Automated Refactoring for Asynchronous Applications

TECHNISCHE UNIVERSITÄT DARMSTADT

Bachelor thesis

Author:
Grebiel José Ifill Brito

Supervisor:
Prof. Dr. Guido Salvaneschi
Reactive Programming Technology

TECHNISCHE UNIVERSITÄT DARMSTADT

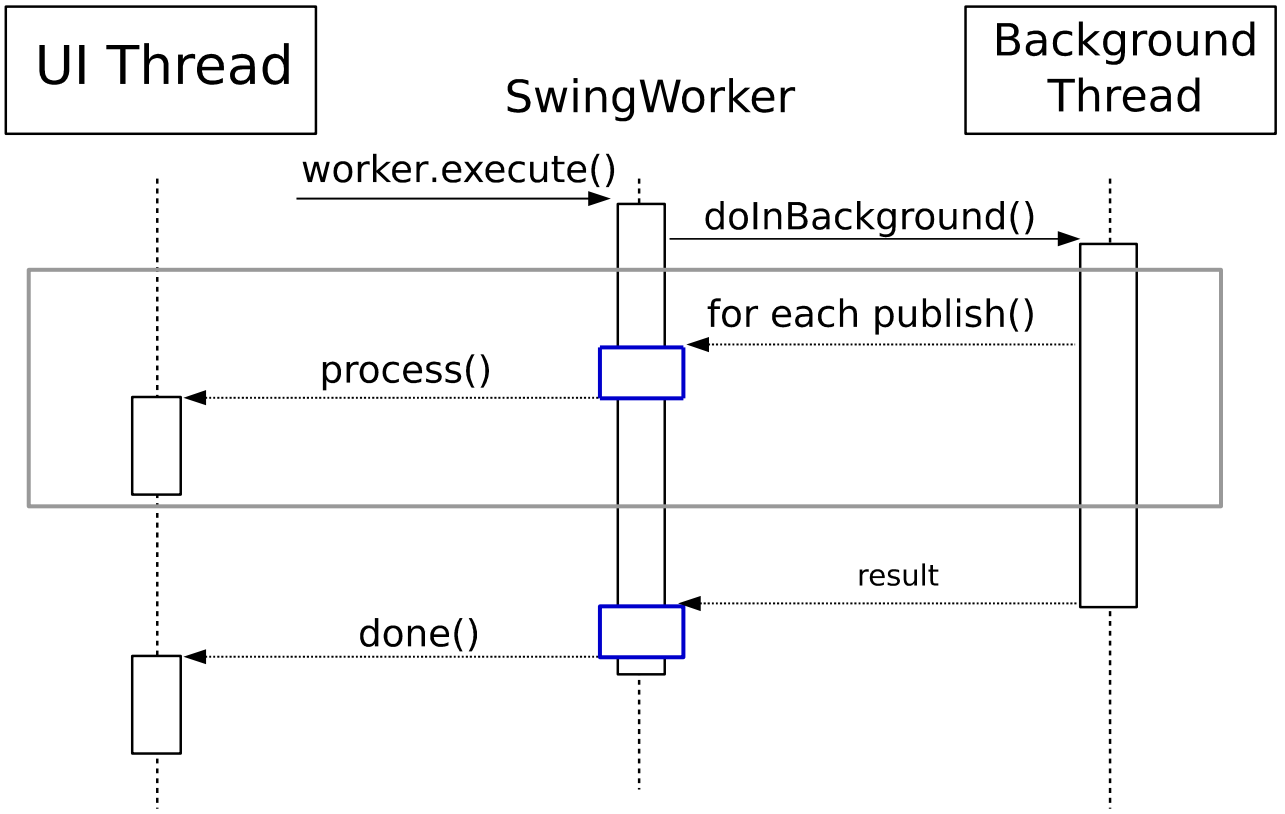
Motivation

- Functional & Reactive Programming → facilitates writing and comprehension
- Asynchronous applications (Callback based) → not trivial
- What about existing applications ? → Refactoring!
- Refactoring is tedious → Automated ☺

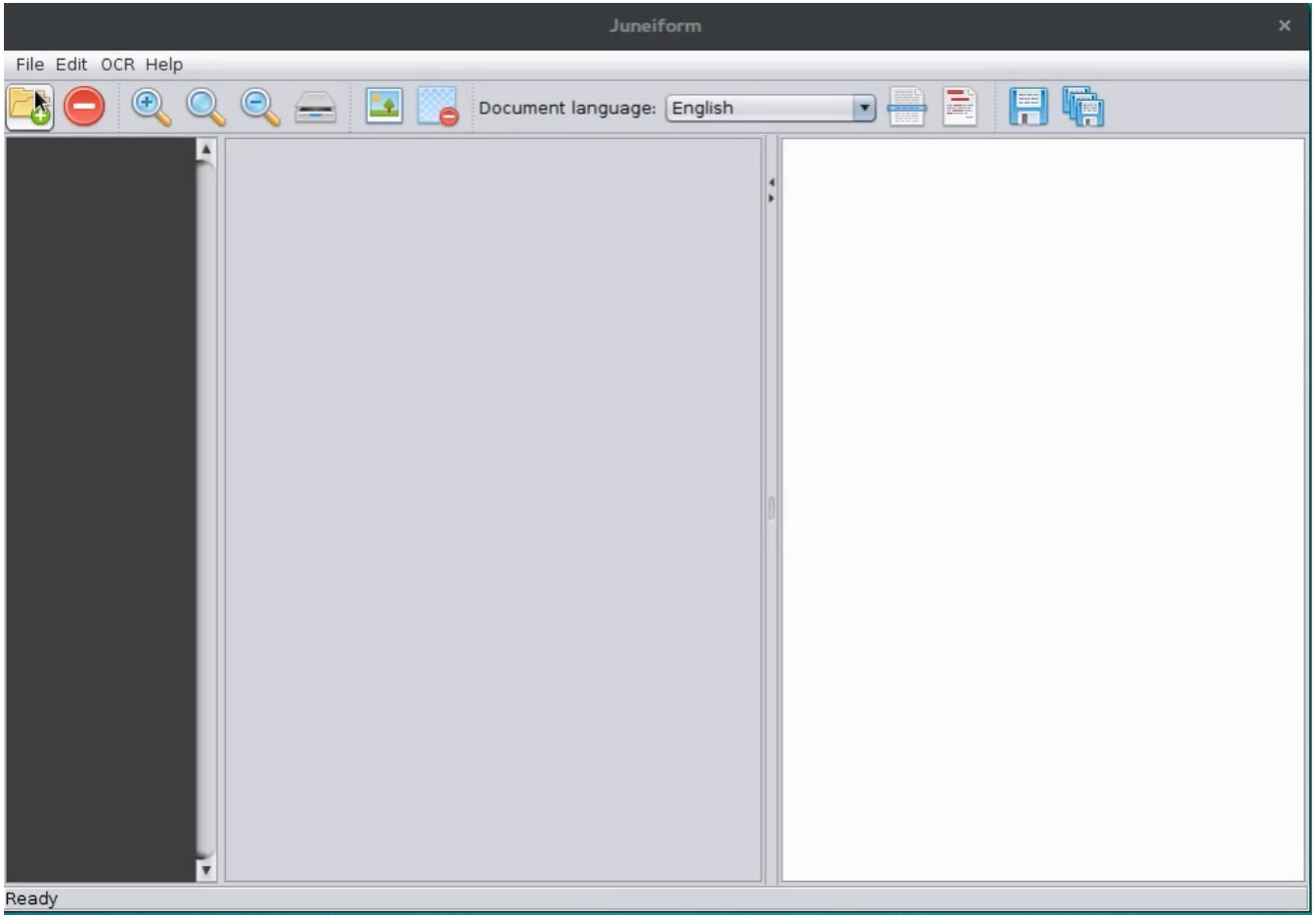
Contribution

- Refactoring approach to convert `SwingWorker`s into RxJava
  - `SwingWorker` async construct for Swing applications
- System to host refactoring tools (`2Rx`)
- Client of `2Rx`: `SwingWorker2Rx`
- Client of `2Rx`: `AsyncTask2Rx` (based on `RxFactor`)

