Automated Refactoring for Asynchronous Applications

Bachelor-Thesis von Grebiel José Ifill Brito aus Caracas, Venezuela
Tag der Einreichung:

1. Gutachten: Prof. Dr. Guido Salvaneschi
2. Gutachten: M.Sc. Pascal Weisenburger

TECHNISCHE UNIVERSITÄT DARMSTADT

Computer Science
Reactive Programming Technology

Automated Refactoring for Asynchronous Applications

Vorgelegte Bachelor-Thesis von Grebiel José Ifill Brito aus Caracas, Venezuela

1. Gutachten: Prof. Dr. Guido Salvaneschi
2. Gutachten: M.Sc. Pascal Weisenburger

Tag der Einreichung:

Abstract

Modern languages are replacing the traditional callback-based approach with event-driven, reactive and functional programming models. According to previous studies, these programming paradigms improve code writing and code comprehension.

Motivated by these facts, we propose a refactoring approach to convert a callback-based async construct into functional reactive programming code. This thesis focuses on refactoring the `SwingWorker` Java Swing construct.

We developed SwingWorker2Rx, a system to perform the mentioned refactoring automatically.

We evaluate the accuracy of SwingWorker2Rx using 58 projects. In these projects there are 678 elements (declarations, instances, invocations, methods, variables, fields, among others) that contained the target async construct. Our tool successfully refactored 676 of these elements, which represents approximately 99.7%. These results show that the tool is highly accurate.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

Asynchronous programming refers to the execution of program code in multiple threads. These threads run concurrently. Synchronous programming, on the contrary, refers to the sequential execution of program code in a single thread. Most UI frameworks are single threaded [17]. When a long-running operation is executed in the UI thread, then the user interface freezes until the operation has completed. This happens because all commands in the UI thread are executed sequentially. To avoid this blocking behavior, developers use asynchronous programming, which allows them to execute the long-running operation in a background thread. Since the background and the UI thread run concurrently, the UI thread remains responsive.

Several tools have been developed in the last years to automatically refactor the source code of asynchronous applications. ASYNCHRONIZER [37], ASYNCDROID [20], ASYNCIFIER [18], ASYNCFIXER [18], PROMISESLAND [26] and RXFACTOR [34] are some of these tools. Each of them solves different problems. ASYNCHRONIZER and ASYNCDROID focus on performance improvement in Android applications, while ASYNCFIER and PROMISSESLAND aim increasing the code readability in C# and JavaScript respectively. ASYNCFIXER helps finding and correcting anti patterns of async constructs in C#. Finally RXFACTOR is a refactoring tool that converts `AsyncTask`s into an implementation that uses the ReactiveX API. RXFACTOR was developed parallel to this thesis by another author.

On the other hand, functional programming has been gaining popularity in the last years. Java 8, for example, offers several classes to support working with data streams in a functional fashion [35]. Other technologies such as Flapjax, ReactiveX and Agera are not only using functional concepts but also focusing on the reactive programming paradigm. Flapjax is a programming language that introduces event-driven reactivity as the natural programming model for web applications [28]. ReactiveX is a library based on the observer pattern that supports data streams and functional programming [11]. Agera also offers classes for functional, asynchronous and reactive programming in Android applications [2].

According to previous researches, reactive and functional programming models improve code writing and code comprehension [10, 22]. One can refactor existing applications in other to facilitate the implementation of these models. Depending on the size of the projects and the amount, the refactoring task can demand a lot of effort. Implementing tools to perform the refactorings automatically minimizes this effort and allows developers focusing on new features which can be added using reactive and functional programming concepts.

1.1 Contribution

In this thesis, we show how a traditional callback-based async construct can be refactored into another construct that facilitates the implementation of reactive and functional programming models. Since performing these refactorings by hand in multiple or large projects demands a lot of time, we designed a tool to accomplish this task.

We focus on refactoring `SwingWorker`s into an implementation that uses the ReactiveX API for Java (RxJava). To perform the refactorings automatically, we developed a tool called 2RX. One can easily add extensions to this tool. We implemented SWINGWORKER2RX, which can convert `SwingWorker`s to RxJava. Additionally, we adapted RXFACTOR, a similar tool that converts `AsyncTask`s into RxJava, to make it a client of 2RX as well. Although RXFACTOR and SWINGWORKER2RX are very similar, the refactoring approaches are different. The approach proposed in this thesis overcomes the following weaknesses of RXFACTOR: Not supporting all methods available in the `AsyncTask` API; Vulnerable to backpressure problems.

Contribution summary:

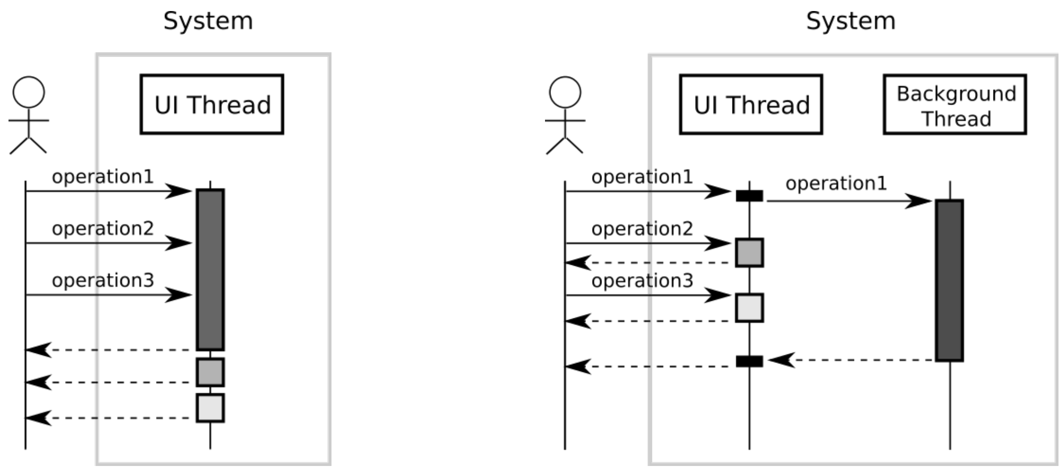
- A refactoring approach to convert `SwingWorker`s into RxJava
- A system to host refactoring tools (2RX)
- A client of 2RX responsible for refactoring `SwingWorker`s into RxJava (SWINGWORKER2RX)
- A client of 2RX responsible for refactoring `AsyncTask` into RxJava based on the implementation of RXFACTOR

1.2 Structure

The thesis is structured as follows: Chapter 2 provides some background about refactoring, asynchronous programming, the observer pattern, and functional and reactive programming. In this chapter we explain the `SwingWorker` API, followed by the state of the art in automated refactoring for asynchronous applications. Finally, we talk about modern technologies that use event-driven, reactive and/or functional programming concepts. We show the design of the refactoring approach and 2RX in Chapter 3. The implementation details are explained in Chapter 4. In Chapter 5 we present the evaluation and its results and in Chapter 6 we summarize the work, present our conclusion and recommendations for future research on this topic.

Asynchronous programming refers to the execution of program code in multiple threads. Since most UI frameworks are single threaded [17], developers use asynchronous programming to improve the responsiveness of the UI. This improvement is accomplished by executing long-running operations on a background thread [18, 33].

In Figure 2.1 we illustrate how asynchronous programming is used in order to avoid that a long-running operation blocks the user interface. In the diagram, we compare the synchronous and asynchronous execution of three operations. The first operation requires more time than the other two. In Figure 2.1a the longest operation is executed first. After that, `operation2` and `operation3` are triggered. As we can see, `operation2` and `operation3` cannot be executed immediately because the UI thread is busy. These operations can be anything, for example, UI events. Since the thread cannot process these events, the system is said to be unresponsive. Figure 2.1b shows how we can solve this problem by using a background thread. Basically, long-running operations are passed to a different thread. As we can see in the diagram, while `operation1` is performed in the background thread, the UI thread remains free and can execute other operations. Eventually, when the long-running operation has completed, the result is sent to the main thread. It is important to consider that delegating operations to background threads generates overhead. This overhead is represented with two small black rectangles.



(a) Synchronous **(b)** Asynchronous

Figure 2.1: Synchronous vs. Asynchronous Execution

Listings 2.1 and 2.2 are two examples of asynchronous programming in Java. The implementation of Listing 2.1 is fire and forget, which means, that no result is expected. The asynchronous operation starts in Line 4 and the rest of the code keeps running sequentially, regardless of the amount of time required to finish the asynchronous operation. The implementation of Listing 2.2 does return a result. The asynchronous operation starts in Line 4 and the result is retrieved in Line 9. These examples are used in the next sections to explain the concepts of data races and promises, also called futures.

```
1  Runnable runnable = () -> performOpAsync();
2
3  Thread thread = new Thread(runnable);
4  thread.start();
5
6  performOperation();
7  // performOpAsync and performOperation run
    ↪  concurrently
```

Listing 2.1: Async Runnable (Fire & Forget)

```
1  Callable<Integer> task = () -> computeResult();
2
3  ExecutorService ex = Executors.newFixedThreadPool( 4 );
4  Future<Integer> future = ex.submit( task );
5
6  performOperation();
7  // computeResult and performOperation run concurrently
8
9  Integer asyncResult = future.get();// blocks until task has
    ↪  completed
```

Listing 2.2: Async Callable (Future)

2.1 Data Races

Asynchronous implementations and shared memory models are vulnerable to data races. A data race occurs when multiple threads write a variable in an unspecific order [24]. Since the order in which threads are executed is not well-defined, the result of the modified variable is non-deterministic. To avoid this problem, the developer has to implement locks to make sure that the order of execution is deterministic.

In Listing 2.1 there are two operations that run concurrently, `performOpAsync` and `performOperation`. Assuming that these methods modify a variable `var` and that there is no synchronization mechanism, then we have a data race because we cannot tell for sure which operation modified the value at last. The same happens in Listing 2.2 with `computeResult` and `performOperation`.

2.2 Promises

Promises, also known as Futures, are language constructs that are used to synchronize concurrent code. These constructs consist of a reference to the result of a running operation [30].

In JavaScript for example, Promises can be `pending` (not started), `fulfilled` (successful operation) or `rejected` (failed operation) [10]. Since Promises can be chained together, developers need not to use traditional callback-based approaches to handle the return values. According to previous studies, chained calls are easier to understand than callback-based approaches [14].

Listing 2.2 shows an example of Futures in Java. First the `task` is executed by the `ExecutorService` in a background thread. This call returns the reference to the object that contains the result, in this case, called `future`. Then, a second operation called `performOperation`, runs in the current thread. After this last operation has completed, we are ready to read the result from the asynchronous task. If the result is not available, then this call waits until the result has been computed. Otherwise, it writes the result in the variable `asyncResult` and continues the execution of the next lines of code.

2.3 Automated Refactoring

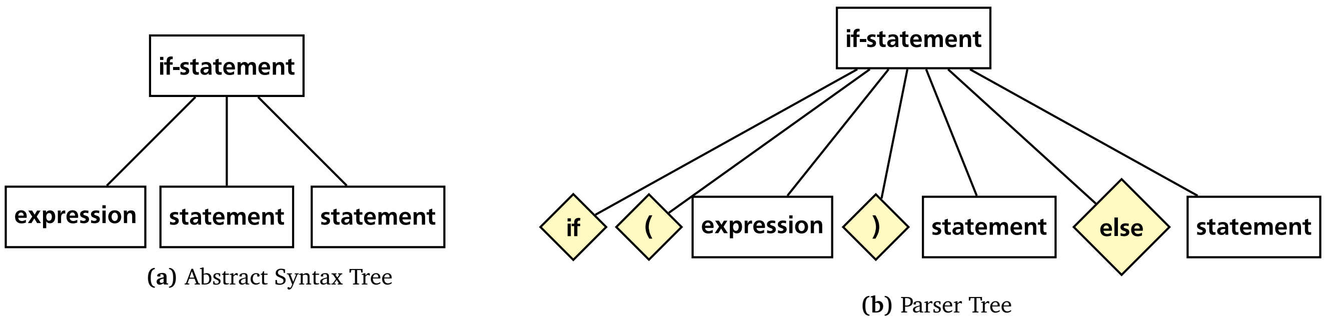
Refactoring consists of modifying the source code of a program without changing its behavior. Refactoring can be applied to extract a reusable component, improve consistency among components, supporting new features, among others [36].

Many IDEs already offer refactoring tools. Some of the common refactoring commands are: rename; move; change method signature; extract methods, local variables, constants, interfaces, superclass; among others [6, 9].

On the other hand, researchers have been developing tools to refactor asynchronous applications. Some of the tools convert synchronous into asynchronous code. Other tools refactor asynchronous code that uses the traditional callback-based approach into method chaining [18, 20, 26, 37]. We explain these tools in Section 2.9.

2.3.1 Abstract Syntax Trees

Abstract syntax trees (AST) are data structures that only contain the essential information about the source code. Each node corresponds to a construct of the specific language. Some examples of these constructs in Java are: assignment, field declaration, variable declaration, method declaration, class instance creation, type declaration, if-statement, try-statement, anonymous class declaration, body declaration, catch clause, among others. In contrast to parse trees, ASTs does not contain the symbols required to compile the code, such as "{", "}", ";", among others [29]. Figure 2.2 shows the difference between both types of trees.



(a) Abstract Syntax Tree

(b) Parser Tree

Figure 2.2: Abstract Syntax Tree vs. Parser Tree [29, p. 216]

As we mentioned in Section 2.3, most IDEs offer refactoring tools. Eclipse, for example, uses AST to get details about the source code and write or modify changes in it (Figure 2.2a).

2.4 Functional Programming

Functional programming is a programming paradigm that uses mathematical functions as its main programming construct [31]. This paradigm avoids using concepts such as state and mutable data. Avoiding these concepts facilitates implementation, testing, debugging and code comprehension [25].

In Listings 2.3 and 2.4 we compare imperative and functional programming. One can see, that the functional implementation is much compacter than the imperative one. However, this does not apply for every case. Developers must reason about which of these paradigms are more suitable to the given problem.

```java
List<Integer> numbers =
  Arrays.asList( 1, 3, 4, 5, 8, 13, 15 );
List<String> evenNumbers = new ArrayList<>();
for ( int i = 0; i < numbers.size() ; i++ ) {
    Integer n = numbers.get(i);
    if ( n % 2 == 0)
        evenNumbers.add(String.valueOf(n));
}
doSomething(evenNumbers);
```

Listing 2.3: Imperative Programming (Java 8)

```java
List<Integer> numbers =
  Arrays.asList( 1, 3, 4, 5, 8, 13, 15 );
List<String> eventNumbers = numbers.stream()
        .filter(n -> n % 2 == 0)
        .map(n -> String.valueOf(n))
        .collect(Collectors.toList());
doSomething(eventNumbers);
```

Listing 2.4: Functional Programming (Java 8)

2.5 Event Driven Programming

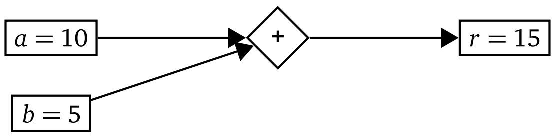
Event driven programming consists of program code that gets executed when an event occurs. Most Windows programs are a good example of this since they are written using event-driven models. If an event never occurs, then the piece of code associated with that event will never be executed. If there is no piece of code associated with an event, then the event will be ignored. [27].

2.5.1 Reactive Programming

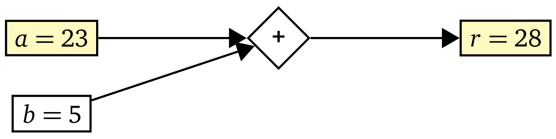
Reactive programming is a programming paradigm that bases in the propagation of change. This programming model is considered a special case of the event-driven paradigm. The events refer mostly to data changes [32]. In reactive programming, there is a data-flow graph that indicates how the changes are propagated.

Figure 2.3 illustrates how reactive programming works. The operation $a + b$ produces the result r. The first state is: $a = 10$, $b = 5$, $r = 15$. In imperative programming, changing the values of a and/or b does not automatically affect the result r, but in reactive programming that is not the case. If the value of a or b changes, the operation $+$ is performed and r is updated.

The previous example also shows a relation between the observer pattern and reactive programming. Let a and b be the subjects and r be the observer. If r updates its value every time a or b change, then the behavior described corresponds to a reactive program as well.



(a) State 1: $a = 10$, $b = 5$, $r = 15$



(b) State 2: $a \to 23$ then $a = 23$, $b = 5$, $r = 28$

Figure 2.3: Dataflow Example

2.6 Observer Pattern

The idea of the observer pattern is to establish a one-to-many relationship between objects. The dependent objects are called observer. The independent objects or subjects notify their observers when their state change. This pattern allows developers modifying observers and subjects independently. Also, observers can be added without applying changes to the existing subject nor the other observers [21].

The usage of the observer pattern in asynchronous applications can cause backpressure problems when the subject notifies changes faster than the observer's capacity to process them. There are several solutions to tackle this problem. One of them is to have a buffer to accumulate the data received from the subject. Since the buffer can get full, this solution does not work for every case. An alternative solution is to block the notifications from the subject until the data has been processed by the observer [12]. The disadvantage of the last approach is that the subject is as fast as the slowest observer. A combination of both approaches is also possible.

2.7 Reactive and/or Functional Technologies

Empirical studies have shown that reactive programming increases the correctness of program comprehension without requiring neither more time to analyze the program nor advance programming skills [22]. Furthermore, there are new technologies that are based on event-driven programming models and reactivity. Flapjax, ReactiveX and Agera are some of them. These technologies also agree that reactive and event-driven approaches simplify code writing and comprehension [2, 11, 28].

2.7.1 Flapjax

Flapjax is a language built on top of JavaScript to enable event-driven programming. It can also be used as a library if developers do not want to use the Flapjax-to-JavaScript compiler [28].

Flapjax introduces two new data types, `Behaviors` and `Event—Streams`. `Behavior`s are values that change over time (i.e: a variable). Changes in `Behaviors` propagate automatically, facilitating developers consistency in their applications. `Event—Stream`s represent the input sources. [28].

Listing 2.5 shows a basic JavaScript example where a string value `id="validationMessage"` (Line 31) is updated when an event "`onChange`" is triggered. Since there is no native JavaScript call to determine if an input have changed, a possible (trivial) implementation could be for example triggering the `validateNumber` function every 200*ms* (Line 22). We chose this time interval because we consider it small enough to be perceived as an immediate change. Notice that the function `startValidation` must be called on load (Line 27). The validation logic is implemented in the function `validateNumber` (Line 5).

Listing 2.6 shows the same example using Flapjax. In this case, Flapjax was downloaded and used as a library (Line 2). The implementation basically defines a behavior for the input `numb` (Line 7). The function `liftB` (Line 8) creates a time varying value which is used for updating the `validationMessage`. The function `loader` defines the events (Line 8) and their reaction (Line 20). This function is also called on load (Line 26). This example shows how Flapjax allows keeping the UI consistency without having to implement a mechanism for updating a particular field. Instead, events and reaction to those events are defined.

More interactive examples can be found at Flapjax's official site [7].

`Event—Streams` can be processed in Flapjax using functions such as: `mapE`, `mergeE`, `filterE`, `andE`, `orE`, `notE`, among others.

These functions are also available in ReactiveX [11].

2.7.2 ReactiveX

ReactiveX is a library that supports data streams and reactivity. It is available in many platforms (Java, JavaScript, C#, C++, Ruby, Android, among others) [11].

The API of ReactiveX for Java programs is called RxJava. By using RxJava it is possible to write asynchronous programs in a functional fashion. Additionally, since RxJava is based on the observer pattern, it is also possible to implement reactive models.

Listing 2.7 shows an example where we use RxJava to select some items, transform them and save the newly generated items. We use a background thread for the operations and update the UI when done. The process is as follows: First, we create a `rx.Observable` object using the method `Observable.from` (Line 4). Then, we invoke further methods to modify the data stream (`filter`, `map`). After that, we specify in which thread the operation must be executed (Line 12). RxJava offers several `Scheduluers` for that purpose. However, one can also use custom executors. Next,

```
<html>
<head>
<title>JavaScript</title>
<script type="text/javascript">
function validateNumber() {
    var x, text;
    x = document.getElementById("numb").value;
    if ( x == ""){
            text = "";
    } else if (isNaN(x)) {
        text = "Not a number";
    } else if (x < 1 || x > 10) {
        text = "Number out of Range";
    } else {
        text = "Valid number";
    }
    document.getElementById("validationMessage")
      .innerHTML = text;
}
function startValidation(){
  validation =
      setInterval("validateNumber()",  200);
}
</script>
</head>

<body onload="startValidation()">
<h1>Number Validator (JavaScript)</h1>
<p>Please input a number between 1 and 10:</p>
<input id="numb">
<p id="validationMessage"></p>
</body>
</html>
```

Listing 2.5: JavaScript

```
<html>
<script type="text/javascript"
 ↪  src="flapjaxLibrary.js"></script>
<title>Flapjax</title>
<script type="text/javascript">

function loader() {
  var valB = extractValueB('numb', 'value');
  var validationStrB = valB.liftB(
    function (s) {
      if ( s == ""){
          return "";
      } else if (isNaN(s)) {
          return "Not a number";
      } else if (s < 1 || s > 10) {
          return "Number out of Range";
      } else {
          return "Valid number";
      }
    });
  insertDomB(validationStrB, 'validationMessage');
}

</script>
</head>

<body onload="loader()">
<h1>Number Validator (Flapjax)</h1>
<p>Please input a number between 1 and 10:</p>
<input id="numb">
<p id="validationMessage"></p>
</body>
</html>
```

Listing 2.6: Flapjax

```
...
List<Product> existingProducts = new ArrayList<>();
  existingProducts.addAll(productDao.getProducts());
  Observable.from(existingProducts)
    .filter(p ->
          p.getPrice() > 50 &&
          p.getStore().equals(STORE_A))
    .map(p -> new Product(
            p.getId(),
            p.getPrice() * 1.10,
            STORE_B))
    .subscribeOn(Schedulers.computation()) // do in background
    .doOnNext(p -> productDao.save(p))
    .doOnError(t -> handleError(t))
    .observeOn(Schedulers.uiThread()) // only as example: this scheduler doesn't exist
    .doOnCompleted(() -> updateUI())
    .subscribe();
  );
...
```

Listing 2.7: RxJava: Streams

```java
1  public static void main( String[] ars ) {
2      BehaviorSubject<Integer> a = BehaviorSubject.create( 1 );
3      BehaviorSubject<Integer> b = BehaviorSubject.create( 2 );
4      final BehaviorSubject<Integer> sum = BehaviorSubject.create( 0 );
5      final BehaviorSubject<Integer> mult = BehaviorSubject.create( 0 );
6      Observable.combineLatest( a, b, ( n1, n2 ) -> n1 + n2 ).subscribe( sum );
7      Observable.combineLatest( a, b, ( n1, n2 ) -> n1 * n2 ).subscribe( mult );
8      // check initial states of sum and mult
9      System.out.println(sum.getValue()); // output: 3
10     System.out.println(mult.getValue()); // output: 2
11     a.onNext(5); // modifies subject a
12     System.out.println(sum.getValue()); // output: 7
13     System.out.println(mult.getValue()); // output: 10
14     b.onNext(10); // modifies subject b
15     System.out.println(sum.getValue()); // output: 15
16     System.out.println(mult.getValue()); // output: 50
17 }
```

Listing 2.8: RxJava: Reactive Code

we define the save operation using `doOnNext` (Line 13). Since in this example the `doOnCompleted` operation must be executed in the UI thread, we place the `observeOn` declaration before `doOnCompleted` (Line 15). Finally, we subscribe the observable. This subscription starts executing the async operation.

Listing 2.8 shows an example where we use RxJava to implement two reactive operations (addition and multiplication). The first step is to define the subjects (Lines 2-3). Then, we declare the subscribers (Lines 4-5), also called observers. Finally, we define the behavior (Lines 6-7). We do this by creating an observable that combines both subjects and defines the result. Then we subscribe the corresponding subscriber to each observable. Lines 11 and 14 show how we manipulate the subjects, so the changes are propagated to the subscribers.

