Mining Extension Point Patterns in Scala

Yunior Pacheco[1,2], Jonas De Bleser[1], Tim Molderez[1], Dario Di Nucci[1], Wolfgang De Meuter[1], Coen De Roover[1]

[1]*Vrije Universiteit Brussel*
Brussels, Belgium
{ypacheco, jdeblese, tmoldere, ddinucci, wdmeuter, cderoove}@vub.be

[2]*Pinar del Rio University*
Pinar del Rio, Cuba

***Abstract*—To use a framework, developers often need to hook into and customize some of its functionality. These customizations are often made by instantiating a type provided by the framework, or by extending or implementing a framework type and instantiating this subtype, and providing the resulting object to other framework objects. Recommending *extension patterns* that frequently occur at such *extension points* can help developers to adopt a new framework correctly and to exploit it fully.**

In this paper we transpose an existing technique for mining extension patterns in Java projects to the Scala context. Our goal is to evaluate whether the unique features of the Scala language have an impact on the mined extension patterns. To this aim, we propose SCALA-XP-MINER, a tool for mining extension point patterns in Scala projects. We preliminary evaluate SCALA-XP-MINER on a corpus of 9 projects using the SWING framework. Our first results reveal that extension points are not very diffused in Scala projects using SWING and that only one type of extension points is adopted by developers.

***Index Terms*—Framework; Extension Points, Usage Patterns, Graph Mining, Scala Language, Mining Software Repository**

I. Introduction

A significant part of software development consists of becoming familiar with APIs from different libraries and frameworks. Libraries and frameworks enable code reuse, provide high-level abstractions for common tasks, and help unify the programming experience [1]. In many cases the flexibility offered by large libraries and frameworks is achieved at the expense of sophisticated APIs that must be accessed by combining API elements into usage patterns, while taking into account constraints and specialised knowledge about the behaviour of the API [2]. Thus, making efficient use and exploiting all the possibilities offered by libraries and frameworks can be quite difficult. The larger and more sophisticated the library or framework, the harder this challenge, due to specific requirements and relations between its components that are difficult to understand. Furthermore, the library or framework may not be documented completely or clearly [1].

When using a framework, it is either necessary or common to extend its functionality. Several studies analysed how developers use libraries in software systems; providing tools to explore and navigate usage examples [3]–[5], documenting techniques [6], and recommending usage patterns obtained from mining code examples ([7], [8]). However, there are relatively few studies that focus on common ways to *extend* a framework and to provide recommendations to developers in this respect.

In this paper, we describe the transposition from Java to Scala of a technique by Asaduzzaman *et al.* [9] for mining framework extension point patterns. Such patterns frequently occur at the points where a framework supports being extended by its users, such as a public method that takes an instance of a framework subtype defined by the users. Once mined for in a sufficient number of projects, extension point patterns can form the basis for recommendations to other projects that use the framework. To this end, we present SCALA-XP-MINER, a tool for mining extension point patterns in Scala projects. SCALA-XP-MINER implements a transposition to Scala of the technique introduced by Asaduzzaman *et al.* [9].

We motivate the transposition not only by the traction that Scala is gaining in industry, but also by the unique features of this programming language that may impact the patterns that occur at framework extension points. We therefore use SCALA-XP-MINER in a preliminary study with the aim of analysing the diffusion and the characteristics of the extension points in Scala projects. In particular, we analysed 9 projects that use the SCALA-SWING framework, a Scala wrapper for the famous SWING GUI framework. Our first results reveal that extension points are not very diffused in our context and that only one type of extension point is adopted by developers.

Structure of the paper. The remainder of this paper is organised as follows: Section II describes the work of Asaduzzaman *et al.* [9], including the definition of *extension points* and *extension point patterns*. Section III presents an overview of the approach, while section IV discusses the results on an initial dataset of Scala projects. Finally, limitations and challenges are discussed in section V.

