Automated Refactoring to Reactive Programming

Calvin Huang and Md Uddin

Asynchronous Programming

Asynchronous programming is a form of parallel programming that allows a unit of work to run separately from the primary application thread. When the work is complete, it notifies the main thread (whether the work was completed or failed).

Java has many Asynchronous constructs some of which are low level complicating communication with the main thread.

Java also provides constructs that provide abstractions for Asynchronous constructs

Constructs	total	%
java.lang.Thread	40,988	14.86%
java.util.concurrent.Executors	11,708	4.24%
java.util.concurrent.Future	5,718	2.07%
javax.swing.SwingWorker	1,651	0.60%
java.util.concurrent.FutureTask	1,372	0.50%
com.google....ListenableFuture	295	0.11%
javafx.concurrent.Task	163	0.06%
akka.dispatch.Futures	122	0.04%
java.util.concurrent.ForkJoinTask	92	0.03%
javax.ejb.AsyncResult	82	0.03%
javax.ws.rs.container.AsyncResponse	31	0.01%
javax.enterprise.concurrent.ManagedTask	19	0.01%
java.util.concurrent.CompletableFuture	7	<0.01%
org.springframework.scheduling.annotation.Async	1	<0.01%
Projects with asynchronous constructs	46,208	16.75%
All Java projects	275,879	100.00%

