Recommending Framework Extension Examples

***Abstract*—The use of software frameworks enables the delivery of common functionality but with significantly less effort than when developing from scratch. To meet application specific requirements, the behavior of a framework needs to be customized via extension points. A common way of customizing framework behavior is by passing a framework related object as an argument to an API call. Such an object can be created by subclassing an existing framework class or interface, or by directly customizing an existing framework object. To do this effectively requires that application developers have extensive knowledge of the framework's extension points and their possible interactions. To aid the developers in this regard, we propose and evaluate a graph mining approach for extension point management. Specifically, our approach mines large amounts of code examples to discover all extension points and their patterns for each framework class. Furthermore, we propose a taxonomy of extension patterns, which categorizes the various ways an extension point has been used in the code examples. Given a framework class that is being used, our approach aids the developer by following a two-step recommendation process. In the first step, it recommends all the extension points that are available in the class. Once the developer chooses an extension point, our approach then discovers all of its usage patterns and recommends the best code examples for each pattern. Using five frameworks, we evaluate the performance of the two-step recommendation, in terms of precision, recall, and F-measure. We also report several statistics related to framework extension points.**

***Index Terms*—API, framework, reuse, graph mining, recommenders**

I. Introduction

A software framework is a reusable implementation of some generic functionality that saves both development time and effort for a client. However, to meet application specific requirements, a developer often needs to customize aspects of the framework (*extension points*). One common way to do so is to pass a framework related object as an argument in an API call. The argument object essentially encapsulates a specific way to customize the framework. The object may be created by subclassing a framework class, implementing a framework interface, or by customizing the properties of an existing object. In this case, we consider the formal parameter of the API call as an extension point. As an example of an extension point, Figure 1(A) shows that the size of a *JFrame* is set to a Dimension object. Some additional but more interesting examples from *JTree* are *TreeModel*, *TreeCellEditor*, *TreeCellRenderer*, and *TreeExpansionListener*. These extension points allow a developer to gain finer control over the behavior of framework classes such as *JFrame* and *JTree*.

Given an extension point, there are often multiple different ways to use it. Consider *TreeCellRenderer* as an example, which controls how a tree node is rendered by passing it as an argument to the *setCellRenderer* method of the *JTree* class.

```
JFrame frame  = new JFrame();
Container c = frame.getContentPane();
c.setLayout(new FlowLayout());
Dimension dim =  new Dimension (...)
frame.setSize(dim);                                   A
```

```
//creating the DefaultTreeCellRenderer object
DefaultTreeCellRenderer renderer = new
            DefaultTreeCellRenderer();
//set the icons to represent open, leaf and closed icon
renderer.setLeafIcon ( new ImageIcon(...));
renderer.setOpenIcon (new ImageIcon(...));
renderer.setClosedIcon (new ImageIcon(...));
//now set the cell renderer to a JTree                B
tree.setCellRenderer (renderer)
```

```
public class CustomTreeCellRenderer extends
    JLabel implements TreeCellRenderer{
    //override
    public Component getTreeCellRendererComponent ( … )
    { //body goes here }
}
//code fragment that shows the
//use of CustomTreeCellRenderer
...
TreeModel treeModel = new DefaultTreeModel(...);
JTree tree = new JTree (treeModel);                   C
tree.setShowsRootHandles(true);
tree.setCellRenderer(new CustomTreeCellRenderer());
```

```
public class MyCellRenderer extends DefaultTreeCellRender
{   //override
    public Component getTreeCellRendererComponent (
        JTree tree, Object value, boolean sel, boolean
         expanded, boolean leaf, int row, boolean hasFocus) {
        super.getTreeCellRendererComponent(...);
        if(value!=null){
            ...   //additional code goes here
            this.setIcon((Icon)value);}
         return this;
    }
}
//following code fragment sets the cell renderer
JTree tree = new JTree (...);                         D
tree.setCellRenderer(new MyCellRenderer());
```

Fig. 1. Examples of framework extension points (Dimension and TreeCellRenderer) and extension patterns

There are three different ways to customize the cell rendering behavior of *JTree*:

1) One can call the *setCellRenderer* method with an object of the existing framework class *DefaultTreeCellRenderer*, which implements the *TreeCellRenderer* interface. One can change the cell rendering behavior by calling a set of methods on the *DefaultTreeCellRenderer* object prior to using that object as an argument (Figure 1(B)).
2) Alternatively, one can create a new class to implement

TABLE I
COMPARISON OF THIS WORK (FEMIR) WITH PRIOR RESEARCH ON FRAMEWORK EXTENSION POINTS (Y: WELL-ADDRESSED, P: PARTIALLY ADDRESSED)

Supported Queries	Pattern Extractor [1]	FrUit [2]	SpotWeb [3]	Core [4]	FEMIR
Summary of all extension points for a framework class	P	P	Y	-	Y
Summary of frequent usage patterns of an extension point	-	-	-	-	Y
Examples that illustrate how extension points are used	-	-	-	-	Y
Extension points that are often used together	-	-	-	-	Y
Framework methods designed to be overridden	Y	Y	Y	Y	Y
Framework methods that are often overridden together	-	-	-	Y	Y
Super-implementations that should be called by overriding methods	-		-	Y	Y
Framework methods that should be called by overriding methods	-	P	-	Y	Y

the *TreeCellRenderer* interface and override the *getTreeCellRenderComponent* method (Figure 1(C)).

3) Or one can subclass *DefaultTreeCellRenderer* and pass it as the argument for *setCellRenderer* (Figure 1(D)).

By studying existing projects a developer can discover many examples of how an extension point is used, however this can be quite time consuming. To address this, we propose to mine large code repositories and automatically locate examples of framework extension point use. We also propose a taxonomy of extension patterns to categorize the located usage examples. Furthermore, to utilize these mined examples, we develop a two-step recommendation system. A developer first selects an extension point related to the framework object that they are working with. Once selected, the recommender then shows code examples of the relevant extension patterns for perusal.

We evaluate the efficacy of our two-step recommendation system using five different frameworks and 1,267 projects collected from GitHub, Our evaluation of the recommender uses the standard ten-fold cross validation. The precision for recommending extension patterns ranges from 78% to 90%, and the recall ranges from 56% to 79% for the top-5 recommendations. Our evaluation indicates that our proposed technique is promising in supporting the use of extension points. Our approach is implemented as an Eclipse plugin called *FEMIR* (**F**ramework **E**xtension **Mi**ner and **R**ecommender). Thus, our paper makes the following contributions:

- A taxonomy of framework extension patterns,
- A technique that combines syntactic analysis and graph-based mining algorithms to recommend framework extension points and their usage patterns,
- An evaluation of our proposed technique in terms of precision and recall, and
- A set of statistics on framework extension points.