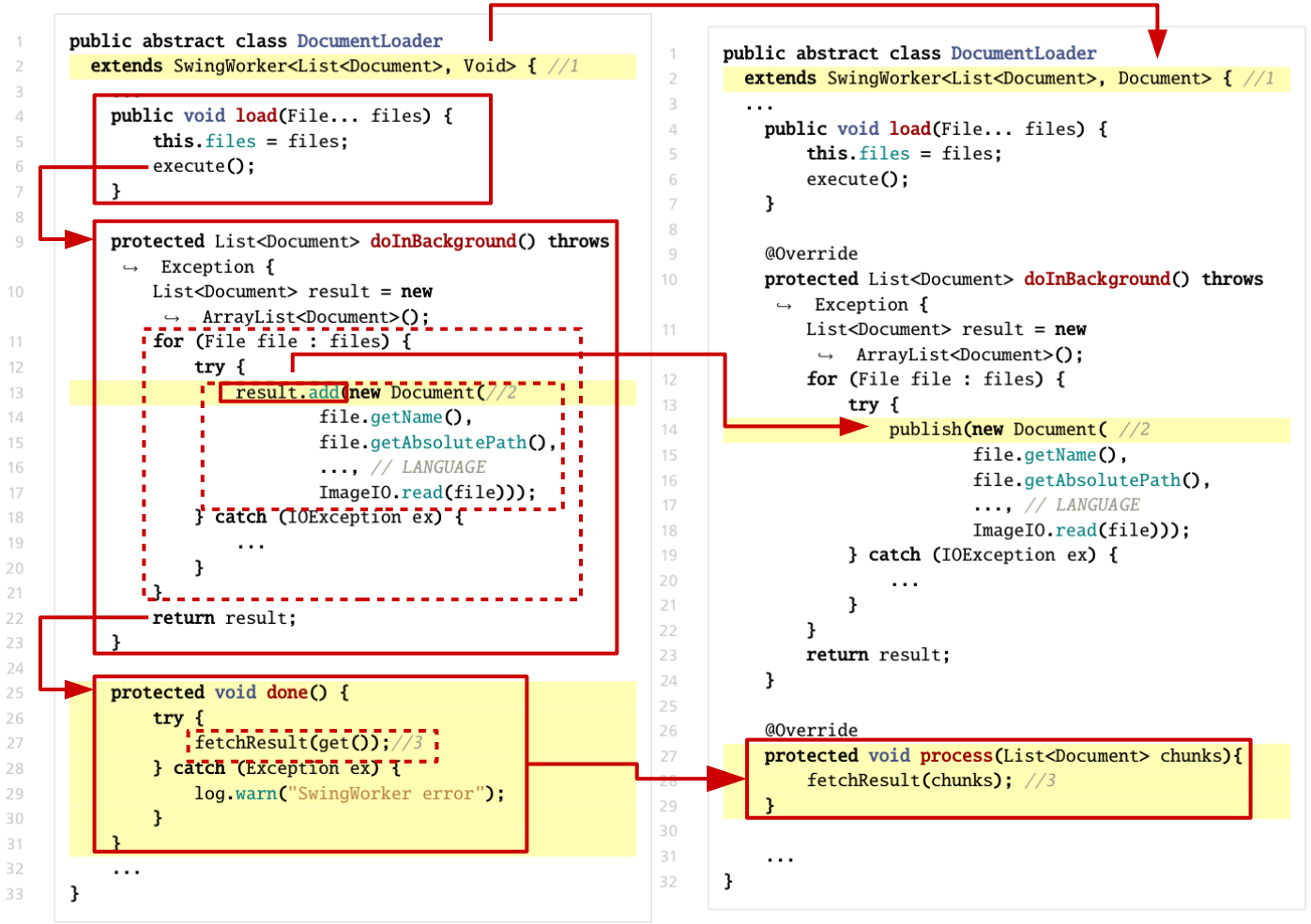
SwingWorker



Juneiform (before refactoring)



DocumentLoader - Pre-Processing



DocumentLoader – Automated Refactoring

Same Semantic

Observer (Subscriber)

```
public abstract class DocumentLoader
  extends SwingWorker<List<Document>, Document> {
    ... // 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 );
    }
  ...
}
```

```
public abstract class DocumentLoader
  extends SWSubscriber<List<Document>, Document> {
    ... // 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 );
  }
  ...
}
```

Subject (Observable)

DocumentLoader – Functional & Reactive Programming

```
...
  public void load( File... files ) {
    this.files = files;
    // setup subject
    ConnectableObservable<Document> connectableObservable = getRxObservable()
        .subscribeOn( Schedulers.computation() )
        .flatMap( swPackage -> Observable.from( swPackage.getChunks() ) )
        .filter( doc -> doc.getName().contains( ".jpg" ) )
        .map( doc -> new Document(
            doc.getName().replace( ".jpg", "" ).toUpperCase(),
            doc.getPath(),
            doc.getLanguage(),
            doc.getImage() ) )
        .observeOn( SwingScheduler.getInstance() )
        .publish();
    // register subscribers
    connectableObservable.subscribe( document -> {
      print( "0 - Updating UI: ", document );
      fetchResult( Arrays.asList( document ) );
    } );
    connectableObservable.subscribe( doc -> print( "1 - ", doc ) );
    connectableObservable.subscribe( doc -> print( "2 - ", doc ) );
    connectableObservable.subscribe( doc -> print( "3 - ", doc ) );
    // connect subject with subscribers
    connectObservable( connectableObservable );
  }
...
```

Editor – Automated Refactoring

Same Semantic

```
...
SwingWorker<String, Void> worker =
  new SwingWorker<String, Void>() {

    @Override
    protected String doInBackground() {
        ... // OCR Logic
    }

    @Override
    protected void done() {
    ... // Update UI
    }
};
worker.execute();
...
```

