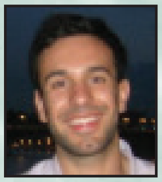
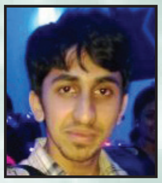
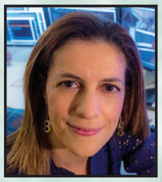
The Java Build Framework: Large Scale Compilation



Pedro Martins
University of California, Irvine
pribeiro@uci.edu



Rohan Achar
University of Nebraska-Lincoln
rachar@uci.edu



Cristina V. Lopes
University of California, Irvine
lopes@uci.edu

April 2018
ISR Technical Report # UCI-ISR-18-3

The Java Build Framework: Large Scale Compilation

Pedro Martins   Rohan Achar   Cristina V. Lopes
Institute for Software Research
University of California, Irvine
{pribeiro,rachar,lopes}@uci.edu

UCI-ISR-18-3, April 2018

Abstract

Large repositories of source code for research tend to limit their utility to static analysis of the code, as they give no guarantees on whether the projects are compilable, much less runnable in any way. The immediate consequence of the lack of large compilable and runnable datasets is that research that requires such properties does not generalize beyond small benchmarks.

We present the Java Build Framework, a method and tool capable of automatically compiling a large percentage of Java projects available in open source repositories like GitHub. Two elements are at the core: a very large repository of JAR files, and techniques of resolution of compilation faults and dependencies.

1 Introduction

`Java` is one of the most popular programming languages, and the third most-used programming language in GitHub. [1] The maturity and wide-spread acceptance that `Java` achieved justifies its popularity among the research community, with considerable research focusing on this language. It is therefore a prime candidate for software mining. A particularly popular type of software mining research is the extraction of source code features with the purpose of performing various studies, such as correlation with bugs, security, etc. These types of studies have motivated the development and curation of repositories and datasets for large-scale source code analysis, such as the Sourcerer Datasets [?], the Qualitas Corpus [?], and Boa [?].

These large repositories of source code for research, however, tend to limit their utility to static analysis of the code, as they give no guarantees on whether the projects are compilable, much less runnable in any way. To the best of our knowledge, no public large-scale repository of `Java` projects provides guarantees that the projects compile, or run, something very relevant to a wide spectrum of analysis techniques. Those kinds of guarantees can be made on small, manually curated datasets, such as those of the DaCapo [?] and SPEC [?] benchmarks. The immediate consequence of the lack of large compilable and runnable datasets is that conclusions from research work that require such properties (e.g. [?, ?, ?]) may not generalize beyond the small benchmarks with which they have been evaluated. The scope and impact of these studies would be much greater if they could leverage the vast amounts of open source `Java` programs available, similarly to what has already been achieved for studies that require vocabulary, software metrics, and static analysis.

But scaling up compilation, testing, and execution to thousands of projects from Internet repositories is a daunting task. In order to compile, test, and run thousands of projects, we need automated techniques for resolving dependencies (compilation), for finding

[1] `https://octoverse.github.com`, Dec. 2017

and driving test suites (testing), and for producing input and workflows (execution). These three types of automation are all challenging in different ways, with the last one (execution) being the most difficult one.

This paper presents a first step into obtaining such datasets by tackling the first challenge: large-scale compilation. We present the Java Build Framework (JBF), a method and tool capable of automatically compiling a large percentage of Java projects available in open source repositories like GitHub. Two elements are at the core of JBF: (1) a very large repository of existing `JAR` files, and (2) a technique of dependency resolution that identifies the external types in the projects and maps them to the most appropriate `JAR`.

Along with JBF, we provide a repository of 50,000 compilable (and compiled) `Java` projects taken from GitHub, called 50K-C. Each project in this dataset comes with references to all the dependencies required to compile it, the resulting bytecode, and the scripts with which the projects were built. The dependencies, bytecode, and scripts are all products of JBF. The dependencies and the build scripts provide a mechanism to re-create compilation of the projects, if needed (to instruct source code for bytecode analysis, for example). The bytecode is ready for testing, execution, and dynamic analysis tools. All the 50,000 projects come with origin information, which means that analyses on them can be cross-referenced with other tools operating on GitHub, such as Boa [?] or the map of GitHub clones DéjàVu[?].

One advantage of the 50K-C dataset with respect to many other large datasets of source code is that all project dependencies have been resolved by JBF. This, in itself, is valuable for many research studies, independent on whether execution is a goal or not. Another advantage is that, even for static analysis, compilation gives solid assurances that the source code is complete and self-contained: having been parsed successfully means that that source code is valid Java code; having been type checked successfully means that all required types, external and internal, are present. For assurances of physical integrity, compilation is the ultimate test.

While large-scale compilation is just the first step towards developing very large collections of runnable projects for research, it is a critical one, without which the other two steps are not possible. Resolving dependencies of arbitrary projects taken from sites like GitHub is a challenge in itself. This paper explains our method of doing it, and compares it to the baseline of trying to use the build scripts included in the projects, when they exist. It also shows how current build techniques, despite sophistication and success on individual projects, struggle to support the large scale buildings for which they were never designed.

The source code, datasets and running tutorials can be found on `http://mondego.ics.uci.edu/projects/jbf`.

This paper is organized as follows. Section 2 gives an overview of JBF. Sections 3 and 4 present the two main parts of the framework. Section 5 describe the evaluation of JBF. Section 6 provides statistical observations of build successes. Section 7 presents related work, and Section 8 concludes the paper.

2 JBF

An overview of the entire process that constitutes JBF can be seen in Figure 1. JBF is split into two main tasks, namely management of dependencies and project compilation, and each of these is further split into sub-tasks.

Starting with dependency management, the first task on the pipeline consists of collecting `JAR` files from the projects. After that, JBF creates an inverted index of Fully Qualified Names (FQN) of `Java` classes and interfaces available in a `JAR`. This is done before any compilation takes place.