II. Mining Extension Point Patterns

EXTENSION POINTS have been defined by Asaduzzaman *et al.* as *means provided by a framework, that allow developers to customise its behaviour, to meet application specific requirements*. A common way to extend a framework is to pass one of its objects as an argument to a framework call [9]. Such an argument may be created by subclassing a framework class, implementing a framework interface, or customising the properties of an existing framework object. Nevertheless, there are other ways of extending a framework, but we only consider the one Asaduzzaman *et al.* proposed.

Thus, an extension point comprises a framework method of which the parameters are related to the framework itself. An EXTENSION POINT USAGE is a call to an extension point method, of which at least one of its arguments is either an instance of a type provided by the framework, or an instance of a user-defined type that extends a framework type.

Figure 1(a) depicts an example of extension point usage in Java. The developer is adding a listener to a framework object. The `addActionListener` method declared in the `Button` class, defines a parameter of type `ActionListener`; in this case, the framework is extended by i) implementing the `ActionListener` interface in the `MyActionListener` class and ii) calling the `addActionListener` method with an instance of the class as argument. The behaviour of the `ActionListener` was customised, by overriding the method `actionPerformed`.

Developers can analyse existing projects to find examples of extension points usage, which can be a time-consuming task. Thus, Asaduzzamann *et al.* [9] propose an approach to automatically locate *extension points* and mine *extension point usage*. The approach represents each extension point usage as a separate graph.

An EXTENSION POINT USAGE GRAPH consists of several types of nodes: receiver type, method call, parameter type, argument type, other method calls, extended class, implemented interface, overriding method. To build the extension point usage graphs, it is necessary to parse and analyse the source code of the project that uses the framework. Asaduzzamann *et al.* use ECLIPSE JDT to this end. The extension point usage graph shown in Figure 2 illustrates the graph representation of the Java code in Figure 1(a).

The subgraphs that occur most frequently in the extension point usage graphs are called EXTENSION POINT PATTERNS. These patterns are useful to describe how an extension point is commonly used. In summary, the approach defined by Asaduzzamann *et al.* [9] generates an extension point usage graph for each extension point usage. The graphs are processed, using a frequent subgraph mining algorithm, to extract the extension patterns. The patterns are then grouped in categories according to a taxonomy defined by the authors that describe the complexity of the pattern:

(i) SIMPLE: an instance of a framework class is passed as an argument to the extension point without modifying it;
(ii) CUSTOMISE: before passing the argument of a framework type to the extension point, a number of state changing methods are called on it;
(iii) EXTEND: the argument to the extension point is an instance of a new class that extends a framework class.
(iv) IMPLEMENT: the argument to the extension point is an instance of a new class that implements a framework class.

Finally, for each category, the patterns that most frequently occur in the codebase are shown.

Figure 3 shows an extension point pattern common to, among others, the extension point usage graph in Figure 2 and thus the code in Figure 1(a). In particular, it shows how the method `addActionListener` is used to extend the behaviour of the `Button` class by i) creating an instance of a client class that overrides the method `actionPerformed`, and ii) passing it as the argument to the method call. It is worth noting that the extension point pattern needs to occur more frequently in the mined extension point usage graphs than a threshold to which the mining algorithm has been configured.

III. OVERVIEW OF THE FRAMEWORK

Our framework for mining EXTENSION POINT PATTERNS in Scala projects consists of three components: a source code importer, a pattern miner, and a visualisation tool. Figure 4 depicts the interactions between these components.

We follow a 3-step approach similar to the one proposed in the reference work [9]. First, we build a graph for each of the extension point usages in the Scala projects that depend on the framework under analysis. Next, we mine extension point patterns using the information previously extracted. Finally, we visualise the input and output of the mining algorithm in a way that allows the developer to browse and understand the results.

Source Code Importer. The importer takes the source code of framework clients as input and collects information on framework usages through a static analysis.

For each framework method call, the importer checks whether it corresponds to an extension point. To be considered, the method call must have at least one parameter that is related to a framework type. For each extension point, we collect the method name, the return type, and the types of the parameters.