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

SWSubscriber<String, Void> rxObserver =
  new SWSubscriber<String, Void>(rxObservable) {
  @Override
  protected void done() {
    ... // Update UI
  }
};
rxObserver.executeObservable();
...
```

Subject (Observable)

Observer (Subscriber)

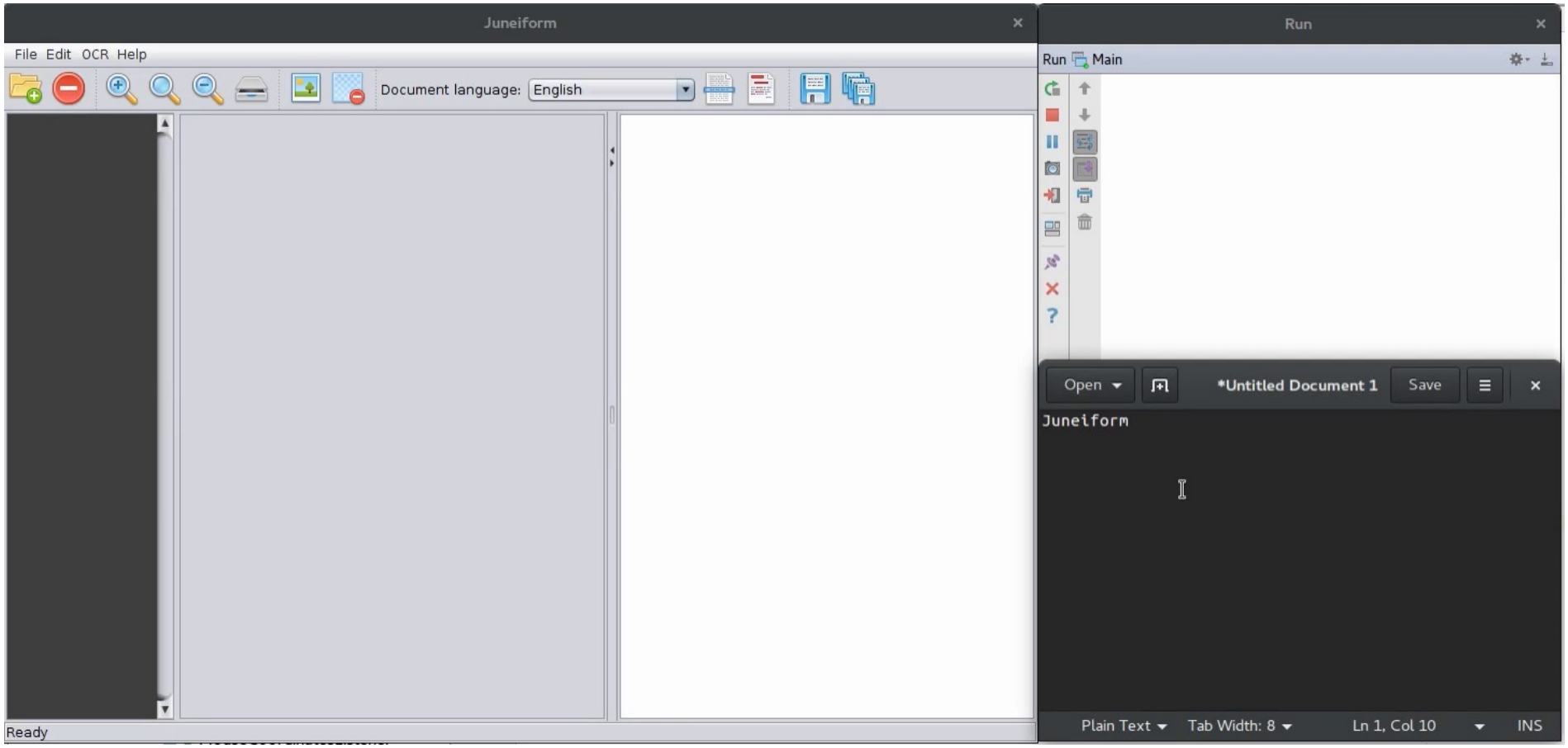
Editor – Functional Programming