The compilation process in JBF is done in up to two rounds for each project. In a first round, an attempt is made to compile it without specifying any external dependencies, just relying on the `Java` standard libraries. The ones that succeed are marked as so; the ones that fail go through a second round of processing. JBF detects compilation errors related to encoding problems and to missing external dependencies. Both are solved through inserting specific directives in our compilation scripts. For resolving ex-

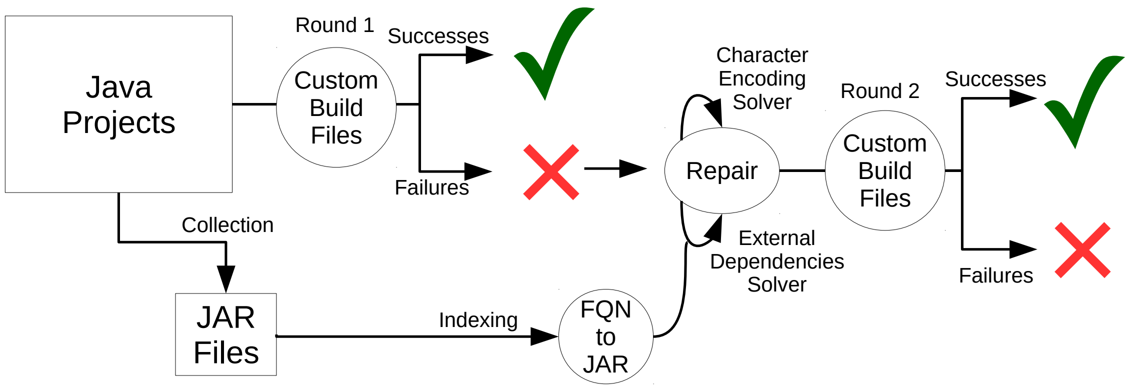


Figure 1: SourcererJBF pipeline.

ternal dependencies, we use the index created in the initial phase. A second round of compilation starts whose results are final: the projects that compile are marked as successes, the failures as failures, and the results of this round together with the results of the first round determine the overall effectiveness of JBF.

In the next sections we will explain these tasks of JBF in detail.

3 Collection and Management of Dependencies

Reliance on external dependencies is nowadays typical of software systems. Therefore, our first concern when attempting to build a set of software projects is to ensure availability of dependencies in order to minimize the number of failed builds.

Generally, `JAR` dependencies required for compilation come from three main sources:

- Boot libraries, with generic functionality provided by the Java Virtual Machine runtime. These dependencies are automatically available to building tools.
- Installed extensions through the Extension Mechanism Architecture[2], which are similar to the core libraries in usability but which vary from environment to environment. The dependencies available through this location extend core functionality and can have a varied origin, but must follow a strict set of guidelines.
- External libraries explicitly referenced in the source code, and that can have any location on the system as long as the building tools are aware of their existence. These dependencies can have any origin. A typical case for the usage of these dependencies is a project that is stored together with a certain `JAR` file that it requires.

[2]`http://docs.oracle.com/javase/6/docs/technotes/guides/extensions/spec.html`

3.1 Collection

For the boot libraries we do not interfere with their typical usage, we simply expect any system under our operational pipeline to contain a standard `Java` installation.

For the extensions, we opt to avoid this mechanism entirely, by directing the compiler `javac` to an empty location, overriding the system's default behavior. The reason for this is environment control: these libraries change between different `Java` versions, with JDK updates within the same `Java` version (`Java` 8.1 to `Java` 8.2) or with personal preferences in different installations of the same versions of the JDK. Instead, we place a handful of commonly used extension libraries along the external libraries.

For the external libraries, we scrape all the projects we have for `JAR` files that were packaged within them, and copy them to a central repository, which consists of a folder with all the JAR files inside (to be precise, a set of folders, because one single folder does not scale).

3.2 Filtering

We observed that some JAR files have invalid signatures that cause javac to crash silently. We use the tool jarsigner[3] to analyze every JAR we collect. In particular, we are only concerned with JAR files that are unusable due to some problem in their digital signature, for which jarsigner produces the following output:

```
java.lang.SecurityException: Invalid
signature file digest for Manifest main
attributes
```

The JAR files that produce this error are dismissed. All the others are accepted into our central repository.

3.3 Indexing

Types in Java are referenced by their Fully Qualified Name (FQN). A FQN is an identifier of a class or an interface which includes the package where it belongs.[4] In order to assist dependency resolution, we create an inverted index that maps every type we find inside JAR files to the JAR files where it occurs.

Specifically, we analyze every JAR file in order to list its table of contents. For example, consider a certain uci-irv.jar with the following contents:

```
META-INF/MANIFEST.MF
edu/uci/ics/algo.class
edu/uci/econ.class
```

This lists two types, each identified by an FQN. Upon index creation, both FQNs are entered as keys, having this specific JAR file in their list of values, generating the following index:

```
edu.uci.ics.algo: {uci-irv.jar}
edu.uci.econ: {uci-irv.jar}
```

Returning to the usage of FQN to resolve types, Java allows resolution through partial FQN to an entire group rather than to a specific type. For example, source code that requires the JAR above can specify this relation as importing edu.uci.*, with the wildcard meaning that the requirement is to any class that is hierarchically below edu.uci, which can be either ics.algo or econ.class, or both. This results in ambiguity that can be solved in a couple of different ways. For performance reasons, we index all possible FQNs that can be used in a project. Returning to our running example, the following FQNs will be indexed, all pointing at the same JAR file:[5]

```
edu.uci.ics.algo: {uci-irv.jar}
edu.uci.econ: {uci-irv.jar}
edu.uci.ics: {uci-irv.jar}
edu.uci: {uci-irv.jar}
```

Note that the mapping between FQN and JAR files is one-to-many, since two JAR files may contain the same FQN. For example, if another JAR had the FQN edu.uci.psico, then it would be indexed in same positions where edu.uci.psico (and its reduced variations) intersect with uci-irv.jar, namely edu.uci.

