Modernizing the Java PathFinder Build Workflow: Migrating from Ant to Gradle

Jeanderson Barros Candido
e-mail: jeandersonbc@gmail.com

August 23, 2022

1 Summary

Developers often perform recurrent tasks during the development process such as testing, managing external libraries, generating API documentation, and managing release artifacts. Build tools help to automate those error-prone and daunt tasks with scripts that abstract those tasks. This proposal aims to modernize the build workflow from the Java PathFinder (JPF) project by migrating from Ant to Gradle. Gradle is a general purpose build system and uses Groovy, a JVM language, to create flexible and highly customizable build workflows. To achieve this goal, my strategy is to have a working Gradle build coexisting with the current Ant script. Ant targets will migrate to Gradle tasks in an incremental and iterative process. By the end of the program, is expected to have the Gradle support fully integrated into the main repository.

2 Outline

The JPF project currently uses Ant to automate the build process. Unfortunately, Ant has drawbacks that hinder developer's productivity for sufficiently complex and large projects. In the context of the JPF project, there are two major issues:

1. **Lack of automatic dependency resolution.** The user needs to manually download and configure dependencies. Ant is often integrated with Ivy[3] as a complementary tool to handle external dependencies. On the other hand, Gradle[5], Maven[11], and other popular build tools resolve declared dependencies automatically out-of-the-box.

2. **Large and verbose script file.** XML has some drawbacks in the context of build automation. Tags are often long names, and, in particular, Ant targets may contain several attributes and nested elements to describe additional properties. For sufficiently large projects, it is challenging to maintain and evolve the build process due to the quick growth and the verbosity of the build script.

Many popular build tools provide features to address those issues. This proposal focuses on migrating from Ant to Gradle; it is relevant because the current build workflow may introduce barriers for maintenance and to newcomers. For further details on how and why I decided to migrate to Gradle, please, refer to the Appendix A.1.

3 Deliverables

- Gradle support on the `jpf-core` module;
- Gradle support on the `jpf-symbc` extension module;
- Updated version of the `jpf-template` auxiliary tool;
- Updated documentation related to the build process from the mentioned projects.
- A build migration guide for replication of this process in other JPF projects.

4 Strategy

The main strategy to succeed with this proposal is to have a working Gradle build since the beginning. This is possible due to the interoperability of Ant withing a Gradle execution [8]. Figure 1 demonstrates this feature with a simple example: the `build.gradle` script imports a simple `build.xml` file and the user can invoke the `foo` target as a Gradle task.

For this proposal, is not sufficient to only import the existing build script. The JPF core (`jpf-core`) and Symbolic PathFinder (`jpf-symbc`) modules have many tasks and dependencies that could be simplified with a simpler and verifiable syntax. Figure 2 illustrates the existing Ant targets and their dependencies. As we can see, critical paths involve mostly the compilation steps and must be a priority since they represent the longer paths on the build workflow. My strategy is to migrate each Ant target until the new build script supports all original targets as Gradle tasks. This is feasible because Gradle interchangeably handles Ant targets and Gradle tasks. Figure 3 demonstrates the migration from Ant to Gradle. The `compile` target, originally on the `build.xml` file, depends on the `init` target and it is a dependency to the `dist` target. After the migration, the `build.gradle` script imports the `build.xml` and implements the `compile` task. The new Gradle script performs the full build workflow despite being incomplete. This feature ensures that Ant and Gradle can coexist during the migration process.