To construct the extension point usage graphs, the importer resolves the types of the receiver and the arguments to the extension point and determines their type hierarchy and the overridden methods in this type hierarchy. It also identifies method calls of which the receiver is either the same receiver of the extension point or refers to one of its arguments. The importer extracts the required syntactic and semantic information through the SCALA-META[1] library.

Figure 1(b) shows how to add a listener to a framework object in Scala using the SCALA-SWING framework. In this example, the extension point is the `listenTo` method, defined in the framework trait `Reactor`, implemented by the `MainFrame` framework class. This method takes a variable number of objects of type `Publisher` as its arguments. In this case the argument is an instance of class `Button`, a class of the framework. The object was passed to the method call without making changes to its state, that is, without calling methods on the `Button` object before the call to the extension point. Figure 5 shows the extension point usage graph generated for the Scala code example in Figure 1(b).

Extension Point Patterns Miner. The miner is responsible for mining the extension point patterns. The frequent subgraph

[1]https://github.com/scalameta/scalameta

```java
class MyActionListener implements ActionListener {

  @Override
  public void actionPerformed(ActionEvent e) {
    System.out.println("Tickles!");
  }

}

class MyFrame extends JFrame {

  private MyFrame() {
    Button convertButton = new Button("Convert");
    convertButton.addActionListener(new MyActionListener());
  }

}
```

(a)

```scala
class MyFrame extends MainFrame {

  val convertButton = new Button {
    text = "Convert"
  }

  this.listenTo(convertButton)

  reactions += {
    case ButtonClicked(_) => println "Tickles!"
  }

}
```

(b)

Fig. 1: Example extension point usage of the SWING framework in Java (a) and the SCALA-SWING framework in Scala (b).

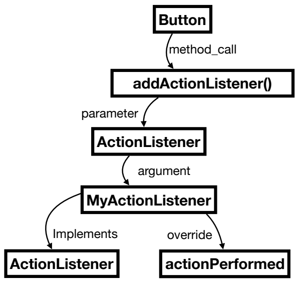

Fig. 2: Example of an EXTENSION POINT USAGE GRAPH representing the code in Figure 1(a)

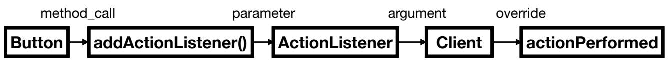

Fig. 3: EXTENSION POINT PATTERN extracted from Figure 2

mining algorithm, used in the miner, takes as input the set of extension point usage graphs generated by the importer in the previous step. The frequent subgraph mining algorithm is a variant of the Apriori algorithm [10]. We mined projects containing code similar to the one shown in Figure 1(b). We first obtained their representation as extension point usage graph as in Figure 5. Afterwards, we used the mining algorithm to obtain the pattern shown in Figure 6. Note that the code in Figure 1(b) correctly matches this pattern.

Visualization Component. The visualization component is used to configure the tool, and to browse through and inspect its results. The developer can select the Scala projects to import through the SOURCE CODE IMPORTER. Then, the EXTENSION POINT PATTERNS MINER analyses the projects to discover extension point usages of a specific framework (given as input). After the computation, the tool displays a list-view and a graph-view of the extension point usage graphs built by the importer and that constitute the input to the mining algorithm. Finally, the tool supports inspecting the frequent extension point patterns uncovered by the mining algorithm, and comparing them to their matches among the input data, to corroborate the validity of the mining process.

IV. PRELIMINARY EVALUATION

This section presents the design and the results of the preliminary study that we conducted to analyse extension point patterns in Scala projects.