The final result of indexing is a large mapping between all FQNs found and the paths of the respective JAR files.

4 Projects Compilation

After having the external dependencies organized as an inverted index, we can start building the projects. This process can take up to three phases: a first round of compilation, a middle task of error repair, and a second round of compilation.

4.1 Building Projects - Round 1

The first round of compilation does not rely on external dependencies and has two objectives: first, to

[3]http://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html

[4]For a detailed explanation of FQN and general naming conventions in Java please see https://docs.oracle.com/javase/tutorial/java/package/namingpkgs.html.

[5]We do not index org or com, for obvious reasons: no project imports com.* or org.*.

immediately build the projects whose compilation is straightforward; and second, to collect a list of errors for projects that fail and provide JBF with information to fix them.

For every project JBF creates a generic `Ant` build file, which simply invokes `javac` for all java files found from the project's root folder and subfolders. After this call, two things can happen: either the project compiles, or it fails. If the project compiles, we store it as a success. For each project that JBF builds we store:

- the resulting Class files;
- the generic build files (they are generic but contain some information related to the file system, so we keep them);
- the exact system call that successfully called the build scripts and built the project;
- and the output produced when building the project.

The projects that fail are passed to the error repair mechanism of JBF.

4.2 Error Repair

For the projects that fail after Round 1, a new task starts, where JBF parses the resulting output of `javac` and searches both for character encoding errors and for missing packages. These type of errors take the forms of

```
error: unmappable character for encoding UTF8
```

for encoding errors, and

```
error: package org.msr does not exist
```

for errors related to missing external dependencies.

JBF parses the output of `javac` looking for signs of these errors. For encoding errors, JBF runs an analysis of the files and passes specific encoding information to `javac`, such as:

```
encoding="windows-1252"
```

For errors related to external dependencies, `javac` reports them in the form of missing FQNs. We then use them as queries to the index to find the `JAR` files in which they exist.

The dependency solver of JBF captures all the missing packages from an unsuccessful build and finds the `JAR` file that contains the largest number of them. For example, if the failed attempt of the project reports 10 missing packages, and we find a `JAR` file that contains a reference to all of them, then this is the `JAR` file chosen. If we find a `JAR` file that contains a reference to 8 of these missing packages, then we choose it, and recursively start the process for the next two. If more than one `JAR` file matches a certain requirement (for example, more than one `JAR` file contains a reference to all of the 10 missing packages), then we choose the first one without any particular preference.

In the end, all the `JAR` files that are found are considered to be the necessary external dependencies, and are given as such to `javac`.

An example of the code that is inserted into `javac` task of `Ant` to fix dependencies is:

```
<pathelement path="path/to/uci-irv.jar" />
```

An important note is that, in this stage, no FQN reduction is performed when trying to resolve dependencies, for a simple reason: when we are scraping and indexing the `JAR` files, we have access to all the complete FQNs that the `JAR` file contains, that is, all the FQNs from the top of the package hierarchy until a certain Class. On the project side, the `package not found` error depends on what the programmer specifically wrote in the source code. For example, the programmer might have specified a requirement for `edu.uci.*` when she is using the Class `edu/uci/ics/algo.class`. For this reason, if we can not find on the indexed `JAR` files a full FQN from a package, we are certain that Class does not exist. On the other side, if the reduction comes from the project side then we can attempt to find a `JAR` file that matches that reduced format, and hope it will contain the required Class. This is the reason why we indexed all the `JAR` files by all the possible FQNs that can be used to refer to them.

If the dependency solver can not find a `JAR` file, it throws a `Missing package error`.

4.3 Building Projects - Round 2

The second round of compilation consists on building the project that failed on the first round, with specific directives that try to resolve both the encoding and the dependency errors as described above.

After this round, the projects that build are stored with the same information as described before, now with build scripts that contain encoding and dependency information that is specific to the actual project.

The projects that fail after this step are considered true fails, and no new attempt is made to try to build them again. Their build information is stored nevertheless, in a format that is similar to the one for successful buildings, except in the output we write the errors the `javac` compiler produced, and there is no folder containing Class files, because no output was generated.

5 Evaluation

Here we present the evaluation of JBF, first alone, then in comparison with the build scripts that come with the projects, for the projects that contain them. Our concrete research questions driving this evaluation are the following:

- What is the effectiveness of JBF on building `Java` projects on scale?
- How does JBF compare to the projects' own build scripts, when present?

5.1 Testing Corpus

To test JBF we downloaded a set of `Java` projects from GitHub. The projects were downloaded using the GHTorrent database and network[6] [?] which contains meta-data from GitHub (number of stars, commits, commiters, whether the projects are forks, main programming language, date of creation, etc.) as well as the download links.[7]

Table 1: Corpus of Java projects.

# projects (total)	3,506,219
# projects (non-fork)	1,859,001
# URLs processed	631,390
# projects (downloaded)	479,113
# projects (excluding Android)	353,709
# jars	245,648
# FQNs	8,164,106

Table 1 shows the information regarding the size of the language corpora in our analysis. We skipped forked projects, because since they represent directly cloned projects and usually contain a large amount of code from the original projects, having explicit forks in the dataset for this study could skew the findings. Figure 2 shows the distribution of files and `JAR` files among the projects.

Downloading the projects was time-consuming. The order of downloaded projects was that of the GHTorrent database projects table, which seems to be, roughly, but not strictly, chronological.[8] Even though for `Java` we processed only 34% of the available, non-forked project URLs, the portion we processed provides a large enough corpus to be significant.