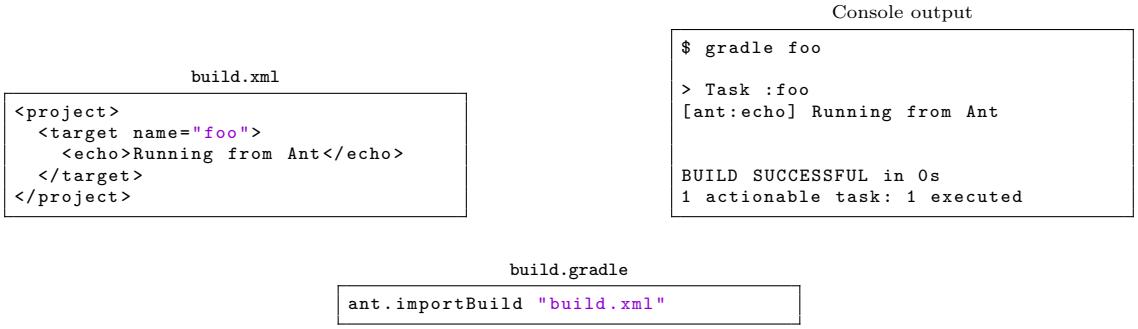
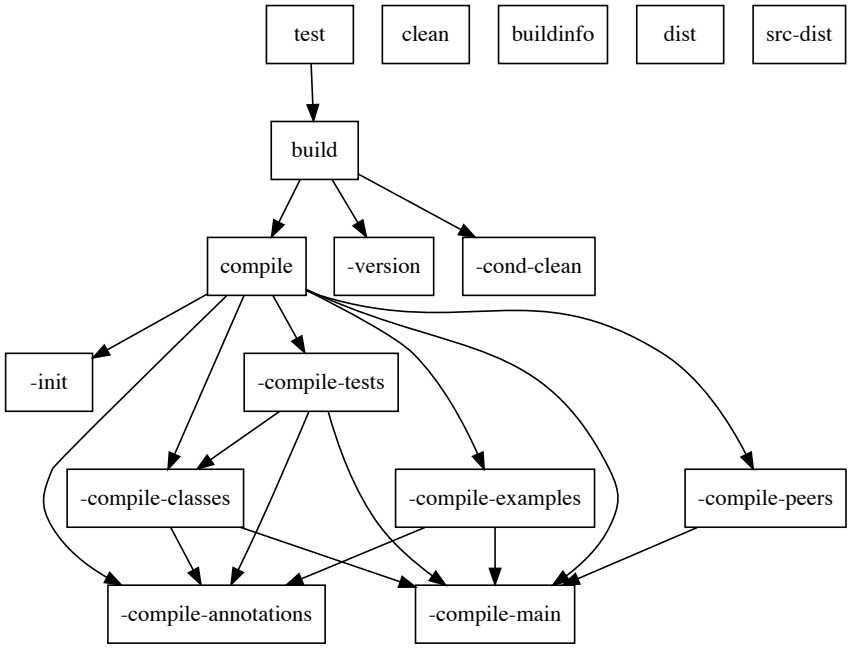
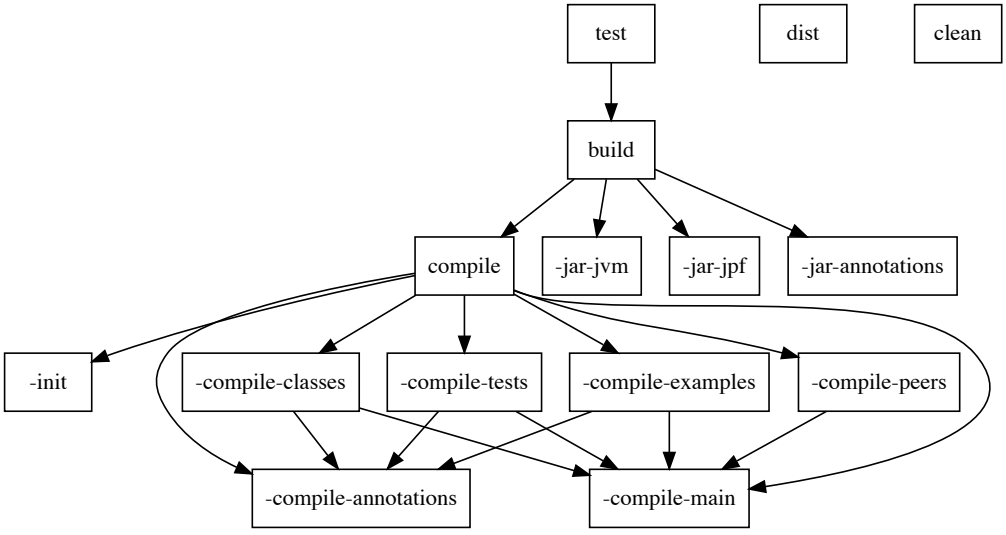


Figure 1: Demonstration of Ant integration with Gradle.

Source: the author.



(a) `jpf-core`



(b) `jpf-symbc`

Figure 2: Ant targets represented as directed acyclic graphs. The edges represent dependencies where the *tail node* depends on the completion of the *head node*.

Source: the author.

build.xml

```
<project default="dist">
  <target name="init">
    <echo>Running init task</echo>
  </target>

  <!-- to be removed from this file -->
  <target name="compile" depends="init">
    <echo>Running compile task</echo>
  </target>

  <target name="dist" depends="compile">
    <echo>Running dist task</echo>
  </target>
</project>
```

(a) Before migration

build.xml

```
<project default="dist">
  <target name="init">
    <echo>Running init task</echo>
  </target>
  <target name="dist" depends="compile">
    <echo>Running dist task</echo>
  </target>
</project>
```

build.gradle

```
ant.importBuild "build.xml"

task compile(dependsOn: "init") {
    doLast {
        println "Running compile task"
    }
}
```

(b) After migration

Figure 3: Ant and Gradle interoperability.
Source: the author.

5 Timeline

The following schedule describes the tasks related to the migration of the `jpf-core`, `jpf-symbc`, and `jpf-template` projects. Depending on the performance during the execution, there is a change to cover other JPF projects, as well. This will be discussed closely with the mentor during the program.

Community Bonding

`Apr/23 - May/13:`

1. Import `jpf-symbc` and `jpf-template` on GitHub.
2. Cleanup and review of the `jpf-core`, `jpf-symbc`, and `jpf-template` projects.