A. Design

We follow the process defined by Wohlin *et al.* [11] to conduct this study. The *goal* of our study is to analyse (i) to what extent extension points occur and (ii) to which category of those defined by Asaduzzaman *et al.* [9] they belong. The *purpose* of this study is to collect extension points of the SCALA-SWING library in order to recommend frequently used patterns to developers. Based on this study, we aim to answer the following research questions:

RQ$_1$: *To what extent do extension points occur in* SCALA-SWING *projects?*

RQ$_2$: *What kind of extension patterns occur and which are the most frequently used for the* SCALA-SWING *library?*

TABLE I: Dataset Characteristics

Project	# Classes	# LOC
https://github.com/MarcinCz/MilkaRecognizer	23	751
https://github.com/enshahar/ScalarTurtle	13	549
https://github.com/Sciss/ScalaInterpreterPane	24	1473
https://github.com/myrjola/scaltris	10	455
https://github.com/gabysbrain/scala-swing-jogl-demo	3	266
https://github.com/amsterdam-scala/AS-Tiles-puzzle-solver	6	614
https://github.com/Sciss/ScalaColliderSwing	30	3980
https://github.com/junxiaosong/AlphaZeroGomoku	19	1143
https://github.com/scala/scala-swing/tree/2.0.x/examples/src/main	17	279

To this end, we collected a dataset of 9 open-source SCALA-SWING projects hosted on Github. The characteristics of each project are reported in Table I. Even if the corpus is relatively small, a manual inspection of the extension points is too time-consuming and error-prone. For that reason, we used the framework described in Section III. We manually determined the precision of SCALA-XP-MINER. We did this by manually inspecting every extension point graph. We found that 16 out of 440 nodes where generated incorrectly. We believe there are multiple reasons for this, but the most common one is due the fact that we cannot resolve the type of expressions that use type parameters (*e.g.*, we are not able to resolve `Int` as

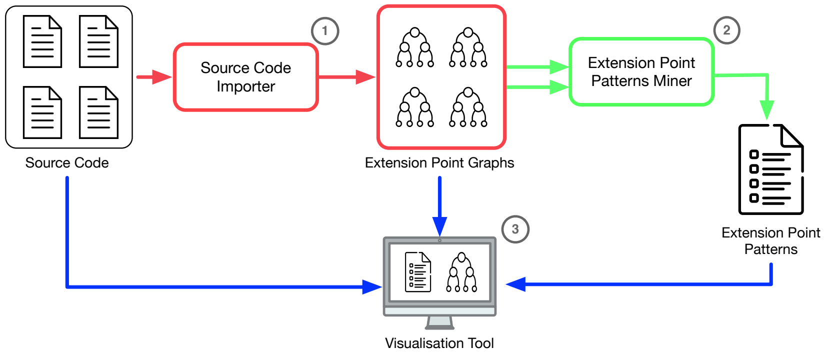

Fig. 4: Overview of the approach

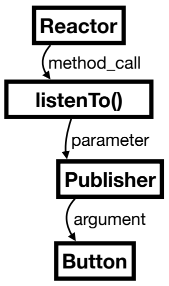

Fig. 5: Example of an EXTENSION POINT USAGE GRAPH for the code in Figure 1(b)

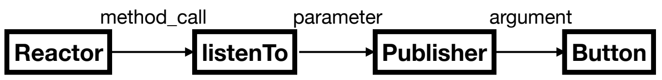

Fig. 6: EXTENSION POINT PATTERN extracted from Figure 5

the result type of the method `List[+A]#head: A` where `List[+A]` was instantiated to `List[Int]`). The precision of our tool is therefore 96%, which we deem sufficiently precise to use this tool as a foundation for our study.

B. Results

We were able to collect 92 extension point usages across 9 projects using SCALA-XP-MINER. These extension point usages are represented by extension point usage graphs as defined by [9]. On average, there are 10 extension point usages per project. We found that the method `Reactor#listenTo` and the method `BorderPanel#add` are the most frequently used extension points, in 40% and 17% of the cases respectively.

To answer **RQ**$_2$, we applied the mining algorithm on the set of 92 extension point usages and obtained 34 patterns which range in size from 2 to 6. Next, we categorized each extension pattern according to the taxonomy as defined by [9]. We found that all of the mined extension patterns are instances of the category SIMPLE.