RxJava offers many other classes and methods for building asynchronous reactive models. Explaining all of them does not belong to the scope of this thesis. More information and examples about this library can be found at [32].

2.7.3 Java 8

Java 8 introduces a series of classes that allows working with data streams in a functional fashion [35]. However, in contrast to RxJava, the new classes of Java 8 do not support defining which operations should be executed in the background and which ones on the UI thread.

```java
Observable.from(productDao.getProducts())
        .filter(p ->
                p.getPrice() > 50 &&
                p.getStore().equals(STORE_A))
        .map(p -> new Product(
                p.getId(),
                p.getPrice() * 1.10,
                STORE_B))
        .subscribeOn(Schedulers.computation())
        .doOnNext(p -> productDao.save(p))
        .doOnError(t -> handleError(t))
        .observeOn(Schedulers.uiThread()) // only as
         ↪   example: this scheduler doesn't exist
        .doOnCompleted(() -> updateUI())
        .subscribe();
```

Listing 2.9: RxJava (Products Example)

```java
productDao.getProducts().stream()
        .filter(p ->
                p.getPrice() > 50 &&
                p.getStore().equals(STORE_A))
        .map(p -> new Product(
                p.getId(),
                p.getPrice() * 1.10,
                STORE_B))
        .forEach(p -> productDao.save(p));
```

Listing 2.10: Java 8 (Products Example)

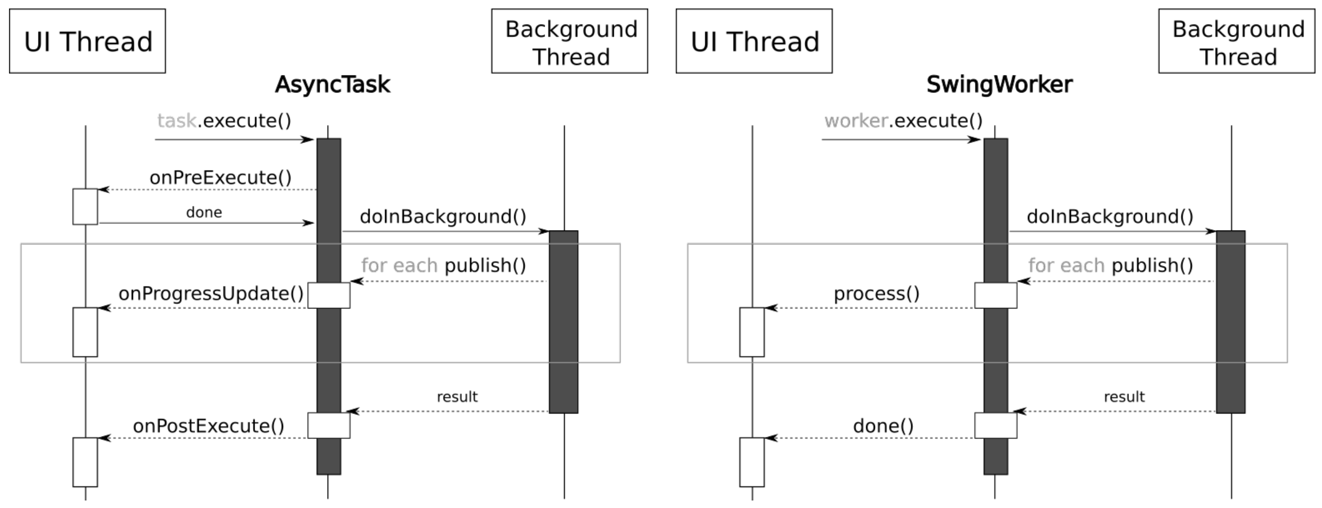
Listings 2.9 and 2.10 show a comparison between RxJava and Java 8. There are two method invocations that are identical and one invocation that is similar but not completely equivalent. Using functions such as `filter` and `map` can be done with both, RxJava and Java 8, in the same way. The call `doOnNext` is similar to `forEach` in the sense of iterating through all items and doing some operations with them. The big difference is that `forEach` actually starts executing the operation while `doOnNext` only defines it. In the second case the operation starts when the observable is subscribed (Listing 2.10, Line 14). Also, `forEach` does not not allow chaining futher methods. Methods such as `subscribeOn`, `doOnError`, `observeOn`, `doOnCompleted`, among others have no equivalent in Java 8.

2.8 Async Constructs

There are several async constructs in many different programming languages and libraries. We want to focus particularly in `SwingWorkers` because that is the construct that we analyze in this thesis. Additionally, we want to explain the basics about `AsyncTasks` because previous research has been analyzing them and developing refactoring tools for different purposes.

Figure 2.4a illustrates in which threads the methods of an `AsyncTask` are executed. In this example, the `AsyncTask` starts with the invocation of `execute`. Then `onPreExecute` is invoked in the UI thread. As soon as this method finishes, `doInBackground` is invoked on a background thread. While the asynchronous operation is running, data can be sent to UI thread throw the invocation of `publish`. This data is processed in the UI thread using the logic specified in `onProgressUpdate`. The `publish` method can be invoked multiple times. Finally, when the asynchronous operation has completed, the result is processed in the UI thread using the `onPostExecute` method.

Figure 2.4b shows which methods of a `SwingWorker` are executed in the background and which ones in the UI thread. The `SwingWorker` also starts when `execute` is invoked. Notice that `SwingWorkers` do not have an `onPreExecute` method. Like `AsyncTasks`, `SwingWorkers` also have a `publish` method to send data to the UI thread. This data is processed according to the logic contained in the `process` method. The `publish` method can be called multiple times too. Finally, after the asynchronous operation has completed, the method `done` is invoked, which is also executed in the UI thread. Inside `done` the method `get` can be invoked, in order to have access to the result of the async operation.



(a) AsyncTask Execution

(b) SwingWorker Execution

Figure 2.4: AsyncTask vs. SwingWorker

`SwingWorkers` and `AsyncTasks` also have some similar methods. We match these methods in Table 2.1. Methods written in **bold** depend on the state of the `SwingWorker` and/or `AsyncTask`.

However, not all methods have an equivalent. Table 2.2 shows all methods that do not have a match. Again, methods written in **bold** depend on the state of the class. As we can see, `SwingWorkers` have more methods that depend on the state than `AsyncTasks`. This point will be discussed again in Chapter 3, when we talk about the refactoring approach that we use to convert `SwingWorkers` into RxJava.

For more details about the `SwingWorker` API go to Appendix 8.1.

2.9 Refactoring Tools

Modern programming languages such as C#, Visual Basic, F# and Scala have introduced `asycn` constructs and `await` calls to facilitate the implementation of asynchrony. With these constructs developers do not need to implement callbacks to manage asynchrony [18].

SwingWorker [16]	AsyncTask [3]
doInBackground()	doInBackground(...)
process(...)	onProgressUpdate(...)
publish(...)	publishProgress(...)
done()	onPostExecute(...)
cancel(...)	cancel(...)
execute()	execute(...)
getState()	**getStatus()**
isCancelled()	**isCancelled()**
get()	get()
get(...)	get(...)

Table 2.1: Equivalent Methods - SwingWorker vs. AsyncTask (1)

SwingWorker [16]	AsyncTask [3]
addPropertyChangeListener(...)	–
firePropertyChange(...)	–
getProgress()	–
getPropertyChangeSupport()	–
isDone()	–
removePropertyChangeListener(...)	–
run()	–
setProgress(...)	–
–	onPreExecute()
–	executeOnExecutor(...)
–	onCancelled()

Table 2.2: Equivalent Methods - SwingWorker vs. AsyncTask (2)

According to previous studies, async constructs are being underused or misused. A common example of misused asynchrony is to introduce elements that appear to be asynchronous, but due to semantic mistakes, the code still runs synchronously. To tackle this problem, refactoring tools have been developed [18].

Refactoring asynchronous applications is not trivial. Therefore researchers have already started developing tools to assist developers in this task. These tools can be classified into three groups:

1. Synchronous code → asynchronous code
2. Callback based asynchronous code → method chained asynchronous code
3. Correction of anti-patterns and performance improvements

2.9.1 Synchronous → Asynchronous

According to a previous research, refactoring synchronous Android applications to be asynchronous is not an easy task [18]. The author of this work, Danny Dig, mentions two main reasons that difficulties this refactoring task. The first one is that most documentation focuses on the design from scratch, rather than on the process of converting synchronous code into asynchronous and the second one is that there are not enough methods neither tools to perform this kind of refactoring.

Motivated by these facts, the author developed ASYNCHRONIZER. ASYNCHRONIZER targets Android applications. It can be used to convert synchronous code into asynchronous by using an `AsyncTask` [37]. ASYNCHRONIZER basically moves the synchronous code into the `doInBackground` method of the `AsyncTask`. Then it analyzes the rest of the code in order to determine which part can be placed into the `onPostExecute` handler. Finally, it creates an instance of the class in the main thread and calls its `execute` method.

Converting synchronous code into asynchronous might produce data races. To make sure that there are no data races after refactoring, they implemented an extension of the ITERACE detector. This component is included with ASYNCHRONIZER. It is important to mention that this check does not run automatically. Developers have to explicitly check for data races after refactoring the code. Reported data races should be analyzed by developers to determine whether they are real or fake. To do that, developers must consider the application's workflow. Some methods in Android are by design never called concurrently (.i.e `onCreateView` and `onDestroyView`) [37].

2.9.2 Callback based → Method chained

ASYNCIFIER is a .NET tool that automatically refactors callback-based asynchronous code into `async/await`. This tool has already been tested using real-world applications [18].

Another tool in this category is PROMISESLAND. This tool converts JavaScript asynchronous callbacks into Promises. This refactoring facilitates code comprehension by replacing callbacks with chained calls [26]. PROMISESLAND consist of a static analyzer and a transformation engine. The static analyzer is in charge of searching for asynchronous patterns that can be refactored into Promises. The transformation engine is responsible for performing the refactoring [26].

In the last months, RXFACTOR was developed. As we mentioned before, this tool has some similarities with 2RX. RXFACTOR takes the code from `AsyncTask`s and generates a functional implementation of that code using the ReactiveX API. Although RXFACTOR and our tool have the same goal, and `AsyncTask`s are very similar to `SwingWorker`s, the refactoring approaches used are very different. We compare both approaches in Chapter 3.

2.9.3 Anti-patterns → Improvements

`AsyncTask` should only be used for short-running operations (approx. less than a second). For long-running operations the class `IntentService` should be used [20]. This refactoring is also not trivial. According to a previous research, this might be due to the developer's lack of knowledge about how to use this class. The author of that study believes that the lack of knowledge is the consequence of not having enough production-level examples that use `IntentService` [20]. ASYNCDROID is a tool that can be used to convert `AsycnTask` into `IntentService`.

ASYNCFIXER is a .NET tool that can be used to recognize performance anti-patterns of `async/await` in mobile applications (i.e. in Windows Phone). This tool also suggests fixes. ASYNCFIXER has already been successfully tested using real-world applications [18].

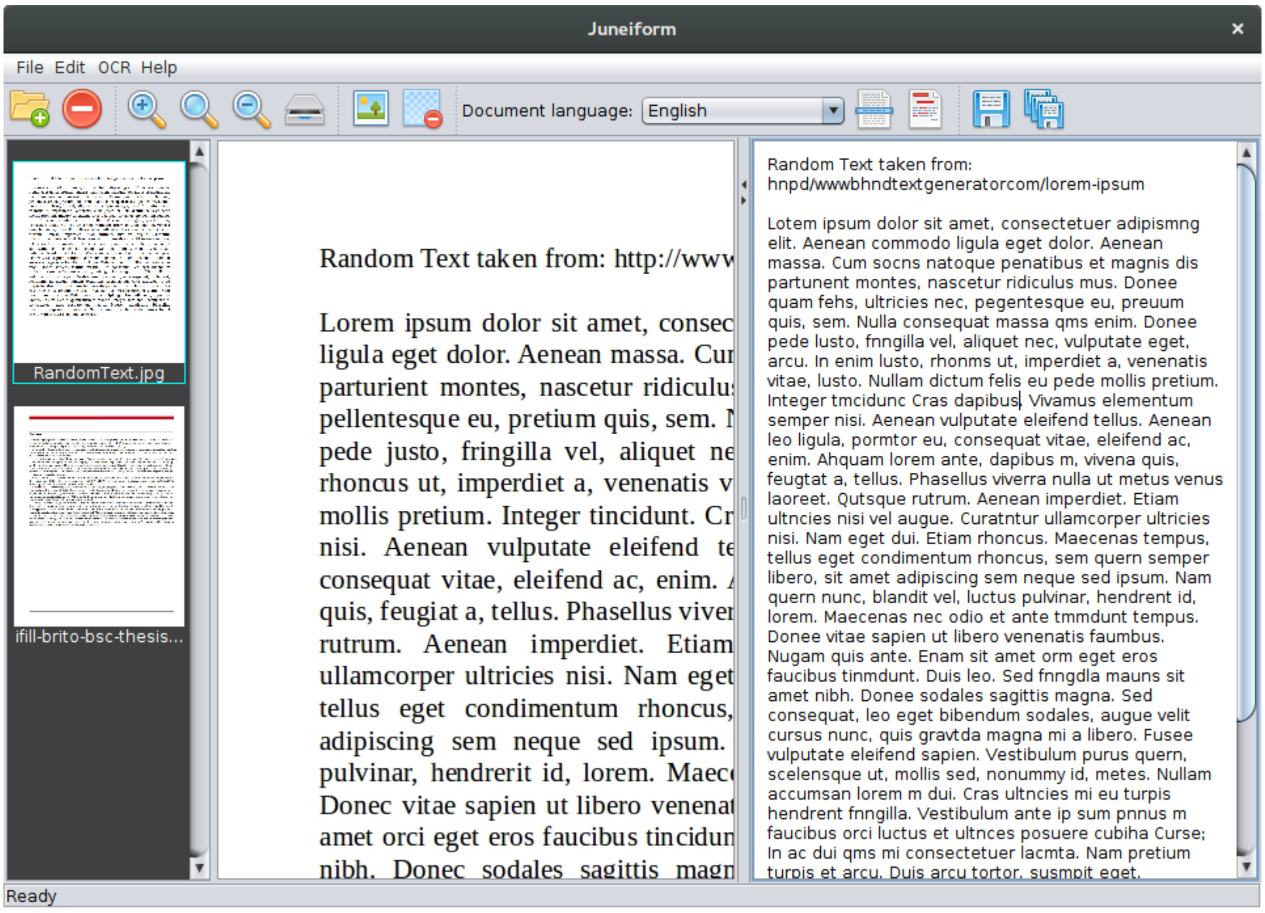
3 Design of the System

Previous studies show that functional and reactive programming models improve code writing and code comprehension [10, 22]. In this thesis, we propose an approach to refactor `SwingWorkers` into RxJava. This refactoring enables constructs that facilitate the implementation of the previously mentioned models. To minimize the effort required to perform the refactoring task, we developed a tool, SWINGWORKER2RX, that refactors `SwingWorkers` automatically. Once the automated refactoring has completed, one can use the constructs available in RxJava to introduce reactive and functional programming concepts into the existing project.

This Chapter starts by presenting a practical example that shows how one can use SWINGWORKER2RX to refactor a real-world application and add new features to it using RxJava. Then we explain the refactoring approach. Finally, we present the design of 2RX and SWINGWORKER2RX.

3.1 Use Case Example

The next example illustrates how the refactoring works. The example bases on Juneiform (Figure 3.1), an application with OCR that extracts text from images.



Random Text taken from: http://www
Lorem ipsum dolor sit amet, consec
ligula eget dolor. Aenean massa. Cu
parturient montes, nascetur ridiculu
pellentesque eu, pretium quis, sem. N
pede justo, fringilla vel, aliquet ne
rhoncus ut, imperdiet a, venenatis v
mollis pretium. Integer tincidunt. Cr
nisi. Aenean vulputate eleifend te
consequat vitae, eleifend ac, enim. A
quis, feugiat a, tellus. Phasellus viver
rutrum. Aenean imperdiet. Etiam
ullamcorper ultricies nisi. Nam eget
tellus eget condimentum rhoncus,
adipiscing sem neque sed ipsum.
pulvinar, hendrerit id, lorem. Maec
Donec vitae sapien ut libero venenat
amet orci eget eros faucibus tincidun
nibh. Donec sodales sagittis magn

Random Text taken from:
hnpd/wwwbhndtextgeneratorcom/lorem-ipsum

Lotem ipsum dolor sit amet, consectetuer adipismng elit. Aenean commodo ligula eget dolor. Aenean massa. Cum socns natoque penatibus et magnis dis partunent montes, nascetur ridiculus mus. Donee quam fehs, ultricies nec, pegentesque eu, preuum quis, sem. Nulla consequat massa qms enim. Donee pede lusto, fnngilla vel, aliquet nec, vulputate eget, arcu. In enim lusto, rhonms ut, imperdiet a, venenatis vitae, lusto. Nullam dictum felis eu pede mollis pretium. Integer tmcidunc Cras dapibus Vivamus elementum semper nisi. Aenean vulputate eleifend tellus. Aenean leo ligula, pormtor eu, consequat vitae, eleifend ac, enim. Ahquam lorem ante, dapibus m, vivena quis, feugtat a, tellus. Phasellus viverra nulla ut metus venus laoreet. Qutsque rutrum. Aenean imperdiet. Etiam ultncies nisi vel augue. Curatntur ullamcorper ultricies nisi. Nam eget dui. Etiam rhoncus. Maecenas tempus, tellus eget condimentum rhoncus, sem quern semper libero, sit amet adipiscing sem neque sed ipsum. Nam quern nunc, blandit vel, luctus pulvinar, hendrent id, lorem. Maecenas nec odio et ante tmmdunt tempus. Donee vitae sapien ut libero venenatis faumbus. Nugam quis ante. Enam sit amet orm eget eros faucibus tinmdunt. Duis leo. Sed fnngdla mauns sit amet nibh. Donee sodales sagittis magna. Sed consequat, leo eget bibendum sodales, augue velit cursus nunc, quis gravtda magna mi a libero. Fusee vulputate eleifend sapien. Vestibulum purus quern, scelensque ut, mollis sed, nonummy id, metes. Nullam accumsan lorem m dui. Cras ultncies mi eu turpis hendrent fnngilla. Vestibulum ante ip sum pnnus m faucibus orci luctus et ultnces posuere cubiha Curse; In ac dui qms mi consectetuer lacmta. Nam pretium turpis et arcu. Duis arcu tortor, susmpit eget,

Figure 3.1: Juneiform: `https://bitbucket.org/Stepuk/juneiform` c07e0bbcf17c2d63137f28109cf5812a231692de

In this example we focus on two classes, `DocumentLoader` and `Editor`. The `DocumentLoader` is the class in charge of importing the images into Juneiform. Here we want to extend the functionality to add four observers to the event “load document”. The `Editor` class is responsible for performing the OCR. Here we want to display a loading spinner and close it after the optical character recognition has completed. Furthermore, we want to automatically copy the output text to the clipboard.

3.1.1 DocumentLoader

The user can import images into Juneiform by clicking on the icon “open Images” (Figure 3.2). This action opens a file chooser dialog. After the user has selected the source images and clicked “Open”, the application starts fetching the images. As soon as the fetching has completed, all images are displayed.

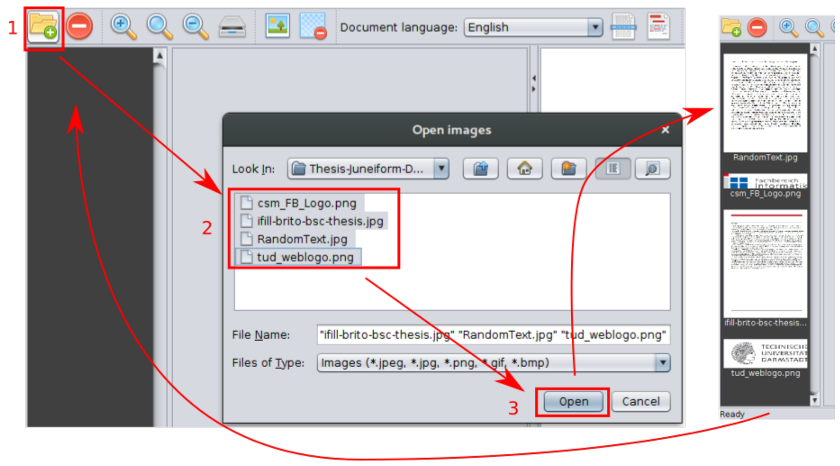


Figure 3.2: Load Documents in Juneiform

The class responsible for loading the images into Juneiform is `DocumentLoader` (Listing 3.1). When the user selects the images from the file chooser dialog (Figure 3.2) and clicks “Open”, the `load` (Listing 3.1: Line 4) method of `DocumentLoader` is invoked using the previously selected files as argument. The files are then cached into a private field called `files` (Listing 3.1: Line 7). After that, `execute` is invoked (Listing 3.1: Line 8). Since `DocumentLoader` is a subclass of the async construct `SwingWorker`, the next invocation is `doInBackground` (Listing 3.1: Line 11) which logic corresponds to the asynchronous operation. The asynchronous operation consists in iterating through all `files` (Listing 3.1: Line 13) and adding them one by one to a list `result` of type `Document` (Listing 3.1: Line 15). The type `Document` is implemented in Juneiform and holds relevant information about a file such as its name, absolute path, image, among others. When the algorithm has iterated through all `files`, then the list `result` contains the previously selected images. The asynchronous operation ends when the method `doInBackground` returns (Listing 3.1: Line 24). After the asynchronous operation has completed, the `done` (Listing 3.1: Line 27) method is invoked by the `SwingWorker`. Notice that `doInBackground` returns a list of type `Document`. The return value of `doInBackground` (in this example: `result`) can be accessed from the method `done` by using `get` (Listing 3.1: Line 29). The method `get` is declared in the superclass of `DocumentLoader` (Appendix 8.1). Since `get` can throw whether an `InterruptedException` or an `ExecutionException`, a `try—catch` block is needed. Finally, if no exceptions are thrown, the `fetchResult` method is called using the returned value of the asynchronous operation as the argument (all files). This method displays the selected images in the UI.