The rest of the paper is organized as follows. Section II presents related work. Section III defines a taxonomy of extension patterns. Section IV presents our approach for mining and recommending extension patterns. Section V evaluates our approach, including its accuracy, a qualitative study of the quality of the recommended patterns, and statistics that characterize framework extensions. We discuss several questions related to our study in Section VI, and threats to validity in Section VII. Finally, Section VIII concludes the paper.

II. RELATED WORK

The most relevant work to our study is *XFinder* by Dagenais and Ossher [5]. *XFinder* is a tool that requires developers to create guides as a sequence of steps for extending a framework. Given a code base, a guide, and a framework, *XFinder* can locate examples that implement each step of the guide. Although they did not focus on finding examples of different ways of using an extension point, their technique can do so only if the developer manually creates a guide first. However, our technique can automatically find extension points, categorize their usage patterns, and locate code examples. Bruch et al. proposed a technique that mines four sub-classing directives of frameworks [4]. These directives are pieces of documentation that describe the methods of a framework class that need to be overridden, super and framework methods that should be called inside an overriding method, and typical co-overridden methods of a framework class. Our technique captures not only sub-classing directive information but also how they can be used with other parts of the code. Michail [6] developed a technique, called *CodeWeb*, that captures reuse relationships between a software library and user applications. The technique was improved by including inheritance hierarchy in the reuse relationships and by using generalized association rules [7]. Bruch et al. [2] developed a technique, called *FRUIT*, that mines Java bytecodes to create framework usage scenarios on five class properties (extends, implements, overrides, calls and instantiates). Patterns are identified by applying an association rule mining technique on those scenarios. However, none of the above techniques focus on finding framework extension points or different ways of interacting with an extension point. Thummalapenta and Xie [3], [8] developed a technique, called *SpotWeb*, that determines both frequently (hotspots) and rarely (coldspots) used framework classes and methods by leveraging code search engines. While *SpotWeb* can provide an overview of framework usage, it does not find extension point usage patterns and their examples. There are also a number of other studies that aim to improve framework reuse. These include the work on documenting framework patterns [9], developing concept implementation template [10], creating cookbooks containing feature recipes [11], and documenting proven solutions to the frequently appearing challenges in understanding frameworks [12]. However, the objectives of these studies are different than ours.

Previous works on mining API usage patterns are related

to our study because we also apply graph mining technique to locate framework extension patterns. A number of techniques have been proposed in the literature for mining API usage patterns. For example, Acharya et al. [13] proposed a technique that mines API usage patterns as partial orders, by leveraging static traces of source code. Zhong et al. [14] proposed an API usage pattern mining framework, called *MAPO*. The technique applies a clustering technique to group related API calls, generates method call sequences for each cluster and then applies a sequential pattern mining technique to discover frequent patterns from those sequences. Nguyen et al. [15] proposed *GrouMiner*, a graph-based approach that can mine frequent usage patterns involving multiple objects from source code. Wang et al. [16] proposed a technique that uses a combination of closed frequent pattern mining approach and two-steps clustering to find succinct and high coverage API usage patterns. The patterns the above techniques mine are typically located in one method, whereas our technique focus on finding different patterns of using extension points that often span across different classes, methods, and files.

Code search techniques are related to our study in that they also focus on finding code examples. A number of techniques have been proposed in the literature. These include but not limited to *Strathcona* [17], *PARSEWeb* [18], XSnippet [19], and the internet-scale code search engine proposed by Keivanloo et al. [20]. However, they are not designed to locate framework extension examples because they typically find code examples in a single method. There are also code search techniques that can find examples from a task description [21] [22] [23]. However, none of these techniques can discover framework extension points and their usage patterns.

III. Taxonomy of Extension Patterns

Extension patterns are common ways of using an extension point. To help manage extension patterns, we categorize them into four broad categories as follows.

A. Simple

A *simple* extension pattern passes an argument object of a framework class to an extension point, without any further customization on the object. This pattern does not require extending a class or implementing a framework interface. Figure 1(A) shows an example of the simple extension pattern for the *Dimension* extension point, which is the formal parameter of the *setSize* method of the *JFrame* class. Notice that if there exist multiple framework classes that can be used as argument types, multiple *simple* patterns may exist for the same extension point.

B. Customize

Developers often need to call a set of methods on the argument object of a framework class to customize its behavior. Such an extension pattern belongs to the *customize* category. As an example, Figure 1(B) shows such a pattern for the *TreeCellRenderer* extension point, where four methods are called on the *DefaultTreeCellRenderer* object before it is passed to *setCellRenderer*. Notice that if there exist multiple framework classes that can be used as argument types, multiple *customize* patterns may exist for the same extension point.

C. Extend

In an *extend* pattern, a new class is created to customize an extension point by extending a framework class. Optionally, additional method calls may be made on the argument object. Figure 1(D) shows an example of this pattern for the *TreeCellRenderer* extension point. In this example, a new tree cell renderer is created by extending the existing *DefaultTreeCellRenderer* class. Notice that if there exist multiple framework classes that can be used as argument types, multiple *extend* patterns may be found for the same extension point.

D. Implement

An *implement* extension pattern occurs when the extension point is a framework interface. To customize the extension point, a client class is created by implementing the interface. Its object is then used as argument. Optionally, additional method calls may be made on the argument object. As an example, consider the *TreeCellRenderer* interface in Figure 1(C). To customize the cell rendering behavior of a *JTree*, a new class implements the *getTreeCellRendererComponent* method of the *TreeCellRenderer* interface. An object of the new class is then passed to the *setCellRenderer* method of the *JTree* class.

IV. Technical Description

Our approach for mining and recommending examples of framework extensions consists of two components, a *graph miner* and a *recommender*, whose working processes are summarized in Figures 2 and 3, respectively. The graph miner is responsible for mining and organizing the usage patterns for a framework extension point. Upon a developer's request for help on a selected extension point from a class, the recommender displays a set of code examples to illustrate all of its relevant extension patterns.

A. Miner

In this section, we briefly describe the seven components shown in Figure 2 of our graph miner.

1) ***Framework Information Collector****:* This component accepts a framework jar file as the input and collects information on framework classes, interfaces, and methods. For each class or interface, we collect its name, super classes, implemented interfaces, and the list of methods. For each method, we collect its name, return type, and types of parameters.

