Mining for Framework Instantiation Pattern Interplays

Yunior Pacheco
Vrije Universiteit Brussel
Brussels, Belgium
yunior.pacheco.correa@vub.be

Ahmed Zerouali
Vrije Universiteit Brussel
Brussels, Belgium
ahmed.zerouali@vub.be

Coen De Roover
Vrije Universiteit Brussel
Brussels, Belgium
coen.de.roover@vub.be

Abstract—**Software frameworks define generic application blueprints which can be instantiated into an application through application-specific *instantiation actions* such as overriding a method or providing an object that implements an interface. In case the framework's documentation falls short, developers may use other instantiations of the same framework as a guide to the required instantiation actions. In this paper, we propose an automated approach to mining *framework instantiation patterns* from existing open-source instantiations. The approach leverages a graph-based representation to capture the common ways of implementing instantiation actions as well as their interplays, so called *instantiation interplays*. As a case study, we mined for patterns in a set of 2,028 Java projects that instantiate four of the most popular Java frameworks. We also classify the extracted interplays according to the different contexts in which they occur. We found that our approach discovers relevant practices and interplays that are not covered by previous approaches. Our results will allow developers to have a better understanding of the frameworks they instantiate.**

Index Terms—**Software Framework, Instantiation Patterns, Graph Mining, Mining Software Repositories**

I. Introduction

Frameworks intend to reduce the cost of application development by providing reusable concepts at the design and code level. Well-designed frameworks adhere to the open-closed principle, i.e., their implementation is closed for modification but open for extension through an API [1]. The process through which developers extend and adapt a framework to the specific requirements of their application is called *framework instantiation*. The corresponding *instantiation actions* taken typically consist of registering callbacks, calling framework methods with a parameter that implements an interface, overriding methods, calling the framework's super methods and declaring application-specific classes that subtype framework types [2], [3]. We call these classes client classes. Instantiating a framework might therefore require a deep understanding of these actions, and their *interplays* (*i.e.,* relationships between framework types in the client code). For instance, which client classes usually collaborate together.

An example of interplays can be found in the documentation of the *ProvidesResize* interface of the Java framework GWT[1]. It reads *"With limited exceptions (such as RootLayoutPanel), widgets that implement this interface will also implement RequiresResize"*. In this case, we can observe an explicit relationship between two framework types: *ProvidesResize* and *RequiresResize*.

[1] http://www.gwtproject.org/javadoc/latest/com/google/gwt/user/client/ui/ProvidesResize.html

Developers who are aware of these internal designs can correctly choose among alternative ways to implement the instantiation actions, to structure their code accordingly, and to use the framework API as efficiently as possible [4]. As a result, the development effort could be reduced. The larger and more sophisticated the framework, the more difficult this challenge will be, due to specific requirements and relationships between its components that are sometimes difficult to unravel.

To help framework users, several approaches [1], [5]–[8] have been proposed that augment a framework's documentation by mining information from example instantiations. However, these approaches do not provide explicit information about the interplays and dependencies that are required between instantiation actions. Following the example above, existing approaches would provide information on how to implement *ProvidesResize* in isolation and independently from *RequiresResize*.

In this paper, we present a novel approach to discovering common framework instantiation actions and their interplays. These common practices are found by mining for frequent subgraph patterns in a graph-based representation of the instantiation actions found in applications that instantiate the framework. Their interplays are found by scanning the resulting graph patterns for connections between nodes that represent client classes.

More specifically, we use a prototype implementation of the approach to empirically investigate the prevalence of instantiation interplays in the process of instantiating frameworks. Our aim is to shed light on how much information is missed by existing mining approaches. To this end, we apply the prototype to 2,028 real-world client applications instantiating four popular Java frameworks (JavaFx, GWT, Spring and Play), and investigate how many interplays developers have to manage in real-world projects as well as of what type they are according to a preliminary classification strategy.

II. Motivating Example

To motivate the need for mining approaches that uncover the interplays between frequent framework instantiation actions, we use the actions required to customize the appearance of the *TableView* element of the JavaFx framework[2]. Its documentation reads as follows: *"The visuals of the TableView can be entirely customized by replacing the default row factory [...] In many cases, this is not what is desired however, as it is more commonly the case that cells be customized on a per-column basis, not a per-row basis [...] You can create custom TableCell instances per column by assigning the appropriate function to the TableColumn cell factory property."*

Listing 1: Example of customizing a JavaFx TableView

```
1  TableView<Item>table=new
       TableView<>(FXCollections.observableArrayList
       (new Item(Feeling.Happy), new
       Item(Feeling.Happy))
2  ....
3  TableColumn<Item, Feeling> feelingColumn =
       new TableColumn<>("Feeling");
4  feelingColumn.setCellFactory(new
       Callback<TableColumn<Item, Feeling>,
       TableCell<Item, Feeling>> {
5        @Override
6        public TableCell<Item, Feeling> call
             (TableColumn<Item, Feeling> param) {
             return new EmojiCell<>();
7     }});
8  ....
9  table.getColumns().add(feelingColumn);
10 public class EmojiCell<T> extends
       TableCell<T, Feeling> {
11    private final ImageView image;
12    public EmojiCell() {...}
13    @Override
14    protected void updateItem(Feeling item,
          boolean empty) {
15      super.updateItem(item, empty);
16      if(empty || item ==null) {
17        setText(null);
18        image.setImage(null);
19      } else {
20        image.setImage(item.getEmoji());
21        setText(item.getValue()); }}}
```

Listing 1 depicts an excerpt of an example implementation of the instantiation process described above. To customize the rendering of the cells displayed in a *TableView* object, new instances of the framework class *TableView* and *TableColumn* are created on line 1 and 3 respectively. Line 4 calls the *setCellFactory()* on the latter instance and passes as an argument a *Callback* implementation in the form of an anonymous class declaration. Next, line 6 overrides the method *call()* to create a *TableCell* instance and return it. The returned instance is of the application-specific subtype *EmojiCell* defined on line 10, which overrides method *updateItem()* on line 14.

As illustrated by the above code extract, the instantiation actions that need to be realized in client code can be intricately related to each other. Moreover, their importance is re-affirmed by the framework's documentation, which describes them more or less explicitly. The most interesting point is the close interplay between the actions to instantiate the framework types *TableView* and *TableCell*. We need to extend the behavior of the *TableCell* through the *setCellFactory()* method in *TableColumn* in order to customize a *TableView* object.