Before applying the automated refactoring, we manually refactored eight lines in the original implementation of `DocumentLoader` so that we can show the images in the UI as soon as they are available. The purpose of this refactoring is to show that our approach can handle dynamically generated items. These items are generated every time the method `publish` is invoked. Basically, we applied three changes:

1. We changed the second parameter of the `SwingWorker` from `Void` (Listing 3.1: Line 2) to `Document` (Listing 3.2: Line 2). The second parameter corresponds to the data type of the intermediate results.
2. We generated intermediate results by using `publish` (Listing 3.2: Line 15), instead of accumulating all `Documents` in a list (Listing 3.1: Line 15). Every time this method is invoked, the `SwingWorker` starts the execution of the method `process` using the intermediate results as the arguments (Listing 3.2: Line 27). The methods `publish` and `process` are declared in the superclass of `DocumentLoader` (Appendix 8.1).
3. We implemented `process`. The logic of `process` is similar to the one in `done`. The only difference is that `process` receives a list `chunks` (Listing 3.1: Line 27) of type `Document` (See the second parameter of `SwingWorker` in Listing 3.2: Line 2) which can directly be used in `fetchResult` to update the images displayed in the UI. Since `chunks` is a variable and `fetchResult` does not throw any exception, there is no `try—catch` block in the method `process`.

```java
public abstract class DocumentLoader
  extends SwingWorker<List<Document>, Void> { //1
    ... // field declarations
    private File[] files;
    ... // field declarations
    public void load(File... files) {
        this.files = files;
        execute();
    }

    protected List<Document> doInBackground() throws
     ↪  Exception {
        List<Document> result = new
         ↪  ArrayList<Document>();
        for (File file : files) {
            try {
                result.add(new Document(//2
                        file.getName(),
                        file.getAbsolutePath(),
                        ..., // LANGUAGE
                        ImageIO.read(file)));
            } catch (IOException ex) {
                ... // handle exception
            }
        }
        return result;
    }

    protected void done() {
        try {
            fetchResult(get());//3
        } catch (Exception ex) {
            log.warn("SwingWorker error");
        }
    }

  /* This method is implemented in the class
   * "Controller". The the implementation
   * consist of loading all images into
   * Juneiform
  */
    public abstract void fetchResult(
      List<Document> result);
}
```

Listing 3.1: DocumentLoader Class - Original Code

```java
public abstract class DocumentLoader
  extends SwingWorker<List<Document>, Document> { //1
    ... // field declarations
    private File[] files;
    ... // field declarations
    public void load(File... files) {
        this.files = files;
        execute();
    }

    protected List<Document> doInBackground() throws
     ↪  Exception {
        List<Document> result = new
         ↪  ArrayList<Document>();
        for (File file : files) {
            try {
                publish(new Document( //2
                        file.getName(),
                        file.getAbsolutePath(),
                        ..., // LANGUAGE
                        ImageIO.read(file)));
            } catch (IOException ex) {
                ... // handle exception
            }
        }
        return result;
    }

    protected void process(List<Document> chunks){
        fetchResult(chunks); //3
    }

  /* This method is implemented in the class
   * "Controller". The the implementation
   * consist of loading all images into
   * Juneiform
  */
    public abstract void fetchResult(
      List<Document> result);
}
```

Listing 3.2: DocumentLoader Class - Pre-Processing

After having manually refactored the class `DocumentLoader`, as showed in Listing 3.2, we used SwingWorker2Rx to refactor Juneiform. Listings 3.3 and 3.4 show the source code of `DocumentLoader` before and after refactoring. The differences in both code snippets are highlighted. Notice that the code snippet presented in Listings 3.3 is an abstraction of the one showed in Listings 3.2.

As showed in Figure 2.4b, the code contained in a `SwingWorker` can be executed whether on a background or on the UI thread. Everything that is called from `doInBackground` is executed in the background. The rest, methods `process` and `done`, is executed in the UI thread. In Juneiform, the background operation consists of reading the selected `Files` and converting them into `Documents` that can be manipulated by the application. The method `process` displays the `Documents` in the UI. Our refactoring separates this logic into two objects. The `Observable` (Listing 3.4: Lines 11-21) is responsible for the asynchronous operation (`doInBackground`) (Listing 3.4: Line 15), while the `Subscriber` (See Listing 3.4: Line 2) handles the synchronous ones (`process` and/or `done`). Our refactoring does not change the implementation contained in any of these three methods. Instead, these methods are wrapped in a `rx.Observable` or a `rx.Subscriber`, which are later use for introducing functional and reactive programming concepts into the existing application. The complete refactoring approach is explained in Section 3.2.1. For now, the relevant part of the code is the `publish` invocation (Listing 3.4: Line 18) because this is the one responsible for generating the items of the `Observable` that can be manipulated using RxJava functional programming constructs. The logic in charge of processing these items is in the method `process` (Listing 3.4: Line 23).

```java
public abstract class DocumentLoader
  extends SwingWorker<List<Document>, Document> {
    ... // field declarations
    private File[] files;
    ... // field declarations
    public void load(File... files) {
        this.files = files;
        execute();
    }

    protected List<Document> doInBackground()
      throws Exception {
        ... // for each file in this.files
        publish(new Document(...
        ...
    }

    protected void process(List<Document> chunks){
        fetchResult( chunks );
    }

  /* This method is implemented in the class
   * "Controller". The the implementation
   * consist of loading all images into
   * Juneiform
  */
    public abstract void fetchResult(
      List<Document> result);
}
```

Listing 3.3: DocumentLoader - Modified Code

```java
public abstract class DocumentLoader
  extends SWSubscriber<List<Document>, Document> {
    ... // field declarations
    private File[] files;
    ... // field declarations
    public void load(File... files) {
        this.files = files;
        executeObservable();
    }

    private rx.Observable<SWPackage<List<Document>, Void>>
      getRxObservable() {
        return rx.Observable.fromEmitter(
        new SWEmitter<List<Document>, Void>() {
        protected List<Document> doInBackground()
          throws Exception {
              ... // for each file in this.files
              publish(new Document(...
              ...
        }
      }, Emitter.BackpressureMode.BUFFER);
  }
  protected void process( List<Document> chunks ){
    fetchResult( chunks );
  }

  /* This method is implemented in the class
   * "Controller". The the implementation
   * consist of loading all images into
   * Juneiform
  */
    public abstract void fetchResult(
      List<Document> result);
}
```

Listing 3.4: DocumentLoader - Automatically Refactored Code

Functional and Reactive Programming in `DocumentLoader`

So far, we have shown how our tool can refactor a `SwingWorker` into a `rx.Observable` and a `rx.Subscriber`. This refactoring enables RxJava constructs that facilitate introducing functional and reactive programming concepts in Juneiform.

Listings 3.5 shows an example where we change the functionality of `DocumentLoader` in a functional fashion. In the new implementation, we add a filter to select only the images which format is jpg. Additionally, we change the display name of the image within Juneiform to uppercase and remove the extension of the file from the name. On the other hand, we implement a reactive model by using four `Subscribers`. The code snippet showed in Listing 3.5 works as follows:

- **Line 10**: we specify the thread in which the `Observable` operates. The scheduler `Schedulers.computation` corresponds to a background thread.
- **Line 11**: the `publish` invocation uses the type `List<Document>`. Since it is more comfortable to work directly with `Documents`, rather than a list of this type, we use `flapMap` to extract the `Document` objects from the lists received from `publish`.
- **Line 12**: at this point we have a stream of `Documents`. We filter these `Documents` to consider only jpg files.
- **Line 13 - 17**: we transform the items by changing their display name to uppercase and simultaneously removing the file extension from them. In functional programming using state should be avoid [31], therefore, we generate a new instance for each `Document` rather than changing the name directly in the current instance.
- **Line 18**: we specify that the operations that follow must be executed in the UI thread.
- **Line 21 - 24**: we register a `Subscriber` using lambda notation that updates the UI and prints the name of the file. Listing 3.6 shows the console output. One can see that the `Subscriber` 0 prints the following text to the console for each file: `Updating UI: <FileName>`
- **Line 25 - 27**: we register further `Subscribers` using lambda notation as well. Each of these `Subscribers` prints the name of the file to the console. Listing 3.6 shows the console output for `Subscribers` 1 to 3.

Notice that the `Subscribers` could perform more complex operations. We used the method `print` (Listing 3.5 Line: 34) to keep the example simple.

- **Line 29**: we connect the `ConnectableObservable` to start the emissions.

```
 1  public abstract class DocumentLoader
 2    extends SWSubscriber<List<Document>, Document> {
 3
 4      ... // field declarations
 5
 6    public void load( File... files ) {
 7      this.files = files;
 8      // setup subject
 9      ConnectableObservable<Document> connectableObservable = getRxObservable()
10          .subscribeOn( Schedulers.computation() )
11          .flatMap( swPackage -> Observable.from( swPackage.getChunks() ) )
12          .filter( doc -> doc.getName().contains( ".jpg" ) )
13          .map( doc -> new Document(
14              doc.getName().replace( ".jpg", "" ).toUpperCase(),
15              doc.getPath(),
16              doc.getLanguage(),
17              doc.getImage() ) )
18          .observeOn( SwingScheduler.getInstance() )
19          .publish();
20      // register subscribers
21      connectableObservable.subscribe( document -> {
22        print( "Subscriber  0 - Updating UI: ", document );
23        fetchResult( Arrays.asList( document ) );
24      } );
25      connectableObservable.subscribe( doc -> print( "Subscriber 1 - ", doc ) );
26      connectableObservable.subscribe( doc -> print( "Subscriber 2 - ", doc ) );
27      connectableObservable.subscribe( doc -> print( "Subscriber 3 - ", doc ) );
28      // connect subject with subscribers
29      connectObservable( connectableObservable );
30    }
31
32    ... // methods: getRxObservable, process and fetchResult
33
34    private void print(String id, Document document ) {
35      System.out.println( "[ " + Thread.currentThread().getName() + " ] " + id + document.getName() );
36    }
37  ...
38  }
```

Listing 3.5: DocumentLoader Class manually modified after SwingWorker2Rx-Refactoring

```
[ AWT-EventQueue-0 ] Subscriber 0 - Updating UI: RANDOMTEXT
[ AWT-EventQueue-0 ] Subscriber 1 - RANDOMTEXT
[ AWT-EventQueue-0 ] Subscriber 2 - RANDOMTEXT
[ AWT-EventQueue-0 ] Subscriber 3 - RANDOMTEXT
[ AWT-EventQueue-0 ] Subscriber 0 - Updating UI: IFILL-BRITO-BSC-THESIS
[ AWT-EventQueue-0 ] Subscriber 1 - IFILL-BRITO-BSC-THESIS
[ AWT-EventQueue-0 ] Subscriber 2 - IFILL-BRITO-BSC-THESIS
[ AWT-EventQueue-0 ] Subscriber 3 - IFILL-BRITO-BSC-THESIS
```

Listing 3.6: Juneiform - Console output after refactoring manually

3.1.2 Editor

Similarly, SwingWorker2Rx refactored the class `Editor`. Listing 3.7 shows the original source code. Here, there is a `SwingWorker` which is responsible for performing the optical character recognition in the background (Listings 3.7: Line 7) and updating the user interface by executing some code in the UI thread (Listings 3.7: Line 12). The specific implementations for these two actions are not relevant for this example. When the user clicks on the icon "Perform OCR" (Figure 3.1), the method containing the code snippet from Listings 3.7 gets executed. As soon as `execute` is reached (Listings 3.7: Line 15), the `SwingWorker` execution starts as described in Figure 2.4b.

```
1   ...
2   SwingWorker<String, Void> worker =
3     new SwingWorker<String, Void>() {
4
5       @Override
6       protected String doInBackground() {
7           ... // OCR Logic
8       }
9
10      @Override
11      protected void done() {
12      ... // Update UI
13      }
14  };
15  worker.execute();
16  ...
```

Listing 3.7: Editor - Original Code

```
1   ...
2   rx.Observable<SWPackage<String, Void>> rxObservable =
3     rx.Observable.fromEmitter(new SWEmitter<String, Void>(){
4     @Override
5     protected String doInBackground() throws Exception {
6       ... // OCR Logic
7     }
8   }, Emitter.BackpressureMode.BUFFER);
9
10  SWSubscriber<String, Void> rxObserver =
11    new SWSubscriber<String, Void>(rxObservable) {
12    @Override
13    protected void done() {
14      ... // Update UI
15    }
16  };
17  rxObserver.executeObservable();
18  ...
```

Listing 3.8: Editor - Automatically Refactored Code

Listing 3.8 shows the refactored code of the class `Editor`. The logic contained in the methods `doInBackground` (Listing 3.8: Line 6) and `done` (Listing 3.8: Line 14) did not change. However, the structure of the code did change. Now we have an `Observable` (Listing 3.8: Lines 2-8) containing the asynchronous operation (`doInBackground`) and a `Subscriber` (Listing 3.8: Lines 10-16) responsible for the synchronous one (`done`). In this case, the `Observable` starts the emissions as soon as the `executeObservable` invocation is reached (Listing 3.8: Line 17). The complete refactoring approach is explained in Section 3.2.1. For now, we only focus on the fact, that we have a `rx.Observable` (Listing 3.8: Lines 2-8) that allows us to add features in a functional fashion.

```
1   public class Editor implements ViewInteractor
2   {
3   ...
4       rx.Observable<SWPackage<String, Void>> rxObservable = Observable.fromEmitter( new SWEmitter<String, Void>(){...
5       }, Emitter.BackpressureMode.BUFFER )
6                   .doOnSubscribe( () -> Utils.showLoadingSpinner() )
7                   .doOnNext( swPackage -> copyToClipBoard( swPackage.getResult() ) )
8                   .doOnCompleted( () -> Utils.closeLoadingSpinner() );
9   ...
10  }
```

Listing 3.9: Editor Class manually modified after SwingWorker2Rx-Refactoring

After performing the refactoring, we modify the source code manually to add new features. In this case, we implement a `Utils` class for showing and closing a loading spinner. Additionally, we add a method to copy the result of the OCR to the clipboard. The implementation details of the `Util` class and the `copyToClipboard` method are not relevant for this example. The important aspect here is that we can add features to the `Observable` using a functional notation, as showed in Listing 3.9 (Lines 6-8). We declare the new features as follows:

- **Line 6:** we show a loading spinner when the observable is subscribed.
- **Line 7:** the last emission corresponds to the result. We copy the result to the clipboard.
- **Line 8:** we hide the loading spinner.

3.2 Refactoring Approach

As we mentioned before RxFactor and SwingWorker2Rx refactor similar asynchronous constructs. However, there are some differences regarding the refactoring approach. In this section, we explain the refactoring approach used in SwingWorker2Rx. After that, we present a summary of the refactoring approach proposed by RxFactor and explain why we decided to use a different one.

3.2.1 SwingWorker2Rx Approach

There were three important aspects that we considered while developing the refactoring approach:

1. Generate emissions on each `publish` invocation rather than only on the result, so that we can add several `Subscribers` to an `Observable` without modifying it. The `Subscribers` are then reactive to each emission of the `Observable`.
2. All methods available in the `SwingWorker` API must be available in the `Subscriber`
3. The `Observable` must stop sending items to the `Subscriber` if the second one has not finished processing the last emission.

In order to fulfill these requirements, we extended RxJava. In this section, we explain the Extension in a very high level. The details follow in Chapter 4.1.

The RxJava Extension is based on three classes: `SWEmitter`, `SWPackage` and `SWSubscriber`.

- The `SWEmitter` produces an emission on each `publish` and one at the end containing the async result.
- The `SWSubscriber` is responsible for processing the emissions. This class implements the methods available in the `SwingWorker` API and holds the state of the operation.
- The `SWPackage` is shared by the `SWEmitter` and the `SWSubscriber`. This structure allows us to send data from the `SWEmitter` to the `SWSubscriber`. This class uses a lock to avoid backpressure problems.

We base our refactoring on these three classes. Basically, we generate a `jar` file and add a dependency to this file in each project by updating the classpath. These classes can be modified post-refactoring, which is good for maintenance and improvements.

SWPackage (Emission)

The class that allows the communication between `SWEmitter` and `SWSubscriber` is called `SWPackage`. This structure contains dedicated fields for data chunks, which are sent on `publish`, the result, and a `ReentrantLock` that is used to stop the emissions from the `Observable` if the `Subscriber` is still processing the last one. See Section 4.1.3 for implementation details.

SWEmitter

The class responsible for generating the sequence is called `SWEmitter`. The `SWEmitter` starts by sending an initialization `SWPackage` to the `SWSubscriber`. Then the asynchronous operation starts. Inside of this operation, the method `publish` can be invoked multiple times. The invocation of `publish` generates an emission, which corresponds to a `SWPackage` that contains chunks of data. Finally, the `SWSubscriber` processes the emission. While it is being processed, the asynchronous operation responsible for generating emissions continues. However, if the `SWEmitter` reaches a `publish` invocation before the `SWSubscriber` finishes processing the last one, then the `SWEmitter` blocks, until the `SWSubscriber` is done. See Section 4.1.4 for implementation details.

SWSubscriber

The class that manages all operations available in the `SwingWorker` API is called `SWSubscriber`. This class is reactive to the emissions generated by the `Observable` and contains all information about the state of the operation. The `SwingWorker` API holds three methods that directly influence the async operation. These methods are `execute`, `run` and `cancel`. Since after refactoring, these methods are called from the `SWSubscriber`, it might not be clear what these operations actually do. Therefore, we rename these methods to make them more clear. Table 3.1 shows how we change the names after refactoring. All other method names remained the same. As we can see, the `SWSubscriber` must keep a reference to the `Observable` to be able to subscribe (execute or run) and cancel it. See Section 4.1.5 for implementation details.

SwingWorker Name	SWSubscriber Name
execute	executeObservable
run	runObservable
cancel	cancelObservable

Table 3.1: SwingWorker Method Names vs. SWSubscriber Method Names

3.2.2 RxFactor Approach

RxFactor is a tool that can refactor `AsyncTasks` into RxJava. Listings 3.10 and 3.11 show an example of how this tool works. First an `Observable` is created by using `Observable.fromCallable` (Listing 3.11: Line 14). The argument of this method is a `Callable` object, which defines the asynchronous operation. Therefore the routine from `doInBackground` (Listing 3.10: Lines 5-7) is written here.

The `publish` (Listing 3.10: Line 7) method cannot be copied inside of `fromCallable`, because this method is not defined in objects of type `Callable`. To be able to refactor `publish` invocations, a `Subscriber` is needed. This `Subscriber` is obtained from the method `getRxUpdateSubscriber` (Listing 3.11: Line 1), where `onNext` (Listing 3.11: Line 5) is implemented. As we can see, `Subscriber.onNext` and `onProgressUpdate` (Listing 3.10: Line 13) are equivalent.

Since `fromCallable` returns a single result, `doOnNext` (Listing 3.11: Line 27) is used to read the only and therefore final emission of the observable, which corresponds to the result from `doInBackground`. The `AsyncTask` processes this result in `onPostExecute` (Listing 3.10: Line 19), which is why we find the same program code in `doOnNext`.

```
 1  new AsyncTask<InputT, PublishT, ResultT>(){
 2    protected ResultT doInBackground(InputT... params){
 3      ...
 4      for (...; ...; ...){
 5        PublishT p = longRunningOperation(...);
 6        ...
 7        publish(p);
 8      }
 9      ...
10      return result;
11    }
12
13    protected onProgressUpdate(PublishT... values){
14      ...
15      process(values);
16      ...
17    }
18
19    protected void onPostExecute(ResultT result){
20      ...
21      finishTask(result);
22      ...
23    }
24  }.execute();
```

Listing 3.10: Before Refactoring with RxFactor

```
 1  private Subscriber<PublishT> getRxUpdateSubscriber() {
 2    return new Subscriber<PublishT[]>(){
 3      public void onCompleted(){ }
 4      public void onError(Throwable t) { }
 5      public void onNext(PublishT[] values){
 6        ...
 7        process(values);
 8        ...
 9      }
10    }
11  }
12  ...
13
14  Observable.fromCallable(new Callable<ResultT>() {
15    public ResultT call() throws Exception{
16      ...
17      for (...; ...; ...){
18        Publish p = longRunningOperation(...);
19        ...
20        getRxUpdateSubscriber().onNext(p);
21      }
22      ...
23      return result;
24    }
25  }).subscribeOn(Schedulers.computation())
26    .observeOn(AndroidSchedulers.mainThread())
27    .doOnNext(new Action1<ResultT>(){
28      public void call(ResultT result) {
29      ...
30      finishTask(result);
31      ...
32      }
33  }).subscribe();
```

Listing 3.11: After Refactoring with RxFactor

This refactoring approach is able to transform `AsyncTasks` into RxJava, as long as the methods `getStatus` and `isCancelled` are not invoked. RxFactor cannot refactor these methods because it does not have a mechanism to keep track of the state of the asynchronous operation. In order to refactor these methods, it is necessary to know whether the asynchronous operation is Pending, Started, Done or Canceled. Neither flags nor methods are available in RxJava for this purpose. A possible solution to this problem is to use a wrapper class. However, there is a drawback to this solution as well. The wrapper class would have to manage all method invocations of the `Subscriber` and the `Observable` in order to determine whether the status should be updated or not. If developers bypass the wrapper by for example subscribing an `Observable` directly, then the wrapper would not have any knowledge about his operation, and the status would remain unknown. Another alternative is to extend RxJava to replicate the workflow of `AsyncTasks`. This is the solution that we used for `SwingWorkers`. We explain how this solution looks like in Section 3.2.1.

According to the evaluation done in the research of RxFactor [34], the approach used by this tool is acceptable because the methods `getStatus` and `isCancelled` are not often used. However, that is not the case for `SwingWorkers`. Nine out of eighteen methods from the `SwingWorker` API require knowledge of the current state. In contrast to `AsyncTasks`, most `SwingWorker` instances use state related methods. This was the main reason for us to not use the approach of RxFactor.

There are also other drawbacks about the approach used in RxFactor. Since `Observable.fromCallable` is used, only one item is generated by the `Observable`. This item is the result of the asynchronous operation. If we connect the `Observable` to multiple `Subscribers`, then these observers will only get the final result. Information about the data that was originally processed in the `AsyncTask` through `onProgressUpdate` will still depend on the `Subscriber` that is directly specified in the `Observable` (Listing 3.11: Line 21).

The current implementation of RxFactor can lead to out of memory exceptions. Listing 3.11 shows that the `Observable` creates an instance of `Subscriber<PublishT>` every time that Line 20 is reached. This line is responsible for generating items and send them to the `Subscriber` so that they are processed in the UI thread. If the `Observable` generates too many items and the `Subscribers` require much time to process them, then there will be eventually too many instances of `Subscriber<PublishT>`. If the number of instances keeps increasing, then an out of memory exception will be thrown at some point.

Rewriting the approach to have a single instance of `Subscriber<PublishT>` would solve the out of memory exception issue, but then backpressure problems could occur. Although `Observable.fromCallable` is backpressure friendly, the way that this construct is combined with the `Subscriber` (Line 20) would make the implementation vulnerable to backpressure problems. These problems occur when the `Observable` produces items faster than the capacity of the `Subscriber` to consume them. RxJava takes care of this problem when the `Observable` and the `Subscriber` are linked by using the `subscribe` method. However, this is not the case in RxFactor. In Listing 3.11 one can see that the `Observable` has no knowledge about the `Subscriber`. If the `Callable` object used in `Observable.fromCallable` (Line 14) generates items too fast, then the buffer of the `Subscriber` will eventually be full. After that happens, the next items will be lost and therefore remain unprocessed. A possible solution to this problem is to use Java locks in the `Observable` and `Subscriber` in order to synchronize both objects and stop generating items in the `Observable`, in case that the `Subscriber` is busy and has no space in its buffer.

Notice that out of memory and backpressure problems only occur under very specific circumstances. However, we consider important to be aware of these issues.

The last disadvantage that we found about the approach used in RxFactor is that two objects from RxJava are needed to refactor the public method invocations from `AsyncTasks`. Table 3.2 shows these objects. The first column of this table contains the public methods of the `AsyncTask` API. The column “Rx Class” corresponds to the RxJava class that must be used to refactor the `AsyncTask` method, while the column “Rx Equivalent Method” shows the name of the method of the corresponding RxJava class. As we already explained, `getStatus` and `isCancelled` cannot be refactored without a wrapper class or similar structure. Basically, most methods can be refactored using `rx.Observable`. However, if `cancel` is invoked, then one needs to generate a `Subscription` from the `Observable` and call `unsubscribe`.

