Analyzing the Build-ability of Open Source Java Projects on Github

Jonathan Lee, Hong-Wun Chen, and Hong-Yan Huang

Abstract—As the number of open source Java projects on Github repository grows tremendously nowadays, many of them are not uptodate to their maintenance. Some research works went through these projects to analyze their compile-time errors, and others provided a classification scheme of build-time errors. In this work, we focus on analyzing the build-ability of these projects by exploring the reason why the open source projects cannot be built through: (1) proposing a revised classification scheme of Gradle build-time errors by conducting an in-depth analysis of the root causes in the error logs, (2) identifying a number of bad coding styles leading to build failure, and (3) shedding some light on how to fix the bugs in an automatic manner.

Index Terms—software build-ability, error classification, Gradle.

1 Introduction

Java, as the third most used programming language on Github [11], has over 6 million repositories. Previous studies showed that 38% to 49% of Java projects on Github ended with a build failure during automatic building [1] [4]. In this work, we investigated the root causes leading to build failure, and classified build errors according to the root causes.

Among the three mature build tools of Java project, Ant [10], Maven [9], and Gradle [8], Gradle is the most well-maintained, with the period between the release date of two versions being about half a month on average. Gradle allows users to write Groovy build script, while the other two are based on XML. Gradle also provides relatively human readable error log in addition to a chain of exception than the other two. These features make the error classification of Gradle more challenging.

In this work, we collected and built 5,188 Java projects using Gradle as build tool from Github (Figure 1). We analyzed the error logs of the failed builds and investigated the root causes leading to the build failures. Based on our findings, we proposed a classification scheme for build errors.

By analyzing the result of error classification, we showed that our classification scheme recognized 78.79% of build errors, the error type taxonomy covered over 99% of build errors, and which type of error occurred the most frequently. Suggestions on improving project build-ability are also proposed.

In Section 2, we survey and compare previous studies on build-ability. We describe the details about our research approach to conduct the automatic build and classification in Section 3. The result of the build-ability test and build error classification, along with some discussion about our findings on bad coding styles, are fully discussed in Section 4. Finally, we summarize our classification scheme and findings, and address the possibility to construct an automatic fix scheme in Section 5.

- *J. Lee is with the Department of Computer Science and Information Engineering, National Taiwan University. E-mail: jlee@csie.ntu.edu.tw*
- *H. Chen is with the Department of Computer Science and Information Engineering, National Taiwan University. E-mail: r07922037@ntu.edu.tw*
- *H. Huang is with the Department of Computer Science and Information Engineering, National Taiwan University. E-mail: r06922050@ntu.edu.tw*

2 Related Work

In this section, we will review previous studies on software build-ability, such as Sulir et al [1], Seo et al [2], Rausch et al [3], Hassan et al [4], and Macho et al [5], and outline how our work differs from them in terms of data source, error type, and research objective.

Rausch et al [3] analyzed the build failures in the continuous integration workflows, called Travis-CI, of 14 Java projects. They simulated the practical software development situation by activating continuous integration workflows over each commit of the 14 Java projects gathered from Github, and analyzed the factors that generated or removed a build failure. Their research focused on the continuous changes on the same project to discuss how factors in software development influence software building; whereas our research considers only the most recent commit to propose a general build error classification criteria.

Seo et al [2] conducted a case study on the build errors on the centralized build system at Google. Their research included 26.6 million builds executed by over 18,000 developers during a period of nine months to calculate practical statistics of how software developers encounter and deal with build errors. Their research focused on compile-time errors that occur when developers built their own code; whereas our work analyzes build-time errors while building open source code written by other developers.

Sulir et al [1] performed a quantitative study over 7200 Java projects on Github. They simulated the environment of developers building open source projects cloned from Github on virtualization platform, and conducted simple

TABLE 1
The related work comparison

Research work	Data source	Error type	Research objective
Our work	5188 Java projects on Github	build-time	automatic classification
Sulir et al [1]	7200 Java projects on Github	build-time	statistical analysis of build-ability
Seo et al [2]	18000 builds at Google	compile-time	collect practical build failure statistics
Rausch et al [3]	14 Java projects on Github	CI workflow	how error in CI workflow happen
Hassan et al [4]	top 200 Java projects on Github	build-time	possibility of a automatic build system
Macho et al [5]	19 Java projects on Github	dependency-related	automatic fix dependency build error

error type classification by extracting key words from error logs generated by build tools. Their research included projects built by the three common Java build tools, while we concentrate on projects built by Gradle. They also conducted statistical analysis of the relation between build errors and project properties; whereas we investigate the root causes leading to build errors, and propose an error type classification.

Hassan et al [4] performed a study on the top 200 Java projects with the most stars on Github. Their research covered not only the three common Java build tools, but also included projects not built by any of them. They found 49.50% of the projects failed during automatic building, and classified the build errors into 9 error types by root cause analysis. Moreover, they performed 3 studies to deal with the 3 most frequent error types, 'non-default build command', 'platform version mismatch', and 'file setup requirement'. They made use of their past research, which is able to extract correct build commands from project readme file through NLP approach, to deal with the 'non-default build command' error type. For the other two error types, simple trial-by-error approach was used. They stated that 57% of build errors could be automatically resolved once an automatic build system that integrated all their works was built.

Macho et al [5] aimed to repair Maven dependency-related build breakage automatically. They built 32,100 commits containing build file revisions from 23 Java projects on Github. They focused on 125 commit pairs which generated and repaired a dependency-related build breakage. They first analyzed 37 pairs, accounting 30%, to learn how developers fixed the breakage. Then, they implemented the 3 most used strategies learned from the previous step, 'version update', 'delete a breaking dependency', and 'add a maven repository', in their dependency-related build breakage repairing tool, BuildMedic. They evaluated BuildMedic with the rest 88 pairs, and found that 45 pairs, accounting 54%, could be fixed automatically.

