and the top results returned from the query are discussing input data validation on the *JTable*, which is a different problem. Even if she is successful, it is difficult to find relevant examples, to learn related customization choices and to identify important insights. All these require to search, analyze, and comprehend a large collection of information.

However, *FEMIR* makes the whole process much easier. Let's consider that she plans to use *FEMIR*. She opens the *FEMIR* tool and search for framework extension points for the *JTable* class. Immediately after typing the dot (.) followed by a variable of the *JTable* class, she encounters a few class names. One of them is the *TableCellRenderer* interface. She realizes that her problem is related with rendering of *JTable* cell. However, she does not know how to use *TableCellRenderer* and thus decides to explore further.

When she selects the *TableCellRenderer* extension point from the list of completion proposals, *FEMIR* groups different patterns of using that extension point and shows them using a *TreeViewer* component. She now sees different ways to interact with *TableCellRenderer* extension point. She decides to implement the *TableCellRenderer* interface because the tool reports that most developers do that when using the *TableCellRenderer* extension point. The developer thinks that an example would be great to complete her implementation. Double clicking on the node visualizes a graph structure that shows the way of implementing and using the *TableCellRenderer* interface. She learns from the graph that she needs to override a method to implement the *TableCellRenderer* interface. When she clicks on the node representing the overridden method, the tool immediately shows how to override the method. She also explores other nodes to learn how to implement the interface and how to use the class implementing the *TableCellRenderer* interface with a *JTable*. She then copies code fragments from those files and adapts that in her own implementation. This completes her task. This scenario shows that not only the *FEMIR* tool helps the developer to learn the framework but also assists in using the extension point whose usage spans on multiple source files.

VI. Evaluation

We briefly summarize the accuracy and runtime performance of *FEMIR* in this section. Full details of the evaluation can be found elsewhere [1].

We evaluated *FEMIR* using five different popular frameworks: *Swing*, *JFace*, *JUnit*, *JUNG*, and *JGraphT*. Given an extension point, we evaluated the effectiveness of the tool in recommending framework extension graphs that matched with actual usage of extension points. We used precision, recall and F-measure to determine the accuracy of recommendations. They were calculated by considering the overlapping of nodes between the original graph under testing and the graph suggested by *FEMIR*. We considered three different recommendation strategies. These were *FEMIR-Local*, *FEMIR-Global*, and *FEMIR-D*. *FEMIR-Local* recommends top-n patterns from

[3]https://www.eclipse.org/gef/zest/

within a category. *FEMIR-Global* recommends top-n graphs regardless of the category. Finally, *FEMIR-D* works as follows. It first determines the top-n graph patterns of an extension pattern category. It then determines all the base graphs in the training data that contain the patterns. Among these base graphs, it recommends those that contain the largest number of different node types. Results from the study revealed that *FEMIR-Global* performed the best. While the precision ranges from 78% to 90%, the recall ranges from 56% to 79% for the top-5 recommendations. The F-measure ranges from 67% to 82%. *FEMIR-Local* performed close to *FEMIR-Global*. While the precision ranges from 82% to 92%, the recall ranges from 49% to 73%. The F-measure ranges from 61% to 80%. Finally, *FEMIR-D* performed the worse.

In addition to the accuracy of recommendations, we also measured the runtime performance of the technique. We found that on overage *FEMIR* required 0.92s for recommending framework extension graphs for the *Java Swing* Framework. Most of this time was contributed by the process of mining framework extension graphs. However, we can reduce the recommendation time by mining framework extension graphs for all framework extension points beforehand. The tool requires significantly more time for analyzing source code files and for generating framework extension graphs. A major part of the time is contributed by the partial program analysis. However, this is only a one time operation.

