M2 IAGL – OPL

Subject #E2: Implement a migrator from java.util.collections to Google Guava

Code transformation

with Spoon librairies

Maxime Chaste

Pierre-Philippe Berenguer

Introduction

Guava is a set of librairies from Google for Java. It adds new methods to java.util.collections objects. These methods are static and could be easily invocated on a List, ArrayList or any JDK collection. Guava also provides collection types that are not in the JDK. They raised many interests and great performances. But these new structures are out of scope for this project.

Indeed, the main job is to parse standard Java programs with Spoon. We have to search for java.util.collections occurrences. When encountered we tends to replace JDK method invocations to Guava ones.

During the project we succeed to modify basic structures like List for example. The Spoon librairy needs less code than expected at the beginning of the project. Our transformation, relies on an unique lightweight Spoon processor helped by « Replacer ». When processing source code, a simple Hashtable is used to apply concordance between JDK types and Guava. Then Replacer tools return Guava element to the Spoon Processor.

We will focus on the way followed in our project to solve the migration from SDK collection to Guava.

We assume that you have few knowledges about Spoon.

Approach

Main transformation relies on a Spoon Processor :

```
« class CtVariableProcessorDispatcher extends
    AbstractProcessor<CtVariable<Object>> »
```

The mechanisme we introduce starts with specifying witch kind of CtElement to process. An ArrayList as class attribute is used to store these elements.

In the init() method inherited from AbstractProcessor, we specify two subclass to be concerned by transformation : CtFieldImpl and CtLocalVariableImpl. Every transformations will be set upon these two types of element. The advantage is to specify definitively, at the beginning of Runtime, the elements to search when browsing Spoon AST.

The second method used is « isToBeProcessed(CtVariable<Object> candidate) ». It looks for a candidate – CtElement, every times Spoon analyze an AST node. Spoon allows an CtElement to be processed if « isToBeProcessed » method return true. So, if the element has been added during init(), then we allow him to be processed by returning true (Code : candidate).

```
public boolean isToBeProcessed(CtVariable<Object> candidate) {
  return elements.contains(candidate.getClass().getSimpleName());
}
```

Code : candidate

The third method, is « process() ». We transform this element if concordance with Guava has been provided. It observe the CtVariable type visited by Spoon.

For this purpose we use an Hashtable. Key are the JDK collection types, for example : « java.util.collections ». Each key of a collection type in the Hashtable got a tranforming tool named « Replacer » as value.

Spoon transformation

Replacer tools

The transformation relies on the CtVariableProcessorDispatcher discuss in the previous section. The term « Dispatcher » in the class name refers to the use of differents tranforming tools. As said above, when « java.util.list » is encountered during process() method, it should be turned into the Guava syntax. Here for java.util.list we will use the corresponding and specific tool implemented : the « ListReplacer ».

Concretely, the processor will dispacth a JDK transformation in process() to the corresponding Replacer. Like collection types, all Replacer should have been set in the init() method (Code : Hashtable Replacer).

```
        this.processorsTable = new Hashtable<String, Replacer>() {
          {
              put("java.util.List", new ListReplacer());
              put("java.util.ArrayList", new ArrayListReplacer());
              put("java.util.Map", new MapReplacer());
              put("java.util.Set", new SetReplacer());
              put("java.util.SortedMap", new SortedMapReplacer());
          }
      };
```

Code : Hashtable Replacer

Thus, we can check method invoctations to any collection type with its distinct tool : these is the purpose of our Replacer. It prevents from confusing code with multiple case of tranformation in process() method.

If the Processor find a CtVariable type corresponding to « java.util.list », then ListReplacer is called. Overall, Processor calls the Replacer set as value for a collection type. Exchanges between Processor and Replacer are the CtVariable as string.

The method invocation is then checked into « ListReplacer ». Any Replacer called, verifies if a method invocation in JDK got a transformation. The Replacer relies on the string to check if a transformation exists. For this job, we compare CtVariable starting string with a string model to Guava. If the comparision works we return the corresponding

Guava code. However, when CtVariable string isn't known by the Replacer, it is retrun has the same. The example behind shows JDK invocation and the specific alternative with Guava :

« java.util.Collections.unmodifiableList(myList) ; » will be replaced by Guava :

« com.google.common.collect.ImmutableList.copyOf(myListe); »

The yellow highlights above corresponds to dynamic values. They won't be identic in different process. To solve this, we copy the argument from original invocation into the Guava invocation. It simply rely on extracting string (argument) between parenthesis.

Finally, the output of the Replacer will be the correct Guava invocation.

Snippet

```
1        Replacer replacer = this.processorsTable.get(type);


2       if (replacer != null) { // when processor exists use it
3          element.setDefaultExpression(getFactory().Code()
                  .createCodeSnippetExpression(
4                         this.processorsTable.get(type).replace(element)));
        }
```

Code : snippet transformation

Back to CtVariableProcessorDispatcher, the code above comes from process() method. Let's have a look into process() method (Code : snippet tranformation) :

1. get the Replacer corresponding to the CtVariable type (encountered in AST)
2. transformation will be processed only if we got a Replacer for the current CtVariable
3. use snippet on the CtVariable named « element » for tranformation
4. replacement : first get the Replacer for the given CtVariable type in Hashtable, then invocat its replace method (has seen previously, a new Guava syntax will be returned if it differs from JDK collection)

PatternProcessor

With this PatternProcessor we search into each method, with an CtMethodImpl<Object>, if code looks like :

```
ArrayList<String> outArr = new ArrayList<String>();
      outArr.add("un");
      outArr.add("deux");
      outArr.add("trois");
```

Then we change them to get only one line :

```
java.util.ArrayList<java.lang.String> outArr =
com.google.common.collect.Lists.newArrayList("
un", "deux", "trois");
```

To do that we start by changing our CtMethodImpl<Object> on a List<CtStatement>. With this list we can navigate into the method code lines.