AsyncTask Method	Rx Class	Rx Equivalent Method
cancel(...)	rx.Subscription	unsubscribe()
execute()	rx.Observable	subscribe()
get(...)	rx.Observable	toBlocking().single()
getStatus()	–	–
isCancelled()	–	–

Table 3.2: Match between AsyncTask and Rx Methods

Listings 3.12 and 3.13 show an example where refactoring the `cancel` invocation automatically is not trivial. In this example the `AsyncTask` is executed in a class and canceled in a different class when an event is triggered. Here we can see that a naive implementation of replacing `AsyncTask` objects by `rx.Observables` does not work, because the Observable does not posses the unsubscribe method. The Subscription is obtained when the Observable is subscribed (Listing 3.13: Line 6). This Subscription must be passed to the class EventListener so that it can be canceled. However, if EventListener does not use the cancel method from the AsyncTask, then it is not necessary to pass the reference to the Subscription to it. The fact of having to work with two objects to refactor all methods from AsyncTasks into RxJava makes the static analysis and automated refactoring more complicated and error prone.

```java
// File 1: Class XYZ
...
  void startAsyncTask()
  {
    task = new AsyncTask<...>(){...};
    task.execute();
    button.addListener(new EventListener(task));
  }
...

// File 2: Class EventListener
...
  EventListener(AsyncTask<...> task)
  {
    this.task = task;
  }

  onEvent(Event e){
    this.task.cancel(true);
  }
...
```

Listing 3.12: Execute and Cancel of AsyncTask in different Classes before Refactoring

```java
// File 1: Class XYZ
...
  void startAsyncTask()
  {
    observable = Observable.fromCallable(...);
    /*Subscription subs = */ observable.subscribe();
    button.addListener(new EventListener(observable));
  }
...

// File 2: Class EventListener
...
  EventListener( Observable<...>  observable ) {
      this.observable = observable;
  }

  onEvent(Event e){
    // COMPILATION ERROR in next line
    // The subscription is needed instead
    this.observable.unsubscribe();
  }
...
```

Listing 3.13: Execute and Cancel of AsyncTask in different Classes after Refactoring

Due to these reasons, we decided to use a different approach to overcome these disadvantages.

3.3 Tool Development

We used the Plugin Development Environment (PDE) from Eclipse to implement a tool to perform automated refactoring of SwingWorkers into RxJava. Since the concepts that we explain in this thesis are not Eclipse specific but can be used for implementing similar tools in other IDEs or even standalone versions, we do not dedicate a section to explain how PDE works. Instead, we focus on the architecture of our tool. For details about PDE, we recommend the official documentation [5] and the thesis of Ramachandra Kamath Arbettu (RxFactor) [34].

The general requirements of the refactoring tool were:

1. Target: Java projects
2. Single Run: refactor one or multiple projects in a single run
3. Extendability: support extensions

To fulfill these requirements, we developed two main projects. The first projects is 2Rx and the second one is SwingWorker2Rx. 2Rx implements the common functionality to all extensions and defines the interface that must be implemented by them. The extensions contain the specific implementation for refactoring a particular construct. In this thesis, we present the implementation of SwingWorker2Rx. One can use this implementation as a reference to create further extensions such as ForLoop2Rx, WhileLoop2Rx, Runnable2Rx, etc. Additionally, we developed a third project for test-driven development.

Figure 3.3 shows a diagram representing the interaction between 2Rx and Extensions. 2Rx is in charge of iterating through all opened projects in the workspace. During this iteration, the Java projects are filtered. The next step is to iterate through all Java projects. Then, we update the classpath of the target project. Each extension must contain at least one jar file in its resources directory (rxjava-x.y.z.jar). We did not add this jar file to 2Rx to allow extensions decide which RxJava version should be used.

Next, 2Rx gets a collector instance from the extension. A collector is a class that contains all relevant AST nodes for performing the refactoring. After that, 2Rx iterates through the compilation units in the project and calls a method from the extension to process each of them. Processing a unit consist of a static analysis of the code and adding relevant AST nodes to the collector.

After processing all the units, 2Rx proceeds to refactor the code using the collector. To be able to refactor the original code, refactoring workers are necessary. These workers are implemented in the extension. 2Rx uses the class `Processor` to invoke the workers concurrently, execute the changes and update a results map. The results map is a map that matches the original compilation unit to the refactored code, saved as a string. This map is only required for testing purposes, where the changes are not written to the files, but read from the map to use them for the assertions.

2Rx also manages confirmation, error, progress and termination dialogs.

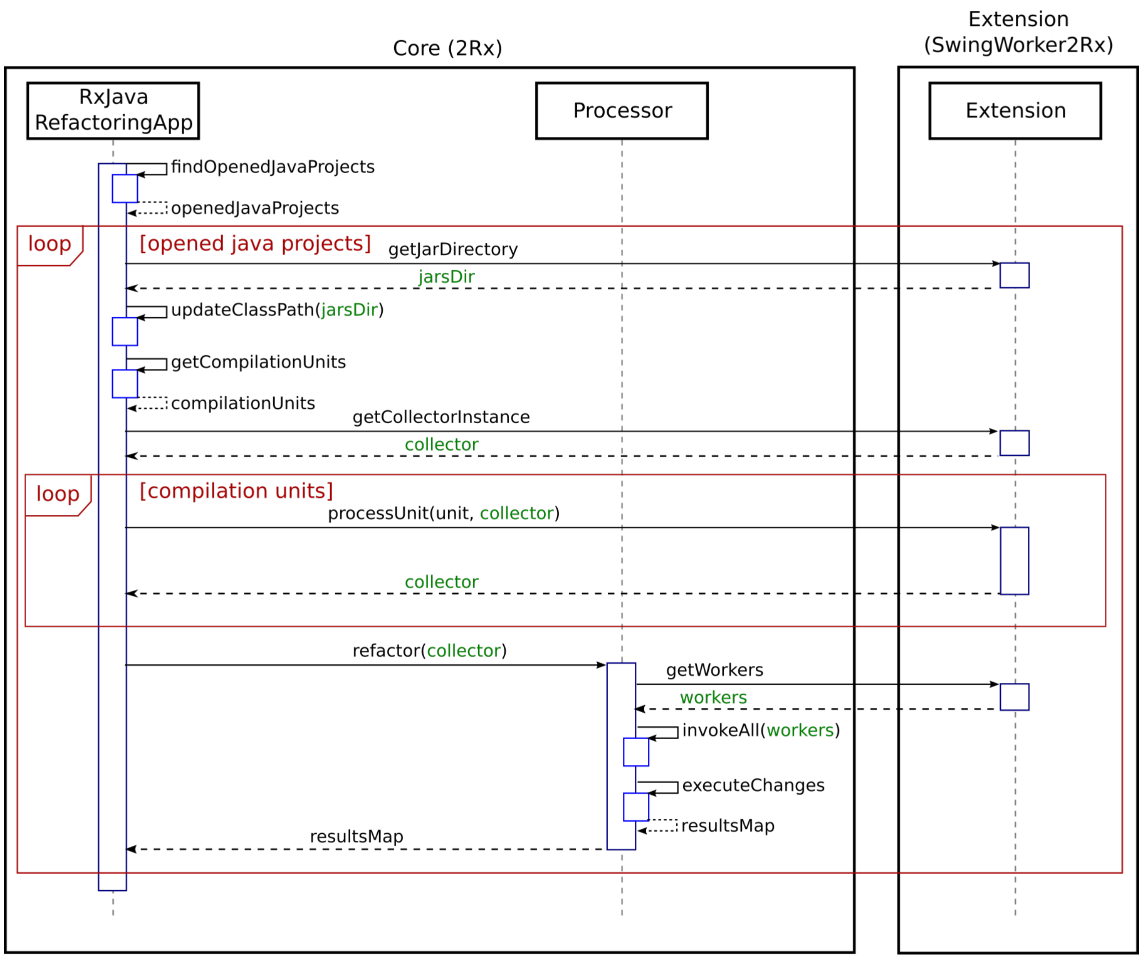
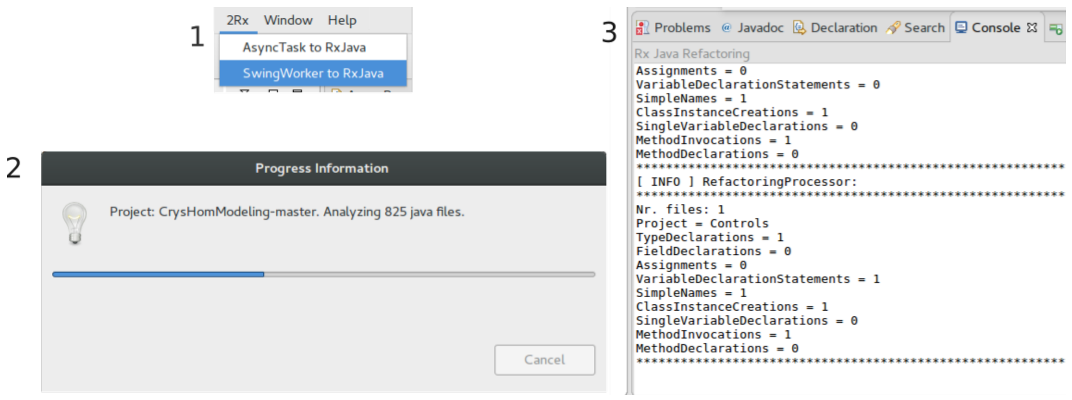


Figure 3.3: Plugin Design



Project: CrysHomModeling-master. Analyzing 825 java files.

```
Assignments = 0
VariableDeclarationStatements = 0
SimpleNames = 1
ClassInstanceCreations = 1
SingleVariableDeclarations = 0
MethodInvocations = 1
MethodDeclarations = 0
***********************************************************
[ INFO ] RefactoringProcessor:
***********************************************************
Nr. files: 1
Project = Controls
TypeDeclarations = 1
FieldDeclarations = 0
Assignments = 0
VariableDeclarationStatements = 1
SimpleNames = 1
ClassInstanceCreations = 1
SingleVariableDeclarations = 0
MethodInvocations = 1
MethodDeclarations = 0
***********************************************************
```

Figure 3.4: 2Rx and SwingWorker2Rx

Figure 3.4 shows what developers see when the plugin is installed. To run the tool, the developer must go to the menu “2Rx” and then select the target refactoring action. This action prompts a confirmation dialog. After clicking “Ok” the refactoring starts. While the projects are being refactored, a progress dialog is shown for each project. When the refactoring of a project has completed, the changes are shown in the console. Finally, an information dialog is shown saying that all projects have been refactored.

Currently 2Rx does not support either interrupting the operation nor “UNDO”. Developers should have a backup of their files before starting the operation.

3.3.1 2Rx Components

2Rx consists of six main components:

1. Action handler: it identifies which extension triggered the refactoring operation and starts the process.
2. Abstract collector: each client must implement a collector because the relevant information for the refactoring task depends on the specific case. In order to benefit from polymorphism, we defined an abstract collector in 2Rx. Collectors in clients must extend this class. Workers use the collector to perform the refactoring task. Since clients can only use one collector instance, all necessary information must be collected in the same object.
3. Abstract worker: workers are the objects that transform the code, based on the collected AST nodes. Each worker is responsible for a specific case. Workers are also implemented in clients by extending the class `AbstractWorker` of 2Rx. It usually makes sense to have multiple workers with clear responsibilities. Workers use a single unit writer to specify the transformations in a single Java file. The refactored compilation units must be registered in a multiple unit writer.
4. Processor: the processor is implemented in 2Rx and it is in charge of executing the workers and updating the target files by using the multiple unit writer.
5. Single unit writer: 2Rx contains a default implementation of this writer. This class can be extended by clients in order to support further refactoring functions. Since multiple workers could access the same single writer simultaneously, this class must be thread-safe.
6. Multiple unit writer: collects the compilation units that have changed. This class contains a method to update the target files. After this method has been executed, the refactoring operation has completed.

2Rx also provides its clients with utility classes and a code generator. The utility classes offer a variety of methods to query information about AST nodes, perform code transformations directly in the AST, send log messages to the console, validate source code strings, among others. The code generator corresponds to a class that contains methods to create several AST node types from source code. The supported AST nodes are single statements, blocks of statements, method declarations, type declarations and field declarations. Defining these AST nodes with JDT (Eclipse Java Development Tools) in the conventional way usually requires a lot of code because every node of the target syntax tree must be specified. Generating these AST nodes from text facilitates the implementation and maintenance.

3.3.2 Extension Setup

2Rx provides an extension point that allows adding clients to the plugin without modifying the existing code. In this extension point we specified that clients must implement the interface `RxJavaRefactoringExtension` (Listing 3.14).

```java
public interface RxJavaRefactoringExtension<CollectorType extends AbstractCollector> {
  CollectorType getASTNodesCollectorInstance(IProject project);
  void processUnit( ICompilationUnit unit, CollectorType collector );
  Set<AbstractRefactorWorker<CollectorType>> getRefactoringWorkers( CollectorType collector );
  String getId();
  default String getJarFilesPath() { return null;}
}
```

Listing 3.14: Extension Interface

Additionally, we created a template project facilitate the implementation of extensions. This project contains a `README` file that explains how to setup the template. The template contains a default package structure. The action handler in the extension is already implemented. This handler is in charge of forwarding the action event to 2Rx. 2Rx reads the command id from the event to identify which extension triggered operation. There is also a default implementation of the

`RxJavaRefactoringExtension` interface. This file contains placeholders in string values that must be replaced and TODOs in methods that must be implemented.

4 Implementation of the System

The refactoring approach used by SwingWorker2Rx is based on the implementation of three classes that complement RxJava. In this chapter, we explain how these classes work. Furthermore, we present some refactoring examples for different AST nodes (input vs. output). After that, we focus on the implementation of SwingWorker2Rx and finally we talk about the template projects available that can be used to implement and test extensions.

4.1 RxJava Extension

We developed an extension to support the `SwingWorker` API in RxJava. The purpose of this extension is to refactor `SwingWorkers` into RxJava considering the observer pattern. To accomplish that, we separate the target `SwingWorker` into a subject and an observer. The subject is responsible for executing the asynchronous operation (`doInBackground` method) while the observer is in charge of processing the intermediate results (`process` method) and the final result (`done` method) in the main thread. Intermediate results are generated on each `publish` invocation, while the final result corresponds to the return value of the method `doInBackground`. In general terms, the goal of the refactoring is to have a subject capable of generating the intermediate and final results and an observer to process them. Subjects and observers can also be called observables and subscribers respectively.

4.1.1 Class Diagram

Figure 4.1 shows an UML Class Diagram of the RxJava extension for `SwingWorkers`. The package `rx` contains classes from RxJava while the package `de.tudarmstadt.stg.rx.swingworker` contains the classes that we implemented. In the rest of this section, we explain the interaction between the classes showed in the UML Class Diagram. The RxJava classes are `Observable`, `Emitter`, `Observer`, `Subscriber`, `Action1`, and the ones implemented by us are `SWEmitter`, `SWPackage`, `SWSubscriber`, `RxSwingWorkerAPI`. It is recommended to look at the Class Diagram while reading to facilitate comprehension.

There are several ways of creating observables in RxJava. One of them is using the class `Observable` which offers different methods to create an `Observable` instance. One can use the method `Observable.fromEmitter` to implement algorithms that define when to generate items. In order to create an `Observable` using this method, one needs an `Action1` of type `Emitter`. The `Action1` interface has only a void method with one argument. The `Emitter` is an interface that extends `Observer`.

```java
Observable.fromEmitter(new Action1<Emitter<Integer>>() {
    public void call(Emitter<Integer> emitter) {
        try {
            for (int i = 1; i < 10; i++) {
                emitter.onNext(i);
            }
            emitter.onCompleted();
        } catch (Exception e) {
            emitter.onError(e);
        }
    }
}, Emitter.BackpressureMode.BUFFER ).subscribe(new Subscriber<Integer>() {
    public void onNext(Integer item) {
        System.out.println("DOUBLE VALUE = " + item.doubleValue());
    }
    public void onError(Throwable error) {
        System.err.println("ERROR = " + error.getMessage());
    }
    public void onCompleted() {
        System.out.println("RESULT = done");
    }
});
```

Listing 4.1: Emitter Example

```
DOUBLE VALUE = 1.0
DOUBLE VALUE = 2.0
DOUBLE VALUE = 3.0
DOUBLE VALUE = 4.0
DOUBLE VALUE = 5.0
DOUBLE VALUE = 6.0
DOUBLE VALUE = 7.0
DOUBLE VALUE = 8.0
DOUBLE VALUE = 9.0
RESULT = done
```

Listing 4.2: Emitter Example Output

Listings 4.1 shows how to create an `Observable` using the method `fromEmitter` and how the `Emitter` and the `Subscriber` interact with each other. The algorithm responsible for generating the items is implemented in the `call` method of `Action1` (Listings 4.1: Line 1). The item generation is done using the class `Emitter` (Listing 4.1: Lines 2 to 11). On the other hand, the `Subscriber` contains the logic to process the generated items (Listings 4.1: Lines 12 to 21).

The algorithm consists of a `for`-loop (Listings 4.1: Line 4). On each iteration the method `Emitter.onNext(i)` is called (Listings 4.1: Line 5). This method generates an item with the integer value `i` which is processed by the `Subscriber` (Listings 4.1: Line 13). After nine iterations, the method `Emitter.onCompleted` is invoked (Listings 4.1: Line 7) which triggers the `onCompleted` method from the `Subscriber` (Listings 4.1: Line 19). If there are any exceptions during the execution, then the `Emitter.onError(e)` method is invoked (Listings 4.1: Line 9). This error is handled in the `Subscriber` as well (Listings 4.1: Line 16). Listings 4.2 shows the console output produced by Listings 4.1. Summarizing, the `Action1` implements an algorithm that specifies when items should be generated (`onNext`), when the operation has completed (`onCompleted`) and when to handle an exception (`onError`). This signals are received and processed by the `Subscriber`.

Due to the following reasons, the example showed in Listing 4.1 cannot be used as it is to replicate the behavior of `SwingWorkers`:

1. `SwingWorkers` allow using a data type for the final result and a different one for intermediate results. `Emitters`, on the contrary, only allow using a single data type.
2. `SwingWorkers` have state flags that are not available neither in `Subscribers` nor in `Observables`. `Subscribers` have only one state related method called `isUnsubscribed`. Other state queries available in `SwingWorkers` such as `getState`, which return `PENDING`, `STARTED` or `DONE`, or queries such as `isCancelled`, `isDone`, `getProgress` have no equivalent in `Subscribers`. The following methods: `addPropertyChangeListener`, `removePropertyChangeListener`, `firePropertyChange` and `getPropertyChange`, are also not available in `Subscribers`. These last four methods require the object `PropertyChangeSupport` in order to be implemented.

In order to tackle the first problem, we implemented `SWPackage`, a generic data structure that accepts two parameters. The first parameter `R` is for the result of the asynchronous operation and the second parameter `P` is for the intermediate results. Listing 4.3 shows how one can create an `Observable` based on an `Emitter` of type `SWPackage`. In this example, we set the parameters `R` and `P` of the `SWPackage` to be `Integer` and `String` respectively. The code snippet showed in Listing 4.3 works as follows:

- **Line 1**: We create the `Observable` from an `Emitter` of type `SWPackage<Integer,String>>`.
- **Lines 4-17**: We define the algorithm to generate the items. The algorithm consists of iterating from zero to ten and sending a `String` to the `Subscriber` on each iteration. The indexes `i` used for the iteration are accumulated in the variable `accum` and returned at the end of the operation as `Integer`.
  - **Line 7**: We create the `SWPackage` instance that is used to generate the next item in the current iteration.
  - **Line 8**: We create the `String` value for the next item based on the iteration index `i`.
  - **Line 9**: We add the intermediate result `text` to the `SWPackage` by using the method `setChunks`. This method accepts multiple arguments of type `String` and returns a `SWPackage`. After that, we invoke the `onNext` method of the `emitter` using the `SWPackage` as argument to finally generate the item.
  - **Line 10**: We update the `accum` variable.
  - **Line 12**: Once the iterations has completed, we create another `SWPackage` to generate the item that contains the final result.
  - **Line 13**: Similar to Line 9, we put the result of the operation into the `SWPackage`. This time, we use `setResult` which accepts a single argument of type `Integer` and returns a `SWPackage` as well. The final result item is then generated by calling `onNext` on the `emitter` using the `SWPackage`.
  - **Lines 14-17**: Same as in Listing 4.1
- **Lines 20-34**: We implemented the methods of the `Subscriber` responsible for handling the events `onNext`, `onCompleted`, `onError`.
  - **Lines 23**: We retrieve the intermediate results from the `item` using `getChunks`.
  - **Lines 24-27**: If there are intermediate results, we take the first one and print it to the console. Notice that this value corresponds to a `String`.
  - **Line 28**: We retrieve the final result from the `item` using `getResult`.
  - **Lines 29-31**: If there is a result, then we print it to the console. Notice that the data type of this value is `Integer`.
  - **Line 33**: Commented out. Same implementation as in Listing 4.1

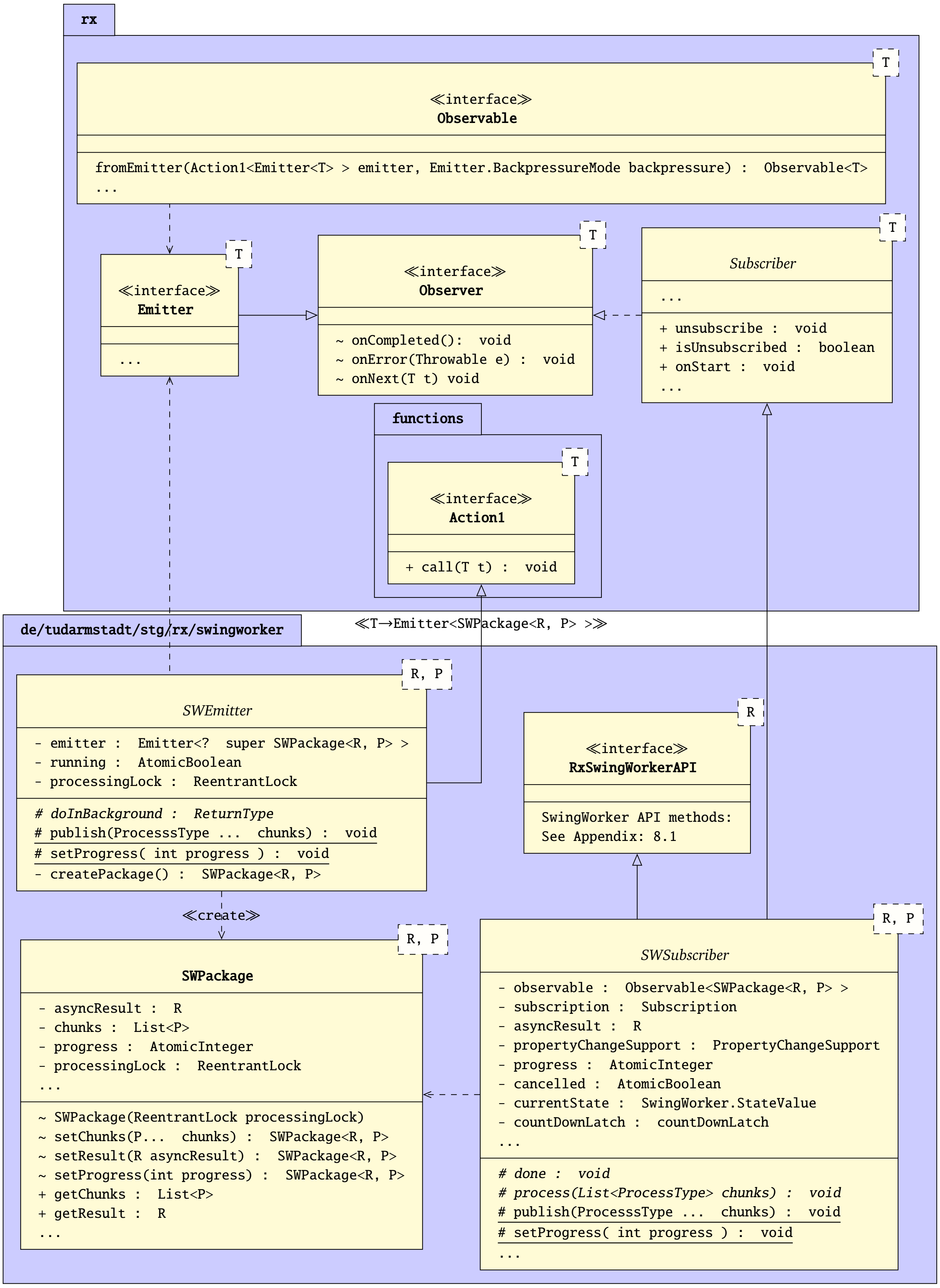

rx
T
≪interface≫
Observable
fromEmitter(Action1<Emitter<T> > emitter, Emitter.BackpressureMode backpressure) : Observable<T>
...
T
≪interface≫
Emitter
...
T
≪interface≫
Observer
~ onCompleted(): void
~ onError(Throwable e) : void
~ onNext(T t) void
T
Subscriber
...
+ unsubscribe : void
+ isUnsubscribed : boolean
+ onStart : void
...
functions
T
≪interface≫
Action1
+ call(T t) : void
≪T→Emitter<SWPackage<R, P> >≫
de/tudarmstadt/stg/rx/swingworker
R, P
SWEmitter
- emitter : Emitter<? super SWPackage<R, P> >
- running : AtomicBoolean
- processingLock : ReentrantLock
# doInBackground : ReturnType
# publish(ProcesssType ... chunks) : void
# setProgress(int progress) : void
- createPackage() : SWPackage<R, P>
R
≪interface≫
RxSwingWorkerAPI
SwingWorker API methods:
See Appendix: 8.1
≪create≫
R, P
SWPackage
- asyncResult : R
- chunks : List<P>
- progress : AtomicInteger
- processingLock : ReentrantLock
...
~ SWPackage(ReentrantLock processingLock)
~ setChunks(P... chunks) : SWPackage<R, P>
~ setResult(R asyncResult) : SWPackage<R, P>
~ setProgress(int progress) : SWPackage<R, P>
+ getChunks : List<P>
+ getResult : R
...
R, P
SWSubscriber
- observable : Observable<SWPackage<R, P> >
- subscription : Subscription
- asyncResult : R
- propertyChangeSupport : PropertyChangeSupport
- progress : AtomicInteger
- cancelled : AtomicBoolean
- currentState : SwingWorker.StateValue
- countDownLatch : countDownLatch
...
# done : void
# process(List<ProcessType> chunks) : void
# publish(ProcesssType ... chunks) : void
# setProgress(int progress) : void
...