The work of Asaduzzaman *et al.* FEMIR [3] is the most relevant and related research to our work. The authors define the concept of *Extension Points* as: *"means provided by a framework, which allow developers to customize its behavior, to meet specific application requirements"*. In the context of their work an extension point is a framework type that constitutes the formal parameter of a call into the framework. A common way of using such an extension point (*i.e.,* an extension point usage) is by calling the framework method with an instance of the framework type itself, or with an instance of a user-defined subtype that overrides some of the inherited methods.

In Listing 1, *TableColumn* in the overridden method *call()* (line 6) as well as *Callback* in the method *setCellFactory()* (line 4) are extension points because these methods take at least one parameter of a framework-related element, *i.e.,* a framework class instance (*i.e., param*) and an anonymous class declaration (*i.e., new Callback* {...}), respectively. Although this approach is capable of mining code examples to discover extension point usages and extension patterns for each framework class and therefore get an idea of how a certain framework element should be instantiated, interplays between different instantiation actions are not captured. For this example, FEMIR can provide recommendations on how to use *TableColumn* or *Callback* but is not capable of capturing the reference to a different instantiation action (subclassing *TableCell*), meaning that the relationship between *TableView* and *TableCell* is missed.

III. Related work

Next to the most closely-related work by Asaduzzaman *et al.* [3] discussed above, more framework instantiation action mining approaches have been proposed earlier. Heydarnoori *et al.* [9] automatically extract concept-implementation templates from dynamic traces of example applications, while Lafeta *et al.* [10] present documentation in a cookbook style, where recipes consist of programming tasks and information about hotspots related to a feature instantiation. Both approaches are supported by dynamic analysis, which considerably limits their applicability in large-scale studies into framework Instantiation.

Michail [11], [12] presented CodeWeb, a tool that addresses the problem of discovering library classes and member functions, using generalized association rules. Bruch *et al.* [6] proposed FrUiT, an Eclipse plug-in relying on data mining techniques that extracts reuse patterns from existing framework instantiations to create framework usage scenarios based on five class properties (extends, implements, overrides, calls, and instantiations). Thummalapenta and Xie [1] presented SpotWeb, which assists developers in understanding and reusing a given framework by detecting hotspots (*i.e.,*

[2]https://docs.oracle.com/javase/8/javafx/api/javafx/scene/control/TableView.html

frequently used APIs) and coldspots (*i.e.,* barely used APIs). Bruch *et al.* [5] proposed CORE, an approach that provides pieces of documentation related to how to subclass a framework class or how to override a framework method, by mining four kinds of documentation items. It bears repeating that none of these approaches fully capture information about the interplays between and the relations among instantiation actions.

Instead of mining examples for instantiation actions, an alternative group of approaches intends to support framework users by requiring framework developers to document the instantiation process in a machine-readable format.

Ortigosa et al. [13] propose an intelligent agent to guide developers through the instantiation process based on rule-based information specified by the framework developer. The agent cannot provide any help on undocumented features. Dagenais and Ossher proposed XFINDER [14] which locates, semi-automatically, implementation examples from a code base given the lightweight documentation of a framework. In other words, the framework documentation is used as a template and XFINDER finds instances of the template within the code base.

Several approaches [15]–[21], leverage the so-called Reuse Description Language, which allows specifying the instantiation process as a series of programming activities such as class extension, method redefinition, and design pattern application. Similarly to Ortigosa's work and XFINDER, the framework design must be annotated using UML-FI stereotypes and tagged values, and a Feature Model should be created to represent the framework features.

Another smaller number of approaches focus on assisting the framework instantiation process using the API documentation. The work of Montperrus [22] describes a comprehensive taxonomy of 23 types of directives present in API documentation, while Yu Zhou *et al.* developed DRONE [23], [24], a framework for automatically detecting and repairing defects in API documentation.

IV. FRAMEWORK INSTANTIATION INTERPLAY MINING

Our approach to mining instantiation patterns consists of two components: 1) a source code importer and 2) a pattern miner. Figure 1 depicts the interactions between these two components.

Listing 2: Instantiation action in source code, client class declaration.

```
public class MessageBoxController implements
    Initializable {
public AnchorPane root;
@Override
public void initialize(URL arg0,
    ResourceBundle arg1) {}
...
private MessageBox getMessageBox() {
return (MessageBox)
    root.getScene().getWindow();
}
private void hide() {
getMessageBox().hide();
}}
public class MessageBox extends Stage { ... }
```

A. Source Code Importer

Client code from open source projects provides valuable scenarios on how to instantiate application frameworks. To extract instantiation information from client code, we have developed a source code importer. This component accepts a framework name as the input and collects, from a repository, open source software projects that contain framework instantiation examples. We search the repository using the import statements in Java source files.

Following that, we perform static analysis on the source code of selected projects. As potential evidence for these realizations, our prototype implementation considers all occurrences of a subtyping relation between a client type and a framework type in the source code (*i.e.,* a named or anonymous class declaration in the client code that extends a super class provided by the framework or that implements an interface provided by the framework). Throughout the remainder of the paper we consider an **Instantiation Action** as the result of a client class declaration that subtypes a framework type. Listing 2 shows an example of an instantiation action in the project `7thwheel/medaka-chan`[3]; a client class declaration (*i.e., MessageBoxController*) implementing a JavaFx framework interface (*i.e., Initializable*).

For each client declaration in an instantiation action, the importer identifies the extended class, the implemented interfaces, the newly declared fields, and the newly declared and overriding method declarations. In addition, their body is scanned for all method calls of which the statically-declared type of the receiver expression is a framework type or a client type that is subtype thereof. The same goes for instance creation expressions. The Eclipse JDT parser is used for parsing and type binding resolution.

All this extracted information is used to construct a graph-based representation called Instantiation Graph

Instantiation Graph. We construct an instantiation graph for each instantiation action to describe how it has been created and how it could be related to other framework elements and instantiation actions. Notice that since multiple instantiation actions can occur within a single project, we can obtain multiple instantiation graphs in each project. Figure 2 depicts the instantiation graph generated for the instantiation action shown in Listing 2. A framework instantiation graph consists of the following types of nodes:

- *Outer call*: a method call where it receives an anonymous class declaration as an argument.
- *Framework type*: a class or interface type provided by the framework.
- *Client type*: a client class declaration, anonymous or not, that extends or implements a framework type. At the core of each instantiation graph there is a *Client type* node, that we call