2) ***Code Downloader****:* This component collects open source software projects hosted on GitHub[1] that contain framework usage examples. GitHub hosts a large number of open source Java projects. It also provides APIs to search for code examples. We search repositories using the import statements in Java source files. For example, to identify repositories that use the *Swing* API, we use the following query: *import AND*

[1] https://github.com/

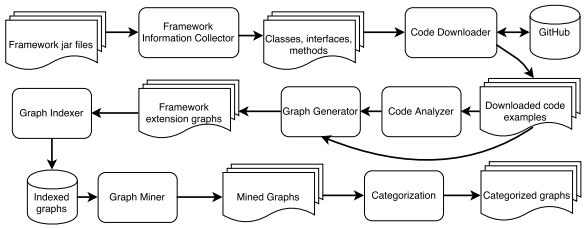

Fig. 2. Working process of the graph miner of FEMIR

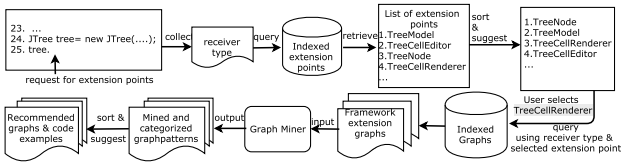

Fig. 3. Overview of the extension patterns recommender of FEMIR

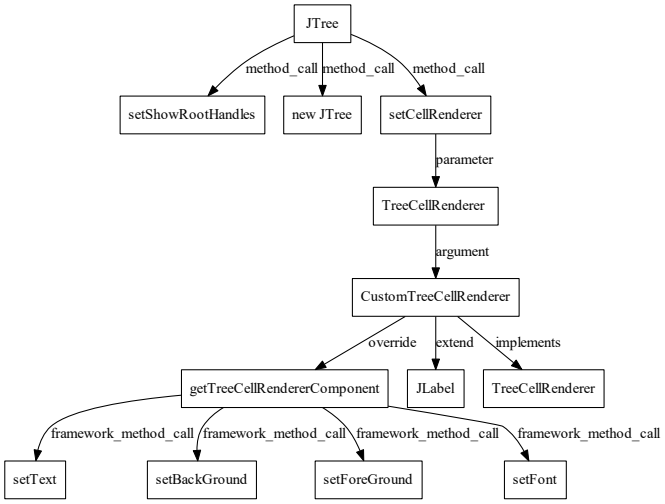

Fig. 4. Framework extension graph for the example in Fig. 1(C))

javax AND swing. After collecting repository information, the technique downloads source code examples.

*3) **Code Analyzer**:* The code analyzer performs static analysis of source code to identify framework extension points and enable the construction of framework extension graphs. It identifies type declarations (classes or interfaces) that are created from framework types, determining their super classes, implemented interfaces and overridden methods. It also identifies those method calls where a receiver is a framework type or a sub-type, and resolves type bindings of both receivers and arguments. The Eclipse JDT parser is used for parsing and we use partial program analysis to resolve type bindings [24].

*4) **Framework Extension Graph Generator**:* At the core of each framework extension graph is a method call that represents a use of a framework extension point. To be counted, the method call must have at least one parameter that is related to a framework type. A framework extension graph thus consists of one or more of the following node types:

- **Receiver type:** A node representing the receiver type of a method call or the class in the case of a constructor call.
- **Method call:** A node representing a method or constructor call.
- **Parameter type:** A node representing the type of a parameter that is either a class or an interface.
- **Argument type:** A node representing the declared type of a method call argument.
- **Other receiver method calls:** All other framework method calls on the receiver variable, including the construction call that creates the variable.
- **Other argument method calls:** All other framework method calls on the argument variable.

If the receiver or the argument of a call is of a client type that extends a framework type, we collect information on the extended classes, implemented interfaces and overridden methods. Thus, a framework extension graph can also contain the following kinds of nodes:

- **Extended class:** A node representing the parent class in the inheritance hierarchy.
- **Implemented interface:** A node representing an implemented interface.
- **Overriding method:** A node representing a method that overrides a framework method. We collect the method name, return type, parameter types, and the type that declare the method.
- **Super method call:** The super method called by an overriding method.
- **Framework method calls:** A set of framework methods that are called by an overriding method.

The extension graph shown in Figure 4 illustrates the various types of nodes introduced above.

*5) **Graph Indexer**:* To enable future retrieval, the extension graphs are indexed by the respective framework-related types of the receiver, the formal parameter, and the argument.

*6) **Graph Miner**:* Given an extension point for a framework class, the goal for the graph miner is to identify the frequent patterns for extending the framework class by generating and counting subgraphs. Given a set of n graphs (also known as base graphs) that belong to the extension point, the miner iteratively and incrementally generates all of their subgraphs as follows. In the first step, it generates all one-node graphs for each base graph. In each subsequent step, the generated subgraphs are counted by comparing their canonical forms (see below). The top-k frequent subgraphs are kept as the starting point for the next step. In the next step, these top-k subgraphs are grown by adding an adjacent node from the base graph. As a result, a new set of subgraphs are generated. The process continues until all nodes of the base graphs are exhausted.

The miner categorizes the graph patterns generated above into the extension pattern categories as defined in Section III. For example, consider the extension patterns for the *TreeCellRenderer* extension point. If a candidate pattern contains an argument node that is created by extending *DefaultTreeCell-*

Node Label	Node index	Neighbor Node Index	Out Degree
JTree	101	21, 35, 40	3
setShowRootHandles	21	-	-
new JTree	35	-	-
setCellRenderer	40	67	1
TreeCellRenderer	67	55	1
Client	55	71, 77, 99	3
getTreeCellRendererComponent	34	30, 45, 87, 201	4
JLabel	99	-	-
TreeCellRenderer	77	-	-
setText	201	-	-
setBackGround	87	-	-
setForeGround	45	-	-
setFont	30	-	-

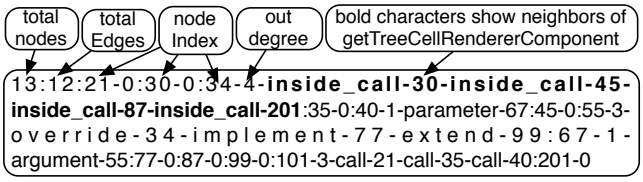

Fig. 5. Canonical form of the graph shown in Fig. 4. Each node is represented by an index, an out degree and a list of neighbor nodes, separated by hyphens. Each neighbor node is represented by an edge label (e.g., inside_call) and its index. Nodes are sorted by index and separated by colons.

Renderer, we assign it to the *extend* category. In contrast, if the argument is an object of the framework class *DefaultTreeCellrenderer* with a set of additional methods called on it, we assign the graph in the *customize* category. Notice that due to the existence of nodes such as argument and receiver calls, more than one pattern may be generated for each extension pattern category. For each category, we pick the top-n candidate graph patterns with the highest frequencies. We call them *mined graph patterns*.

However, since the mined graph patterns are subgraphs generated from the base graphs, they may not be ideal for representing a pattern category because they may miss some essential nodes in the base graphs. Thus, the miner further improves a mined graph pattern p as follows. It first determines the *support* of those nodes that are present in base graphs that contain p but not in p itself. The support of such a node is defined as the ratio of the number of base graphs that contain the node over all the base graphs. If the support for a node exceeds a predefined threshold value δ, it is added to the graph pattern. During our experiment, we find that 0.30 is a good value to work with.

