JaMaBuild: Mass Building of Java Projects

Matúš Sulír
Technical University of Košice
Košice, Slovakia
matus.sulir@tuke.sk

Milan Nosáľ
ValeSoft, s.r.o.
Košice, Slovakia
milan.nosal@gmail.com

Abstract

Many large-scale Java empirical studies require not only source code but also resulting binaries such as JAR files. Precompiled datasets quickly become obsolete, and the creation of a custom corpus for every study is tedious. We present a prototype of JaMaBuild, a tool and a framework for mass building of Java projects from source. Given a list of projects and optional settings, it downloads the projects, filters them by user-definable criteria, builds them using Maven or Gradle, and collects outputs such as JAR files and build logs. Our tool can also be used for local build breakage studies.

CCS Concepts: • **Software and its engineering** → **Compilers**; *Software configuration management and version control systems*; *Object oriented languages.*

Keywords: Java, build tool, Maven, Gradle, corpus

ACM Reference Format:
Matúš Sulír and Milan Nosáľ. 2023. JaMaBuild: Mass Building of Java Projects. In *Companion Proceedings of the 2023 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity (SPLASH Companion '23), October 22–27, 2023, Cascais, Portugal.* ACM, New York, NY, USA, 2 pages. https://doi.org/10.1145/3618305.3623613

1 Introduction

In programming languages and software engineering research, a corpus of Java projects is often needed. Automatically downloading the source code of many projects is fairly trivial. However, the same cannot be said about the resulting binary artifacts often required for certain types of analyses, e.g., JAR files with transitive dependencies. For this reason, multiple preconstructed datasets of binary Java projects exist, including Qualitas.class [4], XCorpus [1], and 50K-C [2]. However, due to the fast pace of progress, pre-built datasets quickly become outdated. Furthermore, the criteria that the authors used to sample the projects do not always match the researchers' requirements. Although NJR [3] provides a way to update the corpus, it does not utilize build tools such as Maven and is based on the plain `javac` compiler with a class file search on failures, which has its shortcomings, such as producing different JARs than the original projects' scripts.

If researchers decide to construct a corpus by themselves specifically for their study, it tends to take a disproportionately large part of the time, and the resulting ad-hoc scripts are often unreadable and error-prone. The generic process outline of project filtering, source building, and artifact collection tends to be the same, while only specific details differ. Our aim is to partially standardize their definition.

Therefore, in this paper, we present an early prototype of JaMaBuild – a mass builder of Java projects. It supports downloading the source code of projects from multiple sources, filtering them according to various criteria, building the projects using Maven or Gradle, saving the build logs, collecting the resulting binary artifacts, and other features.

We see two groups of envisioned users: i) Software engineering researchers in general will be able to easily construct a custom, up-to-date dataset of binary Java projects. ii) Researchers in the area of build tool breakage and repair will be able to collect and analyze local build logs and other artifacts.

The tool, including a screencast with a planned demonstration, is available at https://github.com/sulir/jamabuild.

2 Tool Description

JaMaBuild is based on the principle of "convention over configuration" as it defines the directory structure and file formats and also provides sensible defaults for all configuration options. A mandatory input file is a list of projects with one project definition per row. An optional YAML file contains various settings. Now we will describe the stages of a build process execution for one specific project in detail.

Container Starting. We isolate individual builds by running one Docker container per project. JaMaBuild provides a default Docker image, `sulir/jamabuild`, that contains Java Development Kit, Maven, Gradle, and basic Linux utilities. For cases when it does not suffice, users can create a custom image and define it in the configuration file.

Project Source Code Loading. Every line in the project list file consists of a tab-separated source type and a project identifier (URL). The source type defines how the project will be downloaded, extracted, or located. Currently, we support the "local", "git" and "github" source types. For example, in the case of "`local	google_guava`", the source code

SPLASH Companion '23, October 22–27, 2023, Cascais, Portugal

was already downloaded by an external tool and is present in the directory `projects/google_guava/source`. For "`github apache/commons-lang`", the project is cloned from GitHub into a directory with an automatically derived name.

Pre-Build Filtering. Empirical studies define inclusion and exclusion criteria for the analyzed projects. We suppose simple criteria, such as the creation date, were already applied using external APIs and tools before the construction of the project list, so we focus on the advanced ones.