Figure 4.1: UML: RxJava SwingWorker Extension

```java
Observable.fromEmitter(new Action1<Emitter<SWPackage<Integer,String>>>() {
    @Override
    public void call(Emitter<SWPackage<Integer,String>> emitter) {
        try {
            int accum = 0;
            for (int i = 0; i < 10; i++) {
                SWPackage<Integer, String> pkg = new SWPackage<>();
                String text = new Character((char)(65+i)).toString();
                emitter.onNext(pkg.setChunks(text));
                accum += i;
            }
            SWPackage<Integer, String> pkg = new SWPackage<>();
            emitter.onNext(pkg.setResult(accum));
            emitter.onCompleted();
        } catch (Exception e) {
            emitter.onError(e);
        }
    }
}, Emitter.BackpressureMode.BUFFER )
.subscribe(new Subscriber<SWPackage<Integer,String>>() {
    @Override
    public void onNext(SWPackage<Integer,String> item) {
        List<String> intermediateResults = item.getChunks();
        if (!intermediateResults.isEmpty()) {
            String intermediateResult = intermediateResults.get(0);
            System.out.println("INTERMEDIATE = " + intermediateResult);
        }
        Integer result = item.getResult();
        if (result != null) {
            System.out.println("FINAL RESULT = " + result);
        }
    }
  ... // Implementation of onError and onCompleted
});
```

Listing 4.3: Emitter Example (Two Data Types)

```
INTERMEDIATE = A
INTERMEDIATE = B
INTERMEDIATE = C
INTERMEDIATE = D
INTERMEDIATE = E
INTERMEDIATE = F
INTERMEDIATE = G
INTERMEDIATE = H
INTERMEDIATE = I
INTERMEDIATE = J
FINAL RESULT = 45
RESULT = done
```

Listing 4.4: Emitter Example Output (Two Data Types)

In order to tackle the second problem, we extended the class `Subscriber` from RxJava. The extension of this class, `SWSubscriber`, consists of adding fields that allow us to implement the missing methods. One of these fields is `currentState`. The value of this field is returned when the method `SWSubscriber.getState()` is invoked. Before the asynchronous operation starts, the value of the field is `PENDING`. We overrode the method `onStart` so that the field `currentState` is changed to `STARTED` as soon as the method is invoked (Listing 4.8). The method `onStart` is triggered when the `SWSubscribed` is subscribed. Additionally, we also overrode the method `onCompleted` to make this field change to `DONE` (Listing 4.10). Similarly we used the rest of the fields to replicate the state related behavior of `SwingWorkers` using RxJava.

The classes `SWPackage` and `SWSubscriber` are enough to refactor `SwingWorkers` into RxJava. With `SWPackage` we can define different data types for intermediate and final results. `SWSubscriber` implements all methods of the `SwingWorker` API. The refactoring can be done manually in 15 steps:

1. Copy the `doInBackground` implementation of the `SwingWorker` into the `call` method of `Action1`.
2. Identify all `publish` invocations
3. Create a `SWPackage` for each `publish` invocation
4. Use the method `setChunks` from `SWPackage` and use the arguments specified in `publish`
5. Generate the item by using `Emitter.onNext(SWPackage pkg)`
6. Identify all `setProgress` invocations
7. Create a `SWPackage` for each `setProgress` invocation
8. Use the method `setProgress` from `SWPackage` and use the same progress as argument
9. Repeat step 5
10. Identify the `return` statements
11. Create a `SWPackage` for each `return` statement
12. Use the method `setResult` from `SWPackage` and use the `return` value as argument
13. Repeat step 5

14. At the end of the `call` method, add `Emitter.onCompleted` to signal that the asynchronous operation has completed.
15. Finally, surround everything with a `try—catch` block and use `Emitter.onError` in the catch clause to trigger the `onError` method of the `SWSubscriber` in case that an exception is thrown

One can see that this refactoring is not trivial. Even though the refactoring can be performed automatically, developers might have trouble understanding the output code if they are not familiar with RxJava. Notice that we only need to consider two methods (`publish` and `setProgress`) and the return value for the refactoring. The rest of the algorithm remains the same, regardless of the implementation of `doInBackground`. In order to hide this abstraction from developers and make the refactoring easier and more comprehensible, we implemented `SWEmitter` and `SWSubscriber` using the template method pattern.

In the class `SWEmitter`, we specify that the method `doInBackground` must be implemented by the subclass. The methods `publish` and `setProgress` of the `SWEmitter` take care of the steps 2 to 9 that were mentioned above. Similarly, in the class `SWSubscriber`, we specify that the methods `process` and `done` must be implemented by the subclass. By doing this, developers need not to reason about `Emitters`, `onNext`, `onCompleted`, `onError` and `SWPackage`. See Sections 4.1.2, 4.1.3, 4.1.4 and 4.1.5 for more details about the implementation of these three classes.

4.1.2 Sequence Diagram

Figure 4.2 shows the interactions between `SWEmitter`, `SWPackage` and `SWSubscriber`. In general, the `SWEmitter` produces items of type `SWPackage`. Then, RxJava internally sends these items to the `SWSubscriber`. In the diagram, we use the term “Emissions Pool” to represent this interaction. For the explanation, we can assume that the `SWEmitter` places the items into a pool and the `SWSubcriber` takes them from there.

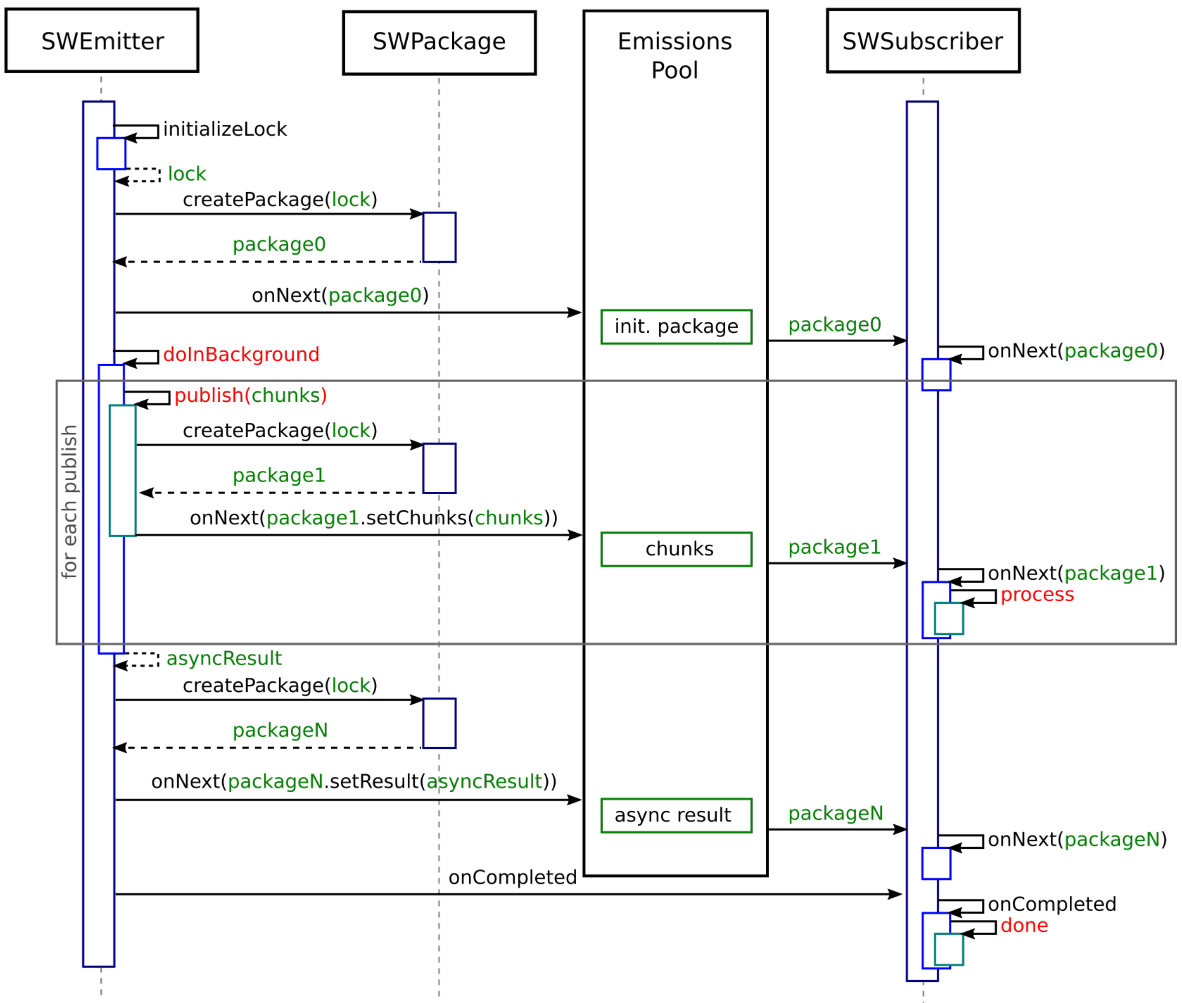


Figure 4.2: RxJava Extension for SwingWorkers

The class SWPackage has two functions. The first one is to support items of different data types and the second one is manage the synchronization between SWEmitter and SWSubscriber in order to avoid backpressure problems.

The asynchronous operation starts by initializing a lock in the SWEmitter. This lock is used for all SWPackages. Then an initialization SWPackage is created and sent to the emissions pool. After that, the asynchronous operation starts (doInBackground). While this operation is running, the methods publish or setProgress can be invoked multiple times. This invocations also generate SWPackages that are pushed into the emissions pool, where they are taken and processed from the SWSubscriber. If the emission was generated by a publish invocation then the process method of the SWSubscriber will be invoked. When the asynchronous operations finishes, a final SWPackage containing the result is sent to the emissions pool again. Finally, the onCompleted method of the SWSubscriber is invoked. This method triggers the piece of code contained in the done method of the SWSubscriber.

While the SWPackages are being processed, the SWSubscriber updates the state of the operation (PENDING, STARTED, DONE). The SWSubscriber contains all methods available in the SwingWorker API including the state relevant ones.

4.1.3 Package Data Structure

The main purpose of the SWPackage is to manage synchronization and encapsulate different types of data. Listings 4.5 how these fields are defined in the generic class. The progress is defined as an AtomicInteger to make the variable thread-safe. The other two types can only be accessed through getters and setters that use locks to guarantee thread-safe access. The processingLock is shared among all SWPackages. This object comes from the SWEmitter and is the one that allows us to avoid backpressure problems.

```java
public final class SWPackage<ReturnType, ProcessType> {
  ...
  private ReturnType asyncResult;
  private List<ProcessType> chunks;
  private AtomicInteger progress;
  private ReentrantLock processingLock;
  ...
  SWPackage(ReentrantLock processingLock ) {
    this.processingLock = processingLock;
    ...
  }
  ...
```

Listing 4.5: SWPackage Contents

4.1.4 Emitter

The SWEmitter generates the SWPackages that are going to be processed by the SWSubscriber. Listings 4.6 shows the implementation of the standard emissions in a SWEmitter. We call them standards because they are always produced. There are two emissions of this kind, initialization (Listing 4.6: Line 3) and result (Listing 4.6: Line 5). Notice that setResult returns an SWPackage as well, otherwise, it would not be possible to use this call as a parameter for onNext (Listing 4.6: Line 4).

```java
1  public final void call( Emitter<SWPackage<ReturnType, ProcessType>> emitter ) {
2    ...
3    this.emitter.onNext( createPackage() ); // init
4    ReturnType asyncResult = doInBackground();
5    this.emitter.onNext( createPackage().setResult( asyncResult ) ); // result
6    this.emitter.onCompleted();
7    ...
8  }
```

Listing 4.6: SWEmitter Standard Emissions

The `doInBackground` method is abstract and must therefore be implemented in the subclass. This implementation can invoke the `publish` (Listing 4.7: Line 11) and/or the `setProgress` (Listing 4.7: Line 15) method multiple times. If these methods are invoked, then they are forwarded to the `SWSubscriber` using a `SWPackage` as well. Listings 4.7 (Lines 12, 16) shows how the forwarding was implemented. Notice that `setChunks` and `setProgress` return an `SWPackage` as well.

```java
1  public abstract class SWEmitter<ReturnType, ProcessType>
2    implements Action1<Emitter<SWPackage<ReturnType, ProcessType>>>
3    ...
4    @Override
5    public final void call( Emitter<SWPackage<ReturnType, ProcessType>> emitter ) {
6      ...
7      // Template Method Pattern:
8      // The doInBackground method (implemented in the subclass) is invoked here
9    }
10
11   protected void publish( ProcessType... chunks ) {
12     this.emitter.onNext( createPackage().setChunks( chunks ) );
13   }
14   ...
15   protected void setProgress( int progress ) {
16     this.emitter.onNext( createPackage().setProgress( progress ) );
17   }
18   ...
```

Listing 4.7: SWEmitter Dynamic Emissions

4.1.5 Subscriber

The `SWSubscriber` is the class responsible for managing state related operations that are available in the `SwingWorker`. In order to manage stateful operations, it is necessary to have private fields that can be updated on specific events. For example: The state of the `SwingWorker` depends on the field `currentState`. At the beginning of the asynchronous operation, this field has the value `PENDING`. When the operation starts, the method `onStart` from the `SWSubscriber` is invoked, making the status change to `STARTED` (Listing 4.8: Line 4). We use the `countDownLatch` to be able to wait for the asynchronous result when the `get` method is invoked(Listing 4.8: Line 3). This is necessary because the `SwingWorker` API specifies that `get` is blocking.

```java
1  public final void onStart() {
2    initialize(); // progress = 0; cancelled = false;
3    this.countDownLatch = new CountDownLatch( 1 );
4    setState( SwingWorker.StateValue.STARTED );
5  }
```

Listing 4.8: SWSubscriber onStart

Listing 4.9 shows how the `SWSubscriber` processes the `SWPackage`. First, it updates the async result. If no result is present, then the value remains `null` (Listing 4.9 Line 3). Then it checks whether a progress value was sent and if so, it updates the `progress` field (Listing 4.9 Lines 4-5). Finally, if the `publish` method in the `SWEmitter` was invoked, the data is taken and processed in the `process` method (Listing 4.9 Lines 7-8).

Listings 4.10 shows the termination of emissions. At the end, the `onCompleted` method of the `SWSubscriber` is invoked. Here the `countDownLatch` is decrease by one to report that the asynchronous operation has finished (Listings 4.10: Line 2). Then the `done` method of the `SWSubscriber` is invoked (Listings 4.10: Line 3) and finally the state is set to `DONE` (Listings 4.10: Line 4).

```java
public final void onNext(SWPackage<ResultType,ProcessType> swPackage){
  ...
  asyncResult = swPackage.getResult();
  if ( swPackage.isProgressValueAvailable() )
    setProgress(swPackage.getProgressAndReset());

  if ( !swPackage.getChunks().isEmpty() )
    process(swPackage.getChunks());
}
```

Listing 4.9: SWSubscriber onNext

```java
public final void onCompleted() {
  countDownLatch.countDown();
  done();
  setState(SwingWorker.StateValue.DONE);
}
```

Listing 4.10: SWSubscriber onCompleted

The `SWSubscriber` implements all methods available in the `SwingWorker` API. Explaining how they are implemented would require copying the source code of the whole class in this section. Therefore we have limited ourselves to explain the RxJava related methods.

4.2 Refactoring Approach

In this thesis, we propose a refactoring approach that allows transforming `SwingWorkers` to RxJava by modifying a few lines of the source code. This is possible because the `SwingWorker` workflow is imitated through the interaction between `SWEmitter`, `SWPackage` and `SWSubscriber`.

In the following subsections, we explain how we refactore different AST nodes. These code snippets can be compared to the example showed at the beginning of Chapter 3, to understand how the transformations in the Juneiform application were performed (Listings 3.3, 3.4).

On the left side we present the original source code and on the right side the refactored one.

4.2.1 Assignments

The usage of the AST node `Assignment` involves working with variables. We observed that often the variable names contain the substrings "swingWorker" or "worker". Since we are refactoring this construct, we replace these substrings by "rxObserver".

```java
anotherWorker = swingWorker;
```

Listing 4.11: Assignment before Refactoring

```java
anotherRxObserver = rxObserver;
```

Listing 4.12: Assignment after Refactoring

There are also cases where a variable can be directly assigned to a class instance creation. The refactoring of class instance creations is shown in subsection 4.2.3.

4.2.2 Variable Declaration Statements

Similar to `Assignments`, `SingleVariableDeclarations` involve variable names that must be adjusted. Additionally, the data type `SwingWorker` must be changed to `SWSubscriber` (Listings 4.13 and 4.14).

```java
SwingWorker anotherWorker = swingWorker;
```

Listing 4.13: Variable Declaration Statement before Refactoring

```java
SWSubscriber anotherRxObserver = rxObserver;
```

Listing 4.14: Variable Declaration Statement after Refactoring

It is also possible to have `ClassInstanceCreations` in `VariableDeclarationStatements` AST nodes. See subsection 4.2.3 for more details about the refactoring of `ClassInstanceCretion` nodes.

4.2.3 Class Instance Creations

ClassInstanceCreations are the AST nodes that actually contain the implementation of the SwingWorker. During the refactoring, we separate this implementation into two objects. The Observable and the SWSubscriber. The Observable contains the logic for the asynchronous operation, while the SWSubscriber has the logic responsible for processing both types of results, intermediate and final. The SWEmitter and the SWSubscriber were designed to support the protected methods of the SwingWorker API. Therefore the blocks doInBackground, process and done do not need to be modified (Listings 4.15 and 4.16).

```java
SwingWorker<String, Integer> swingWorker
  = new SwingWorker<String, Integer>() {
  @Override
  protected String doInBackground() throws Exception {
    ...
  }
  protected void process(List<Integer> chunks){...}
  protected void done() {...}
};
```

Listing 4.15: Class Instance Creation before Refactoring

```java
rx.Observable<SWPackage<String, Integer>> rxObservable
  = rx.Observable
    .fromEmitter(new SWEmitter<String, Integer>() {
      @Override
      protected String doInBackground()
      throws Exception {
        ...
      }
}, Emitter.BackpressureMode.BUFFER );

SWSubscriber<String, Integer> rxObserver
  = new SWSubscriber<String, Integer>(rxObservable) {
  protected void process(List<Integer> chunks){...}
  protected void done() {...}
};
```

Listing 4.16: Class Instance Creation after Refactoring

4.2.4 Field Declarations

FieldDeclarations are very similar to VariableDeclarationStatements. Normally, it is only necessary to adjust the variable name and change the data type from SwingWorker to SWSubscriber. However, we do have an important special case for FieldDeclarations. Our approach creates two objects out of a SwingWorker and the constructor of the SWSubscriber requires an Observable. To make the code more readable, we decided to create an inner class that extends SWSubscriber and generate the Observable there (Listings 4.17 and 4.18).

```java
private SwingWorker<String, Integer> anotherWorker
  = new SwingWorker<String, Integer>(){
  @Override
  protected String doInBackground() throws Exception
  {
    ...
  }
  protected void process( List<Integer> chunks ) {...}
  protected void done() {...}
};
```

Listing 4.17: Field Declaration before Refactoring

```java
private SWSubscriber<String,Integer> anotherRxObserver
  = new RxObserver();

class RxObserver extends SWSubscriber<String,Integer>{
  RxObserver(){ setObservable(getRxObservable()); }

  private rx.Observable<SWPackage<String, Integer>>
    getRxObservable() {
    return rx.Observable.fromEmitter(
      new SWEmitter<String, Integer>() {
        @Override
        protected String doInBackground()
        throws Exception {
          ...
        }
    }, Emitter.BackpressureMode.BUFFER );
  }
  protected void process( List<Integer> chunks ) {...}
  protected void done() {...}
}
```

Listing 4.18: Field Declaration after Refactoring

Another alternative for refactoring this kind of `FieldDeclarations` is to write the `Observable` directly in the constructor (See Listing 4.19), but in our opinion, this code is harder to read, specially if the method `doInBackground` has many lines.

```java
private SWSubscriber<String, Integer> anotherRxObserver = new SWSubscriber<String, Integer>(
      rx.Observable.fromEmitter(
        new SWEmitter<String, Integer>() {
          @Override
          protected String doInBackground()
          throws Exception {...}
      }, Emitter.BackpressureMode.BUFFER )
  ) {
  protected void process( List<Integer> chunks ) {...}
  protected void done() {...}
};
```

Listing 4.19: Alternative Refactoring for Field Declaration

`SwingWorkers` can also contain custom fields and/or methods. In these cases, we also use an inner class such as `RxObserver`, showed in Listing 4.18, that contains all of the custom fields and methods. By doing that we guarantee, that the pieces of code contained in the `Observable` and the `SWSubscriber` still have access to those elements.