The top-5 most-occurring extension patterns are shown in Table II. For example, the third pattern (Figure 6) indicates that

TABLE II: Top 5 most frequent Extension Patterns

#	FrameworkClass.Method	Parameters	Arguments	Frequency
1	BorderPanel.add()	Constraints	Enumeration	16/92 (17.30%)
		Component	-	
2	Reactor.listenTo()	Publisher	Publisher	8/92 (8.60%)
3	Reactor.listenTo()	Publisher	Button	7/92 (7.60%)
4	GridBagPanel.add()	Constraints	Constraints	6/92 (6.50%)
		Component	-	
5	GridBagPanel.add()	Constraints	Constraints	5/92 (5.40%)
		Component	Label	

it is common to extend the framework at the `Reactor` class using its `listenTo` method, passing an instance of `Button` as argument. We use − to indicate that there is no explicit information about the argument type (*i.e.,* patterns #1 and #4).

V. CURRENT LIMITATIONS AND CHALLENGES

In this paper, we have shown how data mining can be used to discover extension point patterns in projects that use the SCALA-SWING framework. These patterns provide useful information to developers that want to extend the behaviour of a framework. We used the approach and the concepts of extension points defined by Asaduzzaman *et al.* [9] as foundation for this work. We applied this approach to the Scala programming language to find extension patterns of the SCALA-SWING framework.

Our initial results show that, for the SCALA-SWING framework, the use of extension points to modify its behaviour is not very widespread. On average, there are only 10 extension point usages per project and both `scala.swing.Reactor#listenTo` and `scala.swing.BorderPanel#add` methods are the most frequently occurring ones. Moreover, we found that all extension patterns are of the category SIMPLE.

Our future work includes (i) a better evaluation technique to assess precision and recall of our approach, (ii) a large empirical study on a corpus of Scala projects, (iii) improved type resolution for complex expressions in the importer, (iv) the definition of Scala-specific extension point categories, and (v) the pruning of uninteresting and redundant patterns from the results of the miner.

References

[1] M. P. Robillard, “What makes apis hard to learn? answers from developers,” *IEEE software*, vol. 26, no. 6, pp. 27–34, 2009.

[2] M. P. Robillard, E. Bodden, D. Kawrykow, M. Mezini, and T. Ratchford, “Automated api property inference techniques,” *IEEE Transactions on Software Engineering*, vol. 39, no. 5, pp. 613–637, 2013.

[3] C. De Roover, R. Lämmel, and E. Pek, “Multi-dimensional exploration of api usage,” in *Proceedings of the 21st IEEE International Conference on Program Comprehension (ICPC13)*, 2013.

[4] D. Mandelin, L. Xu, R. Bodík, and D. Kimelman, “Jungloid mining: Helping to navigate the api jungle,” in *Proceedings of the ACM SIGPLAN 2005 Conference on Programming Language Design and Implementation (PLDI 2005)*, 2005.

[5] S. Thummalapenta and T. Xie, “Parseweb: a programmer assistant for reusing open source code on the web,” in *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. ACM, 2007, pp. 204–213.

[6] R. Alur, P. Černỳ, P. Madhusudan, and W. Nam, “Synthesis of interface specifications for java classes,” *ACM SIGPLAN Notices*, vol. 40, no. 1, pp. 98–109, 2005.

[7] H. Zhong, T. Xie, L. Zhang, J. Pei, and H. Mei, “Mapo: Mining and recommending api usage patterns,” in *Proceedings of the 23rd European Conference on Object-Oriented Programming (ECOOP09)*, 2009, pp. 318–343.

[8] 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 *Proceedings of the the 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*. ACM, 2009, pp. 383–392.

[9] M. Asaduzzaman, C. K. Roy, K. A. Schneider, and D. Hou, “Recommending framework extension examples,” in *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*. IEEE, 2017, pp. 456–466.

[10] C. C. Aggarwal, *Data mining: the textbook*. Springer, 2015.

[11] C. Wohlin, M. Höst, and K. Henningsson, “Empirical research methods in software engineering,” in *Empirical methods and studies in software engineering*. Springer, 2003, pp. 7–23.