[3]https://github.com/7thwheel/medaka-chan

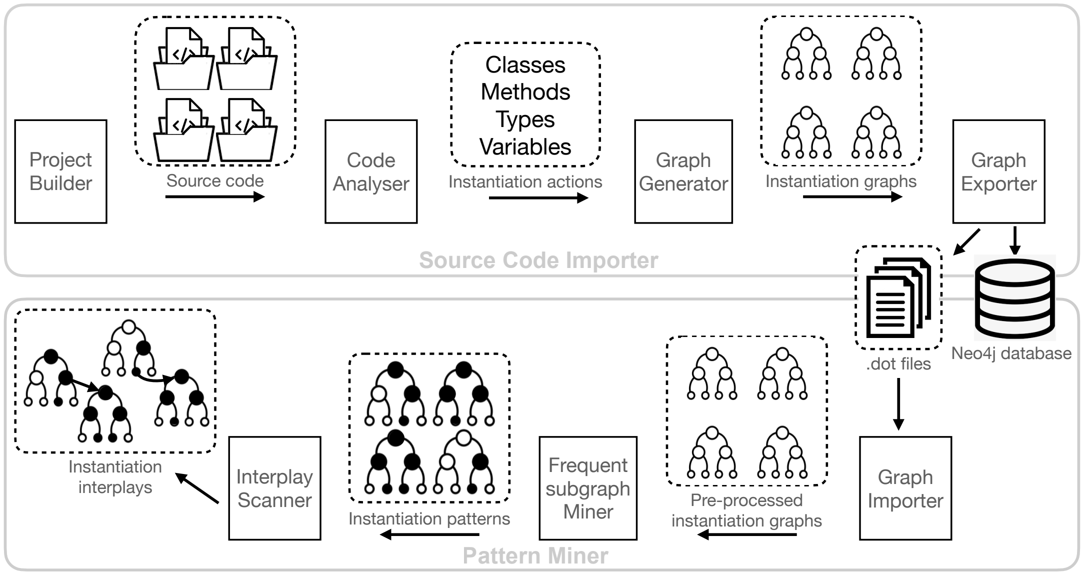


Fig. 1: Overview of the approach

the main *Client type* to differentiate it from other *Client type* nodes in the graph that represent external instantiation actions.
- *Field declaration*: a field of the class whose type is related to the framework.(*i.e.,* either a *Framework type* or a *Client type*).
- *Parameter declaration*: a parameter of a method declaration whose type is related to the framework.
- *Variable expression*: a variable used in the body of a method declaration, whose type is related to the framework. The variable can reference a field of the class, a parameter of the method declaration or can be the result of a constructor or a method call.
- *Overriding method declaration*: a method declared in a client type that overrides a framework method.
- *Client method declaration*: a method declared in a client type. To be included in the graph, the client method declaration must have in its body or signature a reference to an external instantiation action.
- *Framework method call*: a method call whose receiver type is related to the framework.
- *Constructor call*: a constructor that instantiates a type related to the framework.
- *Super method call*: a super method called by an overriding method.

In the instantiation graphs, the edges represent dependencies between the nodes (*i.e.,* subtyping declarations or actions). In general the edge labels are self-explanatory, for example, edges with label *Implement* or *Extend* represent inheritance relationships between a *Client type* node and a *Framework type* node. In the same manner the *Call* label represents actions between a *Variable expression* node and *Framework method call* or *Constructor call* nodes. More details on the generated instantiation graphs can be found in the replication package.[4]

Instantiation Interplays. Figure 2 depicts two *Client type* nodes with labels `seventhwheel.pos.controller.MessageBoxCon-troller` and `seventhwheel.pos.control.Messag-eBox`. The first one is adjacent to the *Framework type* node with label `javafx.fxml.Initializable`. This means that this client class extends the framework interface `Initializable`. The other *Client type* node is adjacent to the *Client method declaration* node with label `seventhwheel.pos.c- ontrol.MessageBox getMessageBox()` and to the *Framework type* node with label: `javafx.stage.Stage`. This means that the method `getMessageBox()` declared in this client class returns a client type that extends the framework class `Stage`. Note that there is a reference to an instantiation action (`MessageBox extends Stage`) inside the body of another instantiation action (`MessageBoxCo- ntroller implements Initializable`). This is an example of what we refer to as instantiation interplay.

We define an instantiation interplay as the relation between an instantiation action on a framework type, which we call main, and another instantiation action on a framework type, which we call related. [5]

In the above example, a relationship is established between the instantiation action of the framework interface

[4]https://github.com/ypacheco/Mining-for-Framework-Instantiation-Pattern-Interplays/

[5]The relation occurs if within the implementation of a main instantiation action we find a reference of another (*i.e.,* related) instantiation action. As we can find various instances of related instantiation actions within the body of a main instantiation action (*i.e.,* different client class declarations that subtype different frameworks types), it is possible to find an interplay involving several frameworks types.

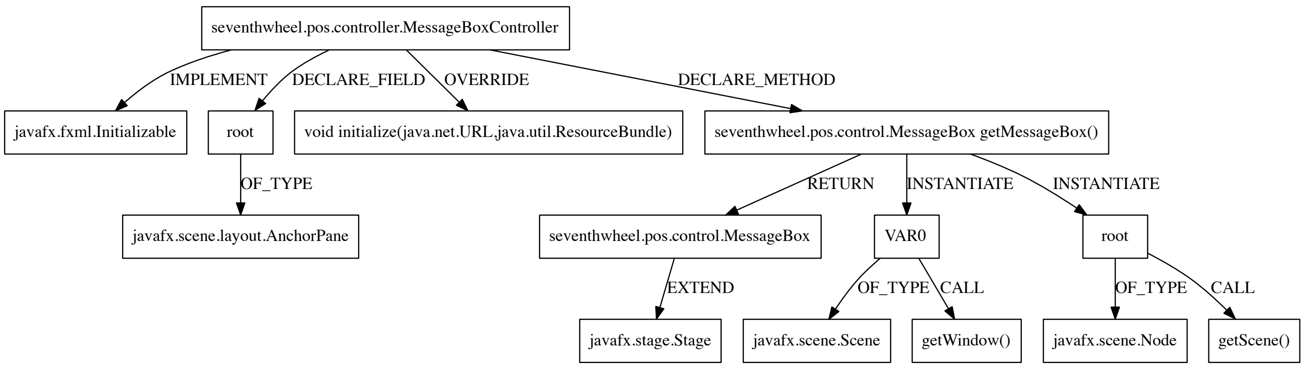

seventhwheel.pos.controller.MessageBoxController
IMPLEMENT
DECLARE_FIELD
OVERRIDE
DECLARE_METHOD
javafx.fxml.Initializable
root
void initialize(java.net.URL,java.util.ResourceBundle)
seventhwheel.pos.control.MessageBox getMessageBox()
OF_TYPE
javafx.scene.layout.AnchorPane
RETURN
INSTANTIATE
INSTANTIATE
seventhwheel.pos.control.MessageBox
VAR0
root
EXTEND
OF_TYPE
CALL
OF_TYPE
CALL
javafx.stage.Stage
javafx.scene.Scene
getWindow()
javafx.scene.Node
getScene()

Fig. 2: Instantiation graph generated from the instantiation action in Listing 2.