Each criterion in the configuration file has a name and an optional parameter. Examples of currently supported criteria include: i) `BashScript`: A user-supplied script is executed in the container. The project is included or excluded based on the exit code. ii) `SourceFile`: A check if the source code contains a file matching the glob pattern. For instance, "`preExclude: ['SourceFile *.jj']`" excludes projects containing a file with the extension "jj". Text content inside the files can be searched too. iii) `AndroidSource`: We search for Java source code files containing the import of Android API.

In practice, many possible criteria exist. JaMaBuild thus also acts as a framework and a collection of reusable filtering criteria, where individual researchers can easily write their own in Java and optionally send us a pull request.

Project Building. Next, the project is built from source code using Maven or Gradle. JaMaBuild determines the used build tool based on the presence of a build configuration file in the project's root directory: `pom.xml` for Maven and `build.gradle` (`.kts`) for Gradle. If no such file is present, the build is considered failed. In the future, we might include fallback strategies, e.g., compiling all files directly using `javac`.

A command assigned to the given build tool is executed, e.g., for Maven, `mvn -B clean package` compiles the project and creates the resulting artifacts. Standard output and error are redirected to the build log file. After the process finishes, the name of the build tool and the exit code (representing success or failure) are written in the results file. Upon failure, we stop the container and continue with the next project.

By default, test execution is skipped as JAR files can often be analyzed even if one of the tests fails. If desirable, e.g., for test failure studies, tests can be enabled using an option.

The duration of one build is limited to one hour by default to prevent stalling. This timeout can be configured too.

Artifact Collection. After a successful build, the resulting JAR files are saved in build-tool-specific locations, such as multiple directories named `target` anywhere inside the project's source tree. Since consistency is one of the key properties of a software corpus, the resulting artifacts are moved to the `jars` directory inside the project's root.

Many analyses require the project's full transitive dependencies. Therefore, they are copied into the `deps` directory. Maven has a special goal for this, which we used with advantage. For Gradle, we copy files from the dependency cache.

Post-Build Filtering. Some inclusion and exclusion criteria cannot be evaluated until the project is successfully built. Thus another round of filtering occurs at this stage. Such filters include: i) `UnresolvedReferences`: Unresolved references in the JAR files cause problems, e.g., during call graph construction. We determine their presence using the `jdeps` tool. ii) `NativeMethods`: We find out if at least one native method definition is present in the project's classes and dependencies. iii) `BashScript`: Each criterion is marked with its expected usage phase (pre-/post-build). Some of them, including script execution, are generic enough to be usable for both stages.

Container Stopping. Finally, the container is stopped and the execution continues with the next project. Since the list of projects potentially contains thousands of items, we restart incomplete project builds in case of interruption.

3 Future Work

As an evaluation, we plan to re-create an existing corpus from one of our previous empirical studies with JaMaBuild. Instead of using a long and unreadable shell script, we will aim to rewrite it as a list of projects and a simple configuration file. Each time we will need custom filters or other new features, we will add them to our tool, having more general scenarios in mind. Next, we could try replicating our previous studies on local build breakage prevalence and reasons similarly.

We have many ideas for improvement: parallelization, support for more input sources (GitLab, local tarballs) and build tools (Ant), etc. We also envision that JaMaBuild could become a framework for automatic build script repair strategies. If a build fails, multiple corrections of the build configuration files will be performed until it eventually succeeds.

Acknowledgments

This work was supported by project VEGA No. 1/0630/22 Lowering Programmers' Cognitive Load Using Context-Dependent Dialogs.

References

[1] Jens Dietrich, Henrik Schole, Li Sui, and Ewan Tempero. 2017. XCorpus–an executable corpus of Java programs. *J. Obj. Tech.* 16, 4 (2017), 1:1–24.

[2] Pedro Martins, Rohan Achar, and Cristina V. Lopes. 2018. 50K-C: A Dataset of Compilable, and Compiled, Java Projects. In *Proceedings of MSR 2018*. ACM, 1–5. https://doi.org/10.1145/3196398.3196450

[3] Jens Palsberg and Cristina V. Lopes. 2018. NJR: A Normalized Java Resource. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*. ACM, 100–106. https://doi.org/10.1145/3236454.3236501

[4] Ricardo Terra, Luis Fernando Miranda, Marco Tulio Valente, and Roberto S. Bigonha. 2013. Qualitas.class Corpus: A Compiled Version of the Qualitas Corpus. *SIGSOFT Softw. Eng. Notes* 38, 5 (2013), 1–4. https://doi.org/10.1145/2507288.2507314