4.2.5 Method Declaration

We also modify `MethodDeclarations` to adjust the return value from `SwingWorker` to `SWSubscriber` (Listings 4.20 and 4.21).

```java
private SwingWorker<String, Integer>
  getSwingWorker(){ ... }
```

Listing 4.20: Method Declaration before Refactoring

```java
private SWSubscriber<String, Integer>
  getSwingWorker(){ ... }
```

Listing 4.21: Method Declaration after Refactoring

4.2.6 Method Invocations

Only three methods out of eighteen were renamed in the `SWSubscriber`. The refactoring of `MethodInvocation` nodes also involves checking the name of the invokers and adjusting them if necessary. The substrings “swingWorker” and “worker” are replaced by “rxObserver”. Listings 4.22 and 4.23 shows the refactoring of the only methods that were renamed.

```java
swingWorker.cancel( true );
swingWorker.execute();
swingWorker.run();
```

Listing 4.22: Method Invocations before Refactoring

```java
rxObserver.cancelObservable( true );
rxObserver.executeObservable();
rxObserver.runObservable();
```

Listing 4.23: Method Invocations after Refactoring

4.2.7 Simple Names

We refactor `SimpleNames` to adjust the argument name of `SwingWorker` types in invocations (Listings 4.24 and 4.25).

```java
doSomething( swingWorker );
```

Listing 4.24: Simple Name before Refactoring

```java
doSomething( rxObserver );
```

Listing 4.25: Simple Name after Refactoring

4.2.8 Single Variable Declarations

SwingWorkers can also be parameters of methods. The variables defined in a MethodDeclaration are called SingleVariableDeclaration. We use this AST node to refactor refactor these SwingWorkers into SWSubscribers (Listings 4.26 and 4.27).

```java
void doSomething ( SwingWorker swingWorker ) {...}
```

Listing 4.26: Single Variable Declaration before Refactoring

```java
void doSomething ( SWSubscriber rxObserver ) {...}
```

Listing 4.27: Single Variable Declaration after Refactoring

4.2.9 Type Declarations

In order to refactor TypeDeclaration nodes, it is necessary to change the superclass of the target node from SwingWorker to SWSubscriber. Furthermore, we need to set the Observable in the constructor. Listings 4.28 and 4.29 illustrate how this is done. Notice that the method getRxObservable is the same that we use for the special case of FieldDeclarations (Listing 4.18).

```java
public class ClassA
  extends SwingWorker<String, Integer> {
  ...
  public ClassA(...) {
    ...
  }
}
```

Listing 4.28: Type Declaration before Refactoring

```java
public class ClassA
  extends SWSubscriber<String, Integer> {
  ...
  public ClassA(...) {
    ...
    setObservable( getRxObservable() );
  }

  private rx.Observable<SWPackage<String, Integer>>
    getRxObservable() {
      return rx.Observable.fromEmitter( ... );
  }
}
```

Listing 4.29: Type Declaration after Refactoring

4.3 2Rx and SwingWorker2Rx

2Rx and SwingWorker2Rx work together to perform the automated refactoring of SwingWorkers to RxJava. The first step to implement an extension of 2Rx is to define its id, name and the location of its resources (i.e. required jar files). After that, the object responsible for collecting all relevant AST nodes for the refactoring must be implemented. To refactor SwingWorkers it is necessary to collect the following AST nodes: TypeDeclaration, FieldDeclarations, Assignment, VariableDeclarationStatement, SimpleName, ClassIntanceCreation, SingleVariableDeclaration, MethodInvocation and MethodDeclaration.

4.3.1 Collectors

The collector has a Map for each of these fields. The Map holds compilation units and a list of the corresponding node. Listing 4.30 shows an example of the fields that we use for the collector of SwingWorker2Rx.

```java
public class RxCollector extends AbstractCollector {
  ...
  private final Map<ICompilationUnit, List<TypeDeclaration>> typeDeclMap;
  private final Map<ICompilationUnit, List<FieldDeclaration>> fieldDeclMap;
  // etc...
}
```

Listing 4.30: SwingWorker2Rx Collector

4.3.2 Processing

At the beginning the collector is empty. The collector is updated on each `processUnit` invocation (Figure 3.3). Since the collector only holds the nodes, another object is needed to analyze the current unit. For that purpose, we use a class that extends `ASTVisitor`. The visitor iterates through all nodes of the compilation units and adds the relevant nodes to a list containing the corresponding node type. There are as many `Lists` in the visitor as `Maps` in the collector.

The difference between the visitor and the collector is that a new visitor instance is used for each compilation unit, meaning that a visitor only contains the relevant information for a single compilation unit, while the collector contains all the relevant nodes for the whole project. Listing 4.31 shows how the visitor and the collector interact with each other.

```java
@Override
public void processUnit( ICompilationUnit unit, RxCollector rxCollector ) {
  ...
  // Initialize Visitor
  String className = SwingWorkerInfo.getBinaryName();
  DiscoveringVisitor discoveringVisitor = new DiscoveringVisitor( className );

  // Collect information using visitors
  compilationUnit.accept( discoveringVisitor );

  // Cache the collected information from visitors in one collector
  rxCollector.add( unit, discoveringVisitor.getTypeDeclarations() );
  rxCollector.add( unit, discoveringVisitor.getFieldDeclarations() );
  rxCollector.add( unit, discoveringVisitor.getAssignments() );
  // etc...
}
```

Listing 4.31: SwingWorker2Rx Processing Compilation Units

4.3.3 Workers

When all compilation units have been processed, then 2Rx uses the collector and a set of workers provided by the extension (See Figure 3.3) to perform the refactorings. In order to have workers with clear responsibilities and avoid long classes, we implemented a worker for each `ASTNode` present in the collector. That makes a total of nine workers.

Listing 4.32 presents the refactoring algorithm used. First, we get the corresponding map from the collector. For each entry in this map, we take the key, which corresponds to a compilation unit. Then, we iterate trough the values of each map entry. The values correspond to the target nodes. In each of these nodes, we run a refactoring visitor that performs a static analysis and caches all relevant information. Then we apply the refactorings using the abstract syntax tree, the compilation unit, the visitor, the single unit writer and the target node. Finally, we register the current compilation unit into the multiple units writer.

The single unit writer does not modify either the compilation units nor the abstract syntax trees. Instead, it registers the changes in a `ASTRewrite` object. By doing this we guarantee, that all workers have access to the original code. After all workers have been executed, the multiple units writer applies the changes to the compilation units.

```
map = collector.xyzMap // assignmentMap, classInstanceCreationMap, etc...

for each map.entry entry
  compilationUnit = entry.key

  for each entry.values node
    ast = node.ast
    singleUnitWriter = WriterHolder.getSingleUnitWriterInstance( ast, compilationUnit )
    refactoringVisitor = new RefactoringVisitor
    node.accept( refactoringVisitor )
    refactor( ast, compilationUnit, refactoringVisitor, singleUnitWriter, node )
    multipleUnitsWriter.addCompilationUnit( icu )
```

Listing 4.32: Worker's Pseudo Code

After we have implemented all workers, we add them to a set, which is used by 2Rx to refactor the original code. Listing 4.33 shows how we build the set of workers.

```java
@Override
public Set<AbstractRefactorWorker<RxCollector>> getRefactoringWorkers( RxCollector rxCollector ) {
  ...
  Set<AbstractRefactorWorker<RxCollector>> workers = new HashSet<>();
  workers.add( new AssignmentWorker( rxCollector ) );
  workers.add( new FieldDeclarationWorker( rxCollector ) );
  workers.add( new MethodInvocationWorker( rxCollector ) );
  // etc...
}
```

Listing 4.33: SwingWorker2Rx Providing Workers

4.3.4 Writers

It is possible to extend the `RxSingleUnitWriter`, explained in Section 3.3.1, to support transformations that might not have been considered in 2Rx. Since all workers are executed simultaneously it is important to make sure that the methods here implemented are thread-safe. To accomplish that, we use the construct `synchronized`. Listing 4.34 shows an example for replacing a `SimpleType`.

```java
@Override
public synchronized void replaceType( SimpleType oldType, String newType ) {
  AST ast = astRewriter.getAST();
  SimpleType newSimpleType = ast.newSimpleType( ast.newName( newType ) );
  astRewriter.replace( oldType, newSimpleType, null );
}
```

Listing 4.34: 2Rx RxSingleUnitWriter - Replace Type (Thread Safe)

4.3.5 Source Code Generation

2Rx provides a couple of methods that can be used to generate code from a string (source code). By using these methods and FreeMarker [8] templates, we generate source code without having to specify all nodes using JDT. Listing 4.35 shows the template used for generating inner class `RxSubscriber` showed in Listing 4.16. Basically, there is a Java object called `model` that contains all necessary information for filling up the templates. This object is passed to the FreeMarker processor in order to produce the source code.

```
class ${model.className} extends SWSubscriber<${model.resultType}, ${model.processType}>{

<#list model.fieldDeclarations as fieldDeclaration> ${fieldDeclaration} </#list>

${model.className}() { setObservable(getRxObservable()); }

<#include "getRxObservable.ftl">
<#include "common/processBlock.ftl">
<#include "common/doneBlock.ftl">

<#list model.methods as method> ${method} </#list>

<#list model.typeDeclarations as typeDeclaration> ${typeDeclaration} </#list>
}
```

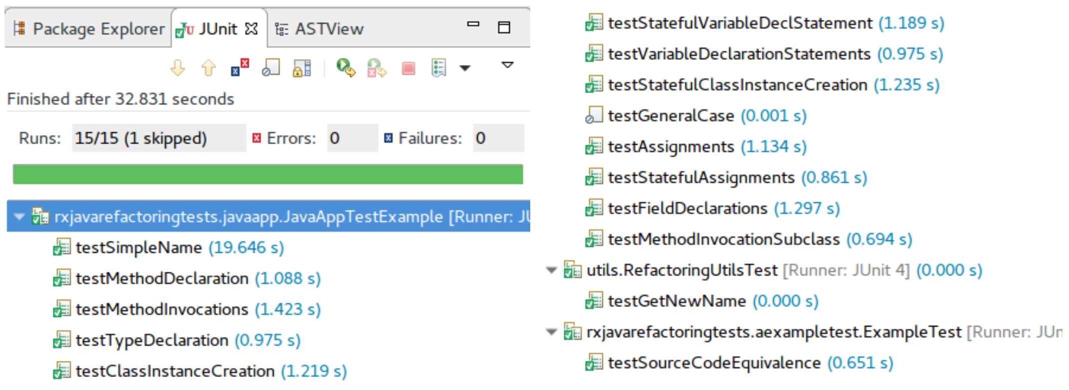
Listing 4.35: Subscriber Freemarker Template

4.3.6 Test-Driven Development

We also developed a third project for test-driven development. This project only contains tests. The main idea is to have a basic project containing the cases to be tested. Each file contains a single case (Figure 4.3). Additionally, there is a second folder containing all expected Java classes, which are used for the assertions.

When the tests are executed, the basic project is refactored without writing the changes to the files. In this way, we guarantee that the input files never change and can always be reused. The resulting code of each compilation unit is saved in a map. For the comparison, we generate an AST for output and expected file. Then the trees are compared. The purpose of doing this is to ignore irrelevant differences such as spaces, empty lines, comments, etc.

Similar to the extensions, there is a template to facilitate setting up the project for the unit tests. This template contains abstract tests classes that can be used to load Android or Java projects, create Java projects and assert source code based on string values.



```
Finished after 32.831 seconds
Runs: 15/15 (1 skipped)   Errors: 0   Failures: 0

rxjavarefactoringtests.javaapp.JavaAppTestExample [Runner: JU
    testSimpleName (19.646 s)
    testMethodDeclaration (1.088 s)
    testMethodInvocations (1.423 s)
    testTypeDeclaration (0.975 s)
    testClassInstanceCreation (1.219 s)
    testStatefulVariableDeclStatement (1.189 s)
    testVariableDeclarationStatements (0.975 s)
    testStatefulClassInstanceCreation (1.235 s)
    testGeneralCase (0.001 s)
    testAssignments (1.134 s)
    testStatefulAssignments (0.861 s)
    testFieldDeclarations (1.297 s)
    testMethodInvocationSubclass (0.694 s)
utils.RefactoringUtilsTest [Runner: JUnit 4] (0.000 s)
    testGetNewName (0.000 s)
rxjavarefactoringtests.aexampletest.ExampleTest [Runner: JUr
    testSourceCodeEquivalence (0.651 s)
```

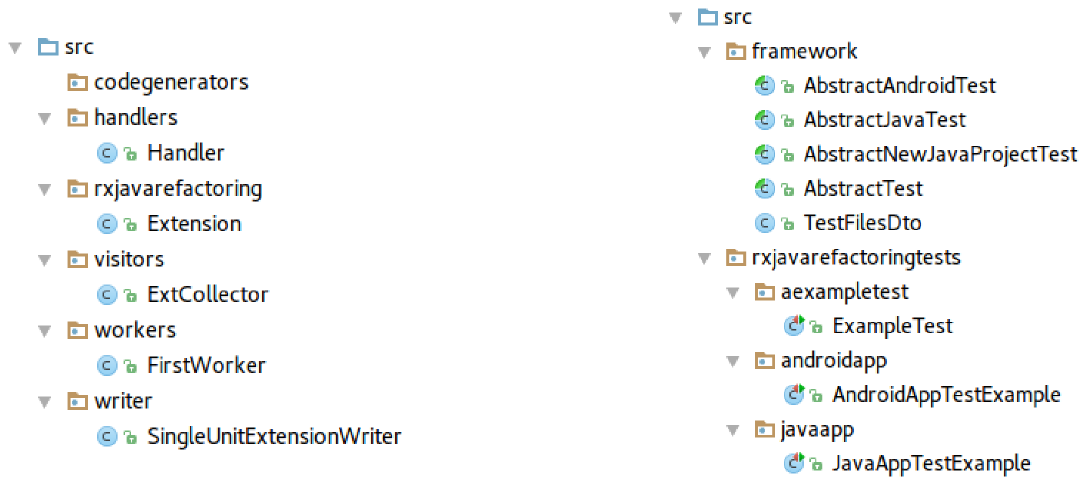
Figure 4.3: Unit Test Results

4.4 Templates

We developed two templates, to facilitate developing and testing extensions for 2Rx.

Figure 4.4a shows the package structure of the template design for extensions. The template consists of five classes. The `Handler` is already implemented. Its function is to forward the event to 2Rx. The classes `Extension`, `ExtCollector` and `FirstWorker` contains "TODOs" to facilitate their implementation. The name "FirstWorker" is a placeholder and should be renamed by developers to improve comprehension. The `SwingleUnitWriter` from 2Rx can directly be used by extensions. However, since the probability of having to add a method to this class is high, we added the `SingleUnitExtensionWriter` to the template, where new methods for manipulating the source code can be added.

Similarly, Figure 4.4b shows the package structure of the template design for writing unit tests. This template consists of four abstract classes and three example tests classes. In order to be able to test 2Rx and its extensions, it is necessary to have an Eclipse project. These abstract classes open or create a project containing the input files. By using this templates developers can skip those steps and start writing the assertions they need. In Listing 4.36 we show how a test would look like. The method `executeTest` is already implemented in the template project.



```
▼ src
    codegenerators
  ▼ handlers
      Handler
  ▼ rxjavarefactoring
      Extension
  ▼ visitors
      ExtCollector
  ▼ workers
      FirstWorker
  ▼ writer
      SingleUnitExtensionWriter
```

(a) Extension Template

```
▼ src
  ▼ framework
      AbstractAndroidTest
      AbstractJavaTest
      AbstractNewJavaProjectTest
      AbstractTest
      TestFilesDto
  ▼ rxjavarefactoringtests
    ▼ aexampletest
        ExampleTest
    ▼ androidapp
        AndroidAppTestExample
    ▼ javaapp
        JavaAppTestExample
```

(b) Test Template

Figure 4.4: Template Projects

```java
public void testRefactoring() throws Exception {
  String targetFile = "FieldDeclaration.java";
  // getSourceCode( String ... path ) The name of the class does not need to match the name of the file
  String expectedSourceCode = getSourceCode( "expected.java.code", "FieldDeclarationRefactored.java" );
  executeTest( targetFile, expectedSourceCode );
}

private void executeTest( String targetFile, String expectedSourceCode ) throws Exception {
  RxJavaRefactoringApp app = new RxJavaRefactoringApp();
  Extension refactoringExtension = new Extension();
  app.setCommandId( refactoringExtension.getId() );
  app.setExtension( refactoringExtension );
  app.refactorOnly( targetFile );
  app.start( null );
  ...
  String actualSourceCode = getSourceCodeByFileName( targetFile, results );
  assertEqualSourceCodes( expectedSourceCode, actualSourceCode );
}
```

Listing 4.36: Unit Test Example

5 Evaluation

We divided the evaluation into two parts. The first one corresponds to the accuracy of the refactoring approach and the second one focuses on the extensibility of the 2Rx.

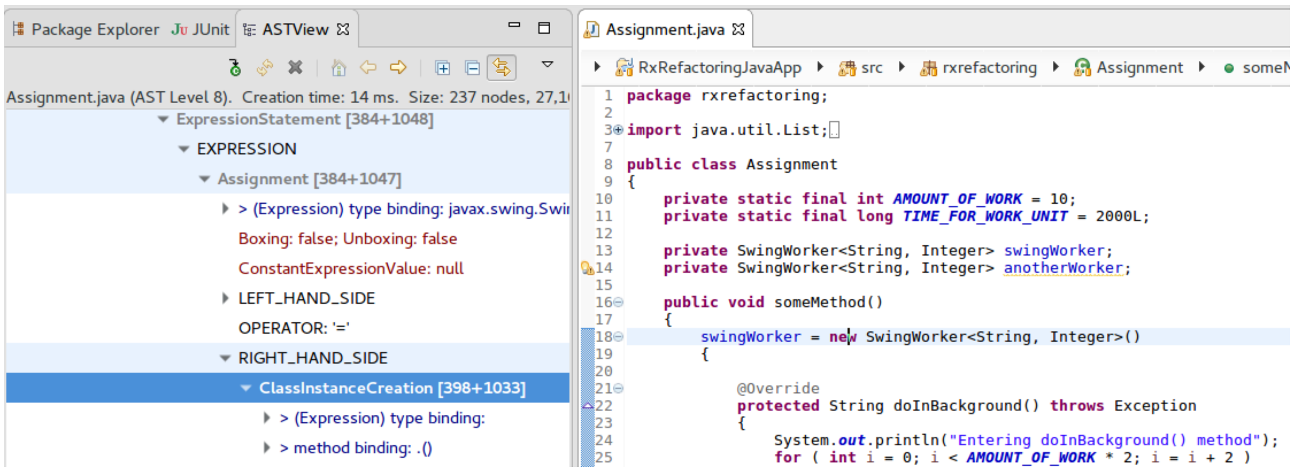
5.1 Refactoring Approach

We used 58 projects for the evaluation of the refactoring approach:

- 1 project for unit tests (Section 5.1.1)
- 10 projects for UI tests (Section 5.1.2)
- 47 projects to analyze the original code versus the refactored one. The analysis consist of code inspections and making sure that there are no compilation errors after refactoring. (Section 5.1.3)

5.1.1 Dataset for Unit Tests

In the implementation phase, we developed a project for unit tests following the test-driven development technique (Section 4.3.6). The test cases in this project were designed considering `SwingWorker` relevant `ASTNodes`. In order to identify these nodes, we used the AST view from Eclipse. The AST view is an Eclipse plugin that allows one to determine which `ASTNode` type corresponds to a particular construct in the source code. Figure 5.1 shows the ASTView panel on the left side and Editor on the right. If the option "Link with Editor" is enabled, then any selection on the ASTView panel will move the cursor in the Editor (source code) to the corresponding `ASTNode`. Similarly, any selection in the Editor highlights the corresponding node in the ASTView panel. In this example, the cursor in the Editor points to `new SwingWorker` and therefore the node `ClassInstanceCreation` is highlighted in the ASTView.



```
Assignment.java (AST Level 8).  Creation time: 14 ms.  Size: 237 nodes, 27,1
      ▾ ExpressionStatement [384+1048]
        ▾ EXPRESSION
          ▾ Assignment [384+1047]
            ▸ > (Expression) type binding: javax.swing.Swi
              Boxing: false; Unboxing: false
              ConstantExpressionValue: null
            ▸ LEFT_HAND_SIDE
              OPERATOR: '='
            ▾ RIGHT_HAND_SIDE
              ▾ ClassInstanceCreation [398+1033]
                ▸ > (Expression) type binding:
                ▸ > method binding: .()
```

```java
package rxrefactoring;

import java.util.List;

public class Assignment
{
    private static final int AMOUNT_OF_WORK = 10;
    private static final long TIME_FOR_WORK_UNIT = 2000L;

    private SwingWorker<String, Integer> swingWorker;
    private SwingWorker<String, Integer> anotherWorker;

    public void someMethod()
    {
        swingWorker = new SwingWorker<String, Integer>()
        {

            @Override
            protected String doInBackground() throws Exception
            {
                System.out.println("Entering doInBackground() method");
                for ( int i = 0; i < AMOUNT_OF_WORK * 2; i = i + 2 )
```

Figure 5.1: AST View

In order to design the test cases, we went through the source code of several projects and used the ASTView panel to identify the `ASTNodes` that are involved with `SwingWorkers`. During this inspection we found 9 `ASTNodes`. We show an example of each of these nodes in Section 4.2:

- Section 4.2.1: `Assignment`
- Section 4.2.2: `VariableDeclarationStatement`
- Section 4.2.3: `ClassInstanceCreation`
- Section 4.2.4: `FieldDeclaration`
- Section 4.2.5: `MethodDeclaration`
- Section 4.2.6: `MethodInvocation`
- Section 4.2.7: `SimpleName`
- Section 4.2.8: `SingleVariableDeclaration`
- Section 4.2.9: `TypeDeclaration`

We then proceeded to design test cases for each of these `ASTNodes`.

5.1.2 Dataset for UI Tests

We chose the projects for UI tests from Bitbucket according to the following criteria: small projects (less than 50 Java files) that compile without having to setup a server, database, etc. We did not select these projects randomly because our goal was to be able to evaluate them at runtime and make sure that the behavior of the applications did not change. Randomly chosen applications are often not compilable due to missing files, complex setups, etc.

5.1.3 Dataset for Source Code Analysis

We used the remaining 47 projects to evaluate the refactoring tool by analyzing its output. We downloaded these projects randomly from Github. The search was performed using the Search site from Github "`https://github.com/search`". We used the following search parameters, Query: "`SwingWorker`", Language: Java, Sort: Recently indexed. The equivalent search URL is:

- `https://github.com/search?l=Java&o=desc&q=SwingWorker&ref=searchresults&s=indexed&type=Code&utf8=%E2%9C%93`

We did not consider further search parameters such as the number of stars or forks because the number of projects found was too small. Table 5.1 shows an overview of the search results when using starts and/or forks as search parameters.

Criteria	Nr. Results
Stars > 0 and Forks > 0	0
Stars > 0	4
Forks > 0	3
Stars > 0 or Forks > 0	7

Table 5.1: Search Results using Nr. of Starts and Forks

From the list of results, we downloaded the first 50 Eclipse or Maven projects. Then we imported them into the Eclipse workspace to make sure that they compile. We analyzed projects that did not compile in order to determined whether the noncompilable source code was `SwingWorker` relevant or not. If the noncompilable code was not `SwingWorker` relevant, then we commented it out to hide the compilation errors showed in the IDE. Commenting out few lines of code does not affect the integrity of the evaluation because these projects were not meant to be tested at runtime. Three out of the fifty projects could not be properly setup to avoid compilation errors before refactoring. Therefore, we removed these projects from the final lists of projects to be evaluated.

5.1.4 Results

Once we finished the implementation of SwingWorker2Rx, including the unit tests, we proceeded to test the tool using 58 projects. Approximately 10% of the projects (6/58) presented errors. We analyzed these errors, identified the missing test cases and added them to the unit tests. Table 5.2 shows an overview of the final results. The 58 projects contain over 10,000 Java files. From these files, SwingWorker2Rx refactored 180 files and a total of 678 `ASTNodes`. 676 `ASTNodes` were refactored successfully, while 2 `ASTNodes` presented errors. In Section 5.1.4 we show the limitations of SwingWorker2Rx. On the right side of Table 5.2 we show which kind of AST nodes containing `SwingWorker` were refactored. Appendix 8.2.2 shows the results of each project.

	Total
Projects	58
Java Files	10,055
Refactored Java Files	180
Refactored ASTNodes	678
Refactored ASTNodes (Successful)	676
Refactored ASTNodes (Failed)	2
Time	5 min 17 s

ASTNode	Total
TypeDeclaration	78
FieldDeclaration	42
Assignment	41
VariableDeclarationStatement	70
SimpleName	116
ClassInstanceCreation	146
SingleVariableDeclaration	9
MethodInvocation	171
MethodDeclaration	5
Total	**678**

Table 5.2: Refactoring Results

We performed a runtime evaluation and code inspections. We used 10 projects for the runtime evaluation (Appendix 8.2.2 - Projects 2-11). This evaluation consisted of running the applications before and after refactoring and making sure that the behavior of the program did not change. For the code inspections, we used 48 projects (Appendix 8.2.2 - Projects 1, 12-58). In this case, we randomly took about 10% of the refactored `ASTNode`s and verified that the output corresponds to the expected source code. We computed the 10% based on the total number of refactored `ASTNode`s in each project, which can be read from the Eclipse console.