`Initializable`, which is considered the main framework type, and the instantiation action of the framework class `Stage`, which is considered the related framework type. In some cases the interplay occurs between two instantiation actions that extend the same framework type. We consider this case an instantiation interplay because although the two instantiation actions extend the same framework type, they are different actions.

Types of Instantiation Interplays. As previously mentioned, the instantiation actions can be the result of a named or an anonymous class declaration that subtype a framework type. Considering that an instantiation interplay is composed of two instantiation actions, main and related, there are four possible scenarios in which an interplay is classified according to how the instantiation actions involved are implemented: *Client-Client*, *Client-Anonymous*, *Anonymous-Client*, *Anonymous-Anonymous.*

In the example of Figure 2 both instantiation actions correspond to named client classes (`MessageBoxController` and `MessageBox`) thus the interplay is classified as *Client-Client*. These types of interplays can help framework designers in verifying how client applications implement elements intended to be reused anonymously, *e.g.,* through the use of events or callbacks.

In order to describe the instantiation interplays not only by the scenarios in which the main and related instantiation actions may be realized, we categorize them according to the context (in the body of the main instantiation action) in which references to the related instantiation action are found:
- *Field declaration*: a field declared in the main client class whose type is another client class that subtypes a framework type.
- *Parameter declaration*: a parameter declared in the signature of a method in the main client class, whose type is another client class that subtypes a framework type.
- *Return type*: the declared return type of a method in the main client class is another client class that subtypes a framework type. The above example occurs in this context.
- *Instantiation*: a client class instantiated in the body of a method in the main client class, while subtyping a framework type.
- *Method call*: a framework method call in the body of a method in the main client class, for which the statically declared type of the receiver is another client class that subtypes a framework type.
- *Client method declaration*: the interplay occurs in the body or the signature of a client method declaration.
- *Overriding method declaration*: the interplay occurs in the body or the signature of an overriding method declaration.

An instantiation interplays may be present in one or more of these contexts. The related client class, the one that extends the related framework type, may appear in several contexts *e.g.,* it may be the declared type of a field of the main client class and the type of an instantiation expression within an overriding method. In this case, we categorize the interplay as the combination of all these contexts: *Field declaration* + *Instantiation* + *Overriding method declaration.*

By categorizing interplays in this manner, our approach does not only produce information about which framework instantiation actions are often related in client projects, but also the form this relation tends to take in the body of the main client class.

B. Pattern Miner

In the second step, the Pattern Miner imports and pre-processes the instantiation graphs previously generated by the importer. The pre-processing is required as the mining algorithm is oblivious to node types and their equality relation. First, the labels of all nodes derived from client code are replaced by a generic label. This operation may introduce ambiguity; thus, for nodes representing variables, the declared type of the variable is added to the node label. The type of each node is also appended to their label. Figure 3 depicts the result of pre-processing the instantiation graph of Figure 2.

After the instantiation graphs have been processed, they can be given as input to the mining algorithm. In addition to the pre-processed instantiation graphs, the algorithm takes as input a minimum support value *minsup* and obtains the so-called instantiation patterns. These patterns are frequent subgraphs present in the set of all instantiation graphs. A subgraph occurs

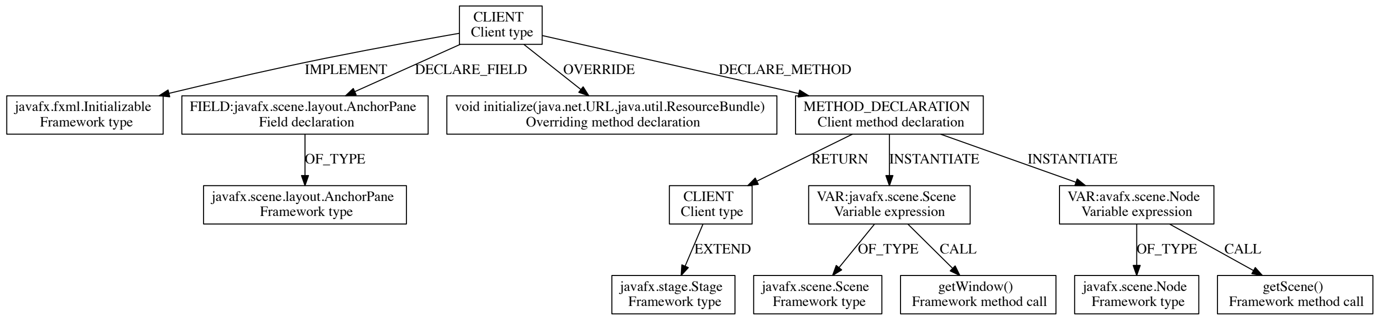


Fig. 3: Pre-processed version of the instantiation graph from Figure 2.

frequently if its support is greater than or equal to the initially defined *minsup*. The support of a subgraph is its number of occurrences in the input set of graphs. It is possible that a subgraph occurs more than once within a graph, in this case, for the calculation of support, we count only one occurrence. We consider this approach since we are interested in finding patterns across multiple instantiation actions.

Our implementation is based on a variant of the frequent subgraph mining algorithm from MuDetect [25], which has been used effectively in other work [26]. This algorithm implements the basic idea of the Apriori algorithm for mining frequent substructures in graphs [27]. It starts by generating all frequent singleton subgraphs and iteratively and incrementally expands them to larger subgraphs. Our variant incorporates a heuristic to determine the nodes that will be considered as singleton graph seeds from which all frequent subgraphs will be grown.

The heuristic only considers nodes of type *Client type* as seed nodes, as those are the only ones involved in potential realizations of the subtyping instantiation action. Moreover, during the first iteration of the algorithm, the heuristic does not consider any possible expansion of the seed nodes that we know to typically have a very high support. By excluding expansions that are very frequent but may have adjacent nodes that are not (*i.e., Variable expression*, *Client method declaration*, *Framework method call*), the heuristic prevents the algorithm from generating a large number of candidate subgraphs which will be discarded later. Note that the initially excluded nodes remain in the search space. They will still be considered for expansions of non-singleton subgraphs later.