```
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 }
```

Juneiform (after refactoring and adding features)

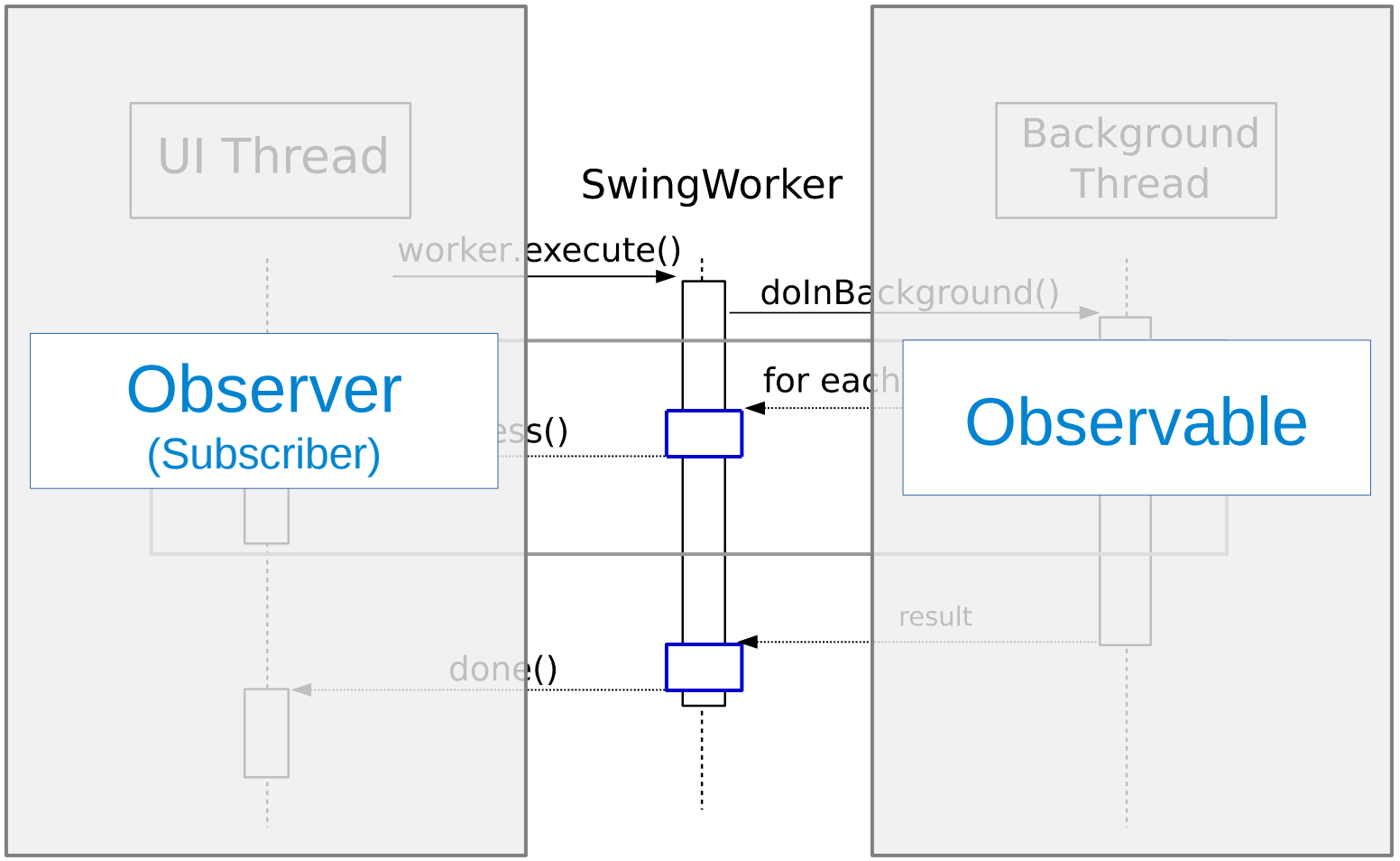
Application

Console



Text Editor

SwingWorker



Evaluation - Accuracy

- Unit tests: 1 project
- Runtime tests: 10 projects
- Code inspections: 47 projects
- Approximately 99,7% accuracy

	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**

Evaluation - Extensibility

- RxFactor → AsyncTask2Rx
- 34 Projects, 24 `ASTNode`s
- diff -r RxFactor/ AsyncTask2Rx/ -x *.classpath -x *.project -x *.jar -x *.class -x *.cache -x gen
  - 1 file differs

RxFactor

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

AsyncTask2Rx

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

Conclusions

- Functional and Reactive programming facilitates code writing and code comprehension.
- Refactoring SwingWorkers to RxJava manually takes time.
- SwingWorker2Rx can refactor SwingWorkers automatically.
- The evaluation shows that …
  - SwingWorker2Rx is accurate
  - 2Rx is extensible

Questions

- Refactoring Approach ? (6 slides)
  - Structure (UML Class Diagram)
  - Dynamic (Sequence Diagram)
- 2Rx & SwingWorker2Rx ? (3 slides)
  - Dynamic (Sequence Diagram)
- Future Work ? (7 slides)
  - Further constructs
  - 2Rx: New Features
  - SwingWorker2Rx: Improvements
- Limitations ? (2 slides)

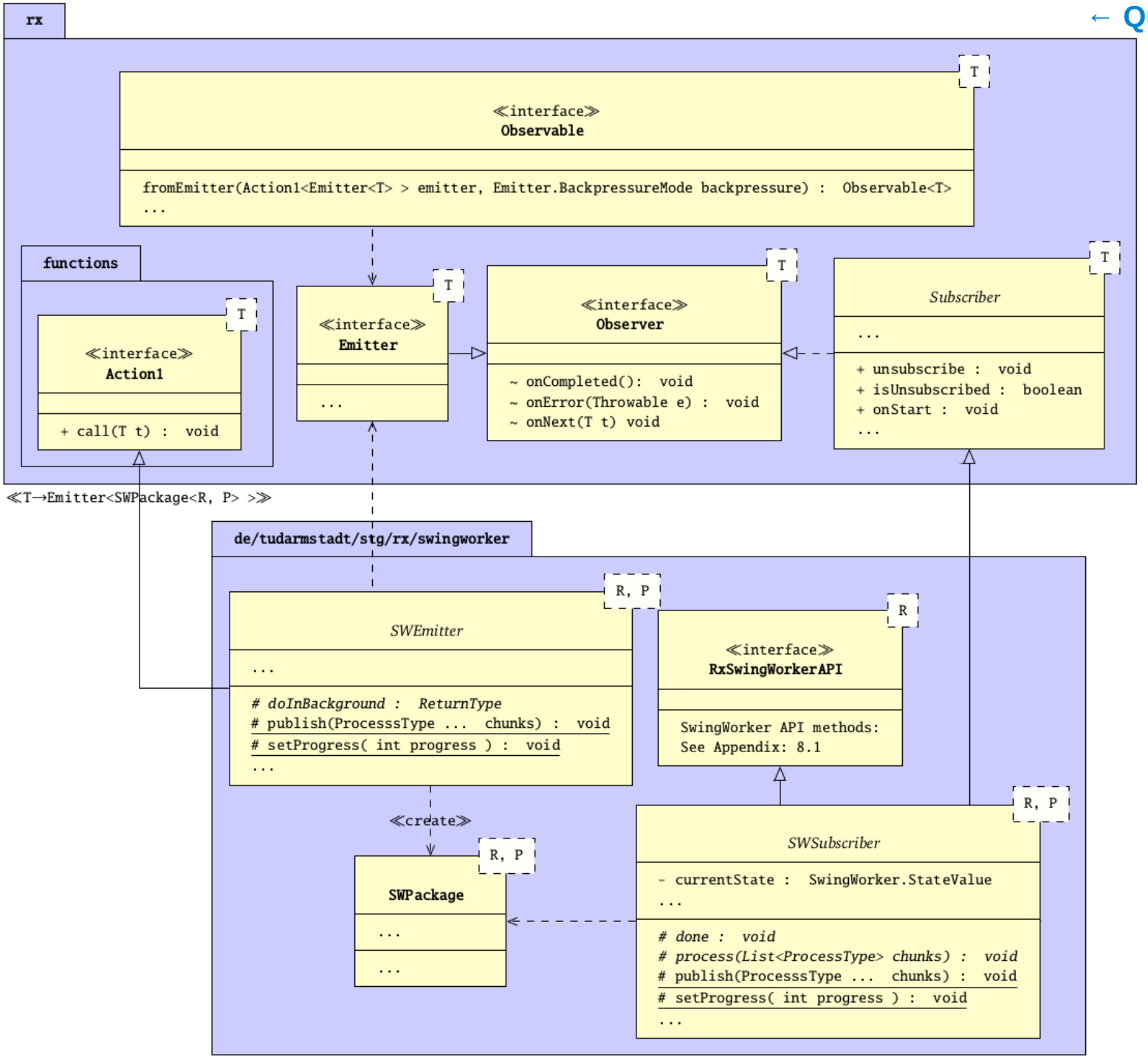
- Other questions are also welcome…

  (but no slides ☺)

Refactoring Approach

UML Class Diagram

(next slide)


rx
T
≪interface≫
Observable
fromEmitter(Action1<Emitter<T> > emitter, Emitter.BackpressureMode backpressure) : Observable<T>
...
functions
T
≪interface≫
Action1
+ call(T t) : void
T
≪interface≫
Emitter
...
T
≪interface≫
Observer
~ onCompleted(): void
~ onError(Throwable e) : void
~ onNext(T t) void
T
Subscriber
...
+ unsubscribe : void
+ isUnsubscribed : boolean
+ onStart : void
...
≪T→Emitter<SWPackage<R, P> >≫
de/tudarmstadt/stg/rx/swingworker
R, P
SWEmitter
...
doInBackground : ReturnType
publish(ProcesssType ... chunks) : void
setProgress(int progress) : void
...
R
≪interface≫
RxSwingWorkerAPI
SwingWorker API methods:
See Appendix: 8.1
≪create≫
R, P
SWPackage
...
...
R, P
SWSubscriber
- currentState : SwingWorker.StateValue
...
done : void
process(List<ProcessType> chunks) : void
publish(ProcesssType ... chunks) : void
setProgress(int progress) : void
...

Observable & Subscriber Example

← Questions

Subject (Observable)

```
1  Observable.fromEmitter(new Action1<Emitter<Integer>>() {
2      public void call(Emitter<Integer> emitter) {
3          try {
4              for (int i = 1; i < 10; i++) {
5                  emitter.onNext(i);
6              }
7              emitter.onCompleted();
8          } catch (Exception e) {
9              emitter.onError(e);
10         }
11     }
12 }, Emitter.BackpressureMode.BUFFER ).subscribe(new Subscriber<Integer>() {
13     public void onNext(Integer item) {
14         System.out.println("DOUBLE VALUE = " + item.doubleValue());
15     }
16     public void onError(Throwable error) {
17         System.err.println("ERROR = " + error.getMessage());
18     }
19     public void onCompleted() {
20         System.out.println("RESULT = done");
21     }
22 });
```

Observer (Subscriber)

Source Code: Before vs. After Refactoring

← Questions

- Class Instance Creation

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

Subject (Observable)