Some of the URLs, when downloaded, failed to produce valid content. This happened in two situations: (1) when the projects had been deleted from GitHub, or marked private, and (2) when development for the project happens in branches other than the master branch. As such, the number of downloaded projects

[6]`http://ghtorrent.org`

[7]While GHTorrent enabled us to crawl large amounts of repositories, we have found its database to have some errors. Specifically, a residual number of duplicated entries (i.e. projects had the same URL); only the youngest of these was kept for the analysis. We filed an issue report at (URL omitted for anonymity).

[8]The documentation of GHTorrent does not specify the exact order, if any, by which projects are added to the projects table. In analyzing the data, we noticed that although the projects are not ordered in strict chronological order, there seems to be a concentration of older projects in the earlier entries of the table. We filed another issue report asking the developers of GHTorrent to clarify this.

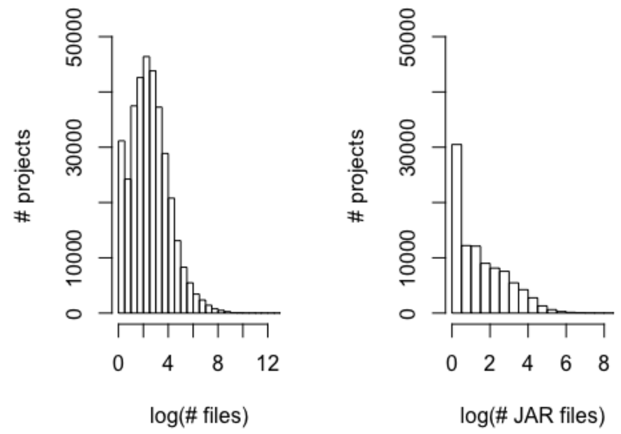


Figure 2: A distribution of the number of files and number of `JAR` files in the `Java` projects.

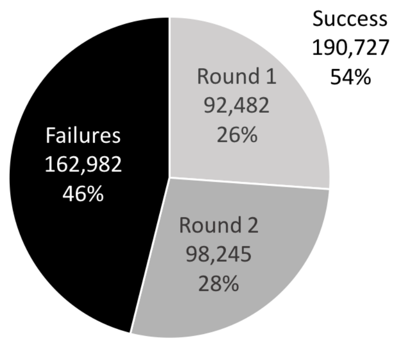


Figure 3: Number of projects built by JBF.

was considerably less than the number of URLs available from GHTorrent. Some projects did not have any source code with the expected extension, and these were excluded from the analysis.

Finally, we also filtered the projects that represent software for the Android ecosystem, since these have special requirements (we would need, for example, to have different versions of the Android JDK.). We did so by searching inside the projects for a `AndroidManifest.xml` file, since according the respective documentation, these files are mandatory for every Android application.[9]

All the tests were performed on a Intel(R) Xeon(R) CPU E5-4650 v4 @ 2.20GHz with 112 cores and 250 Gb of RAM.

5.2 JBF

In this section we present the results of running JBF on the corpus of 353,709 `Java` projects. On total, between the two rounds of compilation JBF compiled more than half the projects, 190,727, which corresponds to 54%.

From all the projects that JBF was capable of building, around half built after the first round of compilation, meaning they did not require any kind of special assistance. The other half required the assistance of the error repair mechanism, which proved successful for 98,245 projects. The exact values of the number of projects built after the two rounds can be seen in Figure 3.

Regarding the work performed by the error repair mechanism we can see in Figure 4 the distribution of errors among the projects that successfully built only after the second run, i.e., the distribution of errors for which JBF behaved successfully. JBF solved encoding errors in 10,303 projects, and resolved the external dependencies of 92,494 projects.

For the projects that failed after the second round of compilation, which amounts to 162,982 projects, we can see on Table 2 the most common errors. An error in special deserves attention, `Missing packages`, which was produced by the type-resolution mechanism, and means that no `JAR` file could be found for the specific type needs of a project. The other errors on the table were extracted directly from the output of the `javac` compiler.

A broad analysis on Table 2 leads us to notice that most errors are related to missing types. This indicates both a lack of soundness on the type structure of projects, and/or a reliance on external type infor-

[9]`https://developer.android.com/guide/topics/manifest/manifest-intro.html`, May 2017.

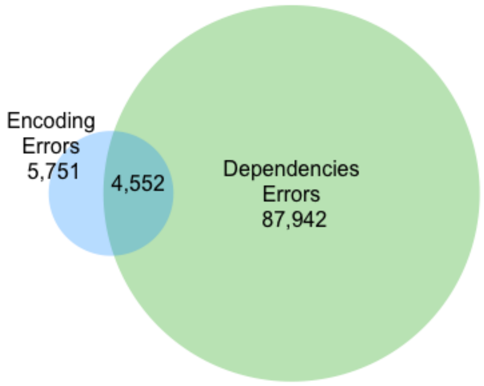


Figure 4: Distribution of errors handled per project built by JBF.

mation for completeness. The conclusion is that any system that attempts to build source code on scale should be prepared to face some form of type problems, an observation we have done before, and that partly motivates this work, and gains here further support.

As a side note, JBF successfully built 16,507 projects that were specifically Android. These projects were not added to the statistics presented until this point, and they are not part of the evaluation corpus we have presented in Section 5.1, since we removed Android projects. We are adding this information here just to note that the general approach of using generic build systems had some success in these cases.

5.3 Build Frameworks

We now compare the effectiveness of the build frameworks stored with the projects with the efficiency of JBF.

The build frameworks within the projects were detected by searching for specific files that would flag their existence. In particular, we searched for a `build.xml` file to detect the existence of `Ant`, a `pom.xml` file to detect the existence of `Maven` and a `build.gradle` file to detect the existence of `Gradle`.

We present, on Figure 5, the distribution of build

Table 2: Top 20 errors for failed projects. Note that these groups intersect, and most projects had more than 1 error type.