Interplays in Instantiation Patterns. In the final step, the instantiation patterns resulting from the frequent subgraph mining algorithm are scanned for possible interplays. The resulting patterns describe common ways of implementing instantiation actions and their interplays, hence, they can help novice developers to implement similar instantiation actions. Figure 4a shows one of the patterns obtained by our approach for the GWT framework. In this example the pattern contains an interplay between the actions that subtype the framework types *EntryPoint* and *ClickHandler*. This interplay is of type *Client-Anonymous* and *Instantiation+Overriding method declaration* according to the two classification strategies described in Section IV-A. Figure 4b shows a source code example of one of the occurrences of the pattern in Figure 4a. These source code examples and the interplays represented in the mined patterns could be used to recommend to a novice developer facing an instantiation process, instantiation actions related to the ones that are being implemented.

V. Empirical Study into Instantiation Interplays

We now apply the prototype implementation of our approach to real-world framework clients. Our goal is to understand the nature of the interplays between framework instantiation actions. More specifically, our study answers the following research questions:

- RQ_1 **How prevalent are instantiation action interplays?**
- RQ_2 **What are the most common instantiation action interplays?**

To answer these questions, we mined for instantiation patterns and their interplays in a corpus of 2,028 projects instantiating four of the most popular Java frameworks: JavaFx (639), GWT (185), Spring (963) and Play (241). As a source for these projects, we used the Sourcerer JBF dataset [28] which contains 50,000 compilable Java projects. Each project in this dataset comes with references to all the dependencies required to compile it, the resulting bytecode, and the scripts with which the projects were built.

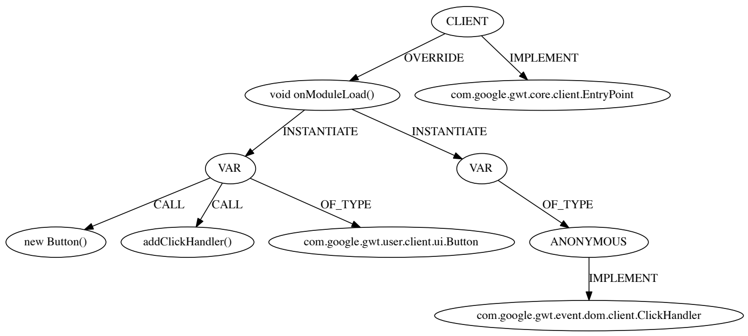
All code and data required to reproduce the analysis in this paper is available in a replication package [7].

RQ_1 *How prevalent are instantiation action interplays?*

To analyze the prevalence of instantiation interplays, we first built two datasets from the Sourcerer JBF dataset. The first dataset consists of all instantiation graphs produced by our Source Code Importer for each of the 2,028 client projects. These graphs provide insights into how developers instantiate the same framework across multiple projects. The second dataset consists of all instantiation patterns found by our Pattern Miner in the graphs. The found patterns reveal the common ways of implementing instantiation actions and their

[6]https://github.com/xinl/feedlosophor

[7]https://github.com/ypacheco/Mining-for-Framework-Instantiation-Pattern-Interplays



(a) Instantiation pattern

```
public class Feedlosophor implements EntryPoint {
  ...
  public void onModuleLoad()
  {
    ...
    // Create the popup dialog box
    final DialogBox dialogBox = new DialogBox();
    dialogBox.setText("Remote Procedure Call");
    dialogBox.setAnimationEnabled(true);
    final Button closeButton = new Button("Close");
    ...
    // Add a handler to close the DialogBox
    closeButton.addClickHandler(new ClickHandler() {
      public void onClick(ClickEvent event) {
        dialogBox.hide();
        sendButton.setEnabled(true);
        sendButton.setFocus(true);
      } });
    ... }}
```

(b) Source code example

Fig. 4: Example of a mined pattern and one of its occurrences from the project `xinl/feedlosophor` [6].

interplays. To find as many relevant patterns as possible, we configured the miner with low support (i.e., *minsup=2*). In what follows, we analyze the first and the second dataset to investigate the overall prevalence of interplays in all framework clients and their most common realizations respectively.

Table I summarizes the number of interplays present in the two datasets for each framework. We found the highest proportion of graphs and patterns containing interplays in the frameworks GWT and JavaFx, *e.g.,* 22% and 24.8% for GWT, respectively. The lowest proportions were found in Play and Spring. It is important to clarify that for instantiation patterns this proportion is not calculated based on the total number of patterns found, but on the number of patterns that contain at least one node of type *Framework type*. These plain statistics give us a general idea of the presence of interplays in the analyzed frameworks. However, to analyze in depth the prevalence of interplays in these datasets, we must also consider other aspects.

Number of interplays per graph. The first aspect to consider is the distribution of the number of interplays present in the instantiation graphs and patterns that contain occurrences of them. Figure 5 shows that for nearly all frameworks, most of the instantiation graphs and patterns have a low number of interplays, which means that the interplays are not exclusive for a specific group of instantiation graphs and patterns. The exception is Play where we only found two graphs and one pattern with interplays. It is important to mention that instantiation graphs and patterns with no interplays are not included.

In Table I, we also observe that for frameworks in which the proportion of graphs and patterns containing interplays is significant, such as GWT and JavaFx, the patterns obtained usually included more interplays than the number that is generally present in the instantiation graphs. For example, in JavaFx, the median of the distribution of interplays in instantiation patterns is 2, which is higher than what we found for interplays in instantiation graphs *i.e.,* 1.

Number of interplays per framework type. Another aspect we consider is the framework types present in the instantiation interplays. As part of the study, we collected, for the two sets of instantiation graphs and instantiation patterns, all the framework types that are extended. We order these types by the number of graphs and patterns in which they are present. For example, we found that the top framework types for JavaFx, Play, GWT and Spring are present in 42%, 29.3%, 24.9% and 4.4% of the instantiation graphs, respectively.

Focusing only on GWT and JavaFx, we found that 79.6% and 92.6% of the interplays from these two frameworks, respectively, are between just two frameworks types, in the sense that one framework type is being used in the context of another framework type. Note that it is possible to find an interplay that involves multiple framework types (*i.e.,* more than 2). In addition, an interplay between specific framework types can appear multiple times accross different projects and within the same project. For JavaFx, we found that framework types with the highest number of unique interplays (*i.e.,* each interplay is counted only once even if it occurs multiple times) are involved in the most popular interplays that are found in nearly half of the projects. On the other hand, we found that the most popular interplay in GWT has a related type that has a low number of unique interplays, *i.e., com.google.gwt.event.dom.client.ClickHandler*. In other words, one framework type can have a low number of relationships with other framework types, but it may be needed in the most popular interplay across projects. Overall, we found that 230 of GWT framework types and 107 of JavaFx's were present in at least one interplay. This shows that the types within these two frameworks are related to each other.