```
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() {...}
};
```

Observer (Subscriber)

Source Code: Before vs. After Refactoring

← Questions

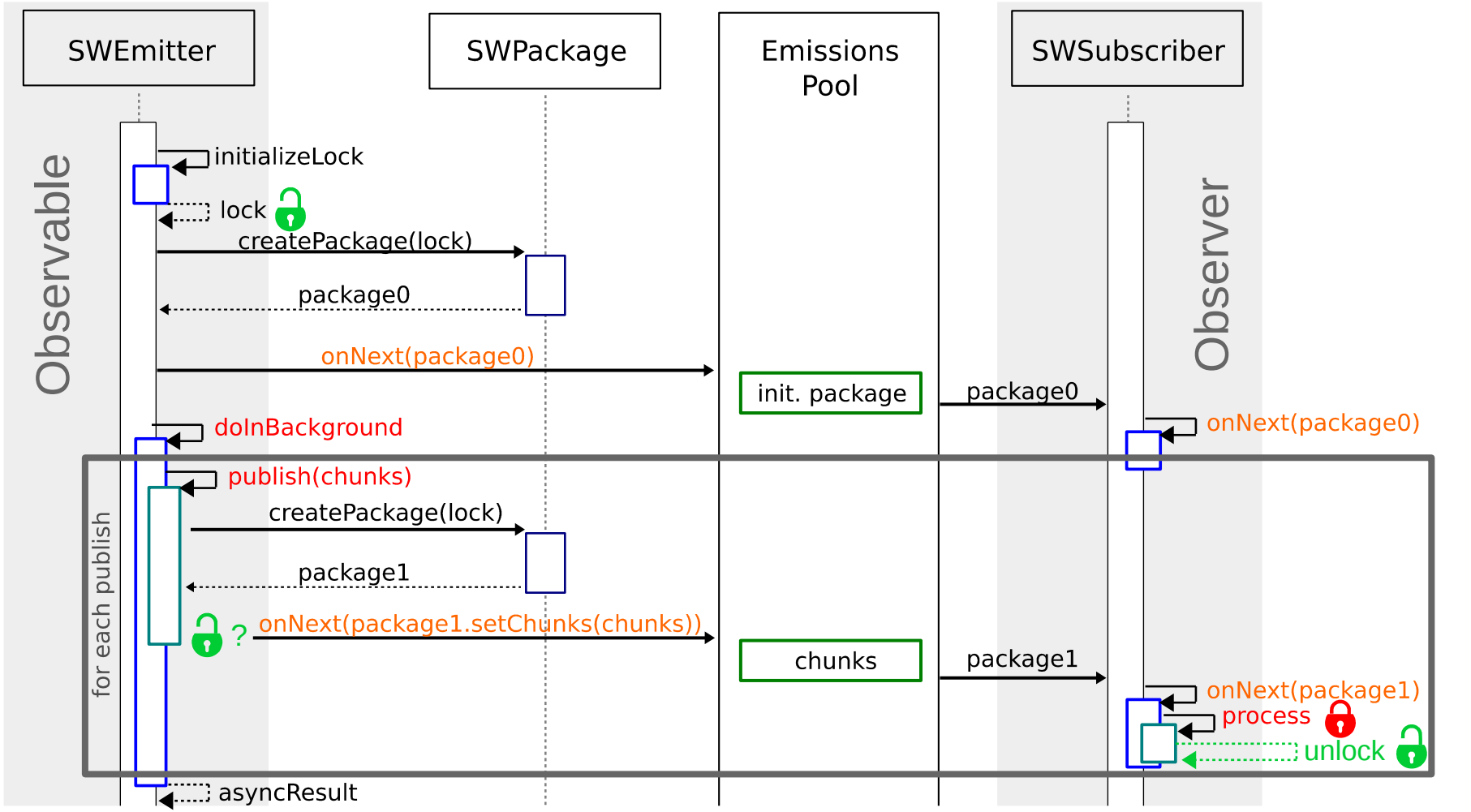
- Method Invocations

```
1  swingWorker.cancel( true );
2  swingWorker.execute();
3  swingWorker.run();
4
5  swingWorker.isCancelled();
6  swingWorker.getState();
7  ...
```