Error	Frequency
cannot find symbol	210,949
package not found	183,614
Missing packages	124,742
method does not override or implement a method from a supertype	68,586
duplicate class	26440
static import only from classes and interfaces	16,703
unmappable character	15,988
expected symbol not found	13,904
illegal access	13804
Too many encoding types detected	12,220
incompatible types	12,155
illegal use	9,367
cannot be applied	7,469
no suitable definition found	6,122
class should be in its own file.	5,541
abstraction error	5,329
not a statement	3,541
reached end of file while parsing	3,081
invalid method declaration; return type required	1,816
too many parameters	1,631

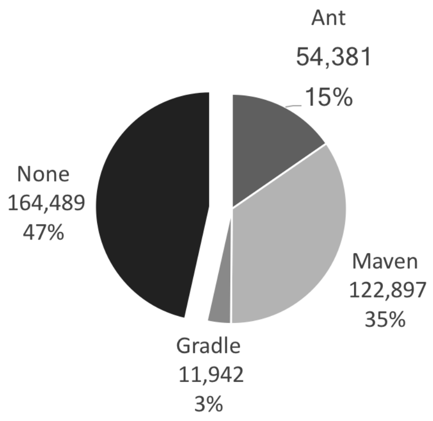


Figure 5: Distribution of build systems across the projects.

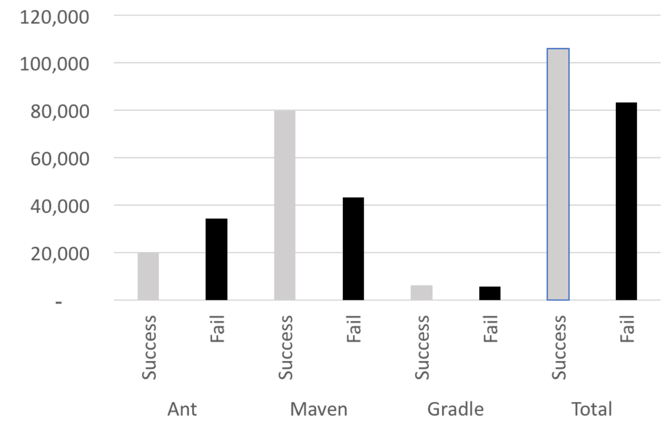


Figure 6: Compilation success rates for each build system individually.

frameworks across the entire corpus of `Java` projects. On total, 189,220 projects (53% of the 353,709 total projects) had build information of some sort. The first important observation is that around half the projects exists without information on how to compile them. This might be due to their nature (not all projects represent a single software solution), or due to the nature of their origin, either for lack of proficiency or personal preference.

Operationally, the test of the build systems was instructed to directly call whichever build framework existed with the projects. In cases where there was more than one, the order of prioritization was `Ant` first, then `Maven` and then `Gradle` (the number of projects in this situation was residual). For each of the projects with a build framework we made a generic system call that would start the process of building it. For `Ant`, `Maven` and `Gradle` the calls were, respectively:

- `ant -f <path to build file>`
- `mvn -f <path to pom file> compile`
- `gradle -b <path to gradle file> compileJava`

Only one try was allowed per project, without any type of error resolution: either it immediately succeeded, or immediately failed.

We start by observing, in Figure 6, the effectiveness of the build frameworks in isolation as well as their aggregated performance. `Ant` is the build framework that performs the worst, and is only capable of building 20,095 projects, failing in the other 34,286 projects. `Maven` is the most successful, and builds more than half the projects; on total, `Maven` is capable of building 79,672 `Java` projects, and fails only for 43,225. We have seen before that type errors, in particular missing external types are frequent. Knowing that `Maven` has the capacity of automatically managing external dependencies, namely obtaining them from online sources, provides a likely explanation as to why this tool performed so well. `Gradle` successfully built 6,206 `Java` projects and failed for 5,736, splitting more or less in half the success rate of this tool.

Going now to the comparison between JBF and the build systems, in Figure 7 we can see individual and aggregated comparisons with JBF. Starting with `Ant` (first chart), JBF performed better, managing to build 15,332 of the projects the tool was not capable to, while the inverse is only true for 6,321 projects. Both JBF and `Ant` successfully built 13,774, and both failed building 18,954 projects.

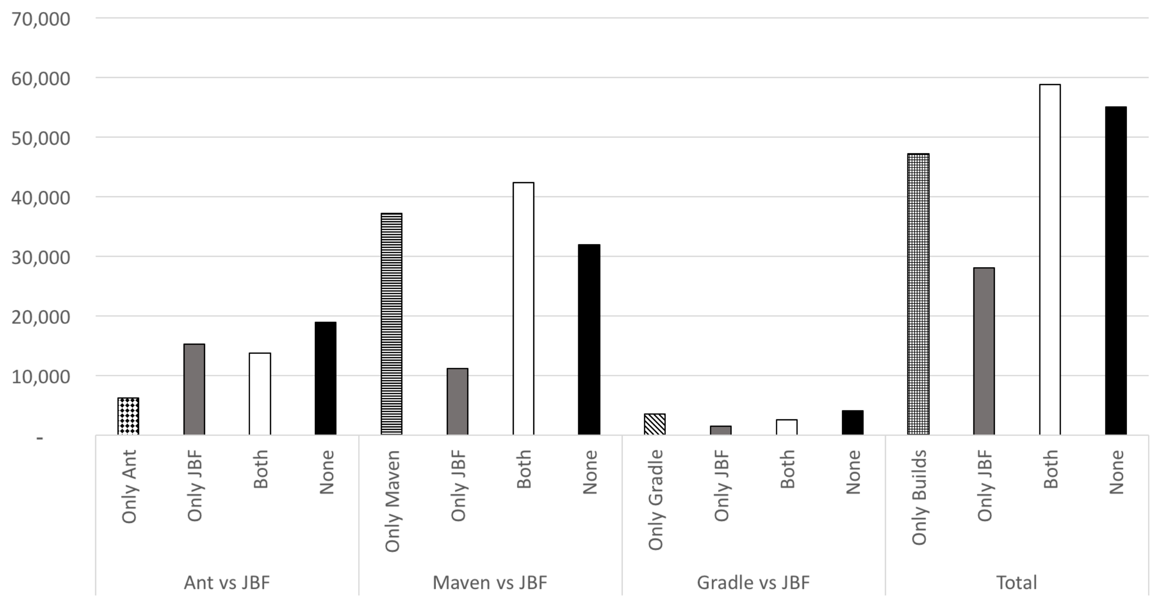


Figure 7: All build systems vs. JBF, 189,220 projects total.