Number of interplays per project. Finally, we study how the instantiation interplays behave within the client application where they are implemented. In this way we will not only have a general overview of the extent of the instantiation interplays in the analyzed frameworks, but also of the context in which

TABLE I: Summary of the datasets used in this study for each framework. Nodes here refer to patterns containing *Framework type* nodes.

		Instantiation Graphs			Instantiation Patterns			
Framework	projects	total	interplays	%	total	nodes	interplays	%
GWT	185	4720	1040	22	1433	817	203	24.8
JavaFx	639	4938	965	19.5	2740	978	332	33.9
Play	241	58	2	5.2	47	26	1	3.8
Spring	963	904	89	9.8	373	243	16	6.6

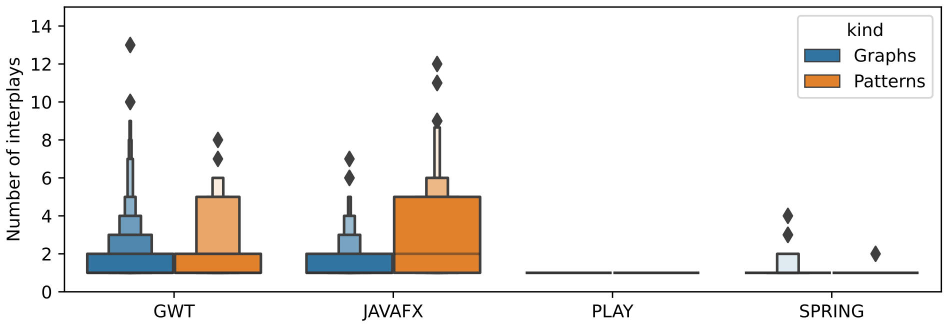


Fig. 5: Distribution of the number of interplays present in the instantiation graphs and patterns for each framework.

TABLE II: Number of graphs and interplays per projects.

Framework	total	graphs (%)	interplays (%)	% projects with interplays from graphs
GWT	185	168 (90.8)	145 (78.4)	86
JavaFx	639	480 (75.1)	308 (48.2)	64.2
Play	241	25 (10.4)	2 (0.8)	8
Spring	963	356 (37)	52 (5.4)	14.6

they appear in a client application. Table II shows the number of instantiation actions and interplays present in the analyzed projects. The proportion refers to the number of projects with interplays out of all projects with instantiation graphs. For the GWT and JavaFx frameworks a considerable number of projects containing instantiation actions have interplays. Play and Spring are opposite cases. In these two frameworks, very few projects contain instantiation actions and therefore instantiation graphs. For GWT and JavaFx, 86% and 64.2% of the projects that performed an instantiation process included at least one interplay, respectively. Focusing only on these projects, we found that the median number of interplays in them is 5 and 3 for GWT and JavaFx, respectively. These numbers decrease to 4 and 2 when only computing the unique number of interplays present in each project. This means it is possible to find the same interplays several times within the same project.

Instantiation interplays occur consistently in instantiation actions regardless of the different ways of implementing them. Instantiation interplays are prevalent within software projects instantiating GWT and JavaFx frameworks. They are also distributed in different parts within the same project.

RQ_2 What are the most common instantiation action interplays?

To identify the most common instantiation interplays, we must collect the interplays present in the instantiation patterns. To distinguish the different types of instantiation interactions, we use the categorization described in Section IV-A. Once classified, we identify the most frequent interplays by computing the average support of the patterns in which the interplays occur.

Table III shows for each framework, the most frequent class of interplays according to two different aspects. The first one (Class category) categorizes interplays depending on the context in which the interplay occurs, while the second one (Class scenario) is based on the scenarios in which the instantiation actions present in the interplay are realized.

TABLE III: Most frequent interplay classes for each framework.

	Class Category	Class Scenario	
Framework	Context	Main client type	Related client type
gwt	return type of client method declaration	Client	Anonymous
javafx	instantiation in the body of client method	Client	Anonymous
play	field declaration	Client	Client
spring	field declaration	Client	Client

For GWT framework, the most frequent interplays, depending on the context where they are found, are those that appear as a return type of a client method declaration. For JavaFx, the most common ones are instantiations in the body of a client method. If we analyze the client classes involved in the interplay, the most common class for GWT and JavaFx is when an anonymous declaration is instantiated in the body of a client class declaration. The most popular interplays in GWT and JavaFx contain the framework types *(javafx.fxml.Initializable, javafx.event.EventHandler)* and *(javafx.application.Application, javafx.event.EventHandler)* [8], respectively. The only interplay we were able to extract for Play is a field declaration in the body of a client class declaration, similar to the most frequent classes for Spring.

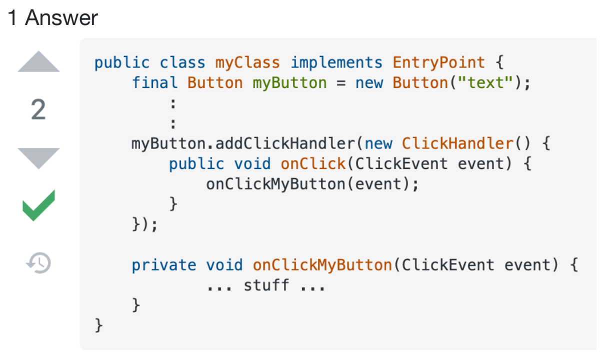
In our study of interplay categories present in the instantiation patterns, we found that interplays present in the context of the body of a client method declaration, such as parameter type declarations (*Parameter declaration-Client method declaration*), instantiations (*Instantiation-Client method declaration*) or return type declarations (*Return type-Client method declaration*), are the most common. On the other hand, the most common scenarios we found, include the creation of an anonymous class to instantiate an object in the body of a client class (*Client-Anonymous*).

The most frequent interplays are present in the body or signature of a client method declaration and usually comprise an anonymous class declaration to subtype the related framework type.

VI. Discussion

Prevalence of instantiation action interplays. In frameworks that are intended to be instantiated through subtyping and that have a relatively high number of instantiation actions, such as GWT and JavaFx, there is a significant prevalence of these interplays. In Spring and Play we find much fewer instantiation actions and therefore much fewer interplays. We believe that this is likely explained by the fact that in the case of Spring the instantiation process is heavily based on the use of annotations. Play, on the other hand, is a framework built on the Model-View-Controller (MVC) architecture and relies, for the manipulation of its Model components, on Object/Relational Mapping (ORM) tools that also make extensive use of annotations. We conclude that interplays are not exclusive of a specific group of instantiation actions nor a specific type of client projects, although they do depend on the reuse mechanisms provided by the frameworks. The presence of interplays between instantiation actions is a common phenomenon in framework instantiation processes.