```
1  rxObserver.cancelObservable( true );
2  rxObserver.executeObservable();
3  rxObserver.runObservable();
4
5  rxObserver.isCancelled();
6  rxObserver.getState();
7  ...
```

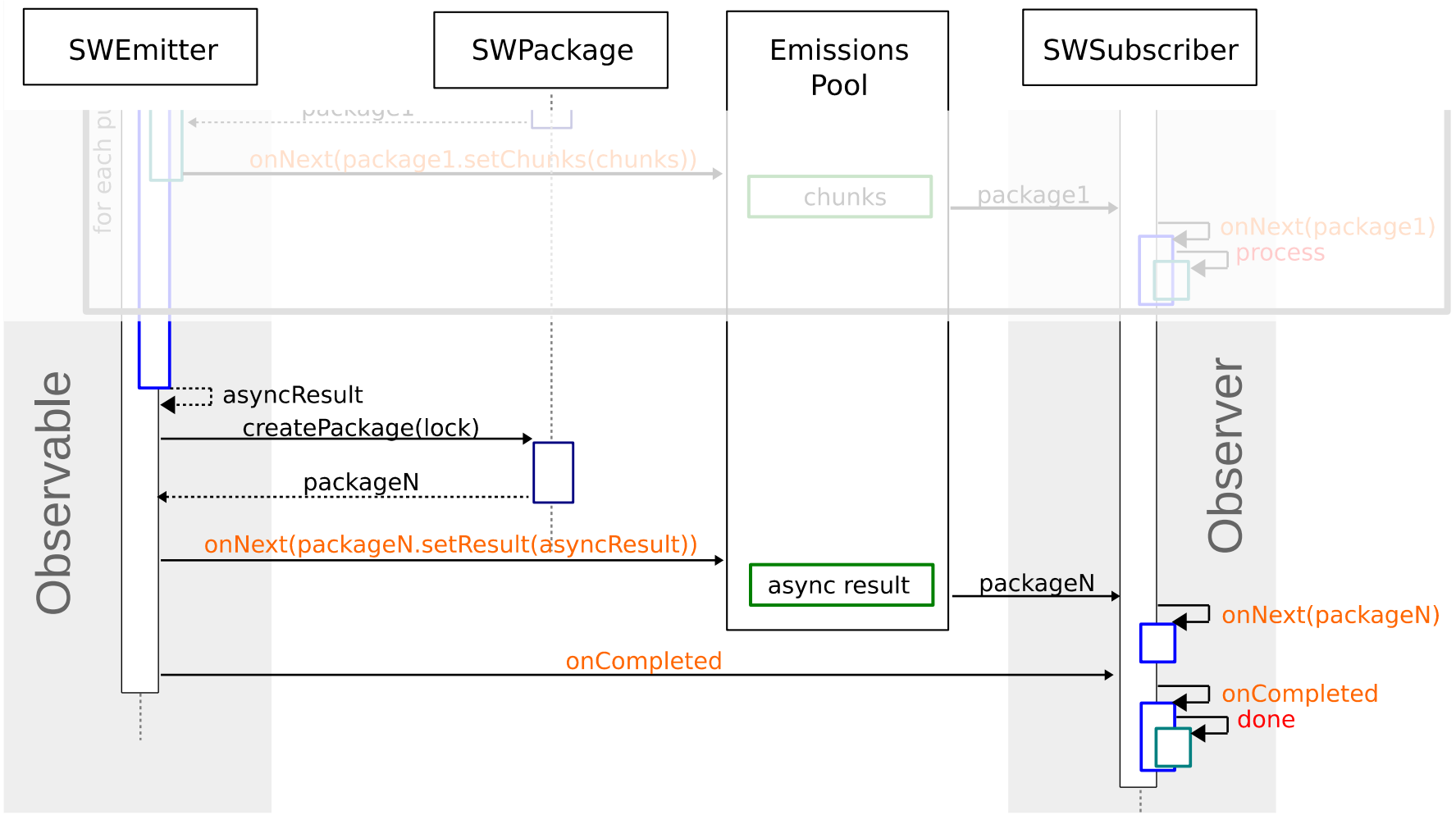
RxJava Extension for SwingWorkers (I)

← Questions



RxJava Extension for SwingWorkers (II)

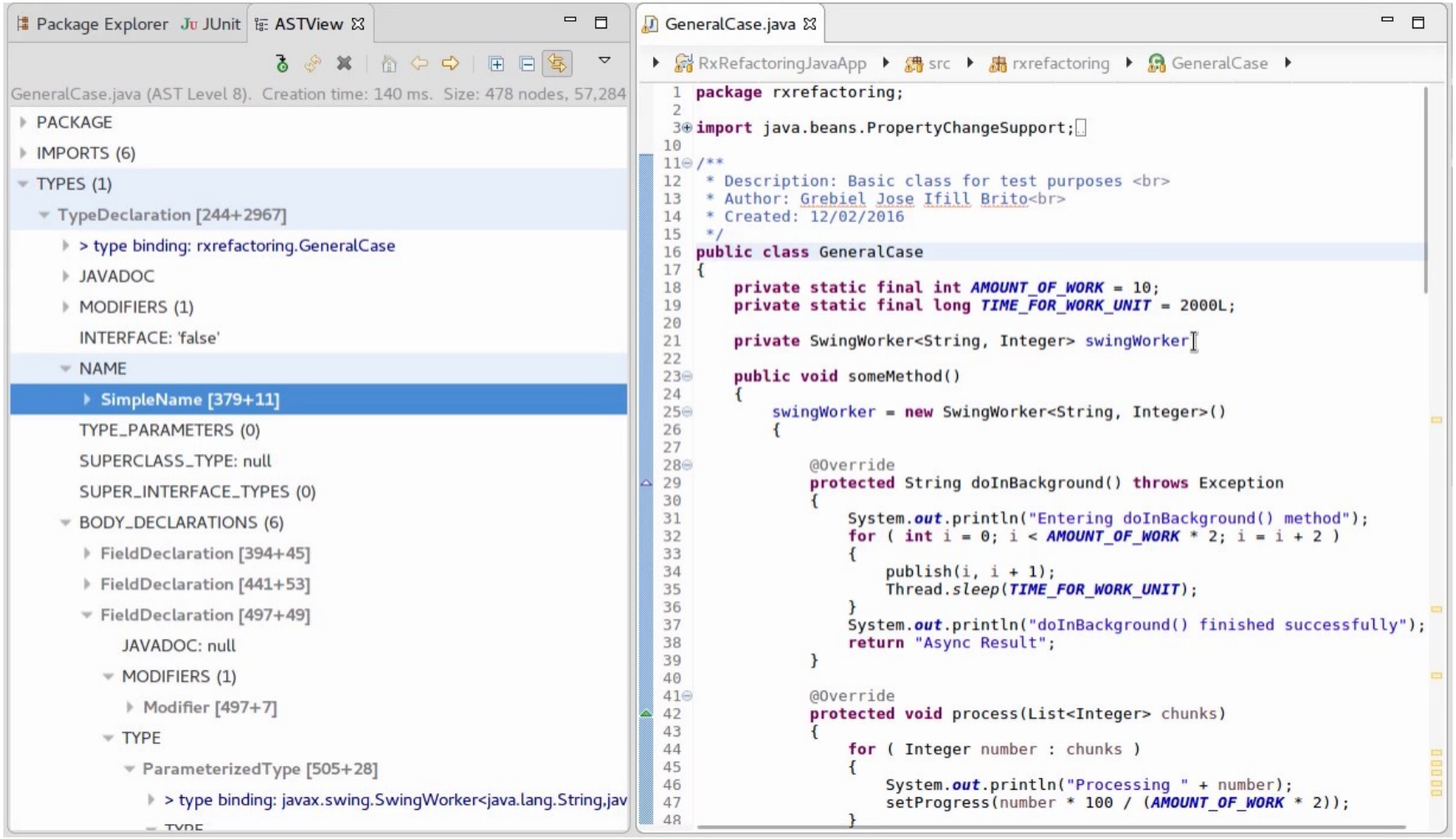
← Questions



2Rx & SwingWorker2Rx