VII. User Study

To understand the usefulness of the *FEMIR* and to identify any usability issues quickly, we conducted a preliminary user study. We used a simple observation process where each participant worked on three tasks that require customizing the behavior of framework classes using framework extension points. We then conducted a semi-structured interview to gain more insights about the design of the tool. Five volunteers participated in this study. The participants were all graduate students and had previous experience working with *Eclipse IDE* and *Java Swing/AWT* libraries. However, the levels of experience differ across the participant pool. We provided written description of the tasks and asked the participants to complete the tasks in 30 minutes. We observed user actions and interviewed each participant at the end of the study.

In general, the five participants agreed that the tool was easy to use and recommendations were useful. Three out of the five users mentioned that they were not aware of some extension points until *FEMIR* recommended those extension points to them. Overall, all users agreed that the tool helped them to quickly learn usages of framework extension points and they were successful to make necessary changes to their code. However, we also learned a few important lessons from the study. First, three out of the five users searched on the Internet to learn more about the extension points. This is because *FEMIR* did not provide any explanation of extension points. After talking with users, it became clear that integrating Javadoc information would have helped them to understand the goal of using extension points. We also learned that code examples in GitHub often have dependencies on other parts of software systems which make them difficult to comprehend. Thus, some users emphasized on including code examples that shows the basic usages of framework extension points. One possible way to improve *FEMIR* is to enable the tool to automatically discover web tutorials and include code examples from those locations. Finally, one user requested to include the suggestions of framework extension directives or related framework extension points during extending a framework class or implementing a framework interface. We are currently working to address all these issues.

VIII. Conclusion

This paper presents *FEMIR*, a tool that supports automatic detection of framework extension points and their usages. Since there can be multiple ways to work with an extension point, *FEMIR* supports grouping of extensions by categories. Developers can use the *FEMIR* tool to learn extension points and their different usages. By leveraging a graph mining technique, *FEMIR* can rank extension points as well as their usages based on their frequencies in code examples. This demonstration will show how *FEMIR* can be utilized to learn and use framework extension points while developing applications. The tool is currently available as an Eclipse plugin. In the future, we plan to conduct more thorough user studies to further investigate the usefulness of the *FEMIR* tool.

Acknowledgment

This work is supported in part by the Natural Science and Engineering Research Council of Canada (NSERC).

References

[1] Muhammad Asaduzzaman, Chanchal K. Roy, Kevin A. Schneider, and Daqing Hou. Recommending Framework Extension Examples. In *Proc. of the 33rd International Conference on Software Maintenance and Evolution*, page 11, 2017. Accepted and to be published.

[2] Marcel Bruch, Mira Mezini, and Martin Monperrus. Mining subclassing directives to improve framework reuse. In *Proc. of the 7th IEEE Working Conference on Mining Software Repositories*, pages 141–150, 2010.

[3] Marcel Bruch, Thorsten Schäfer, and Mira Mezini. FrUiT: IDE Support for Framework Understanding. In *Proc. of the OOPSLA Workshop on Eclipse Technology eXchange*, pages 55–59, 2006.

[4] Barthélémy Dagenais and Laurie Hendren. Enabling static analysis for partial java programs. In *Proc. of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, pages 313–328, 2008.

[5] Barthélémy Dagenais and Harold Ossher. Automatically Locating Framework Extension Examples. In *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 203–213, 2008.

[6] Tung Thanh Nguyen, Hoan Anh Nguyen, Nam H Pham, Jafar M Al-Kofahi, and Tien N Nguyen. Graph-based Mining of Multiple Object Usage Patterns. In *Proc. of the 7th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, pages 383–392, 2009.

[7] Suresh Thummalapenta and Tao Xie. SpotWeb: Detecting Framework Hotspots and Coldspots via Mining Open Source Code on the Web. In *Proc. of the 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336, 2008.

[8] Hao Zhong, Tao Xie, Lu Zhang, Jian Pei, and Hong Mei. MAPO: Mining and Recommending API Usage Patterns. In *Proc. of the 23rd European Conference on Object-Oriented Programming*, pages 318–343, 2009.