Common instantiation interplays. Existing approaches focus on program elements such as overriding method declarations and named class declarations, since these are considered to be the fundamental elements in the instantiation process driven by subtyping. On the other hand, these approaches suffer from technical limitations such as the lack of support for anonymous classes or lambda expressions. From the results of our study we can conclude that the most frequent interplays are closely bound to program elements that are not traditionally considered as direct participants in the framework instantiation process. This is the case for instantiation in the body of client method declarations and anonymous class declarations. This indicates that for the JavaFx and GWT frameworks, the mechanisms for client applications to implement the reuse elements they provide are significantly driven by the use of events or callbacks. Dependency injection is another reuse mechanism provided by the frameworks. This mechanism generally involves the use of annotations. The low number of instantiation graphs and interplays obtained in Play and Spring, discussed above, corresponds to the use of this form of inversion of control. In this case, the frameworks Play and Spring rely on dependency injection through the use of annotations to make client applications observe the reuse mechanisms they provide. These results indicate that there are multiple factors such as interplays, in addition to those traditionally analyzed, that play a significant role in the instantiation process.



```
public class myClass implements EntryPoint {
    final Button myButton = new Button("text");
        :
        :
    myButton.addClickHandler(new ClickHandler() {
        public void onClick(ClickEvent event) {
            onClickMyButton(event);
        }
    });

    private void onClickMyButton(ClickEvent event) {
            ... stuff ...
    }
}
```

Fig. 6: Accepted answer to a Stack Overflow question involving an instantiation interplay [9].

Interplays in practice. Our approach discovers relevant practices that can support developers in the framework instantiation process. Using the interplays present in these practices we can suggest instantiation actions that should be implemented in conjunction with those already implemented. Thereby a novice programmer can perform the instantiation of a framework efficiently without having deep knowledge of its covert relations. A possible idea would be to match the developer's ongoing edits to one of the mined patterns and then collect the occurrences of that pattern and translate them into user-friendly code skeletons depicting the related instantiation actions that should be completed. We can find an example of this possible use in a question from *Stack Overflow* [9]. In this question the user wants to know how to implement a *ClickHandler* in the body of an already defined class: `"myClass implements EntryPoint"`. This clearly means that this developer is going to perform an interplay between two instantiation actions: a client class that implements the *EntryPoint* interface and another client class that should implements the *ClickHandler* interface. If we compare the accepted answer to this question shown in Figure 6 with the code snippet in Figure 4b, we observe that they are remarkably similar. The pattern obtained in Figure 4a and the corresponding source code example of its occurrences, provide the solution that the user expected. This is a clear example of the relevance of the instantiation patterns and

[8]In the tuple (Type1, Type2), Type1 is the main framework type whereas Type2 is the related framework type.

[9]https://stackoverflow.com/questions/18323473/

interplays obtained by our approach to assist novice developers in the process of instantiating a framework.

Framework designers can also benefit from these concepts. Using knowledge from the interplays found in client applications, they can analyze whether the API's high-level architecture complies with the design principles for which it was conceived. Incorrect implementations of these design principles in client code do not tend to make systems unstable or malfunction, but they can make software maintenance difficult. As an example, JavaFx designers can verify how client applications observe the "Hollywood Principle" [29] when reusing their framework by analyzing the extracted interplays. Among the most common frameworks types involved in the extracted interplays we find: *javafx.event.EventHandler*, *javafx.beans.value.ChangeListener* and *javafx.util.Callback*. More precisely, one of the most popular ways of obeying the Hollywood principle is to use events or callbacks. A deeper and more comprehensive analysis of these interplays can help framework designers answer questions such as: do framework instantiation interplays affect other properties like instance, modularity, reusability, error-proneness, reliability or performance?

Finally, other researchers can use this work as a baseline for future work. We believe that instantiation interplays constitute an interesting phenomenon that through their study may provide potential insights for frameworks designers and users to enhance maintainability and augment existing documentation. As such, it is a promising area for further research.

VII. Threats to Validity

Internal: We decided to study four of the most popular Java frameworks. In addition to their popularity, which served to find a large number of client applications, we decided to include frameworks designed for different purposes, which is difficult to accomplish since most of the current Java frameworks share similarities with each other, including the selected ones. However, we believe that the selection made provides the study with a variety of different scenarios to analyze. Thus, we have a framework for handling AJAX technology (GWT), for the creation of rich internet applications (JavaFx), for the development of web applications (Play) and finally one for general purposes (Spring).

Construct: The main threat to construct validity arises from the technical limitations of the Importer component of our approach. Our prototype implementation does not account for the latest Java language features such as Sealed Classes. Recent client applications may use these features as part of the framework instantiation process while our approach is not able to capture them. However, we believe that by exploring a large number of projects, we have reduced the impact of possible missed instantiation actions in our results. Furthermore, our approach does not take into account the version of the framework used by the client applications. The possible inconsistencies that could be introduced by analyzing different versions of the same framework are small compared to the total number of instantiation actions obtained. Possibly, our approach could find workarounds that are not intended or recommended by framework developers, *i.e.,* that can become deprecated in future versions and probably should not be used.

External: Our findings are specific to our problem domain. Our approach is designed to analyze instantiation actions in client applications that instantiate Java frameworks. Hence, the Importer component heavily relies on the Eclipse JDT to parse the source code of these applications. Moreover, the proposed graph-based model contains elements specific to the Java programming language. Nevertheless, the idea of representing relationships between source code elements in a graph-based structure and applying a frequent subgraph mining algorithm to find patterns can be generalized to other programming languages and other problem domains where the relationships to uncover are different.

Conclusion: Our conclusions are based mainly on empirical observations of the results obtained from applying our approach. On the other hand, the graph-based model used to represent instantiation actions and the interplays between them, is inspired from another graph-based model [2], widely used for mining software repositories. Similarly, the mining algorithm used in our approach is based on the mining algorithm from MuDetect [25]. Therefore, there is a reduced risk of reaching incorrect conclusions when applying our approach.