To represent and compare graphs, the graph miner uses an approximated canonical form. To answer the question we need to determine whether two graphs are isomorphic to each other or not. Although there are practical algorithms for testing graph isomorphism, they are computationally expensive [25]. As an alternative, we use their canonical forms [26], because the canonical forms of two isomorphic graphs should be identical too. Unfortunately, determining the canonical form of a graph is also computationally expensive [15]. Therefore, we approximate the canonical form by using graph invariants, which are structural properties of a graph. Specifically, we create the canonical form of a graph by concatenating the following graph invariants into a string:

1) The total number of nodes in the graph
2) The total number of edges in the graph
3) The list of nodes ordered by index
4) Each node is represented by an index, out degree, and a list of neighbors ordered by index

A node index is calculated for each graph node by hashing a string concatenating its out degree, name, and node type. Figure 5 shows an example on how the canonical form is calculated for the framework extension graph of Figure 4. Lastly, to enable comparison, all client classes derived from a framework type (such as the *CustomTreeCellRenderer* in Figure 4) are represented using the same label *client*.

An analysis of the computational complexity follows. To mine m base graphs each containing n nodes, the graph miner would need to generate a total of $O(m*2^n)$ subgraphs. While the value of m can be very large for a large repository, due to the practice of writing shorter methods, on average, the value of n is not. To control the number of generated subgraphs, we limit the maximum size of the candidate graph patterns to 20. Furthermore, each step of the graph mining process passes on only the top k frequent candidate patterns to the next step. In our experiment, we choose $k = 500$ because we believe that practically no extension point would have more than 500 interesting patterns.

B. Recommender

FEMIR recommends the most likely patterns for each extension point that a developer asks for help. To learn how to extend the functionality of a framework class, the developer requests the help from FEMIR by typing a dot after a variable of the class. FEMIR first shows the developer the list of extension points that are applicable to the class. Once the developer selects an extension point, FEMIR then shows multiple patterns for each of its extension categories.

Specifically, FEMIR first retrieves the subset of extension graphs that belong to the extension point from the set of all framework extension graphs that are built by the miner previously. To do so, it utilizes the names of the framework calss and the selected extension point as indices. FEMIR then utilizes the graph miner described above to generate and group the graph patterns by extension pattern categories. FEMIR sorts the patterns belonging to each category based on their usage frequency in the training data. Finally, FEMIR recommends the top-n patterns for each category for the developer's perusal. Once the developer selects a recommmended pattern, FEMIR shows the actual code examples that contain the pattern.

V. Evaluation

In this section, we first evaluate the performance of FEMIR in recommending code examples. To further illustrate

TABLE II
SUMMARY OF FRAMEWORK EXTENSION DATASET USED IN THIS STUDY

Framework	Classes	Methods	Projects	Files	LOC
Swing	466	10,251	263	107,380	9.94 M
JFace	524	7880	300	151,523	11.20 M
JUnit	121	944	242	123,220	10.50 M
JGraphT	201	1493	295	82, 621	9.50 M
JUNG	342	3306	167	76,376	8.80 M

FEMIR's capability qualitatively, we also present a set of extension patterns that are recommended by FEMIR. Lastly, we present a set of statistics to characterize framework extensions.

We choose five different open source frameworks for our evaluation: *Swing*, *JFace*, *JUnit*, *JUNG*, and *JGraphT*. These frameworks are widely used for application development. They are also used in prior research [3]. Table II summarizes the five frameworks used in this study, including the numbers of projects and source files analyzed. FEMIR is realized as an Eclipse plug-in. All experiments are conducted on a machine with INTEL Core i7 CPU (2.93 GHz), 16 GB RAM.

A. Accuracy of FEMIR recommendation

This section presents an experimental study to understand the accuracy of our proposed technique FEMIR. To evaluate FEMIR, we assume a scenario where a developer decides to extend the functionality of a framework class, but does not know how to do that. She requests the help from FEMIR after declaring a variable of that framework class. This is done by typing a dot(.) after the variable name. FEMIR first shows the names of a list of extension points that the developer could use to extend the functionality of the framework class. After she selects an extension point, FEMIR shows the different patterns of using it. Given an extension point, we thus evaluate the effectiveness of the technique in recommending a framework extension graph that matches with the actual usage of the extension point.

1) Evaluation procedure and metrics: We develop an automatic evaluation system that analyzes the source code files and collects framework extension graphs from the subject systems for each framework. Thus, for each framework variable v, we know the fully qualified type name of the variable (t_v), the extension points used ($e_1, e_2, e_3, ..., e_n$), and the framework extension graphs ($g_1, g_2, g_3, ..., g_n$) that illustrate how those extension points are used in the code.

Each extension is considered as a data point in our evaluation. We apply the popular ten-fold cross validation to measure the performance of FEMIR. Specifically, we divide the data set into ten folds, each containing an equal number of extensions. Next, for each fold, we use code examples from the nine other folds to train FEMIR for recommendation. The remaining fold is used to test the performance of the technique. For each test data point, FEMIR recommends the top-n extension graphs for each extension category to show the different ways of using the extension point.

Precision and recall are calculated as follows. Let *O* denotes the original graph under testing and *S* is the graph suggested by FEMIR. Because *S* can be useful even if it is not identical to *O*, we do not simply determine whether S and O are identical or not. Instead, we determine the common nodes shared between these two graphs. We use precision, recall and F-measure metrics to measure the performance, which are defined as follows. If a node in *O* occurs in *S*, we consider it a correctly recommended node (increasing precision). On the contrary, if a node in *O* does not occur in *S*, we consider it a missing node (lowering recall).

$$Precision = \frac{total\ correctly\ recommended\ nodes}{total\ recommended\ nodes} \quad (1)$$

$$Recall = \frac{total\ correctly\ recommended\ nodes}{total\ recommendation\ needed\ nodes} \quad (2)$$

$$F - measure = 2\ .\ \frac{precision\ .\ recall}{precision + recall} \quad (3)$$

To further illustrate the calculation of precision and recall, consider the example depicted in Figure $1(c)$, where a new class implements the *TreeCellRenderer* interface. Suppose this is what a developer eventually wants to create but does not know how initially. So the developer requests FEMIR to help. FEMIR then suggests the list of extension points of *JTree*, such as *TreeCellRenderer*, *TreeNode*, and *TreeModel*. Suppose that the developer selects the *TreeCellRenderer* extension point. In response, FEMIR recommends the top-n graph patterns from each category, for the developer's perusal. To calculate precision and recall, FEMIR compares the test case with the top-n graphs from the same category and report the precision and recall from the graph that yield the best F-measure. Figure 6 shows the best graph recommended by FEMIR, where missing nodes are highlighted in red, and incorrect nodes are highlighted using blue. Thus, the precision and recall for this recommendation are 10/12 and 10/13, respectively.

We refer to the strategy described above as the *local* strategy because it recommends the top-n patterns from within a category, which is denoted as FEMIR-Local in Table III. An opposite strategy is to recommend the global top-n patterns regardless of their categories. We denote the global strategy as FEMIR-Global in Table III.