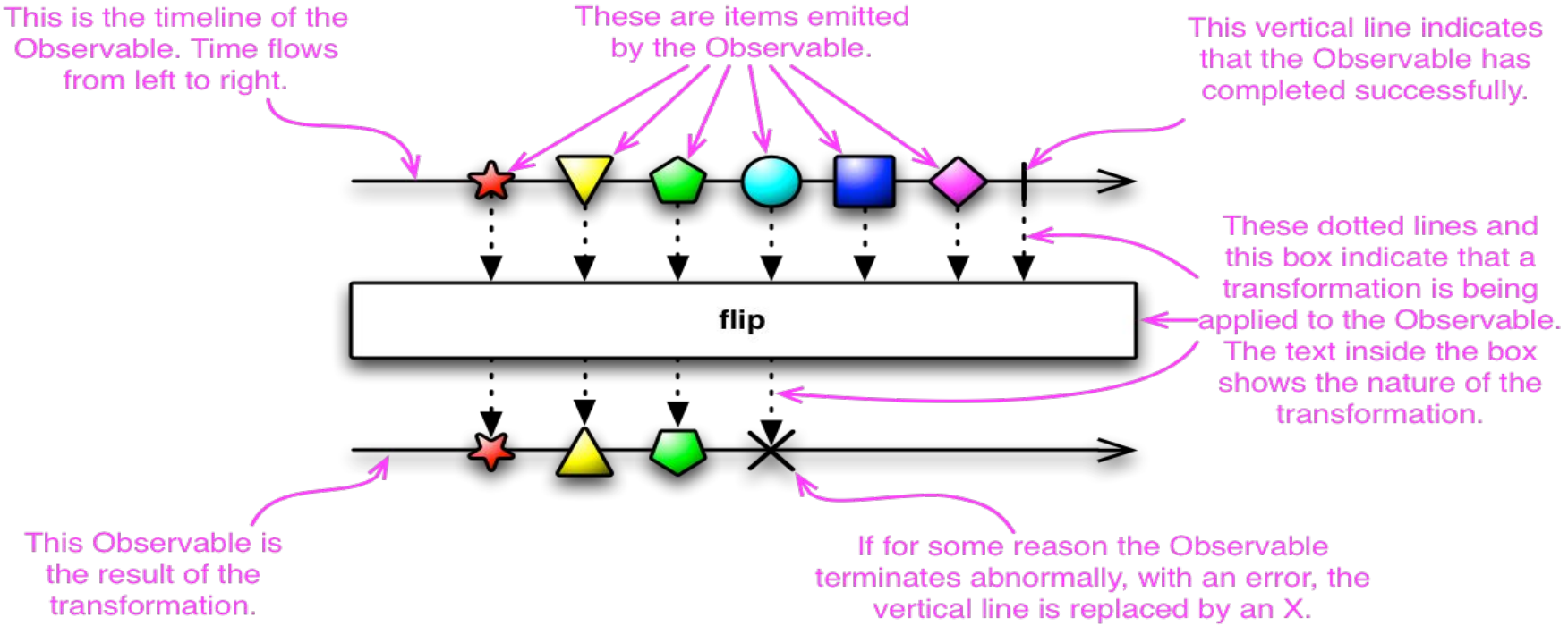
Reactive Programming

Reactive Programming is a development model structured around asynchronous data streams.
ReactiveX is a library for asynchronous RP that provides abstractions and operators to process and combine event streams
Observable, which is the source of an event stream
An Observer can register to an Observable and an Observer can be notified of event occurrences
Observables can be chained and executed on different threads

```
1 Observable<Data> data = getDataFromNetwork();
2 data
3   .filter(d -> d != null)
4   .map(d -> d.toString() + " transformed")
5   .subscribeOn(Schedulers.computation())
6   .subscribe(d ->
7     System.out.println("onNext => " + d));
```

obtain a stream of Data from the network as an Observable
all data that is null gets filtered out
the data is transformed to a String and " transformed" is added to the end
Finally, the result is printed to the command line

Observable Visual



Reason for Automated Refactoring of Asynchronous programs to Reactive Programs.

Asynchronous applications are notoriously error-prone to implement and to maintain – greatly benefit from reactive programming because they can be defined in a declarative style, which improves code clarity and extensibility.

Declarative Style - Write down what you want not how.

.filter()
.map()
.reduce()
Or other functions that you declare

studies indicate that RP increases the correctness of program comprehension not requiring more time or advanced programming skills

Purpose of this paper

Produce a technique to refactor common Java asynchronous constructs to RP.

Design 2RX, a plugin that refactors code into RxJava Observables

Evaluate the approach with automatic testing and code inspection, showing that it is applicable on a broad amount of code that uses asynchronous constructs, providing a large coverage of construct and exhibits good execution time

Release a new large dataset of third-party projects suitable for research on asynchronous programming in Java.

Refactoring an OCR Application

```
1 public abstract class DocumentLoader
2 extends SwingWorker<List<Document>, Document> {
3
4  private File[] files;
5
6
7
8
9
10 protected List<Document> doInBackground()
11 throws Exception {
12  List<Document> results = ...
13  ...
14  for (File f : files) {
15   ...
16   Document d = new Document(... f ...);
17   results.add(d);
18   publish(d);
19   ...
20  }
21  return results;
22 }
23
24
```

```
27 public void load(File... files) {
28  this.files = files;
29  execute();
30 }
31 protected void process(List<Document> chunks){
32  fetchResults(chunks);
33 }
34 protected void done() {
35  ...
36  List<Document> documents = get();
37  ...
38 }
39 public abstract void fetchResult(
40  List<Document> result);
41 ...
42 }
```

Refactoring on OCR Application

```
1 public abstract class DocumentLoader
2  extends SWSubscriber<List<Document>, Document>{
3
4  private File[] files;
5
6  private Observable<SWEvent<List<Document>,
7    Document>> getObservable() {
8   Emitter<List<Document>, Document> emitter =
9    new SWEmitter<List<Document>, Document>() {
10     protected List<Document> doInBackground()
11     throws Exception {
12      List<Document> results = ...
13      ...
14      for (File f : files) {
15       ...
16       Document d = new Document(... f ...);
17       results.add(d);
18       publish(d);
19       ...
20      }
21      return results;
22   } };
23  return Observable.fromEmitter(emitter,
24   Emitter.BackpressureMode.BUFFER);
25 }
```

```
27  public void load(File... files) {
28   this.files = files;
29   executeObservable();
30  }
31  protected void process(List<Document> chunks){
32   fetchResult(chunks);
33  }
34  protected void done() {
35   ...
36   List<Document> documents = get();
37   ...
38  }
39  public abstract void fetchResult(
40   List<Document> result);
41  ...
42 }
```

Refactoring to RP

The key idea of the refactoring is to transform the values generated by asynchronous computations into an event stream that emits an event whenever a new value is generated

Observable emits an event for each results generated by the asynchronous computation.