The results presented above show that the refactoring approach presented in this thesis and the tool SwingWorker2Rx are accurate. Approximately 99,7% of the `SwingWorker` relevant `ASTNode`s were refactored successfully.

Limitations

The limitations of SwingWorker2Rx are not linked to the refactoring approach, but to the implementation of the tool. Listings 5.1 and 5.2 show the refactoring of a `SwingWorker` instance creation. From the developer's point of view, the refactoring consists of changing one line (Listing 5.1: Line 1 by Listing 5.2: Line 1) and adding three new lines (Listing 5.2: Lines 2,7,9). However, what SwingWorker2Rx actually does is to read the whole `SwingWorker` as showed in Listing 5.1, and use that source code to generate the complete code showed in Listing 5.2. This implementation makes impossible to refactore nested `SwingWorker`s because only the outer `SwingWorker` is considered. In order to support refactoring nested `SwingWorker`s, the implementation of SwingWorker2Rx have to be modified, so that the tool only refactors the lines highlighted in Listing 5.2. This way, the tool would also be able to refactor the code inside `doInBackground`, `process`, `done` and any other method inside of a `SwingWorker`.

```
1  SwingWorker<String, Integer> swingWorker = new
   ↪  SwingWorker<String, Integer>() {
2    protected String doInBackground() throws Exception {
3      ...
4    }
5    protected void process(List<Integer> chunks){...}
6    protected void done() {...}
7  };
```

Listing 5.1: Original Code

```
1  rx.Observable<SWPackage<String, Integer>> rxObservable
   ↪  = rx.Observable
2     .fromEmitter(new SWEmitter<String, Integer>() {
3       protected String doInBackground()
4       throws Exception {
5         ...
6       }
7  }, Emitter.BackpressureMode.BUFFER );
8
9  SWSubscriber<String, Integer> rxObserver = new
   ↪  SWSubscriber<String, Integer>(rxObservable) {
10   protected void process(List<Integer> chunks){...}
11   protected void done() {...}
12 };
```

Listing 5.2: Refactored Code

Listing 5.3 shows an example of nested `SwingWorker`s. Originally an `ExecutorService` submits the instance `singleRun` by invoking `executorSC.submit(singleRun)`. The variable `singleRun` is an instance of `SCWRLrunner`, which is a subclass of `SwingWorker`. In order to fix this compilation error manually, one must replace the error line by "`singleRun.executeObservable()`". Notice that the manual fix is not necessary if SwingWorker2Rx is modified as described above.

```
rx.Observable<SWPackage<Void, Void>> rxObservable = rx.Observable.fromEmitter(new SWEmitter<Void, Void>(){
  @Override
  protected Void doInBackground() throws Exception {
    ...
        for (SCWRLrunner singleRun : SCWRLTasks) {
          ...
          executorSC.submit(singleRun); // --> COMPILATION ERROR
        }
    ...
  }
}, Emitter.BackpressureMode.BUFFER);
```

Listing 5.3: Project: CrysHomModeling-master
Class: modellingTool.MainMenu

Another limitation found during the evaluation of the projects was method names clashes. Not all method names that can be used in `SwingWorker` subclasses, can also be used in `rx.Subscriber` subclasses. Listing 5.4 shows an example of this problem. Since `SwingWorkers` do not have any method matching the name "`add`", the original code compiles. However the new subclass of `rx.Subscriber` does have an "`add`" method and therefore, the code does not compile.

The solution is to replace Line 6 (Listing 5.4) by "`AsyncPanel.this.add(targetComponent, BorderLayout.CENTER);`" because `AsyncPanel` is the name of the class containing the target method. One could implement an analyzer to check the validity of method names while performing the refactoring. In case that conflicts are found, then one could change the method invocation to `<ClassName>.this.<MethodName>(<arguments>)`.

Due to time limitations we did not adjust SwingWorker2Rx to support nested `SwingWorkers`, neither to avoid method clashes.

```
1  public AsyncPanel extends JPanel {
2    private class InitWorker extends SWSubscriber<Void, Void> {
3      ...
4        protected void done() {
5            ...
6            add(targetComponent, BorderLayout.CENTER); // --> COMPILE ERROR
7            ...
8        }
9    }
10 }
```

Listing 5.4: Project: trol-commander
Class: com.mucommander.ui.layout.AsyncPanel

5.2 Extensibility of 2Rx

In order to evaluate the extensibility of 2Rx, we refactored the implementation of RxFactor to make it a client of 2Rx. We call this extension AsyncTask2Rx. This refactoring did not require applying any changes in 2Rx.

5.2.1 Experimental Setup

We evaluated the extensibility of 2Rx using 34 projects. These projects were taken from `https://github.com/vogellacompany/codeexamples-android/tree/ae69e4c9a89b577659185381eef7dce206afc2cd`, a public repository in Github that contains 257 Android projects. We downloaded all projects and selected the ones that use `AsyncTasks`.

The validation consists of performing the automated refactoring of the 34 projects using RxFactor and AsyncTask2Rx and comparing the outputs of both tools.

5.2.2 Validation Approach

To validate that AsyncTask2Rx and RxFactor produce the same exact result, we refactored the Android projects two times. The first time using RxFactor and the second time using AsyncTask2Rx. We placed the output from both tools in different directories and ran the following Linux command to compare the directories, their files, and the contents of each file:

```
diff -r RxFactor/ AsyncTask2Rx/ -x *.classpath -x *.project -x *.jar -x *.class -x *.cache -x gen
```

The argument `r` is used to specify that the command must be executed recursively (includes all subdirectories). The arguments `RxFactor` and `AsyncTask2Rx` correspond to the directories where we saved the outputs locally. Finally, in order to compare only Java files, we used the argument `x` to exclude the files with extensions `classpath`, `project`, `jar`, `class` and `cache`. Additionally we also excluded the directory `gen`. The files contained in this directory are automatically generated when the project is built.

5.2.3 Results

The refactoring was applied to 24 `ASTNode`s. The “diff” operation found a difference in both folders (Listing 5.5). However, one can see that there are no semantic changes in the source code. Due to this result, we inspected both files and found that they differ because of the position of the method `getRxUpdateSubscriber`. While RxFactor places the method `getRxUpdateSubscriber` in Line 54, 2Rx places it in Line 24. When we refactored RxFactor, we used several methods from a class of 2Rx called `RxSingleUnitWriter`. Specifically, we used the method `addMethod` for inserting `getRxUpdateSubscriber` into the source code. RxFactor has a similar method for this purpose. Listings 5.6 and 5.7 show the implementation of `addMethod` in RxFactor and 2Rx respectively. In both code snippets, one can see that Line 3 defines the position of the method to be inserted. RxFactor uses `insertLast` and 2Rx uses `insertFirst`. Therefore, the Java files obtained after refactoring the class `AsyncLoadingActivity` with both tools (See Listing 5.5) are not equal. However, they are semantic equivalent.

These results show that the RxFactor was successfully adapted into an extension of 2Rx.

```
diff -r -x '*.classpath' -x '*.project' -x '*.jar' -x '*.class' -x '*.cache' -x gen
 ↪ RxFactor/codeexamples-android-master/de.vogella.android.async/src/de/vogella/android/async/
 ↪ AsyncLoadingActivity.java
 ↪ AsyncTask2Rx/codeexamples-android-master/de.vogella.android.async/src/de/vogella/android/async/
 ↪ AsyncLoadingActivity.java
23a24,41
>     private Subscriber<Integer[]> getRxUpdateSubscriber() {
>       return new Subscriber<Integer[]>() {
>         @Override
>         public void onCompleted() {
>         }
>
>         @Override
>         public void onError(Throwable throwable) {
>         }
>
>         @Override
>         public void onNext(Integer values[]) {
>           TextView text = (TextView) activity.findViewById(R.id.text1);
>           text.setText(String.format("%s / 7", values[0] + 1));
>         }
52,69d69
<     private Subscriber<Integer[]> getRxUpdateSubscriber() {
<       return new Subscriber<Integer[]>() {
<         @Override
<         public void onCompleted() {
<         }
<
<         @Override
<         public void onError(Throwable throwable) {
<         }
<
<         @Override
<         public void onNext(Integer values[]) {
<           TextView text = (TextView) activity.findViewById(R.id.text1);
<           text.setText(String.format("%s / 7", values[0] + 1));
<         }
```

Listing 5.5: Diff Command Output

```
1  public void addMethod( MethodDeclaration newMethod,
    ↪ TypeDeclaration typeDeclaration ) {
2    ...
3    listRewrite.insertLast( newMethod, null );
4  }
```

Listing 5.6: RxFactor - `RxSingleUnitWriter.addMethod`

```
1  public synchronized void addMethod( MethodDeclaration
    ↪ newMethod, TypeDeclaration typeDeclaration ) {
2    ...
3    listRewrite.insertFirst( newMethod, null );
4  }
```

Listing 5.7: 2Rx - `RxSingleUnitWriter.addMethod`

6 Conclusion

Modern programming languages are making use of event-driven programming models and reactivity to facilitate both code writing and code comprehension, specially when developing asynchronous applications.

Asynchrony can improve the responsiveness of applications. Since refactoring asynchronous code is not trivial, researchers have developed tools to perform this task. Each tool targets a specific language and problem:

- PromisesLand (JavaScript): converts asynchronous callbacks into `Promises`.
- Asyncifer (C#): refactors callback-based asynchronous code into `async/wait` constructs
- AsyncFixer (C#): finds anti-patterns of `async/wait` and suggest fixes
- Asynchronizer (Android): converts synchronous code into `AsyncTask`
- AsyncDroid (Android): converts `AsyncTask` into `IntentService`
- RxFactor (Android): converts `AsyncTasks` into RxJava.

Previous studies have shown that these tools are needed, highly applicable and accurate [18, 20, 26].

Furthermore, other studies agree that functional and reactive programming models improve code writing and code comprehension [10, 22]. In this thesis, we show how to automatically refactor `SwingWorkers` into RxJava in order to facilitate introducing reactive programming concepts in Java applications that use this async construct. The results show that the developed tool is reliable. From 678 AST nodes (declarations, instances, invocations, methods, variables, fields, among others) that contained `SwingWorkers`, SwingWorker2Rx successfully refactored 676, which represents approximately 99.7%.

6.1 Future Work

Since the plugin was divided into core (2Rx) and extension (SwingWorker2Rx), it is possible to use 2Rx in future research to develop further extensions. The core could also be refactored to improve the user experience, adding settings and/or a help section, and implementing a "UNDO" feature would be some examples of where to start.

6.1.1 Further Async Constructs

In this thesis, we focused on refactoring `SwingWorkers`. There are also other async constructs that can be refactored to RxJava. As showed in Listing 6.1, fire and forget operations are usually implemented using the standard Java classes `Runnable` and `Thread`. This construct can be refactored to RxJava in order to facilitate modifying the behavior of the program using functional programming (Listing 6.2). Notice that the method `performOpAsync` changed from being a `void` method (Listing 6.1: Line 11) to return an object of type `Void` (Listing 6.2: Line 11). This change is mandatory because an object is needed for the method `fromCallable` (Listing 6.2: Line 3).

```
1   void execute() {
2     Runnable runnable = () -> performOpAsync();
3
4     Thread thread = new Thread( runnable );
5     thread.start();
6
7     performOperation();
8     // performOpAsync and performOperation run
        ↪  concurrently
9   }
10
11  void performOpAsync() {
12    ...// no return statement here
13  }
```

Listing 6.1: Runnable before Refactoring

```
1   void execute() {
2     Observable
3             .fromCallable( () -> performOpAsync())
4             .subscribeOn(Schedulers.computation())
5             .subscribe();
6
7     performOperation();
8     // performOpAsync and performOperation run
        ↪  concurrently
9   }
10
11  Void performOpAsync() {
12    ...
13    return null;
14  }
```

Listing 6.2: Runnable after Refactoring

Since `Runnables` are fire and forget, they do not return any value. Assuming that we modify the method `performOpAsync` after refactoring, so that it returns a list, then it is possible to use RxJava methods to filter (Listing 6.3: Line 5) and transform (Listing 6.3: Line 5) the items of the list. Additionally, we could execute an operation based on each of these items (Listing 6.3: Line 7). Furthermore, the Observable object can be used to create reactive programming models in the existing code (See Listing 2.8).

```
1  void execute() {
2    Observable
3            .fromCallable( () -> performOpAsync())
4            .subscribeOn(Schedulers.computation())
5            .filter(item -> validate(item))
6            .map(item -> transform(item))
7            .doOnNext(item -> execute(item))
8            .subscribe();
9
10   performOperation();
11   // performOpAsync and performOperation run concurrently
12 }
13
14 List performOpAsync() {
15   List list;
16   ... // fill list
17   return list;
18 }
```

Listing 6.3: Enabled Methods after Refactoring (Runnable)

Callables can also be used in Java for implementing asynchronous operations. Listing 6.4 presents an example where a Callable is used to perform the operation computeResult (Listing 6.4: Line 1). In contrast to Runnables, Callables does return a result. This result can be obtained by using the reference to the corresponding Future (Listing 6.4: Line 4). Listing 6.5 shows how the original code can be refactored to RxJava. In this case the Callable can be directly used in the fromCallable method (Listing 6.5: Line 1). The result of the asynchronous operation can be accessed through doOnNext (Listing 6.5: Line 4).

```
1  Callable<Integer> task = () -> computeResult();
2  ExecutorService executor =
3    Executors.newFixedThreadPool( 4 );
4  Future<Integer> future = executor.submit( task );
5
6  performOperation();
7
8  // blocks until task has completed
9  Integer asyncResult = future.get();
10 /*
11   Some code here that uses asyncResult
12  */
```

Listing 6.4: Callable Future before Refactoring

```
1  Observable.fromCallable( () -> computeResult())
2      .subscribeOn(Schedulers.computation())
3      .observeOn(SwingScheduler.getInstance())
4      .doOnNext(asyncResult -> {
5        // enters here when computeResult() is done
6        /*
7          Some code here that uses asyncResult
8         */
9      })
10     .subscribe();
11
12 performOperation();
```

Listing 6.5: Callable Future Equivalent after Refactoring

6.1.2 Java 8 and Functional Programming

Java 8 introduces a series of classes to work with data streams. However, Java 8 does not offer methods to specify in which thread an operation should be performed. Refactoring these constructs to RxJava would facilitate introducing asynchrony in functional models implemented in Java 8.

Listing 6.6 shows how data streams can be manipulated in Java 8. First, we filter the products from store STORE_A which price is greater than 50 (Listing 6.6: Lines 2-4). Then, we create new products based on the filtered items by incrementing the price on 10% (Listing 6.6: Lines 5-7). These products are assigned to the store STORE_B (Listing 6.6: Line 8). Finally, we save the new products using a data access object (Listing 6.6: Line 9).

Listing 6.7 shows how the previous example looks after refactoring. Instead of creating a stream of data (Listing 6.6: Line 1), it is necessary to create an Observable that contains the products (Listing 6.7: Line 1). The functions filter and map are identical. Then, we need to specify the operation that should be perform on each item (Listing 6.7: Line 9). Since doOnNext only declares the function, it is necessary to subscribe the Observable in order to actually perform the operation (Listing 6.7: Line 10).

```java
1  productDao.getProducts().stream()
2          .filter(p ->
3                  p.getPrice() > 50 &&
4                  p.getStore().equals(STORE_A))
5          .map(p -> new Product(
6                  p.getId(),
7                  p.getPrice() * 1.10,
8                  STORE_B))
9          .forEach(p -> productDao.save(p));
```

Listing 6.6: Java 8 - Functional Programming

```java
1  Observable.from(productDao.getProducts())
2          .filter(p ->
3                  p.getPrice() > 50 &&
4                  p.getStore().equals(STORE_A))
5          .map(p -> new Product(
6                  p.getId(),
7                  p.getPrice() * 1.10,
8                  STORE_B))
9          .doOnNext(p -> productDao.save(p))
10         .subscribe();
```

Listing 6.7: Java 8 - Refactored to RxJava

Listing 6.8 shows some of the methods that were enabled thanks to this refactoring:

- **Line 9** (subscribeOn): to specify in which thread the operations should be executed (i.e. a background thread)
- **Line 11** (doOnError): to handle errors that might occur while processing the items
- **Line 12** (observeOn): to specify in which thread the following operations should be executed (i.e. main thread)
- **Line 13** (doOnCompleted): to execute an action after the data processing has completed (i.e. updating the UI)

```java
1  Observable.from(productDao.getProducts())
2          .filter(p ->
3                  p.getPrice() > 50 &&
4                  p.getStore().equals(STORE_A))
5          .map(p -> new Product(
6                  p.getId(),
7                  p.getPrice() * 1.10,
8                  STORE_B))
9          .subscribeOn(Schedulers.computation())
10         .doOnNext(p -> productDao.save(p))
11         .doOnError(t -> handleError(t))
12         .observeOn(SwingSchedulers.getInstance())
13         .doOnCompleted(() -> updateUI())
14         .subscribe();
```

Listing 6.8: Enabled Methods after Refactoring

6.1.3 RxJava Extension

We developed an RxJava extension targeting SwingWorkers. We suggest that future research considers analyzing this extension to evaluate the potential of improvement. One of the improvements involves refactoring the communication between the emitter SWEmitter and the subscriber SWSubscriber in order to avoid blocking the publish method of the Observable when the SWSubscriber is processing the previous SWPackage. A possible solution would be to implement a buffer and block only when the buffer is full.

7 References

[1] Agera: Observables and updatables. https://github.com/google/agera/wiki/Observables-and-updatables#threading.
Accessed: 2016-11-24.

[2] Agera: Reactive programming for android. https://github.com/google/agera.
Accessed: 2016-11-27.

[3] Android: Asynctask. https://developer.android.com/reference/android/os/AsyncTask.html.
Accessed: 2016-11-20.

[4] Android: Specifying the code to run on a thread. https://developer.android.com/training/multiple-threads/define-runnable.html.
Accessed: 2016-11-20.

[5] Eclipse: Pde (plug-in development environment). http://www.eclipse.org/pde/.
Accessed: 2017-02-31.

[6] Eclipse: Refator actions. http://help.eclipse.org/juno/index.jsp?topic=%2Forg.eclipse.jdt.doc.user%2Fconcepts%2Fconcept-refactoring.htm.
Accessed: 2016-11-27.

[7] Flapjax demos. http://www.flapjax-lang.org/demos/.
Accessed: 2016-11-24.

[8] Freemarker: Template engine. http://freemarker.org/.
Accessed: 2017-01-13.

[9] Intellij: Refactoring source code. https://www.jetbrains.com/help/idea/2016.1/refactoring-source-code.html.
Accessed: 2016-11-27.

[10] Promises. https://www.promisejs.org/.
Accessed: 2016-11-27.

[11] Reactivex. http://reactivex.io/intro.html.
Accessed: 2016-11-27.

[12] Rxjava: How to use rxjava. https://github.com/ReactiveX/RxJava/wiki/How-To-Use-RxJava.
Accessed: 2017-01-11.

[13] Rxjava: How to use rxjava. https://github.com/ReactiveX/RxJava/wiki/How-To-Use-RxJava.
Accessed: 2016-11-28.

[14] Spring: Understanding javascript promises. https://spring.io/understanding/javascript-promises.
Accessed: 2016-11-27.

[15] A study and toolkit for asynchronous programming in c#. http://www.learnasync.net/.
Accessed: 2016-11-27.

[16] Swingworker api. https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html.
Accessed: 2017-01-11.

[17] Why are guis single-threaded? (2007). http://codeidol.com/java/java-concurrency/GUI-Applications/Why-are-GUIs-Single-threaded/. Accessed: 2016-11-24.

[18] Dig Danny. Refactoring for asynchronous execution on mobile. *IEEE Software*, 2015.

[19] Dig Danny, Franklin Lyle, Gyori Alex, and Lahoda Jan. Lambdaficator: From imperative to functional programming through automated refactoring. *IEEE Software*, 2013.

[20] Dig Danny and Semih Okur Yu Lin. Study and refactoring of android asynchronous programming. *IEEE Software*, 2015.

[21] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

[22] Salvaneschi Guido, Amann Sven, Proksch Sebastian, and Mezini Mira. An empirical study on program comprehension with reactive programming. *ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, 2014.

[23] David Money Harris & Sarah L. Harris. *Digital Design and Computer Architecture*. Packt Publishing Ltd., Livery Place, 35 Livery Street, Birmingham B3 2PB, UK, 2015.

[24] D. Helmbold, C.E. McDowell, C. Wang, and Santa Cruz. Computer Research Laboratory University of California. *Detecting Data Races by Analyzing Sequential Traces*. Technical report (University of California, Santa Cruz. Computer Research Laboratory). University of California, Santa Cruz, Computer Research Laboratory, 1990.

[25] John Hunt. *A Beginner's Guide to Scala, Object Orientation and Functional Programming*. Springer Publishing Company, Incorporated, 2014.

[26] Gallaba Keheliya. Characterizing and refactoring asynchronous javascript callbacks. Master's thesis, The University of British Columbia, 2015.

[27] Adrian Kingsley-Hughes, Kathie Kingsley-Hughes, and Daniel Read. *VBScript Programmer's Reference*. Wrox Press Ltd., Birmingham, UK, UK, 3rd edition, 2007.

[28] Meyerovich Leo A., Guha Arjun, Baskin Jacob, Cooper Gregory H., Greenberg Michael, Bromfield Aleks, and Krishnamurthi Shriram. Flapjax: A programming language for ajax applications. *Brown University Tech Report CS-09-04*, 2009.

[29] Kenneth C. Louden. *Programming Languages: Principles and Practice*. Wadsworth Publ. Co., Belmont, CA, USA, 1993.

[30] Miller Mark S., Tribble E. Dean, and Shapiro Jonathan. Concurrency among strangers. *Springer-Verlag Berlin Heidelberg 2005*, 2005.

[31] Greg Michaelson. *An Introduction to Functional Programming Through Lambda Calculus*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[32] Tsvetinov Nickolay. *Learning Reactive Programming with Java 8*. Elsevier, 225 Wyman Street, Waltham, MA 02451, USA, 2013.

[33] Deligiannis Pantazis, F. Donaldson Alastair, Ketema Jeroen, Lal Akash, and Thomson Paul. Asynchronous programming, analysis and testing with state machines. *ACM*, 2015.

[34] Kamath Arbettu Ramachandra. Automatic refactoring of android application to reactive programming. Master's thesis, Technical University of Darmstadt, 2017.