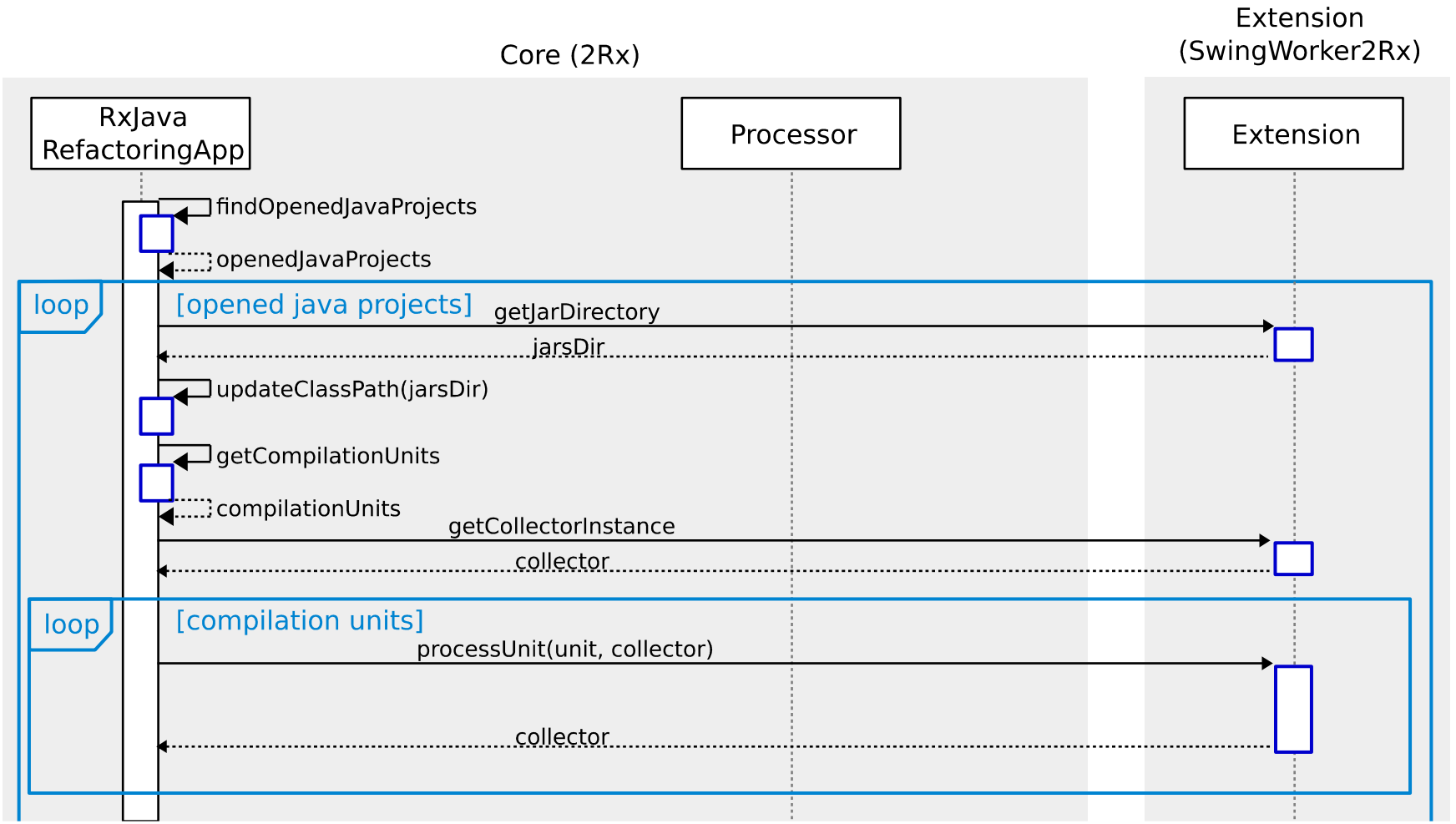
ASTNodes - ASTView

← Questions



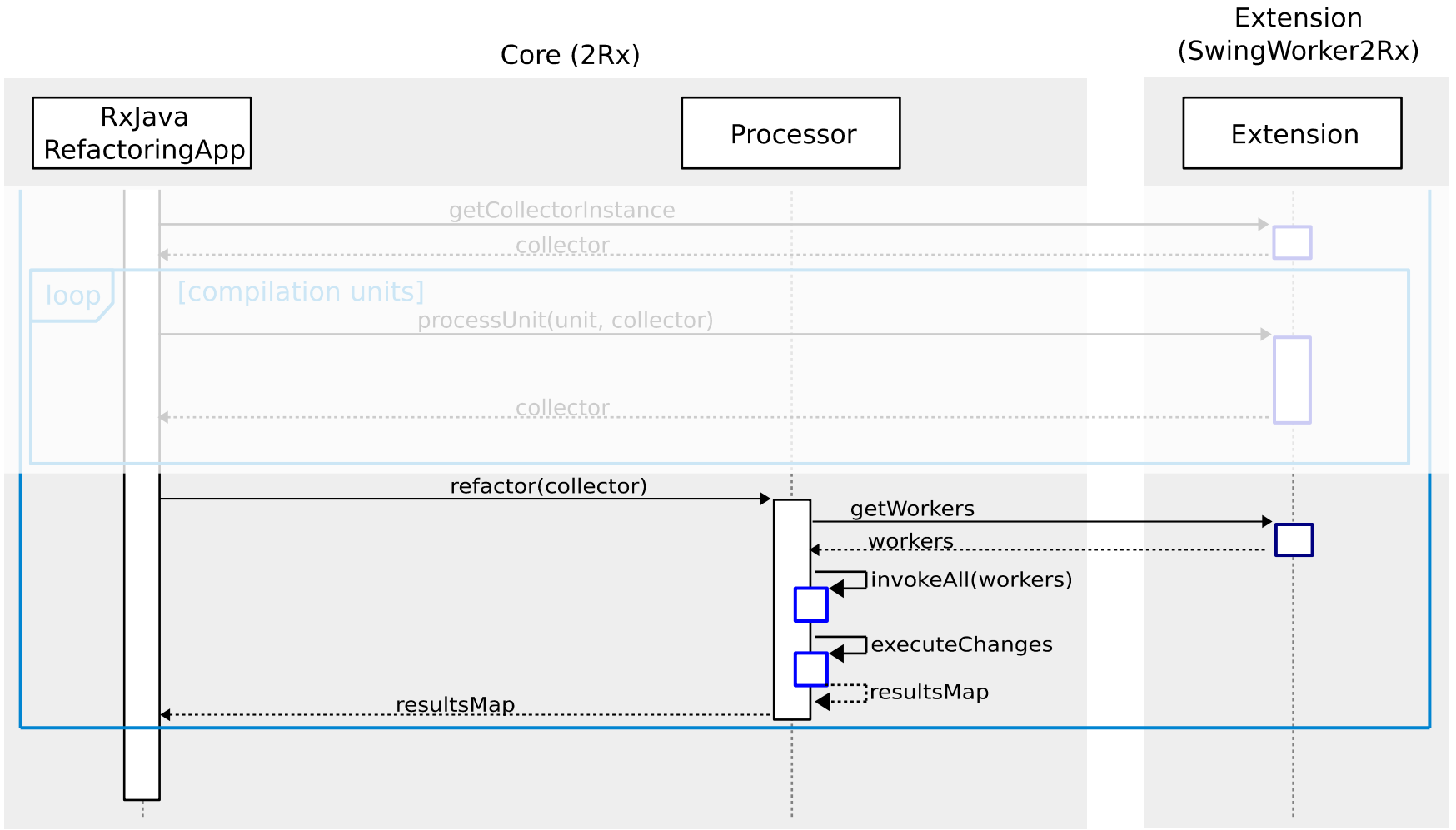
2Rx & SwingWorker2Rx

← Questions



2Rx & SwingWorker2Rx

← Questions

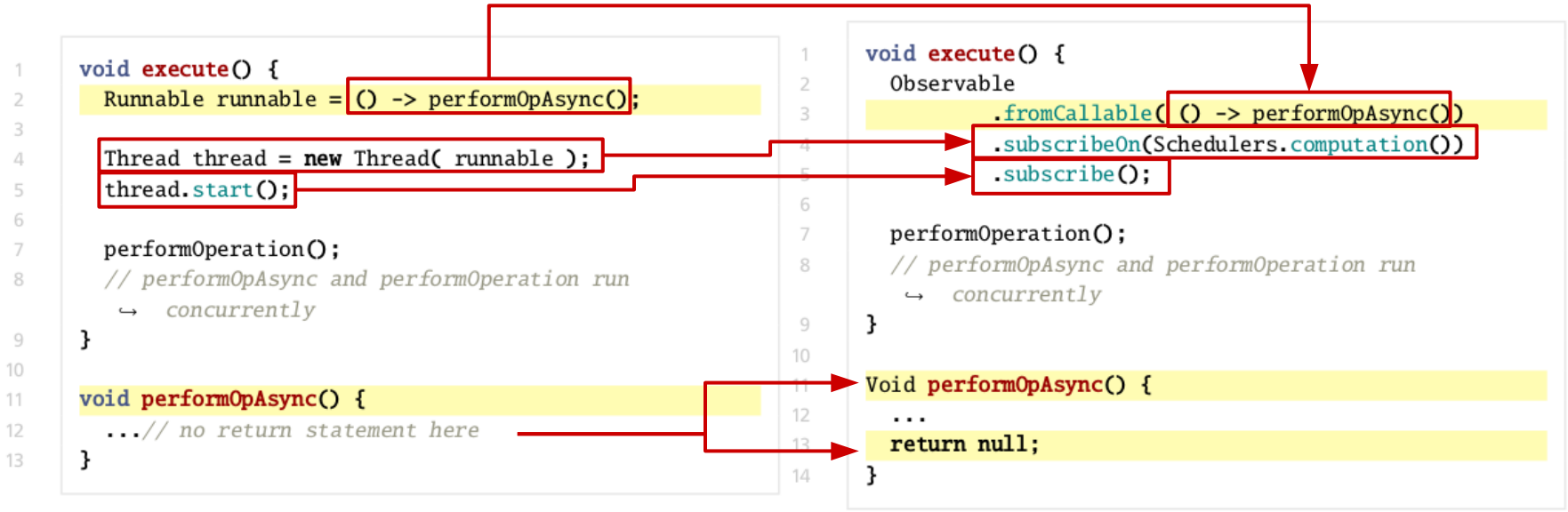


Future Work

Future Work

← Questions

- Further async constructs



Future Work

← Questions

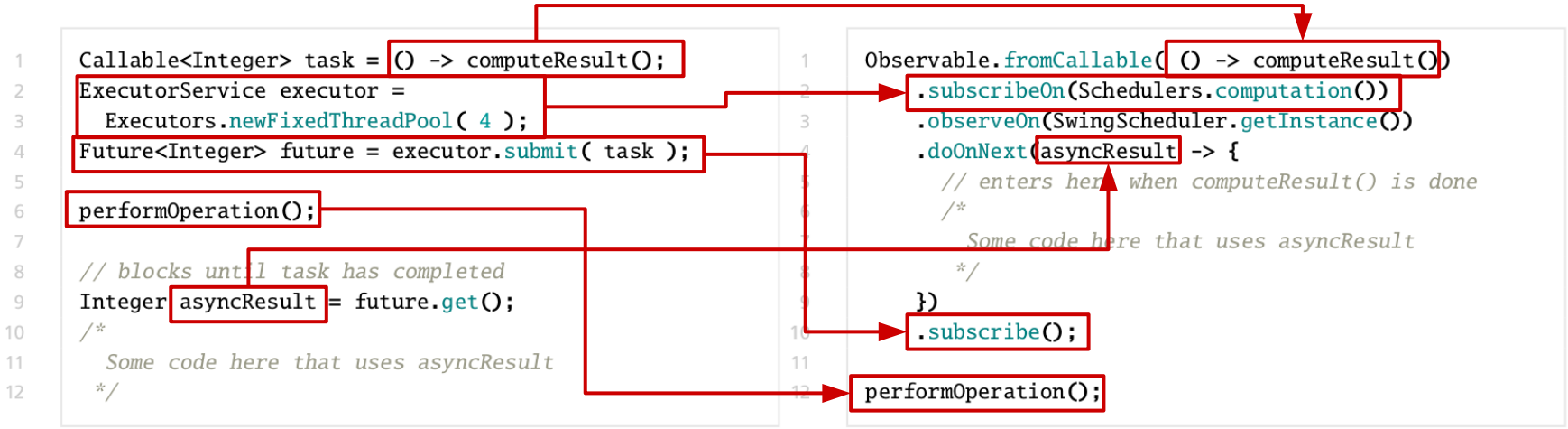
- Further async constructs

