Spoon Report

Bailleul Quentin, Douylliez Maxime
November 5, 2014

CONTENTS

1. INTRODUCTION

Nowadays, IDE are intensively used to develop programs. IDE allows the developer to design better programs by using a set of useful tools while staying in the same environment. IDE usually includes refactoring functions as a tools to speed up code manipulation safely.

While most IDE users don't use all implemented refactorings, Some refactorings are valuable enough to an user but not implemented in mainstream IDE. Moreover, there are some specific refactoring that can be very useful to a specific project, but it's up to the project Team to create such a refactoring. What tools a developer can use to create his own refactoring in Java?

This project aims at developing a refactoring that converts Java source code using JDK collection to a source code using Guava collection.

2. CONTEXT

Spoon has been used to create refactor while Guava is the library that will replace the JDK one for a small portion of collection part. Both will be presented in this part.

2.1. WHAT IS SPOON ?

Spoon is a framework for the analysis and transformation of Java source Code. It can be used to write specific analyses and transformation far more easily than with other tools. Spoon user writes his domain specific analysis and transformation in Java , avoiding to learn another language to process Java source code.

Spoon API allows to manipulate an abstract tree syntax containing all meaningful informations of a Java source code in order to extract informations, edit, remove, add part of Java code and generate modified source code.

Spoon project site can be found at `http://spoon.gforge.inria.fr/`

2.2. WHY GUAVA ?

Guava is the open-sourced version of Google's core Java libraries. Libraries let you use easily a set of functions that you don't need to write and test. Guava have been heavily tested and optimized by Google's developers and is used intensively.

The JDK Collections API, is widely used amongst developers and has simplified collections use. Guava aims to make working in the Java language faster and easier by creating new classes or working around JDK classes.

For instance, The JDK offers to create a unmodifiable collection from a mutable collection. But it actually returns a view of the original collection. Change the mutable collection and the unmodifiable collection will be too. Developers often use unmodifiable collections as a base to create immutable collections. Guava offers you to create truly immutable and distinct version of a collection from a mutable collection or directly by instantiation.

For instance,this is how to create an immutable List with JDK then with Guava.

```
List<String> COLOR_NAMES = new ArrayList<String>();
COLOR_NAME.add("red");.....

List<String> COLOR_NAMES_UNMODIFIABLE=Collections.unmodifiableList(
COLORNAME);

//delete all reference to COLOR_NAMES...

Guava
ImmutableSet<String> COLOR_NAMES = ImmutableSet.of(
"red",  "orange","yellow",  "green",  "blue",  "purple");
```

Guava pushes interesting part of JDK further in order to make the life of a java developer smoother.

Guava libraries can be found at `https://code.google.com/p/guava-libraries/`

3. CONTRIBUTION

3.1. BOUNDARIES

Doing the entire refactoring can represent a huge amount of work to be done smartly because several lines of "JDK code" can be converted into a single line of "Guava code".

Furthermore, due to the multiple manners (good or esoterics) to write a process logic, determine the complete set of code patterns that ‘ do something’ can be unproductive.

That’s why refactoring code patterns have to accept a certain level of inaccuracy , discussed in pattern code detections part.

3.2. REFACTORING ARRAYLIST AND LINKEDLIST INSTANTIATION

This part introduces the use of Spoon abstract syntax tree by changing the way ArrayList and LinkedList are instancied. These transformations are easy and the Guava resulted is of little interest when java 7 or over is used. Theses functions are quoted as "Not all Guava features have much utility (see e.g. Lists.newArrayList)",cf (`https://code.google.com/p/guava-libraries/wiki/PhilosophyExplained`)

Before Java 7, compiler could not infer of parameterizable classes instantiation types. for instance:

```
List<TypeThatsTooLongForItsOwnGood> list = new
ArrayList<TypeThatsTooLongForItsOwnGood>();
```

Guava lets you use an easier way by writing:

```
List<TypeThatsTooLongForItsOwnGood> list = Lists.newArrayList();
```

Now by using Java 7 or over you can write:

```
List<TypeThatsTooLongForItsOwnGood> list = new ArrayList<>();
```

We get all CTvariables then we extract CtExpressions of them. if the CtExpression if an ArrayList or an LinkedList, then we create a new CTExpression containing the new instanciation, using the following code:

```
CtExpression<LinkedList<java.lang.String>> value =
getFactory().Code().createCodeSnippetExpression(s);
```

We replace the old expression by the new one and the transformation is done.

To go further into refactoring, we need to use code patterns analysis, in the next subsection.

3.3. CODE PATTERNS DETECTIONS AND TRANSFORMATIONS

This part focus on code patterns detections and transformations in order to translate it to a single Guava call. It offers a glimpse of what Spoon is capable of.

In Java, with JDK, you have to create a List, then you can add elements inside:

```
ArrayList<String>() list = new ArrayList<>();
this.list.add("First");
this.list.add("Second");
this.list.add("Third");
this.list.add("Fourth");
```

Guava offers a more pleasant way to do this:

```
ArrayList<String>() list =
Lists.newArrayList("First","Second","Third","Fourth");
```

The major issue encountered during analysis phase is that you might encounter this code to analyse:

```
ls = new LinkedList<Integer>();
ls.add(99);
ls.removeFirst();
if(randomVar.isTrue()){
  AnotherRefOfls.destroy();
}
ls.addFirst(45)
ls.add(1)
```

Check Statement and possible interactions with processed list by another references of them is another pain.

Check three types of invocation on a same list that might influence the final order of parameters is quite hard to implement and should not be encountered in well written classes.

"Well written class" is blur itself, so there is the worst case where our processor will preserve initial programming logic:
In the constructor only, you can create as many lists as you want to, then only use as many list.add () as you want with other variables instructions. Within these other instructions do not interact with processed lists.

In the Appendix, there is an example of most complex Java Source code successfully processed by our processor.

The limitation "Constructor only" come from different Methods and Constructor representation in Spoon metamodel. CtConstructor and CtMethod have to be processed in two different classes in our implementation.

4. CONCLUSION

By analysing and transforming Java code source with Spoon, we make the first move toward a full migration from JDK collections to guava equivalent. Creating an API to convert from Java to guava would need a lot of work and intelligence to be really useable on big project. Google could afford to spend months to create such a converter, increasing attractiveness of their libraries even more.

A. Appendix

```
ls = new ArrayList<String>();
fal= new ArrayList<String>();

ls.add("Hello");
anotherVar2.add(1);
fal.add("Love");
System.Out.println("i do not break analysis")
ls.add(Mister);

var.add("hello3");
fal.add("chocolate");
another.add(2);
```

Should produce:

```
ls = Lists.newArrayList("Hello","Mister","Monperrus");
fal= Lists.newArrayList("Love","chocolate");

anotherVar2.add(1);
System.Out.println("i do not break analysis")
var.add("hello3");
another.add(2);
```