Then we have some stapes to get our result :

1 get all the lane where we find a new arrayList declaration and store their index on an arrayList ; here this list is called listNewArray

2 for each new arrayList we count how many lane contains code lines which looks like "ArrayListName.add" ;
each time we find a lane like that we keep argument and add this one on our String which contains our new lane code

3 we delete the old lane ; then put the new instead

4 if an other "new declaration" exists, we recalculate the new index for the next "new declaration" then go 2) else go 5)

5 change the List<CtStatement> by the new one

This pattern can be improve if we do that for all change , currently if we've got something like :

```
ArrayList<String> outArr = new ArrayList<String>();
      outArr.add("un");
ArrayList<String>outArr2 = new ArrayList<String>();
      outArr2.add("un");
      outArr.add("deux");
```

… We will only get :

```
java.util.ArrayList<java.lang.String> outArr =
com.google.common.collect.Lists.newArrayList("un");
java.util.ArrayList<java.lang.String> outArr2 =
com.google.common.collect.Lists.newArrayList("un");
outArr.add("deux");
```

We can also imagine a pattern which will check all the methods to determinate if the ArrayList is immutable or not but currently.

Validation

The validation is manual. We have started to build a short class with simple « java.util.collections » types. When running the CtVariableProcessorDispatcher we succeed to recognize JDK collection with the process() method.

The next step have been to validate tranformation for « java.util.list » inputs. When it officialy work correctly we expand model to other collection types. Thus, we tried to mix them (different types in the same class), and the « spooned » result shows exact expectations. Here is one Guava successful transformation :

Input	Output – Spooned
`List<String> unmodifiableList = Collections.unmodifiableList(arr);`	`List<java.lang.String> unmodifiableList = Collections.unmodifiableList(arr);`

Tab : spooned result example

After that, we have compiled by hand spooned files and original ones (with standard JDK). No compilation errors appears. And, we obtain identical results when running these two Java class.

Discussion

We have tried as possible to make the processor lightweight for comprehension, and density of code. In opposite, these class are not « Fined grained object ». In fact, we just wants to put transformations in distinct objects with their respective collection scope.

Whereas using « if then else » or « switch case », our CtVariableProcessorDispatcher relies on an Hashtable. It references multiple Replacer. These Replacer used for transforming a particular JDK collection type, are instanciate once. In our tests we « Spooned » simple input files with few lines of code. Maybe all Replacer we instanciate could be not usefull in our own test during the project.

However, we thought that in real voluminous projects the majority of the JDK collections should be found. It would be better not to instanciate every time the same Processor for a type encountered multiple times.

After checking[1] on internet, we have also validated that an Hashtable will be faster and

1 http://www.derrickwilliams.com/?p=13

better for referencing possible tranformations (Replacer).

Difficulties

Analysis of source code with Spoon is easy to understand. When extending AbstractProcessor we match correctly a specific CtElement. However, first attempt in the project faced difficulties to transform this CtElement.

Multiple attempts use : « snippet.setValue(…) ; » for example, but with no concrete results. We use many different possibilities to transform something in process(). And, our solution has come from this snippet usage:

```
element.setDefaultExpression(getFactory().Code()
                  .createCodeSnippetExpression(
                          ...);
```

Further works

- Automate tests
- Add migrator to Apache Common Collection, by adapting or merging with other models
- Parse more CtElement, and try to transform different pattern usages of collections to tranform them into a Guava way

Conclusion

Spoon visits source code and builds an AST. When visiting node we tell our CtVariableProcessorDispatcher to take care only on CtVariable. More precisely, only CtFieldImpl, CtLocalVariableImpl are concerned.

When the process() method of CtVariableProcessorDispatcher leads to a Spoon type, we have to translate it to the source code type. It is how we could got Java type. This type is searched into an unique Hashtable. When, the type is found as key, we get its specific « Replacer » set as value.

The replacer is a class defining translation from java.util.collections to Guava for a precise type. We develop Replacer for : List, ArrayList, Map, Set, and SortedMap.

Then, the process() method witch have called correct Replacer got the translation to Guava. Here a snippet from Spoon lets us exchange SDK collection type or method with the similar one in Guava.

We succeed to convert Collections objects and validate results manually. Automated tests could be better in the futur, and we also need to test project on wide source codes. None of the efficient, speed, bugs have been rised or evaluated.

For pattern migrator, our first attempt, succeed to replace multiple lines for adding object in an ArrayList, by only one line with a Guava method. To conclude, an interesting improvement could be, to transform larger patterns of Collections from creation, update, or invocation, into a better Guava form.

References

- Spoon : https://gforge.inria.fr/frs/?group_id=73
- Guava utility classes : https://code.google.com/p/guavalibraries/wiki/CollectionUtilitiesExplained
- Guava doc : http://docs.guava-libraries.googlecode.com/git-history/release/javadoc/index.html
- Hashmap interest : http://www.derrickwilliams.com/?p=13