2) Alternative strategies for maximizing accuracy: Recall that when recommending framework extension graphs (Section IV-A6), FEMIR first mines the most frequent graph patterns for each extension pattern category. To improve accuracy, it then applies a greedy strategy to add to each graph pattern more nodes that are deemed important but missed during the mining process. To further explore other options, we compare the greedy strategy with an alternative that is called the diversity strategy.

The diversity strategy works as follows. After determining the top-n graph patterns of an extension pattern category, we determine all the base graphs in the training data that contain the patterns. Among these base graphs, we recommend the ones that contain the largest number of different node types. We refer to this strategy as the diversity strategy, and denote it as FEMIR-D in Table III.

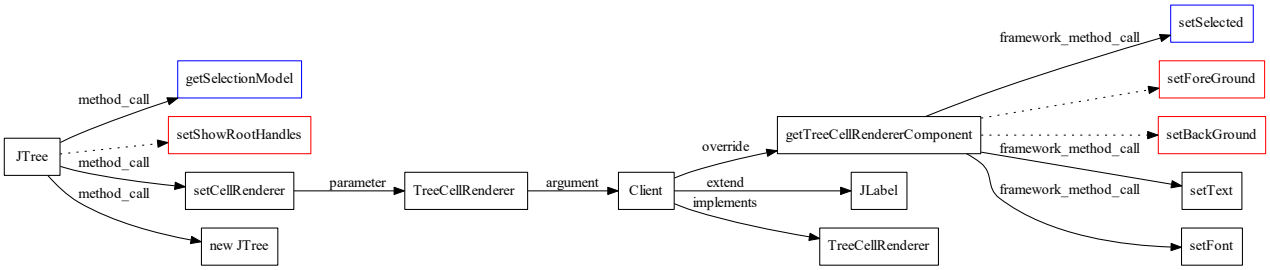

Fig. 6. Extension pattern recommended for Figure 1(C), where matching nodes are shown in black, missing nodes in red, and incorrect nodes in blue.

TABLE III
EVALUATION RESULTS OF RECOMMENDING FRAMEWORK EXTENSION GRAPHS (RMC: RECEIVER METHOD CALL, AMC: ARGUMENT METHOD CALL, E: CLASS EXTENSION, I: INTERFACE IMPLEMENTATION, FMC: FRAMEWORK METHOD CALL, O: OVERRIDDEN METHOD, OTHER: OTHER NODE TYPES)

Framework	Technique	RMC AMC	E I	O FMC	Other	Precision			Recall			F-Measure		
						Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Swing	FEMIR-Global	30%	6%	14%	50%	0.89	0.87	0.85	0.66	0.76	0.79	0.76	0.81	0.82
	FEMIR-Local					0.89	0.89	0.89	0.71	0.72	0.73	0.79	0.79	0.80
	FEMIR-D					0.20	0.41	0.42	0.31	0.67	0.73	0.24	0.51	0.53
JFace	FEMIR-Global	12%	6%	26%	56%	0.76	0.78	0.78	0.51	0.61	0.64	0.61	0.68	0.70
	FEMIR-Local					0.81	0.82	0.82	0.44	0.48	0.49	0.57	0.60	0.61
	FEMIR-D					0.25	0.34	0.43	0.37	0.56	0.63	0.30	0.42	0.51
JUnit	FEMIR-Global	17%	5%	12%	66%	0.90	0.90	0.90	0.60	0.67	0.69	0.72	0.77	0.78
	FEMIR-Local					0.89	0.92	0.92	0.61	0.68	0.70	0.72	0.78	0.80
	FEMIR-D					0.64	0.70	0.70	0.61	0.70	0.71	0.63	0.70	0.71
JGraphT	FEMIR-Global	31%	7%	5%	57%	0.86	0.85	0.85	0.41	0.53	0.56	0.55	0.65	0.67
	FEMIR-Local					0.86	0.87	0.87	0.42	0.52	0.55	0.57	0.65	0.67
	FEMIR-D					0.72	0.75	0.76	0.40	0.61	0.64	0.52	0.67	0.70
JUNG	FEMIR-Global	35%	6%	5%	54%	0.87	0.86	0.86	0.59	0.67	0.71	0.70	0.76	0.78
	FEMIR-Local					0.88	0.88	0.88	0.61	0.67	0.70	0.71	0.76	0.78
	FEMIR-D					0.56	0.70	0.71	0.51	0.67	0.70	0.56	0.68	0.70

3) Evaluation results: Table III shows the precision, recall, and F-measure values for recommending target framework extension graphs. Furthermore, columns three to six show the percentages of different kinds of nodes in the test cases. The largest number of nodes are of *other* type. Together *RMC* (receiver method calls) and *AMC* (argument method calls) represent the second largest group of nodes. Although the percentages of E, I, O and FMC nodes are much smaller in number, they would be the most difficult to use because of the inheritance structure and limited usage examples.

In general, FEMIR-Global performs well in all these test cases, with precision ranging from 78% to 90% and recall 56% to 79% for the top-5 recommendations. FEMIR-Local performs close to FEMIR-Global. While the precision of FEMIR-Local is slightly better, ranging from 82% to 92%, the recall ranges between 49% to 73% for the top-5 recommendations, which are slightly lower than FEMIR-Global. More investigation will be needed to explain the difference between FEMIR-Local and FEMIR-Global.

The result of FEMIR-D is not better than FEMIR-Global either. In fact, the precision and recall values are lower than FEMIR-Global in all cases except for *JGraphT*, where we observe that FEMIR-D has slightly better F-measure than FEMIR-Global.

B. Examples of framework extension patterns

In this section, we report a qualitative evaluation of the quality of the patterns recommended by FEMIR. We consult tutorials (both official and community-based) to find a set of relevant patterns. Our goal is to check whether FEMIR is able to identify these patterns by mining code bases. We focus on Java *Swing* and *JFace* due to our familiarity with them. [2] [3] [4].

1) TableRowSorter: Tables in Java Swing support sorting and filtering capability. To do so, it is required to pass an instance of *TableRowSorter* to the *setRowSorter()* method. However, to gain more control, one can override *TableRowSorter* or its parent class *DefaultRowSorter*. Figure 7(A) depicts an example pattern mined by FEMIR for using *TableRowSorter*. FEMIR is able to mine patterns for controlling table sorting in all three different ways. Furthermore, a developer can provide a filter object by calling *setRowFilter()* method on the sorter object to control which rows will be displayed. Our mined pattern is able to collect that information too. All of these indicate that FEMIR is able to show different ways of using an extension point, providing the developer the opportunity to select the one that best matches with her goal.