Target two Java constructs for asynchronous programming:
SwingWorker and Futures

SwingWorker

Java.swing.SwingWorker
SwingWorker is a construct defined in the Java Swing library. It asynchronously executes the code in its doInBackground method, which returns the result of the computation.
SwingWorker can emit intermediate values during the asynchronous execution.
Refactoring SwingWorker to Observable requires to consider two major differences between the two constructs:
SwingWorker does not only emit a final result, but also intermediate results with a different type
SwingWorker keeps track of the current status of the computation – if it is still running or if it has already finished

SwingWorker

To achieve the functionalities of SwingWorker with Observable there needs to be three helper classes

SWEmitter produces an event for each call to the a certain method and for the final result
SWSubscriber implements the SwingWorker API on top of the emitter
SWEvent is the type of events produced by the Observable, holding either an intermediate or a final result
Helper classes enable refactoring more cases, as they take over some of the responsibility of preserving the functionality during the refactoring.
Helper classes complicate the code introducing additional classes and functionalities

Futures

Java.util.concurrent.Future
A future is a code block that has not been computed yet, but is available eventually.
Refactoring to RP relieves developers from handling the emission of the value explicitly as RP will handle the emitted value using the Observer
Enables RP’s support for functional composition and asynchronous execution

ExecutorService()

The ExecutorService is the alternative to Java's Timer and perhaps debated to be optimized for the following reasons.

- Unlike Timer, ExecutorService is not sensitive to changes within the systems internal clock
- Timer has only one thread which can backlog other tasks that need to be done when a long running task is using the thread
- Runtime exceptions can kill the only thread which will kill the Timer

Elastic Search

```
1 ExecutorService pool = ...
2 List<Future<List<T>>> list =
3  new ArrayList<Future<List<T>>>();
4
5 for (int i = 0; i < numTasks; i++) {
6  list.add(
7   pool.submit(new Callable<List<T>>() {
8    @Override
9    public List<T> call() throws Exception {
10    List<T> results = new ArrayList<T>();
11    latch.await();
12    while(count.decrementAndGet() >= 0) {
13     results.add(executor.run()); }
14    return results;
15   }
16 })); }
17 ...
18
19 for (Future<List<T>> future : list) {
20  results.addAll(future.get()); }
```

(a) Original code.

- In line 1, an ExecutorService creates a pool (scheduler)
- In line 2-3, a List is defined that stores a Future of type List<T>. This will be used to create a task to be sent to the scheduler.
- In Line 7, we push a task into the ExecutorService
- Line 9-15, task is defined in the call method of Callable (Lines 9-15) which returns a Future for the task result that is then added to the list

Elastic Search

```
1 ExecutorService pool = ...
2 List<Future<List<T>>> list =
3  new ArrayList<Future<List<T>>>();
4
5 for (int i = 0; i < numTasks; i++) {
6  list.add(
7   pool.submit(new Callable<List<T>>() {
8    @Override
9    public List<T> call() throws Exception {
10    List<T> results = new ArrayList<T>();
11    latch.await();
12    while(count.decrementAndGet() >= 0) {
13     results.add(executor.run()); }
14    return results;
15   }
16 })); }
17 ...
18
19 for (Future<List<T>> future : list) {
20  results.addAll(future.get()); }
```

(a) Original code.