Coding

Phase 1

`May/14 - Jun/10:`

1. Implement scripts for acceptance tests of the `jpf-core` jar files[1];
2. Migration of the JPF's `compile` subtree (see Figure 2a);
3. Migration of the remaining Ant targets from the JPF project;
4. Update JPF documentation related to Ant tasks;
5. Elaborate a draft to the migration guide based on the experience.

`Jun/11 - Jun/15:` First Evaluation.

[1] This will be useful to compare jar files generated from Gradle and Ant

Phase 2

`Jun/11 - Jul/08:`

1. Implement scripts for acceptance tests of the `jpf-symbc` jar files as in the previous phase;
2. Migration of the Symbolic PathFinder's `compile` subtree (see Figure 2b);
3. Migration of the remaining Ant targets;
4. Update documentation related to Ant tasks;
5. Update the migration guide.

`Jul/09 - Jul/13:` Second Evaluation.

Phase 3

`Jul/09 - Aug/05`

1. Add Gradle support on the `jpf-template` project;
2. Final version of the migration guide;
3. Update `jpf-template`'s documentation;
4. Migrate changes from `jpf-core` to the main repository;
5. Migrate `jpf-symbc` and `jpf-template` to the Java PathFinder Github user.

`Aug/06 - Aug/14:` Final evaluation.

A Appendix

A.1 Build Tools Evaluation

There are several build tools available in the Java community. Maven and Gradle are two mainstream tools popular in Android and web development. SBT[10] is another build tool that is becoming popular, and it was suggested in the GSOC idea's list[12]. Which one fits better to JPF's needs? To answer this question, I evaluated Maven, Gradle, and SBT in respect to the following aspects:

- *Q1. How are dependencies managed?* Configuring paths and jar files manually can be error-prone. Ideally, the build tool would take care of paths and dependency versions automatically.
- *Q2. How brief and powerful is the build script format?* XML syntax may lead to overly verbose and large files for sufficiently large projects. Ideally, the chosen build tool offers a friendly and brief syntax and provides an easy way to create user-defined tasks.

Results

- *Answering Q1:* One of the features introduced by Maven was the automatic management of external dependencies. By default, Maven fetches jar files from the Maven Central Repository[2] and keep them locally. In addition, Maven allows access to other repositories[1] with minor configuration. Not surprisingly, Gradle and SBT not only adopted this feature but *are also compatible* with Maven repositories. Therefore, **all options are tied in this question**.
- *Answering Q2:* In this aspect, Maven inherits drawbacks from Ant since it is also based on XML. On the other hand, Gradle and SBT are different from Maven: scripts not only *describe* the build process but also *specify* how to perform tasks. Build scripts are actual code. In the case of SBT and Gradle, both are compatible with Java API. This feature opens many possibilities and conveniences to create custom build processes and user-defined tasks. Therefore, **SBT (Scala-based) and Gradle (Groovy-based) are both valid options**.

Conclusion

Maven is a mature tool with many advantages compared to Ant. However, the build script lacks *expressiveness* since it is also based on XML. SBT, on the other hand, empowers the developer by specifying the build process with real code. In addition, Scala features interoperability with Java. SBT is a promising tool but it is relatively recent. The first stable release is from February 9$^{\text{th}}$, 2018[9]. **Gradle is a mature tool and combines the best of Maven and SBT** including features like incremental building, a daemon for efficient execution[6], and also a convenient *wrapper* mechanism[7] for reproducible builds. One of the most important Gradle features that support directly this proposal is the interoperability with Ant build scripts[8][1]. **For these reasons, Gradle seems to be a better fit for the context of JPF.** A list of useful links and snippets to demonstrate some Gradle basics are available on GitHub[4] for further reference. Note that this evaluation is a sanity check and does not intend to be generalized to all contexts in Java development.

[1] Section 4 elaborates how this feature is used in this proposal

References

[1] MVN Repository. https://mvnrepository.com/.

[2] The Central Repository. https://search.maven.org/.

[3] Apache Ivy: The agile dependency manager. https://ant.apache.org/ivy/.

[4] Jeanderson Candido. Gradle Labs. https://github.com/jeandersonbc/gradle-labs.

[5] Gradle Build Tool. https://gradle.org/.

[6] Gradle User Manual. The Gradle Daemon. https://docs.gradle.org/current/userguide/gradle_daemon.html.

[7] Gradle User Manual. The Gradle Wrapper. https://docs.gradle.org/current/userguide/gradle_wrapper.html.

[8] Gradle User Manual. Using Ant from Gradle. https://docs.gradle.org/current/userguide/ant.html.

[9] SBT. Official Releases. https://github.com/sbt/sbt/releases/.

[10] SBT: the interactive build tool. https://www.scala-sbt.org/.

[11] The Apache Maven project. https://maven.apache.org/.

[12] The Java PathFinder Project. GSoC 2018 Project Ideas. http://goo.gl/PAVibm.