Maven was the framework that performed better in comparison to JBF, managing to build 37,252 projects where JBF failed, whereas JBF built only 11,223 of the projects in which Maven failed. For Maven we have a large intersection of projects built by both tools, 42,420, which means that both strategies performed with a reasonable degree of success on the group of Maven projects.

The results for Gradle (third chart) are similar to Maven. Both strategies successfully built around a quarter of the projects, and Gradle performed better than JBF but, in comparison, not as good as Maven.

Finally, we can analyze the aggregate of the building tools against JBF, in the fourth chart of Figure 7. Both the build frameworks and JBF successfully built 58,785 projects, and the build frameworks built exclusively 47,188 projects, being the JBF counterpart 28,141 projects. Contextualizing with all the projects, the successes and failures shared by JBF and the build frameworks cover around two thirds of all the projects, for the other third, the build frameworks perform more effectively.

5.4 Build Frameworks and Scale

There are a few notes worth mentioning for understanding the performance of build frameworks on environments where they have to scale.

The first aspect is that when comparing JBF with existing build systems we are inherently neglecting a large number of projects that simply do not contain one. Within all the 353,709 projects we obtained, if we used build files we would be able to build 105,973 projects, whereas with JBF we were capable of building 190,727 projects, out of which 103,801 projects contain no build information.

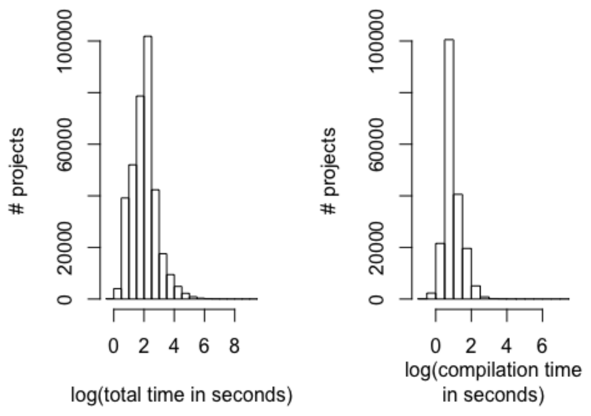
Second, the build frameworks analyzed are in their essence scripts that allow a possible infinite number of actions on a large number of domains. They can remove, add or move information in the system or connect to online servers. It is easy to see the security holes that running a large number of unknown script creates, and these should not be taken lightly.

Third, the utility of building software is in the access of the produced runnable elements, in our case class files. But the use of unknown scripts can - and often does - cripple this advantage. We do not know where the targets are being allocated, or if the script is creating a JAR file instead of class files.

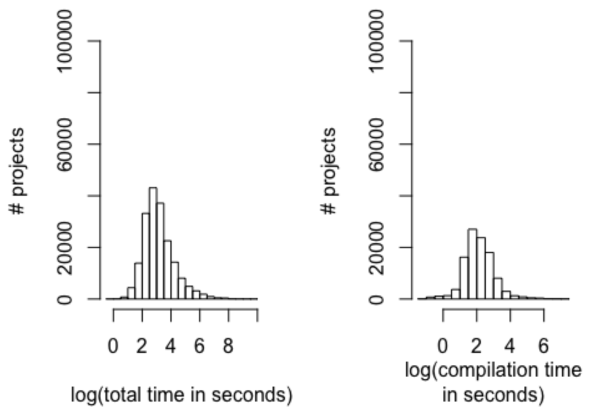
Fourth, the whole notion of success is dependent only on the full execution of the build framework, not on its intrinsic actions. This means a script can be successful but everything it does is printing the famous 'Hello World!'. On efforts of building on scale this introduces an uncertainty that easily becomes unreasonable. For example, when we provide the results of the build scripts in the previous sections these represent successes from the perspective of the operating system (the process terminated without an error code), not from the perspective of the inherent compilation task. This problem is of hard leverage.

Fifth, and related with the previous two, the time it takes to run a build script is as much of a mystery as its contents. The building of the 353,709 `Java` projects using JBF required a median of 7.5 seconds per thread, where each thread was responsible for the complete process from the management and extraction of the archived files to the compilation and any necessary clean-ups or error recoveries (Figure 8). The actual compile time (a sub-task of each thread) had per project a median of 2.3 seconds. The equivalent times for the building frameworks were 20 seconds in median, and just to compile 7.8 seconds in median. The times, especially the compile time, are substantially longer than JBF times. On universes of hundreds of thousands of projects this is relevant as it represents differences of weeks between one strategy or the other.

During our experiments running the build systems we found a little of all of this: random files appearing in the file system, 'stuck processes', lost target files, and so on. It is important to emphasize that this is not a critic of the build systems themselves: `Ant`, `Maven` and `Gradle`. These are quite sophisticated, very capable, and on single entities potentially more capable than JBF. The problem is that they do not scale, they do not possess the characteristics of control, predictability or security that JBF does simply because they were never designed with that concern in the first place.



(a) JBF.



(b) Build frameworks.

Figure 8: Time used to completely handle each project, and time used to compile each project using their building information (only for the ones that compiled successfully).