[2]https://wiki.eclipse.org/Eclipse_Corner
[3]http://www.javaworld.com/
[4]https://docs.oracle.com/javase/tutorial/uiswing/

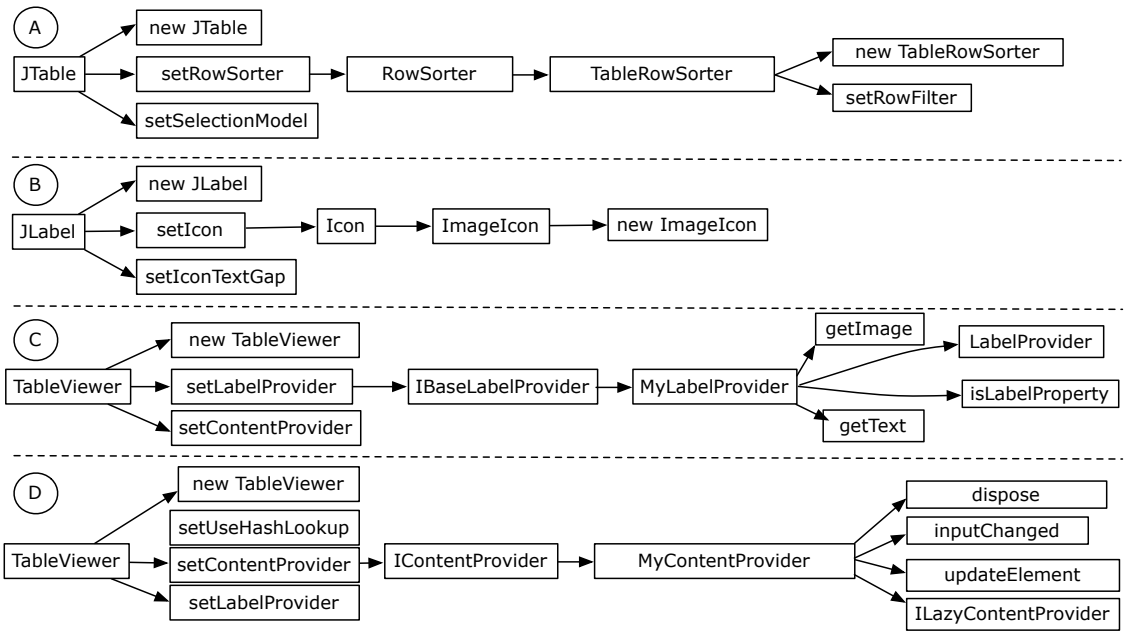

Fig. 7. Example framework extension patterns mined by FEMIR: A *customize* extension pattern for the *RowSorter* extension point (A), a *simple* pattern for *Icon* (B), and two *extend* extension patterns for *IBaseLabelProvider* (C) and *IContentProvider* (D), respectively.

2) Icon: Many swing components can be decorated with an icon, a small fixed size picture. Figure $7(B)$ shows a pattern where an icon is used jointly with a *JLabel*. The pattern also shows that developers frequently call other methods to set the properties of an icon. An alternative to this pattern is to implement the Icon interface to create a custom icon (not shown). FEMIR is able to mine both patterns. Again, this example shows that FEMIR can collect all popular ways to complete a task and allow the developer to select the option that best fits her needs.

3) LabelProvider: A label provider allows viewers to customize the display of labels. By default, a label provider uses the element's *toString* value to display text and null for image. Figure $7(C)$ shows a pattern of using a custom *LabelProvider* for the *TreeViewer* component. It also shows that creating a label provider typically involves sub-classing the *LabelProvider* class and overriding the following methods: *isLabelProperty*, *getImage*, *getText*, and *dispose*. This matches with information provided in the documentation of the JFace. However, mined patterns often contain more important details than framework documentation. Thus, results mined by FEMIR could complement documentation as a developer aid.

4) ContentProvider: Eclipse *JFace* viewers support a content provider that establishes the connection between the data and the viewer. Figure $7(D)$ shows a pattern of creating a content provider in order to customize a *TableViewer*. The pattern also shows that when using the content provider, developers also set a label provider by calling the *setLabelProvider* method, which is another extension point for the *TableViewer* (Figure $7(C)$). Thus, this example illustrates that multiple extension points may be used together and a pattern of using one also helps to discover others.

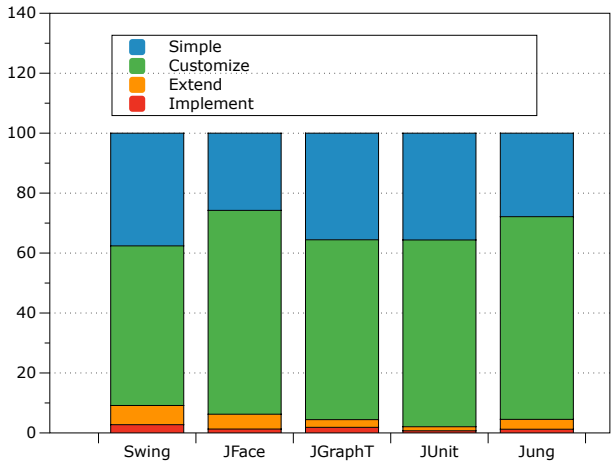

Fig. 8. Distribution of extension patterns by categories

C. Distribution of extension patterns by categories

Further to the evaluation, we want to see the distribution of extension points by categories using the five frameworks. As shown in Figure 8, the distributions are similar across all five frameworks. The most popular category of framework extension patterns is the *customize* usage patterns, which by definition require to call a set of methods on the method argument. The *simple* category, which does not require extending a framework type, or using a variable with several call sites, is the second most popular. Lastly, although the *extend* or

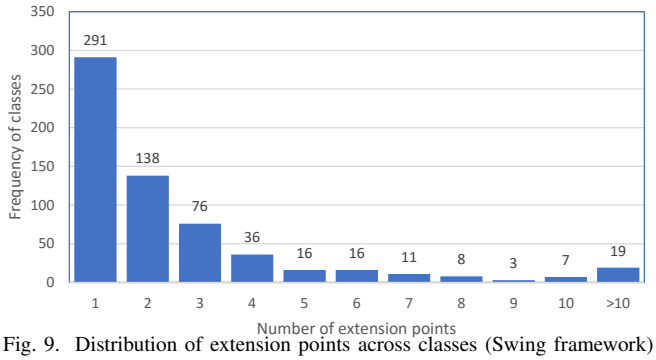

Fig. 9. Distribution of extension points across classes (Swing framework)

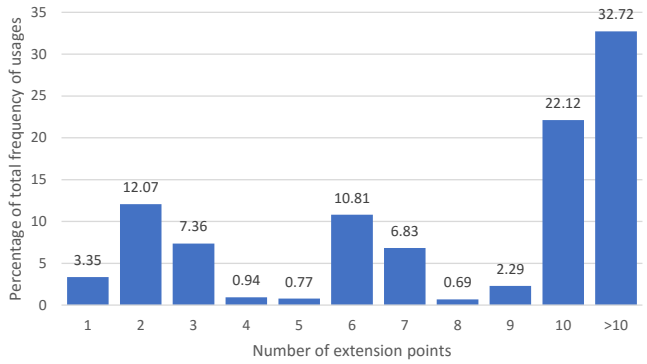

Fig. 10. Usage frequencies of Swing framework classes with different numbers of extension points

implement categories appear to be minor, they require more efforts to learn and use than the first two categories.