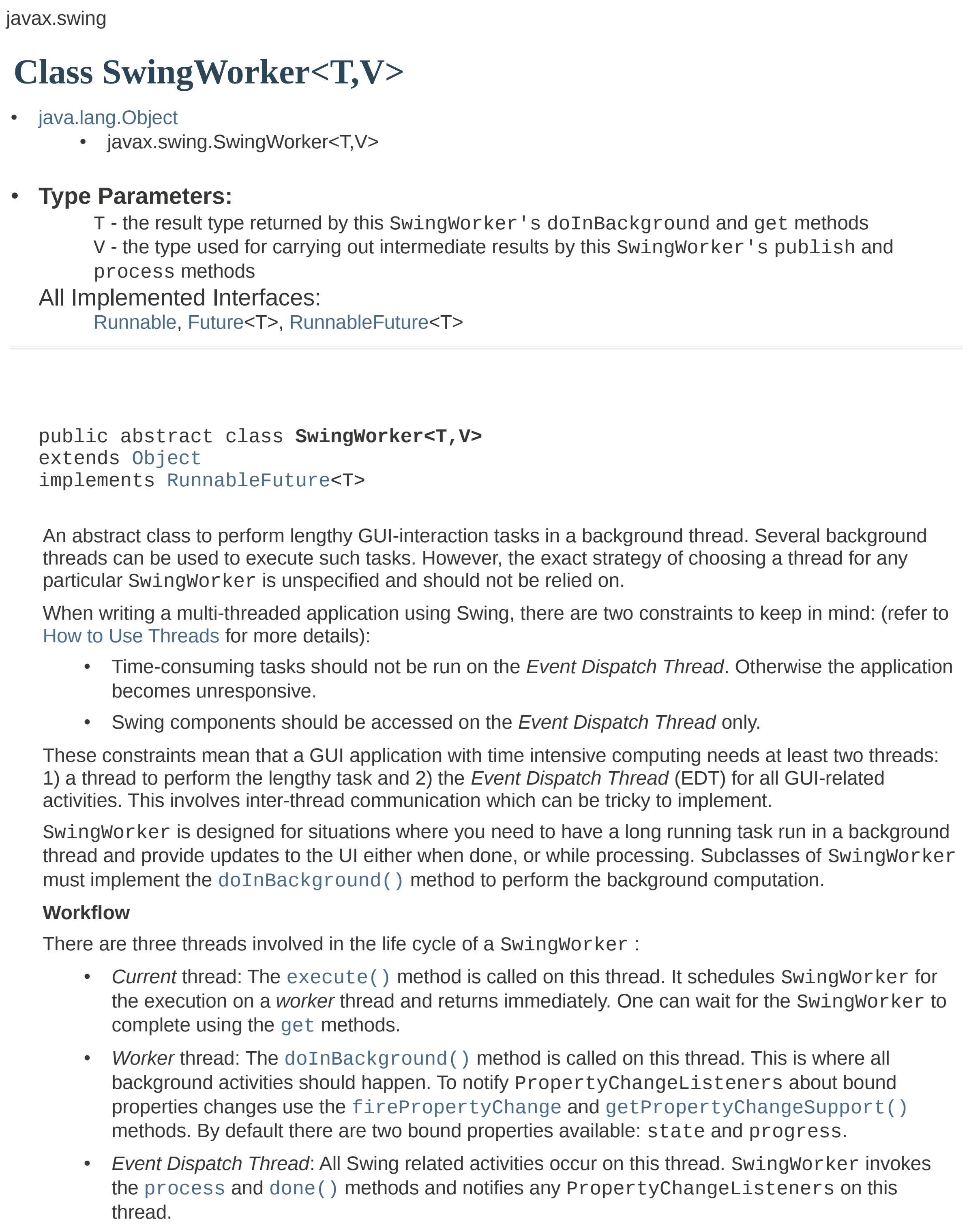
[35] Richard Warburton. *Java 8 Lambdas: Pragmatic Functional Programming*. O'Reilly Media, Inc., 1 edition, 2014.

[36] Opdyke William F. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champaign, 1992.

[37] Lin Yu, Radoi Cosmin, and Dig Danny. Retrofitting concurrency for android applications through refactoring. *FSE'14*, 2014.

8 Appendix

8.1 SwingWorker API



javax.swing

Class SwingWorker<T,V>

- java.lang.Object
    - javax.swing.SwingWorker<T,V>

- **Type Parameters:**

  `T` - the result type returned by this `SwingWorker's doInBackground` and `get` methods
  `V` - the type used for carrying out intermediate results by this `SwingWorker's publish` and `process` methods

  All Implemented Interfaces:

  Runnable, Future<T>, RunnableFuture<T>

```
public abstract class SwingWorker<T,V>
extends Object
implements RunnableFuture<T>
```

An abstract class to perform lengthy GUI-interaction tasks in a background thread. Several background threads can be used to execute such tasks. However, the exact strategy of choosing a thread for any particular `SwingWorker` is unspecified and should not be relied on.

When writing a multi-threaded application using Swing, there are two constraints to keep in mind: (refer to How to Use Threads for more details):

- Time-consuming tasks should not be run on the *Event Dispatch Thread*. Otherwise the application becomes unresponsive.
- Swing components should be accessed on the *Event Dispatch Thread* only.

These constraints mean that a GUI application with time intensive computing needs at least two threads: 1) a thread to perform the lengthy task and 2) the *Event Dispatch Thread* (EDT) for all GUI-related activities. This involves inter-thread communication which can be tricky to implement.

`SwingWorker` is designed for situations where you need to have a long running task run in a background thread and provide updates to the UI either when done, or while processing. Subclasses of `SwingWorker` must implement the `doInBackground()` method to perform the background computation.

Workflow

There are three threads involved in the life cycle of a `SwingWorker` :

- *Current* thread: The `execute()` method is called on this thread. It schedules `SwingWorker` for the execution on a *worker* thread and returns immediately. One can wait for the `SwingWorker` to complete using the `get` methods.
- *Worker* thread: The `doInBackground()` method is called on this thread. This is where all background activities should happen. To notify `PropertyChangeListeners` about bound properties changes use the `firePropertyChange` and `getPropertyChangeSupport()` methods. By default there are two bound properties available: `state` and `progress`.
- *Event Dispatch Thread*: All Swing related activities occur on this thread. `SwingWorker` invokes the `process` and `done()` methods and notifies any `PropertyChangeListeners` on this thread.

https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html

Often, the *Current* thread is the *Event Dispatch Thread*.

Before the `doInBackground` method is invoked on a *worker* thread, `SwingWorker` notifies any `PropertyChangeListeners` about the `state` property change to `StateValue.STARTED`. After the `doInBackground` method is finished the `done` method is executed. Then `SwingWorker` notifies any `PropertyChangeListeners` about the `state` property change to `StateValue.DONE`.

`SwingWorker` is only designed to be executed once. Executing a `SwingWorker` more than once will not result in invoking the `doInBackground` method twice.

- **Nested Class Summary**

Modifier and Type	Class and Description
`static class`	**`SwingWorker.StateValue`** Values for the `state` bound property.

- **Constructor Summary**

Constructor and Description
`SwingWorker``()`   Constructs this `SwingWorker`.

- **Method Summary**

Modifier and Type	Method and Description
`void`	**addPropertyChangeListener**`(PropertyChangeListener listener)` Adds a `PropertyChangeListener` to the listener list.
`boolean`	**cancel**`(boolean mayInterruptIfRunning)` Attempts to cancel execution of this task.
`protected abstract T`	**doInBackground**`()` Computes a result, or throws an exception if unable to do so.
`protected void`	**done**`()` Executed on the *Event Dispatch Thread* after the `doInBackground` method is finished.
`void`	**execute**`()` Schedules this `SwingWorker` for execution on a *worker* thread.
`void`	**firePropertyChange**`(String propertyName, Object oldValue, Object newValue)` Reports a bound property update to any registered listeners.
`T`	**get**`()` Waits if necessary for the computation to complete, and then retrieves its result.
`T`	**get**`(long timeout, TimeUnit unit)` Waits if necessary for at most the given time for the computation to complete, and then retrieves its result, if available.
`int`	**getProgress**`()` Returns the `progress` bound property.
`PropertyChangeSupport`	**getPropertyChangeSupport**`()` Returns the `PropertyChangeSupport` for this `SwingWorker`.
`SwingWorker.StateValue`	**getState**`()`

https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html

	Returns the `SwingWorker` state bound property.
`boolean`	**isCancelled**`()` Returns `true` if this task was cancelled before it completed normally.
`boolean`	**isDone**`()` Returns `true` if this task completed.
`protected void`	**process**`(List<V> chunks)` Receives data chunks from the `publish` method asynchronously on the *Event Dispatch Thread*.
`protected void`	**publish**`(V... chunks)` Sends data chunks to the `process(java.util.List<V>)` method.
`void`	**removePropertyChangeListener**`(PropertyChangeListener listener)` Removes a `PropertyChangeListener` from the listener list.
`void`	**run**`()` Sets this `Future` to the result of computation unless it has been canceled.
`protected void`	**setProgress**`(int progress)` Sets the `progress` bound property.

- **Methods inherited from class java.lang.Object**

  `clone, equals, finalize, getClass, hashCode, notify, notifyAll, toString, wait, wait, wait`

https://docs.oracle.com/javase/7/docs/api/javax/swing/SwingWorker.html

8.2 Evaluation Projects

8.2.1 Projects

The following table contains a list of the projects used for the evaluation. The first project was developed by us for unit-test purposes. Projects 2 to 11 were chosen from Bitbucket, in order to perform UI tests. These projects were not selected randomly. The rest of the projects were selected randomly from Github.

Nr.	Project	Link to Project	Link to Commit
1	RxRefactoringJavaApp	— Developed for unit tests —	—
2	AnkitGupta-image-viewer	`https://bitbucket.org/AnkitGupta/image-viewer`	6382e64849c0fda6c3ffb6102d1f405a407fc4b4
3	antlogik-drinksmachine	`https://bitbucket.org/antlogik/drinksmachine`	c729e079208b8ef252e278f9889d906c6624dc68
4	fkhannouf-tomate	`https://bitbucket.org/fkhannouf/tomate`	4e54f91da712360e2aee0ee05792a622f950295a
5	flimm-polygon	`https://bitbucket.org/flimm/polygon`	d60f039bb4ac6555bef6445af90a47b9d832b85b
6	johnnywsd-java-extractemailfromdocdocxpdftxt	`https://bitbucket.org/johnnywsd/java-extractemailfromdocdocxpdftxt`	9ba3238af0cc057ccad55a28e601fc584c5007af
7	juneiform	`https://bitbucket.org/Stepuk/juneiform`	c07e0bbcf17c2d63137f28109cf5812a231692de
8	RecepieApp	`https://bitbucket.org/majidhumayou/cookbookapplication`	fac2741bd501424e122aca6c21383763c100cfdb
9	reichart-deexifier	`https://bitbucket.org/reichart/deexifier`	2d17cc3d0df776f8df525f5fa4e9724c060aabe6
10	SwingBasics	`https://bitbucket.org/dhiller/swingbasics`	8c0509e7abd9355418bb8bf6ebd59c37fc213fda
11	Tong	`https://bitbucket.org/vehk/footong`	0a0c957e5bb499f94806510ab1380e65674c4ef0
12	Accountant-master	`https://github.com/KaProjects/Accountant`	edade8306fdb28f4a307de2d15b3219ca36ef5da
13	altimeter	`https://github.com/olopes/altimeter`	ef8b5ec0dfeb2beea391061af9bcbc8fb1b62abb
14	atlauncher	`https://github.com/ATLauncher/ATLauncher`	4a9dd4b51a734178abbf16c1d013c8ebcff71678
15	backup-master	`https://github.com/BabitsPaul/backup`	5f5b1c16bf1aa4a62af36f52294ac0776ea1e2af
16	Controls	`https://github.com/vasiliev-alexey/control_compare_utility`	3cbbbab673869b02a2070b079e0e64e54db96f1d
17	corejava8	`https://github.com/demo-from-book/corejava8`	a9ca25566686a92dee12ee76fcd26bd392c42640
18	CrysHomModeling-master	`https://github.com/zivbenster/CrysHomModeling`	8dee4c02fee4c7076efea41daee815e8dc4c052a
19	EdtEvaluation	`https://github.com/Floishy/ba_edtEvaluation`	012e05d27640fed5f3a890c8ce7886ca0059e5e6
20	EInvVatIncoming	`https://github.com/mcfloonyloo/EInvVatIncoming`	4556f5b768786b9ffb165d912a5538ff1c4feaec
21	FCA	`https://github.com/jpotoniec/FCA-ML`	95ccef1f9bfbf6f2dd634025baeefba5577a1991
22	fll-sw	`https://github.com/jpschewe/fll-sw`	e9836f6a05c61e0cb2bdcfc64f28ae4c4c472bc0
23	Fractal	`https://github.com/missweizhang/fractal`	39e637d9e59046da45c05b08aa554448a6ca306c
24	frc-game-sim	`https://github.com/TheLocust3/FRC-Game-Simulator`	349a61466d5ded559b1c235860226b72f84ee055
25	Gitblit	`https://github.com/amyounis/gitblit_assignment`	6964a5cd6774721f494b760c409b4baa0217365b
26	GrainGrowth-master	`https://github.com/piotrek005/GrainGrowth`	9d218b438225c8017f5a80c997c0523b5f2d26e2
27	imondb-collector	`https://github.com/bittremieux/iMonDB`	15f32b58925b6fc0aa3cc07e0046a8ca85c20bf0
28	imondb-core	`https://github.com/bittremieux/iMonDB`	15f32b58925b6fc0aa3cc07e0046a8ca85c20bf0
29	imondb-viewer	`https://github.com/bittremieux/iMonDB`	15f32b58925b6fc0aa3cc07e0046a8ca85c20bf0
30	iris	`https://github.com/jeremybrooks/iris`	e7e318b9b79519cb9bd4edb96105a303415f2e92
31	java-hello-world-master	`https://github.com/yemreu/java-hello-world`	d9405036156b4cce9409d79db1284a5b5d1c56c0
32	JGeckoU	`https://github.com/BullyWiiPlaza/JGeckoU`	c9b78db31f41dddbd9db5b14875b6103146c19b2
33	mandelbrotJava-master	`https://github.com/philiprlarie/mandelbrotJava`	47b7f97bdbe74efc37de6a2cbee2189c2ef38bde
34	meka	`https://github.com/palmer0914/meka`	884f30fa687cba247cab92910ef49fe7c18bbb03
35	MTGDeckEditor	`https://github.com/aroelke/deck-editor-java`	3c6878e82020174507b35050ba4a775a581fd77a
36	MultithreadingwithSwingWorker	`https://github.com/shevapato2008/java-multithreading`	31396dadf964577047fbdfbdf3639dea8b0b7f35
37	my_java	`https://github.com/rexnie/my_java`	22bbf5e1e999a65c0ab6cd7431e8679cab7b30c5
38	NormalMAPP	`https://github.com/SedlaSi/NormalMAPP`	c2f66055be4ba9fe5959b825c00917d1a1a74374
39	ONCClient	`https://github.com/oneilljw/ONC-Client`	9c7cec4a07933f4fbec6a49466f14e37781facfc

Nr.	Project	Link to Project	Link to Commit
40	OssetianCheckers	`https://github.com/MikeColtaine/OssetianCheckers`	95af6ed00c197c71564c46eadeb2d44194093365
41	PF-CORE	`https://github.com/powerfolder/PF-CORE`	d8cf1408fb58b03bbd5f397c8b5c5030e1a17ca3
42	PicturesManager-master	`https://github.com/kboutin/PicturesManager`	bfcc9a2d6ed83f505eadfe12253a63cee792e014
43	PiiL-master	`https://github.com/behroozt/PiiL`	4bbe2032b8e4f7c2e5a1827ef56edcb260fa9cd8
44	PipeCutter	`https://github.com/zhivko/PipeCutter`	c4f25890952777e66e50f1f3dbe5910c42f4dcd3
45	ProiectGeometrie-master	`https://github.com/CretuCalin/ProiectGeometrie`	1677266b397ff5d28af6e3218baaa62d750ad906
46	pwirBank-master	`https://github.com/KrzysztofPytel/pwirBank`	971903906e26ad91ec87120fa23d737aee8a0b82
47	QT-Platform	`https://github.com/msasc/QT-Platform`	c83d36faf7435fd241ac5418c6b4404bc90c55d7
48	rapaio	`https://github.com/palmer0914/rapaio`	7a894923ad06393aff4ad7ca257afbba9c44eee4
49	Repeated_Phrases-master	`https://github.com/fiveham/Repeated_Phrases`	eada5fcfc70a59078a5a086a431d8cb6793c70e9
50	safetyLock	`https://github.com/djzhao627/safetyLock-LeWei`	a9444fe401090ca22ad7e71fa625b249872d2df6
51	Study-Guide-Generator-master	`https://github.com/joshdon/Study-Guide-Generator`	f453b6d6768ab87c4016bf2972a513f447b4cbf2
52	Sudoku	`https://github.com/szepfejuede1/Sudoku`	496f6a2f2a4a955c21b78a534a074ac6404e6229
53	SUST_BackGammon-master	`https://github.com/Rownak/SUST_BackGammon`	7c7663fe1645244e5bbb3ee369d1a006475e7c21
54	Tpad	`https://github.com/reheda/Tpad`	11c67ac7846fa2f19c66af9c3c16b2b2cf7c03ee
55	trol-commander	`https://github.com/trol73/mucommander`	97f41012a6ba3523abe0da249f46ceda3be51480
56	ultttt_model-master	`https://github.com/mrchan64/ultttt_model`	d63ba21379d28facf8e12f08d638b026dd0220f9
57	VritualMonitor	`https://github.com/bixuefeng/VmMonitor`	5fcfa2f20bbef27174267befe58ee2eb48931ff0
58	weka	`https://github.com/palmer0914/weka`	91c7e5460a16267479129f787cddbe4adb818238

Table 8.1: Projects for the Evaluation

8.2.2 Results

Overview

The following table contains an overview of the results. The column changes corresponds to the sum of all refactored AST nodes in each project.

Nr.	Project	Total Java Files	Refactored Files	ASTNodes
1	RxRefactoringJavaApp	13	13	48
2	AnkitGupta-image-viewer	2	1	4
3	antlogik-drinksmachine	19	4	15
4	fkhannouf-tomate	6	1	5
5	flimm-polygon	9	2	6
6	johnnywsd-java-extractemailfrom-docdocxpdftxt	10	3	18
7	juneiform	35	3	10
8	RecepieApp	8	2	6
9	reichart-deexifier	7	2	6
10	SwingBasics	41	5	13
11	Tong	44	1	5
12	Accountant-master	70	1	2
13	altimeter	27	1	3
14	atlauncher	184	10	32
15	backup-master	23	1	2
16	Controls	32	1	5
17	corejava8	387	6	26
18	CrysHomModeling-master	825	6	19
19	EdtEvaluation	9	0	0
20	EInvVatIncoming	19	0	0
21	FCA	61	2	4
22	fll-sw	366	1	5
23	Fractal	10	1	5
24	frc-game-sim	48	3	6
25	Gitblit	533	9	32
26	GrainGrowth-master	25	1	5
27	imondb-collector	41	8	27
28	imondb-core	35	0	0
29	imondb-viewer	72	4	13
30	iris	7	1	3
31	java-hello-world-master	48	1	6
32	JGeckoU	174	7	23
33	mandelbrotJava-master	5	1	3
34	meka	353	3	14
35	MTGDeckEditor	114	3	17
36	Multithreadingwith-SwingWorker	2	1	4
37	my_java	311	10	34
38	NormalMAPP	14	0	0
39	ONCClient	192	5	22
40	OssetianCheckers	29	1	5
41	PF-CORE	895	20	83
42	PicturesManager-master	45	2	11
43	PiiL-master	36	4	18
44	PipeCutter	126	7	26
45	ProiectGeometrie-master	11	0	0
46	pwirBank-master	13	1	5
47	QT-Platform	698	1	6
48	rapaio	425	1	5
49	Repeated_Phrases-master	22	1	3
50	safetyLock	20	2	10
51	Study-Guide-Generator-master	4	2	6
52	Sudoku	14	1	4
53	SUST_BackGammon-master	15	2	7
54	Tpad	28	1	4
55	trol-commander	1,352	8	28
56	ultttt_model-master	11	1	4
57	VritualMonitor	47	0	0
58	weka	2,083	1	5
—	**Totals**	**10,055**	**180**	**678**

Table 8.2: Evaluation Results - Overview

Refactored ASTNodes

The following table contains the information about the specific AST nodes that were refactored in each project.

Nr.	Project	ASTNodes	Type Declarations	Field Declarations	Assignments	Var. Decl. Statement	Simple Name	Instance Creation	Single Variable Decl.	Method Invocations	Method Declarations
1	RxRefactoringJavaApp	48	2	5	3	4	8	11	2	12	1
2	AnkitGupta-image-viewer	4	0	1	0	0	1	1	0	1	0
3	antlogik-drinksmachine	15	2	1	1	2	3	3	0	3	0
4	fkhannouf-tomate	5	0	1	1	0	1	1	0	1	0
5	flimm-polygon	6	0	0	0	2	2	2	0	0	0
6	johnnywsd-java-extractemailfromdocdocxpdftxt	18	3	3	3	0	3	3	0	3	0
7	juneiform	10	1	0	0	2	2	2	0	3	0
8	RecepieApp	6	2	0	0	0	0	2	0	2	0
9	reichart-deexifier	6	1	0	0	1	1	1	0	2	0
10	SwingBasics	13	2	1	1	0	1	3	0	5	0
11	Tong	5	0	1	1	0	1	1	0	1	0
12	Accountant-master	2	0	0	0	0	0	1	0	1	0
13	altimeter	3	1	0	0	0	0	1	0	1	0
14	atlauncher	32	2	2	2	1	7	2	6	10	0
15	backup-master	2	0	0	0	0	0	1	0	1	0
16	Controls	5	1	0	0	1	1	1	0	1	0
17	corejava8	26	3	3	3	1	4	6	0	6	0
18	CrysHomModeling-master	19	3	1	1	2	3	3	1	4	1
19	EdtEvaluation	0	0	0	0	0	0	0	0	0	0
20	EInvVatIncoming	0	0	0	0	0	0	0	0	0	0
21	FCA	4	0	0	0	0	0	2	0	2	0
22	fll-sw	5	1	0	0	1	1	1	0	1	0
23	Fractal	5	1	0	0	1	1	1	0	1	0
24	frc-game-sim	6	2	0	0	1	1	1	0	1	0
25	Gitblit	32	2	0	0	7	7	7	0	9	0
26	GrainGrowth-master	5	1	0	0	1	1	1	0	1	0
27	imondb-collector	27	3	1	1	4	5	4	0	8	1
28	imondb-core	0	0	0	0	0	0	0	0	0	0
29	imondb-viewer	13	2	0	0	2	2	3	0	4	0
30	iris	3	1	0	0	0	0	1	0	1	0
31	java-hello-world-master	6	1	1	1	0	1	1	0	1	0
32	JGeckoU	23	3	0	0	3	3	7	0	7	0
33	mandelbrotJava-master	3	1	0	0	0	0	1	0	1	0
34	meka	14	0	0	2	3	3	3	0	3	0
35	MTGDeckEditor	17	3	2	2	1	3	3	0	3	0
36	MultithreadingwithSwingWorker	4	0	0	0	1	1	1	0	1	0
37	my_java	34	3	3	3	1	4	10	0	10	0
38	NormalMAPP	0	0	0	0	0	0	0	0	0	0
39	ONCClient	22	2	3	3	1	4	4	0	5	0
40	OssetianCheckers	5	1	0	0	1	1	1	0	1	0
41	PF-CORE	83	13	4	4	9	13	20	0	20	0
42	PicturesManager-master	11	2	1	1	1	2	2	0	2	0
43	PiiL-master	18	1	1	1	3	4	4	0	4	0
44	PipeCutter	26	1	1	1	5	7	5	0	6	0
45	ProiectGeometrie-master	0	0	0	0	0	0	0	0	0	0
46	pwirBank-master	5	1	0	0	1	1	1	0	1	0
47	QT-Platform	6	1	1	1	0	1	1	0	1	0
48	rapaio	5	0	1	1	0	1	1	0	1	0
49	Repeated_Phrases-master	3	1	0	0	0	0	1	0	1	0
50	safetyLock	10	0	0	0	2	2	2	0	2	2
51	Study-Guide-Generator-master	6	1	0	0	1	1	1	0	2	0
52	Sudoku	4	0	0	0	1	1	1	0	1	0
53	SUST_BackGammon-master	7	1	1	1	0	1	1	0	2	0
54	Tpad	4	0	0	0	1	1	1	0	1	0
55	trol-commander	28	5	3	3	0	3	6	0	8	0
56	ultttt_model-master	4	0	0	0	1	1	1	0	1	0
57	VritualMonitor	0	0	0	0	0	0	0	0	0	0
58	weka	5	1	0	0	1	1	1	0	1	0
—	Totals	678	**78**	**42**	**41**	**70**	**116**	**146**	**9**	**171**	**5**

Table 8.3: Evaluations Results - Refactored ASTNodes