VIII. Conclusion

In this paper, we have introduced a graph-based representation for framework instantiation actions and their interplays. This representation enables mining example clients of the same framework for frequent subgraph patterns. The mined patterns can serve to augment a framework's documentation. The results of our empirical study, on 2,028 projects using four popular Java frameworks, show that the extracted interplays describe how application developers adhere to the reuse mechanisms provided through the framework's API. Instantiation interplays are common among instantiation actions, and are not restricted to a particular type of these nor to a particular type of client applications. In many cases, the most common method of realizing instantiation actions involves several of these interplays. Moreover, the most common interplays appear as part of elements that have been ignored by existing approaches on framework instantiation.

The potential benefits of the study of interplays include assisting novice developers in the instantiation process as well as analyzing relevant properties in the design of software frameworks. Due to their importance, framework instantiation interplays should be studied in greater depth. Our approach is the first to uncover them automatically.

Acknowledgments

This research was partially funded by the Excellence of Science project 30446992 SECO-Assist financed by F.R.S.-FNRS and FWO-Vlaanderen.

References

[1] Suresh Thummalapenta and Tao Xie. Spotweb: Detecting framework hotspots and coldspots via mining open source code on the web. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, pages 327–336. IEEE Computer Society, 2008.

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

[3] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Daqing Hou. FEMIR: A tool for recommending framework extension examples. In *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, pages 967–972. IEEE Press, 2017.

[4] Martin P Robillard. What makes APIs hard to learn? Answers from developers. *IEEE software*, 26(6):27–34, 2009.

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

[6] Marcel Bruch, Thorsten Schäfer, and Mira Mezini. Fruit: Ide support for framework understanding. In *Proceedings of the 2006 OOPSLA workshop on eclipse technology eXchange*, pages 55–59. ACM, 2006.

[7] Muhammad Asaduzzaman, Chanchal K Roy, Kevin A Schneider, and Daqing Hou. Recommending framework extension examples. In *2017 IEEE International Conference on Software Maintenance and Evolution*, pages 456–466. IEEE, 2017.

[8] Yunior Pacheco, Jonas De Bleser, Tim Molderez, Dario Di Nucci, Wolfgang De Meuter, and Coen De Roover. Mining Scala Framework Extensions for Recommendation Patterns. In *SANER 2019 - Proceedings of the 2019 IEEE 26th International Conference on Software Analysis, Evolution, and Reengineering*, 2019.

[9] Abbas Heydarnoori, Krzysztof Czarnecki, Walter Binder, and Thiago Tonelli Bartolomei. Two studies of framework-usage templates extracted from dynamic traces. *IEEE Transactions on Software Engineering*, 38(6):1464–1487, 2012.

[10] Raquel FQ Lafetá, Marcelo A Maia, and David Röthlisberger. Framework instantiation using cookbooks constructed with static and dynamic analysis. In *2015 IEEE 23rd International Conference on Program Comprehension*, pages 125–128. IEEE, 2015.

[11] Amir Michail. Data mining library reuse patterns in user-selected applications. In *ase*, page 24. IEEE, 1999.

[12] Amir Michail. Data mining library reuse patterns using generalized association rules. In *Proceedings of the 22nd international conference on Software engineering*, pages 167–176. ACM, 2000.

[13] Alvaro Ortigosa, Marcelo Campo, and Roberto Moriyón. Towards agent-oriented assistance for framework instantiation. *SIGPLAN Notices (ACM Special Interest Group on Programming Languages)*, 35(10):253–263, 2000.

[14] Barthélémy Dagenais and Harold Ossher. Automatically locating framework extension examples. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 203–213. ACM, 2008.

[15] Talita Gomes, Toacy Cavalcante de Oliveira, Donald D Cowan, and Paulo SC Alencar. Mining reuse processes. In *CIbSE*, pages 179–190, 2014.

[16] Ivan Mathias Filho, Toacy C. De Oliveira, and Carlos J.P. De Lucena. A framework instantiation approach based on the Features Model. *Journal of Systems and Software*, 73(2):333–349, 2004.

[17] Talita Lopes Gomes. *REUSE MINER: MINING FRAMEWORK INSTANTIATION PROCESSES*. PhD thesis, 2015.

[18] Edson M. Lucas, Toacy C. Oliveira, Kleinner Farias, and Paulo S.C. Alencar. CollabRDL: A language to coordinate collaborative reuse. *Journal of Systems and Software*, 131:505–527, 2017.

[19] Toacy C. Oliveira, Paulo S.C. Alencar, Carlos J.P. de Lucena, and Donald D. Cowan. RDL: A language for framework instantiation representation. *Journal of Systems and Software*, 80(11):1902–1929, 2007.

[20] Toacy Oliveira, Paulo Alencar, Marcilio Mendonca, and Donald Cowan. Assisting Framework Instantiation: Enhancements to Process-Language-based Approaches. Technical Report August, 2005.

[21] Toacy C. Oliveira, Paulo S.C. Alencar, Ivan M. Filho, Carlos J.P. De Lucena, and Donald D. Cowan. Software process representation and analysis for framework instantiation. *IEEE Transactions on Software Engineering*, 30(3):145–159, 2004.

[22] Martin Monperrus, Michael Eichberg, Elif Tekes, and Mira Mezini. What should developers be aware of? An empirical study on the directives of API documentation. *Empirical Software Engineering*, 17(6):703–737, 2012.

[23] Yu Zhou, Changzhi Wang, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald Gall. Automatic detection and repair recommendation of directive defects in java api documentation. *IEEE Transactions on Software Engineering*, 46(9):1004–1023, 2018.

[24] Yu Zhou, Xin Yan, Taolue Chen, Sebastiano Panichella, and Harald Gall. Drone: a tool to detect and repair directive defects in java apis documentation. In *2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*, pages 115–118. IEEE, 2019.

[25] Amann Sven, Hoan Anh Nguyen, Sarah Nadi, Tien N. Nguyen, and Mira Mezini. Investigating Next Steps in Static API-Misuse Detection. pages 265–275, 2019.

[26] Ruben Opdebeeck, Johan Fabry, Tim Molderez, Jonas De Bleser, and Coen De Roover. Mining for graph-based library usage patterns in cobol systems. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 595–599. IEEE, 2021.

[27] Charu C Aggarwal and Rodrigo Goyena. *Data mining: the textbook*, volume 53. Springer, 2015.

[28] Pedro Martins, Rohan Achar, and Cristina V Lopes. The java build framework: Large scale compilation. *arXiv preprint arXiv:1804.04621*, 2018.

[29] Michael Mattsson, Jan Bosch, and Mohamed E Fayad. Framework integration problems, causes, solutions. *Communications of the ACM*, 42(10):80–87, 1999.