D. Distribution of extension points by classes

To understand how usable it is to show extension points to developers, Figure 9 depicts the distribution of extension points across framework classes for Java Swing. Only 4% of the classes have 10 or more extension points. However, this small fraction of framework classes appear to be more frequently used. As shown in Figure 10, more than 54% of all framework extension examples in Java Swing involve such a class.

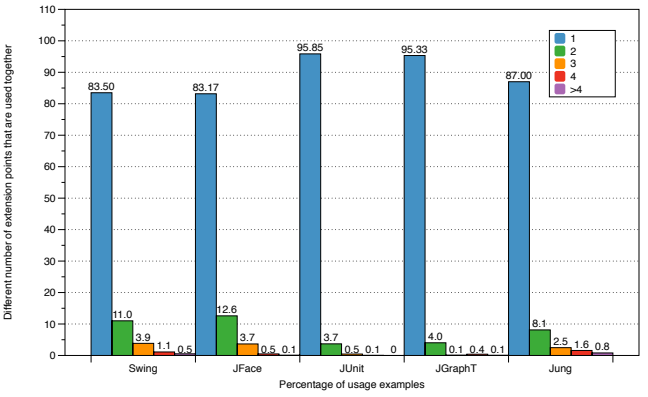

Fig. 11. The percentage of cases where multiple different kinds of framework extension points are used together

E. How often are multiple extension points used together?

A framework class may have multiple extension points. To understand the association between framework extension points, we are interested in knowing how often developers use different extension points together. To answer this question, we collect the data as follows. While parsing the source code, we collect the set of methods that are called on the same receiver object, including the constructor call that creates that object. The framework extension graphs are then grouped together based on the receiver objects. Each group of framework extension graphs provides the set of extension points that are used together. Figure 11 shows how frequently different kinds of extension points are used together.

As shown in Figure 11, most commonly (more than 83% for all frameworks), developers use only one framework extension point. When they do use multiple extension points together, they most often use two (from 3.66% to 8.12%). The percentages decrease as the numbers of different extension points that are used together increase. It seems that developers rarely use five or more different extension points together.

VI. Discussion

This section discusses a set of questions related to our study.

A. Detecting examples of extension patterns by categories

The goal of this section is to explore how well FEMIR performs in detecting examples in each of the four extension pattern categories. We use the same experiment settings (but use only the Swing framework) and evaluation metrics as in Section V. We first categorize the test cases of Java Swing framework into four different extension pattern categories. For test cases of each category, we run the experiment and determine the precision, recall, and F-measure values. Table IV shows the results of the experiment. The result shows that FEMIR is indeed useful in recommending extension patterns for all four categories. Interestingly, the simple extension pattern category produces the lowest precision (0.80) and recall (0.63), and the extend pattern category produces the highest precision (0.92) and recall (0.91). More investigation is needed to understand what causes the differences between the categories, which remains as future work.

B. Quality of canonical forms

Two graphs with identical canonical forms are not necessarily isomorphic. To test the accuracy of our canonical form representation, we generate a graph database using one hundred subject systems for the Java *Swing* framework. Then, we enumerate through the list of canonical forms in the graph database. We collect the set of graphs that share the same canonical form. We then apply the graph isomorphism detection algorithm by Mckay [27] to all pairs of graphs in the set. The Mckay algorithm indeed identifies all pairs of graphs as being isomorphic. This confirms that the canonical form works correctly.

TABLE IV
EVALUATION RESULTS OF FEMIR FOR EACH EXTENSION PATTERN CATEGORY

Extension Pattern Category	Precision			Recall			F-Measure		
	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5	Top-1	Top-3	Top-5
Simple	0.83	0.82	0.80	0.55	0.62	0.63	0.66	0.71	0.71
Customize	0.92	0.89	0.87	0.73	0.76	0.87	0.82	0.82	0.87
Extend	0.91	0.92	0.92	0.77	0.90	0.91	0.83	0.91	0.91
Implement	0.88	0.85	0.85	0.57	0.65	0.65	0.69	0.73	0.74

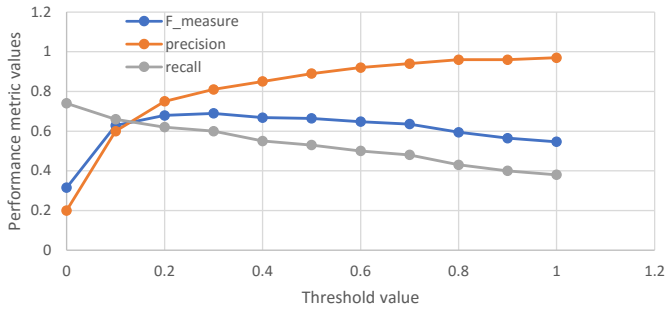

Fig. 12. Precision, recall, and F-measure at different threshold values

C. Effect of the threshold value

As explained in Section IV.A, our proposed technique uses a threshold value to determine the missing nodes that need to be added to the extension graph. To understand the effect of this threshold value, we conduct a study using *JFace* and its subject systems. It is the largest of the five different frameworks we have considered in our study. We use the ten-fold cross validation startegy and determine the precision, recall and F-Measure values by changing the threshold value from 0 to 1, at 0.10 interval. Figure 12 shows that the precision increases at the expense of recall. To balance the effect, our goal is to select a value that maximizes the F-measure value. The value increases with the increase of the threshold value and reaches to its highest value at 0.3 threshold. After that we observe a gradual decrease of the F-measure value. Hence we recommend to use 0.3 in our study.

It should be noted that we assume in this paper that the costs of precision and recall errors are the same. If a precision error and a recall error have a different cost (for example, it would be reasonable to assume that a recall error is more costly than a precision error because developers could easily filter out irrelevant nodes), a different threshold will be chosen.

D. Runtime performance of FEMIR

To measure the runtime performance of the technique, we measure the time required to make recommendations. Based on executing a total of 16,268 queries, the average time required to recommend framework extension graphs per query was 0.92s for Java Swing. The major part of the recommendation time is needed for mining framework extension graphs. However, a significant fraction of the time can be saved if we mine the graphs for all extension points beforehand and index them to be used by the recommender later.

The time required to analyze the source code files and generate framework extension graphs requires significantly long time. For example, for *JFace* alone, it takes around 78 hours on a single node machine for FEMIR to generate framework extension graphs from the source code files. The time is mostly contributed by partial program analysis of the source code. However, this is only a one time operation.

VII. THREATS TO VALIDITY

There are a couple of threats to the validity of this study.

