Practical Solution For Automation Of Libraries Migration In Java Code

Guseyn Ismayylov, 7th December 2020

1 Abstract

Migration of libraries and other transformations in the code is a huge percentage of work that every programmer does most of the time. And in the big companies the size of legacy reaches unimaginable numbers. This is why problem of automation such work is so tempting.

In this report we will discuss two possible solutions that can be practically implemented to solve the problem:

- Mining migration patterns from open sources of existing code
- Creating manually specific migration scenarios from initially convenient structures that can be used in there.

Also we are going to make a conclusion (as a result of research of the current problem) about why second approach is more efficient in terms of practical implementation and application.

2 Problem

Let's say we have a Java program with **.pom** file as configuration file with dependencies and we use functionality from those dependencies in our program. Now, we need to update such dependencies and update them to specified versions or even to other libraries automatically. And of course, we also need to update Java code correspondingly and it must compile.

So, on input we've got

- Folder with Java code
- **.pom** file with dependencies
- Specified library/libraries where we want to migrate from
- Specified library/libraries where we want to migrate to

And as output we need to get:

- Updated **.pom** file with dependencies
- Updated valid Java code in the original folder

3 Related works

3.1 MigrationMiner: Mining patterns from open source

First solution that was found is **Migration Miner** by user **@hussien89aa** on GitHub. The idea is that we generate migration patterns from open source by list of links of repositories in GitHub that can potentially contain migrations. The list is stored in **.csv** file.

The logic of how this project works can be explained in following steps:

1. We collect diffs from git commits from the list with git repositories.
2. We parse different versions of dependencies in **.pom** file.
3. We create structures that describe java changes in each commit.
4. We iterate over java changes using new java imports in corresponding java files and using pom.xml file (to get version of libraries) to generate migration rules. In each iteration we check types and import + if needed new changes in dependencies of pom.xml versions (for each commit).
5. We apply criteria for Migration Rule(it can be describe as a pair of libraries before and after that are included in migration scenario): if we have new java imports in file and in changed lines of java code with complete statements where we remove all types of classes or scopes from one library and add types from another library (which we can get in new imports(or jdk) and pom.xml) we can 100% be sure that it's some migration rule. Also we can reduce expectations for the criteria and hope we can get good migration rules. But we do it if the criteria is failed to give big number of rules.
6. We build structures of migration rules. While we are getting migration rules, we can get name of classes and/or methods around core variables in migrations. For increasing efficiency we also need to load source code of libs from maven central or other repositories.

7. We get migration segments from code that are attached to migration rules that we found. Migration segments are consist of Java statements, which are parsed java structures so they can be applied on existing code base.
8. We build migration patterns for migration segments that can be applied for real code.

More detailed explanation you can find in this presentation.

I can list following found issues with this found library:

- Solution is not complete for all the problems: generating patterns, storing and also applying them on existing code.
- A lot of custom solution for common problems: working with network, GitHub, file system, parsing Java. And these solution are not stable and full of bugs. Instead we should use well proven solutions.
- Using MySQL(where all structures are stored) is forced, which is not flexible. Also there is no convenient API for data storage.
- Algorithms are slow and complex. For example, we need to do processing for all libraries, instead of moving from one library to another one.

3.2 Rewrite: Creating manually specific migration scenarios from initially convenient structures that can be used in there.

Another solution with completely different approach is Rewrite project.

The idea is that we create so called recipes in Java or Yaml format that consistent of visitors that apply changes (that we declare there) to specified java file/or folder with java files.

Detailed explanation of how Rewrite works you can find in the docs on Rewrite site above and in this presentation. In few words, logic can be describe in following steps:

1. We create Java visitors for java code transformations.
2. We can use visitors in refactor processor like in this example project.
3. We can create **.yml** definitions out of such visitors and use maven/gradle plugin to apply them on specified folder with code.

Considering the progress that the core team of Rewrite project has made and still making, it's decided to support them by creating example solutions that based on Rewrite, filing bugs and submitting feature requests instead of creating another solution.

4 Solution

Following first approach project Mea was created. But quite soon I realized disadvantages of mining patters:

- Difficult to get commits from open source that can be applicable for migration scenarios.
- Found applicable patterns are not complete and don't guarantee complete migration from lib "A" to lib "B" for all cases, all methods and so on.
- It's difficult to predict what kind of changes user wants to apply: either it's just migration or some smart replacement of code which can be hardly to achieve just by using regular expressions. In another words, user might have very specific path and approach for migration which most probably would not be covered from mining structures.
- There is no common approach for creating missing patterns, we just cannot force user to find some commits and it's better to provide an end user with some instruction on how to create such patterns just by typing examples of migrations.

This is why I decided to stop working on **Mea** and support **Rewrite** by submitting feature requests, filing bugs and creating following own projects: Rewrite JUnit-4 To JUnit-5, Rewrite Java Definitions. All these projects are based on usage of visitors from Rewrite Project(and some additional that I created).

5 Deliverables

Project	Lines of code	Time spent	Contributors
Mea	7565	40 days	@guseyn(GitHub)
Migrator	4216	50 days	@guseyn(GitHub)
Rewrite JUnit-4 To JUnit-5	2010	60 days	@guseyn(GitHub)
Rewrite JUnit-4 To JUnit-5	6356	40 days	@guseyn(GitHub)

6 Conclusion

I've done really huge research of the problem, found and created many interesting ideas and projects. I realized that mining patterns is useless activity in practical sense, this is why I help and support Rewrite project, because the approach that's been used there gives real practical results like this tool. I created example project that can help newcomers to understand how to use Java definitions. It helps open source community and Rewrite project itself because by creating such example projects I found many bugs in real practical cases. By this, I improve Rewrite project and motivate core team to work on the project and this helps us to solve this quite complex and ambitious problem.