```
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 }
```

Future Work

← Questions

- Further async constructs



```
Callable<Integer> task = () -> computeResult();
ExecutorService executor =
  Executors.newFixedThreadPool( 4 );
Future<Integer> future = executor.submit( task );

performOperation();

// blocks until task has completed
Integer asyncResult = future.get();
/*
  Some code here that uses asyncResult
 */
```

```
Observable.fromCallable( () -> computeResult())
    .subscribeOn(Schedulers.computation())
    .observeOn(SwingScheduler.getInstance())
    .doOnNext(asyncResult -> {
      // enters here when computeResult() is done
      /*
        Some code here that uses asyncResult
       */
    })
    .subscribe();

performOperation();
```

Future Work

← Questions

- Java 8 and Functional Programming

```
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));
```

```
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))
        .doOnNext(p -> productDao.save(p))
        .subscribe();
```

Future Work

← Questions

- Java 8 and Functional Programming

```
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();
```

Future Work

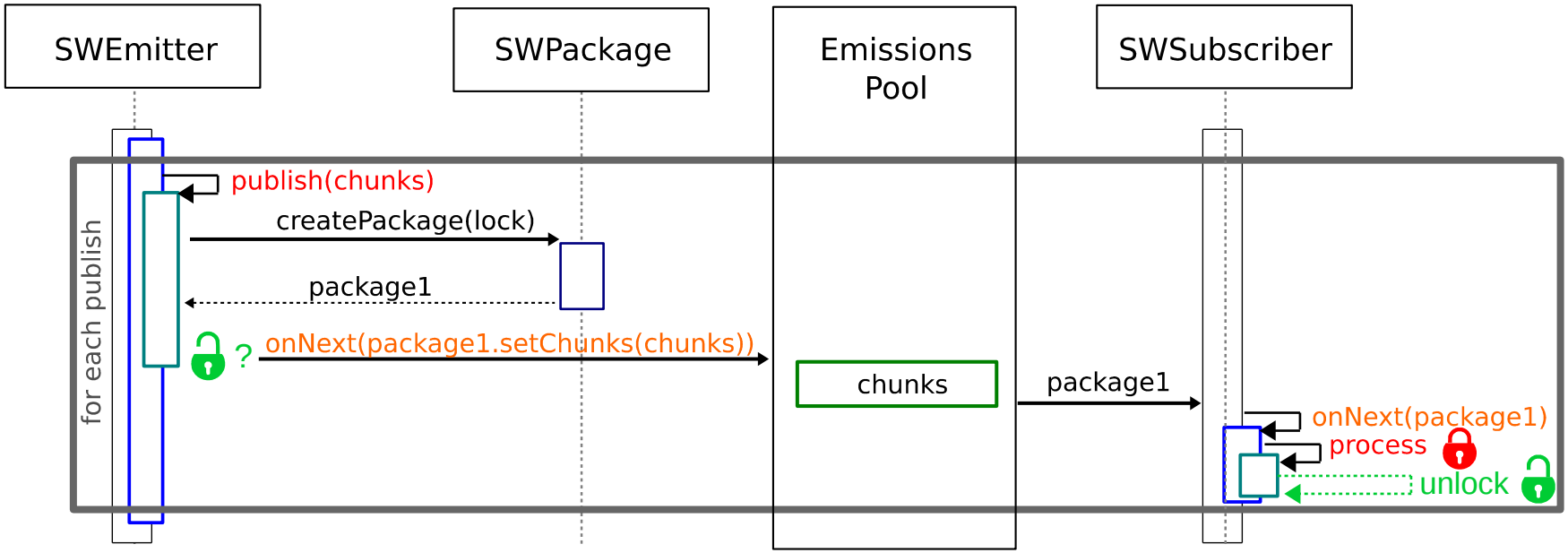
← Questions

2Rx

- UNDO feature
- Cancel or Stop button
- Settings
- ...

RxJava Extension for `SwingWorkers`

- Implement buffer to avoid blocking the `Observable` on each `publish` invocation.



Limitations

Limitations

← Questions

Nested `SwingWorker`s

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

Another SwingWorker here

```
rx.Observable<SWPackage<String, Integer>> rxObservable
 ↪  = rx.Observable
    .fromEmitter(new SWEmitter<String, Integer>() {
      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() {...}
};
```

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

SingleRun: instance of SWSubscriber

Limitations

← Questions

Name clashes

Originally a SwingWorker

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