First, we consider five frameworks to evaluate our proposed technique. One can argue that the conclusions can be different for other frameworks. However, we would like to point out to the fact that the frameworks we consider in our study are popular and a large number of Java systems are actively using them. Given that our proposed technique does not directly depend on a particular framework, we believe that the results we obtain should largely carry over to other frameworks.

Second, in this study, we collect software systems that are publicly hosted in GitHub. It is possible that software systems hosted in a different project hosting site other than GitHub, or close source projects can exhibit different framework extension point usage patterns. To mitigate this effect, we consider those projects in GitHub that are active in development, have a long development history, and are large in size. Instead of considering only a few Java projects, we also consider a large number of projects in our study.

VIII. CONCLUSION

In this paper, we propose an approach to recommending framework extensions by mining previously written source code examples. We first define the concept of framework extension points and propose a taxonomy of framework extension patterns. Based on these, we then develop a graph mining approach for recommending the top-n frequent extension patterns. We hypothesize that showing these patterns and code examples will usefully aid developers in learning how to use the extension points. Using ten-fold cross validation, we evaluate the accuracy of our recommender using a large set of applications built on top of five popular frameworks. We show that our proposed technique can help developers automatically discover not only the framework extension points, but also patterns of using those extension points, with reasonable accuracy. We also collect several statistics to characterize the framework extensions in our code base.

In the future, user studies will be needed to capture feedback from developers in order to further improve our technique. Furthermore, more investigation is also needed to better understand the inner working of our recommender.

REFERENCES

[1] J. Viljamaa, "Reverse engineering framework reuse interfaces," *SIGSOFT Softw. Eng. Notes*, vol. 28, no. 5, pp. 217–226, 2003.

[2] M. Bruch, T. Schäfer, and M. Mezini, "FrUiT: IDE Support for Framework Understanding," in *Proc. of the OOPSLA Workshop on Eclipse Technology eXchange*, 2006, pp. 55–59.

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

[4] M. Bruch, M. Mezini, and M. Monperrus, "Mining subclassing directives to improve framework reuse," in *Proc. of the 7th IEEE Working Conference on Mining Software Repositories*, 2010, pp. 141–150.

[5] B. Dagenais and H. Ossher, "Automatically Locating Framework Extension Examples," in *Proc. of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008, pp. 203–213.

[6] A. Michail, "Data Mining Library Reuse Patterns Using Generalized Association Rules," in *Proc. of the 22nd International Conference on Software Engineering*, 2000, pp. 167–176.

[7] ——, "Data Mining Library Reuse Patterns in User-Selected Applications," in *Proc. of the 14th IEEE International Conference on Automated Software Engineering*, 1999, pp. 24–33.

[8] S. Thummalapenta and T. Xie, "SpotWeb: Detecting Framework Hotspots via Mining Open Source Repositories on the Web," in *Proc. of the 2008 International Working Conference on Mining Software Repositories*, 2008, pp. 109–112.

[9] R. E. Johnson, "Documenting Frameworks Using Patterns," in *Proc. on Object-oriented Programming Systems, Languages, and Applications*, 1992, pp. 63–76.

[10] A. Heydarnoori, K. Czarnecki, W. Binder, and T. T. Bartolomei, "Two Studies of Framework-Usage Templates Extracted from Dynamic Traces," *IEEE Transactions on Software Engineering*, vol. 38, no. 6, pp. 1464–1487, 2012.

[11] R. F. Q. Lafetá, M. A. Maia, and D. Röthlisberger, "Framework Instantiation Using Cookbooks Constructed with Static and Dynamic Analysis," in *Proc. of the 2015 IEEE 23rd International Conference on Program Comprehension*, 2015, pp. 125–128.

[12] N. Flores and A. Aguiar, "Patterns for Understanding Frameworks," in *Proc. of the 15th Conference on Pattern Languages of Programs*, 2008, pp. 1–11.

[13] M. Acharya, T. Xie, J. Pei, and J. Xu, "Mining API Patterns As Partial Orders from Source Code: From Usage Scenarios to Specifications," in *Proc. of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, 2007, pp. 25–34.

[14] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, "MAPO: Mining and Recommending API Usage Patterns," in *Proc. of the 23rd European Conference on Object-Oriented Programming*, 2009, pp. 318–343.

[15] T. T. Nguyen, H. A. Nguyen, N. H. Pham, J. M. Al-Kofahi, and T. 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*, 2009.

[16] J. Wang, Y. Dang, H. Zhang, K. Chen, T. Xie, and D. Zhang, "Mining succinct and high-coverage API usage patterns from source code," in *Proc. of the 10th IEEE Working Conference on Mining Software Repositories*, 2013, pp. 319–328.

[17] R. Holmes and G. C. Murphy, "Using Structural Context to Recommend Source Code Examples," in *Proc. of the 27th International Conference on Software Engineering*, 2005, pp. 117–125.

[18] S. Thummalapenta and T. Xie, "Parseweb: A Programmer Assistant for Reusing Open Source Code on the Web," in *Proc. of the Twenty-second IEEE/ACM International Conference on Automated Software Engineering*, 2007, pp. 204–213.

[19] N. Sahavechaphan and K. Claypool, "XSnippet: Mining For Sample Code," in *Proc. of the 21st Annual ACM SIGPLAN Conference on Object-oriented Programming Systems, Languages, and Applications*, 2006, pp. 413–430.

[20] I. Keivanloo, J. Rilling, and Y. Zou, "Spotting Working Code Examples," in *Proc. of the 36th International Conference on Software Engineering*, 2014, pp. 664–675.

[21] J. Stylos and B. A. Myers, "Mica: A Web-Search Tool for Finding API Components and Examples," in *Proc. of the IEEE Symposium on Visual Languages and Human-Centric Computing*, 2006, pp. 195–202.

[22] C. McMillan, M. Grechanik, D. Poshyvanyk, Q. Xie, and C. Fu, "Portfolio: finding relevant functions and their usage," in *Proc. of the 33rd International Conference on Software Engineering*, 2011, pp. 111–120.

[23] S. Chatterjee, S. Juvekar, and K. Sen, "SNIFF: A Search Engine for Java Using Free-Form Queries," in *Proc. of the 12th International Conference on Fundamental Approaches to Software Engineering: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, 2009, pp. 385–400.

[24] B. Dagenais and L. Hendren, "Enabling static analysis for partial java programs," in *Proc. of the 23rd ACM SIGPLAN Conference on Object-oriented Programming Systems Languages and Applications*, 2008, pp. 313–328.

[25] X. Yan and J. Han, "gspan: graph-based substructure pattern mining," in *Proc. of the IEEE International Conference on Data Mining*, 2002, pp. 721–724.

[26] L. Babai and E. M. Luks, "Canonical labeling of graphs," in *Proc. of the Fifteenth Annual ACM Symposium on Theory of Computing*, 1983, pp. 171–183.

[27] B. D. McKay and A. Piperno, "Practical graph isomorphism, II," *CoRR*, vol. abs/1301.1493, 2013.