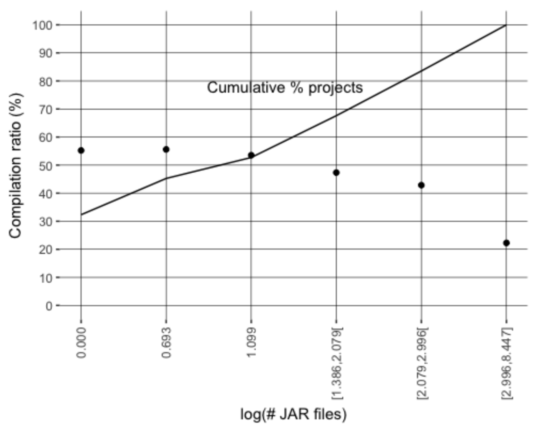


Figure 11: JBF success ratios per number of JAR files inside the project.

6 Analyzes of Success Builds

While executing and testing JBF we realized some characteristics of Java source code are more likely to result in success builds than others. This provides statistic indicators for successful compilations which adds to the understanding of source code and can help in sample selection, relevant even on scale by limited resources (for example, which projects to download from GitHub to maximize build performance). For this reason we provide them here.

Between the two building rounds that exist in the JBF pipeline, we have seen that each one successfully built around half the total successes of JBF. Figure 9 shows a distribution of the number of files and inside JAR files for the group of projects that was successfully built after the first round (left) and after the second round (right). It is noticeable that the first group has a smaller number of files (top-left) which leads us to speculate that the reason some projects build immediately is due to their simpler structure, at least if complexity of a project is measured in relation to the number of files it has. The distribution of JAR files for the projects that required some sort of error resolution (which, as we saw, was mostly related to external dependencies) have a larger number of JAR files inside them comparing to the ones that did not.

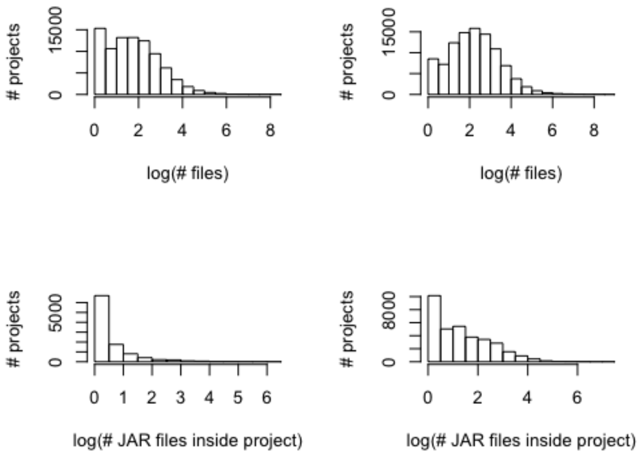


Figure 9: A comparison of projects that did not require error resolution (left) and projects that did (right). On top we have the distribution of the number of files, and on the bottom we have the number of JAR files that the projects contained.

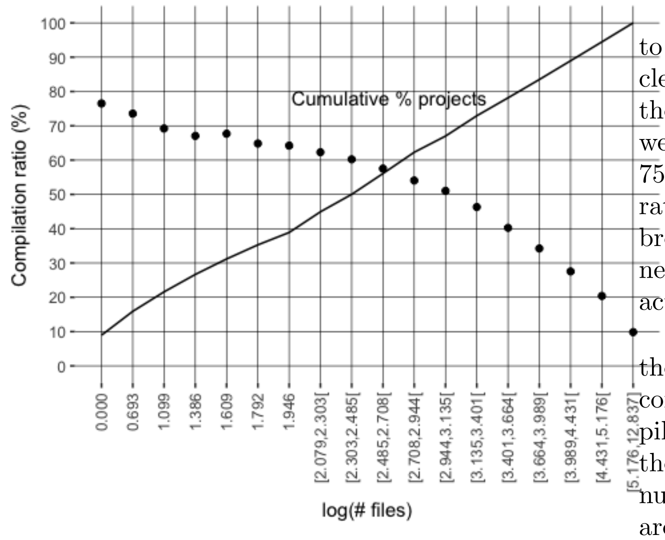


Figure 10: JBF success ratios per project size.

For the relation of the number of files of a project with its compilation ratio we can look at Figure 10. The black dots are small intervals on the scale of the x-axis, representing subsets of similar sized of projects with different number of files ('bins'). The y-axis represents the compilation ratio; a certain, random value of 70% implies that whichever bin (or black dot, or interval of size) is within it's range, it contains a subset of projects where 70% successfully compile.

The line labeled 'Cumulative % projects' is a visual indication of how well the partition of bins splits the number of projects (the y-axis, showing percentage, has the same unit for compilation ratio and cumulative % of projects). The line has an almost perfect growth, showing a good distribution of projects across the x-axis.

If we look at the first, highest point of the figure we see that its x-value is close to 0, which means this point represents projects that are very small. The y-value of the same point is around 75%. This means that for projects that are very small, 75% compile and 25% fail. If we now look at the right-most point, its y-value is 10%, so the subset of projects that are on the largest side of size have a small compilation ratios.

Observing the entire of the x-axis, and comparing to the evolution of the values on the y-axis, we can clearly see a trend of lowering compilation ratios as the size of the projects increases. For small projects we start with a compilation success ratio of around 75%, and as we progress towards larger projects the ratio at which they compile is successively lower. The broad observation is that the number of files affects negatively the likelihood of a project compiling. The actual Pearson correlation coefficient is -0.95.

On Figure 11 we can see a similar assessment of the impact of the number of enclosed `JAR` files under compilation, which is negative. The impact on compilation is negative as the number of `JAR` files within the project grows. For projects with a very small number of dependencies the ratio of compilation is around 55%, and as the number `JAR` files increases we see the compilation ratio decreasing to around 21%. We found a correlation between the number of enclosed `JAR` files and the compilation success ratios, with a Pearson correlation coefficient is -0.88.

7 Related Work

7.1 Dependency Resolution

Dependency resolution plays a big role in the success or failure of building projects. Various other works focus on this topic and have employed various techniques to mine open source repositories and obtain project dependency information.