- In line 14, the task waits for the result of another executor and retrieves the result – the List returned by the Future
- Between lines 5 and 14 ,the task is executed asynchronously

How can we refactor?

- There are several ways we can reap the benefits of refactoring original code into reactive programming paradigm
- Most influential is using Rx’s Observable instead of Future
- Stop blocking during asynchronous calculation

```
1 ExecutorService pool = ...
2 List<Observable<List<T>>> list =
3  new ArrayList<Observable<List<T>>>();
4
5 for (int i = 0; i < numTasks; i++) {
6  list.add(Observable.fromFuture(
7   pool.submit(new Callable<List<T>>() {
8    @Override
9    public List<T> call() throws Exception {
10    List<T> results = new ArrayList<T>();
11    latch.await();
12    while(count.decrementAndGet() >= 0) {
13     results.add(executor.run()); }
14    return results;
15   }
16  }), Schedulers.computation())); }
17 ...
18
19 for (Observable<List<T>> future : list) {
20  results.addAll(future.blockingSingle()); }
```

- The list now stores Observable instead of Future(3)
- Observable is created from the same Callable that was submitted to the executor(9-15)
- The Observable uses a Scheduler to run asynchronously (16)
- The Future is still executed according to the ExecutorService – only the Observable operates on the Scheduler.

So what has changed?

- Many improvements have been made by replacing the Future type with an Observable.
- Unlike future, Observable doesn’t block when returning the data opposed to Future.get() in the initial implementation
- It becomes truly asynchronous as the tasks can be done in the future while older tasks are still being computed.
- With Future, the call to .get() causes our program to run momentarily synchronously as it blocks the program from attending to other tasks.
- Future is not easy to optimize for asynchronous execution flows as the latency varies for each execution.

Preconditions

Before applying a refactoring, it is crucial to check whether certain conditions are correct in order to see if the certain parts of code can be refactored. Like if a future is a standard future.
Three preconditions for applying refactoring on a portion of asynchronous code(Future or SwingWorker)

The asynchronous computation isn’t cancelled. ReactiveX provides no way to cancel asynchronous computations of Observables, but only to unsubscribe an observer (which does not cancel the running computation). Why is that? Suppose we have a network request being scheduled and we decide to cancel the request, we won’t be able to cancel the internal computations as RxJava is not aware of what lies inside the observable therefore cannot cancel it from happening until finish. We are able to however unsubscribe it from the scheduler.

Preconditions (Continued)

A notable disadvantage that using Observable takes away is the ability to check the internal state during the asynchronous process. Most asynchronous constructs provide us with ways where we can directly retrieve the current state of an asynchronous execution. Therefore, since we do not have this functionality, a user should consider this before using it. Furthermore, Future.isDone() is used to solve such problems and has the upper hand over Observables.

Solution for Retrieving Old State

One of the reasons we can’t retrieve current (old state) with observables is because reactive programming is against shared state. The solution to this is convoluted as it can break other things not seen at first. A programmer can extend the Observable to subclasses but this creates problems with the onSubscribe() callback.

Implementation

designed 2RX as an Eclipse plugin for refactoring Java projects.

2RX is an API that allows retrieving the AST of a compilation unit, performing static data flow analyses, identifying a specific Java construct, manipulating the AST, and outputting the refactored code.
Implemented an automatic precondition checker for both the constructs currently supported.
The checker is based on a flow-sensitive static analysis on Java source code.

Evaluation: Research Questions

Which fraction of asynchronous constructs used in real-world projects is supported by 2RX?

How many occurrences of the supported asynchronous constructs can 2RX correctly refactor?(how many cases satisfy the preconditions that 2RX requires to perform the refactoring and lead to refactored code that is correct and achieves the same functionality of the original code.)

Is 2RX fast enough to be usable by developers?

Dataset:

Used Boa, which provides a snapshot of all public GitHub projects from 2015, consisting of 380,125 projects.
275,879 projects containing Java source files.
46,208 Java projects contain at least one asynchronous construct.
Found a total of 7,133 projects that use at least one of the two supported constructs(SwingWorker and Java Future)
5,718 projects use Java Future and 1,651 projects use Swingworker.
Removed projects that do not use one of the popular build tools Maven or Ant to automate the evaluation (automatically tested the refactored code as well as checked whether the refactored code compiles)
4,652 projects for Future and 1,118 projects for SwingWorker

Automatic Test Generation:

To evaluate the correctness of the refactorings, they ran unit tests for each project.
The refactoring does not change the functionality of the code.
To automatically test the refactored code, they implemented a framework based on Randoop. The generated tests capture the behavior of the original code, and are run again on the refactored code to ascertain that programs are behaviorally equivalent to the original code

Results

Stars	Project	cond y	cond n	Time (ms)	LOC	files	c?	t?
3058	Zookeeper	8	0	16,051	83,956	671	✓	⌀
1777	Disunity	1	0	7,733	5,720	113	✓	✓
1285	Gitblit	3	0	13,378	63,910	415	✓	✓
661	OptaPlanner	2	0	31,138	60,219	946	✓	⌀
565	Jabref	1	0	20,756	93,268	698	✓	✓
486	Nodebox	2	0	9,926	32,244	283	✓	✓
150	ATLauncher	1	0	5,941	46,292	348	✓	⌀
109	CookieCadger	4	0	1,976	4,415	18	✓	✓
89	PIPE	3	0	13,676	73,597	732	✓	✓
70	BurpSentinel	3	0	1,056	10,217	132	✓	✓

(a) SwingWorker.

Stars	Project	cond y	cond n	Time (ms)	LOC	files	c?	t?
23495	Elasticsearch	4	0	261,132	370,006	3,595	✓	✓
6152	JUnit	1	0	4,577	24,218	375	✓	✓
5820	DropWizard	2	1	48,910	17,708	361	✓	✓
4871	Mockito	2	0	104,409	52,871	822	✓	⌀
4790	Springside4	0	2	8,266	20,293	199	✓	✓
4424	Titan	1	3	116,117	40,301	531	✓	⌀
3774	AsyncHttpClient	105	0	15,402	29,739	344	✓	⌀
3327	Graylog2Server	0	5	47,839	138,663	2,014	✓	✓
3018	Java Websocket	0	1	1,653	5,117	52	✓	✓
2840	B3log	0	1	16,217	14,635	173	✓	✓

(b) Java Future.

SwingWorker: all occurrences of asynchronous constructs pass the preconditions

Compilation after refactoring succeeds for every project. Automatic test generation fails for 3 projects.

Future: all occurrences passed the preconditions and 3 projects failed the automatic test generation. 89.8% of occurrences were capable of being refactored.

2RX is capable of refactoring 91.7% of cases in total with 6 projects that failed the automatic test generation.

~366 ms/1K LOC for SwingWorker, and ~1121 ms/1K LOC for Java Future

Threats To Validity

Internal Validity:There can be some unnoticed differences in behavior that the Automatic test doesn’t pick up.
They mitigated this by inspecting all refactoring manually and determined if they were correct with a third party of people and themselves.
External Validity: Whether their results can be generalized
Consider codebases developed by different teams, which promises variety in coding style.
Manual inspection showed a diverse usage of asynchronous constructs
Futures used with Executor, custom implementations, or as part of collections, amongst others.

Increasing confidence that the results presented in this paper generalize to most Java projects

Conclusion

They are currently extending 2RX to support more constructs and improve its applicability.
They hope that, equipped with 2RX, more and more programmers can bring the design benefits of RP to their projects.
The SwingWorker refactoring is faster than the Future refactoring, because the precondition analysis of Future is more involved because the helper classes in the SwingWorker refactoring remove some of the need of certain preconditions.
They consider these speeds acceptable as it only took at least 3 mins to refactor a large program and you only need to refactor a program once.