The work closest to ours is that presented by Ossher [**?**] (Chapter 4). Like in our work, the approach presented there also relies on the collection of a very large number of JAR files, and on the creation of an index that maps FQNs to JAR files, which is then used to find the dependencies for the projects. Ossher's work had a different purpose than ours: they produced a map of project-to-project dependencies across the Java ecosystem, and were not trying to compile the projects. This, in turn, carries a couple of technical differences between our work and theirs. First, the JARs used by Ossher were a combination of those present in the Maven Central Repository as well as a relatively complicated system for identifying *projects* inside JAR files. Instead of doing that, we simply collect all JAR files in all the projects, and we use the ones that match the needed FQNs, independent of what else they contain that may not be relevant for the projects. Our approach is much simpler, and seems to work well for purposes of compilation. Second, Ossher's dependency resolution did not give priority to the JARs present in the projects, and looked for matches in the entire collection of JARs. We started by doing the same, but soon realized that the results of compilation were much better if we gave priority to those local JARs.

Lungu et al. worked on extracting inter project dependencies with a specific focus on Smalltalk ecosystems [?]. Zhang, Hanyu details four techniques for dependency resolution in C/C++ projects [?], through a) analyzing the source code of the project; b) parsing build scripts; c) reading binary files with the *ELF* file format and discovering dynamically linked libraries; and d) using a specification based technique that looks at the source specifications of projects recorded in *Debian* repositories. Clearly, each programming language has its own characteristics regarding dependencies; Smalltalk, C/C++ and Java are all quite different. The idea of parsing build scripts in search for dependencies, however, is one that we plan to explore in the future as a way of improving JBF.

API usage analysis, like [?], can be used to design alternative techniques for finding the right dependencies for projects. Lãmmel et al. extracted API usage and package footprint from large corpus of built java projects using *AST* based approaches. Here, missing packages required to build the java projects were obtained by searching the web for the jars using the unresolved package names as the search queries. They were able to improve their build success rate by 15% by using the resolution method in [?].

7.2 Empirical Studies of Builds

An analysis of build systems in multi-language software can be found in the work by Neitsch *et al.* [?] where the authors performed a qualitative study on a set of five multi-language open source software packages. They found that many build problems can be systematically addressed, and make the general point that build quality of software is rarely the subject of research, even though it is paramount to maintaining and reusing software, a message we endorse.

Sulir *et al.* [?] report on work similar to ours. They attempted to build a collection of 7,264 Java projects from GitHub, from which around 60% succeeded. This is a higher rate than that obatined by JBF in the entire dataset, but it is close to the rate we obtained for the smaller dataset of projects that have build scripts, 56%. Indeed, they only build projects which contain build scripts from `Ant`, `Maven` or `Gradle`. Additionally, their criteria to sample projects was much stricter than ours: at least one fork, and the enforcement of a black list for specific API's. We had no such constraints. Similar to what we observed, most failures were due to missing dependencies. They also observed that projects that built were smaller and older on average, and that the number of stars does not seem to have any correlation with building success. We did not analyze meta information related to age or other social and GitHub aspects, but their conclusion regarding the impact of scale goes in the same direction as our comparison between the projects built in the first, and in the second round of compilation.

Seo *et al.* [?] present a large-scale empirical study of 26.6 million builds produced during a period of nine months by thousands of developers at Google. They analyze failure frequency, compiler error types, and resolution efforts to fix those compiler errors. Two languages are analyzed, C++ and Java, and 37.4% and 29.7%, respectively, of these builds fail. The most common errors are associated with dependencies between components. The build environment is a Google's proprietary cloud-based build process that utilizes a proprietary build language, which is different from our generic building framework, but problems of external dependencies were nevertheless similar to ours.

On the subject of recovering type information for Java programs, Dagenais *et al.* [?] present a framework with an heuristics-based type inference system to recover declared types of expressions and resolve ambiguities in partial programs obtained from the Web. Their technique relies on the generation of ASTs for each file. They test with 4 projects, three 'self-contained' and the fourth with more than 90 external dependencies. They report around 90% success among the types the framework was able to recover, but between 35% and 50% of the types remained unknown. Not surprisingly, the projects that contained more external dependencies were on the higher range of unknown types. JBF does not explicitly generate ASTs, relying instead on javac to do that work.

8 Conclusion

We have presented JBF, a tool to build `Java` programs on scale, with automatic error repair and management of external dependencies.

Under direct comparison, the tool performed similarly to standard `Java` build frameworks, building less 10% of the projects. However, on a corpus of 353,709 `Java` projects, JBF build 24% more projects than what was possible with existing build frameworks, reaching 54% of the total projects built while introducing assurances of security, performance and integrity.

Two details about the results we obtain suggest there is a good potential to increase the success ratios of JBF. First, the errors obtained after the second round of compilation were mostly due to unknown types, one of them explicitly created by JBF signaling this exact problem. Second, the build framework that has the capability of obtaining external dependencies, `Maven`, was the one that outperformed JBF. These suggest that an important threshold on success ratios had its origin on the availability of `JAR` files rather than any idiosyncrasy of the tool.

Increasing the size of the `JAR` corpus is ongoing work, either manually or through automation. For example, a new task of rating missing dependencies by frequency, after the second round of compilation, associating them with `JAR` files, obtain them on online sources and put them in our central repository is operationally simple[10] and will further improve the built projects without affecting the JBF pipeline.

[10]The Maven Central Repository, for example, allows searching `JAR` files by Classname, aka FQN.

Attached to the introduction of JBF we created 50K-C, a dataset of `Java` programs together with all their dependencies, with individual building scripts and with the class files generated by them.

JBF and 50K-C can benefit and accelerate research projects that target: the actual process of large-scale building and running software, improvements on the `Java` programming language, and general domains of language engineering and understanding.