3 Build-ability Analysis

In this section, we will fully discuss our research approach in this study, including projects collection, building environment, building procedure, and error classification.

3.1 Projects collection

The projects collection process includes two parts: obtaining a list of qualified projects, and screening them with several filters. (See Figure 1)

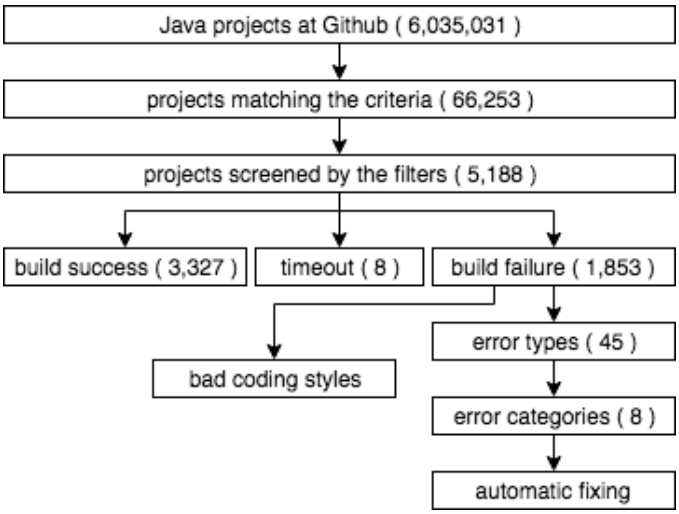

Fig. 1. Projects Collection and Findings

Among all the repositories on Github, we first retrieved the repository list consisting of repositories satisfying the following criteria:

- repositories written mainly in Java,
- repositories publicly available,
- repositories not a fork repository,
- repositories with last commit over a month ago,
- repositories with number of stars not less than 5,
- repositories without keyword Android or Minecraft in their titles, descriptions, and readme files.

Then, we cloned all repositories and screened them with the following filters:

- all submodules need to be available,
- a Gradle build file in root directory,
- not using Android APIs.

We will then explain the reasons why choosing these criteria and go into details of the projects collection process.

In this work, we only take Java repositories into consideration, since Java is the compiled language with the most number of repositories on Github [11], and it has mature build tools like Maven and Gradle. There were over 6 millions Java repositories at the time this study is conducted.

We excluded fork repositories to balance the weight of popular open source projects, since forks and the forked project may share similar build errors. Projects using Android API were preliminarily eliminated to avoid build errors due to incompatible environment. Projects that were

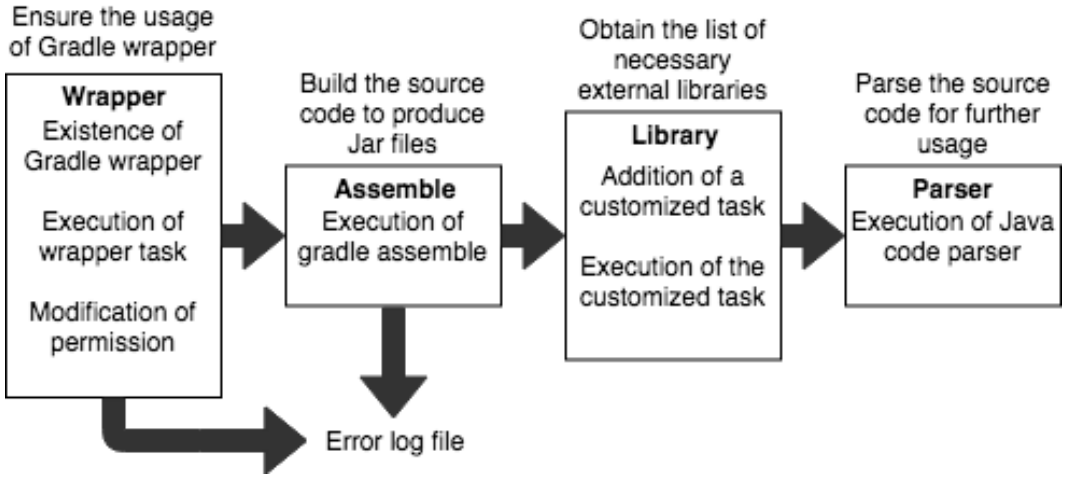

Fig. 2. Building process

frequently committed during the time this study was conducted were considered to be not stable and were excluded. Also, we set a Github star threshold to ensure that the chosen repositories were popular to the developers.

The Github API was used to query the repositories satisfying the above criteria in the first part. The number of qualified repositories was 66,253.

In the second part, the qualified repositories were cloned and were filtered with BASH script automatically.

Repositories with unaccessible submodules were considered imperfect and were excluded. Double check of usage of Android API was done by scanning all files with .gradle extension. We focused only on projects built by Gradle. Projects without a recognizable ”build.gradle” file in their root directory were excluded. The number of projects left after the second part was 5,188.

3.2 Building Environment

Automatic building of the 5,188 projects were executed on Jenkins CI and were controlled by Groovy script. The building environment and software versions used were as follows:

- Ubuntu 16.04.6 LTS,
- Java SE Runtime Environment 1.8.0_201,
- Gradle version 4.8,
- Jenkins CI version 2.164.1.

Since the syntax of Gradle keeps changing with the upgrade of Gradle version, especially when Gradle adopts a built-in wrapper task and disallows any user-defined wrapper task from version 5.0 making a large number of projects unbuildable, we chose version 4.8 as default version in this work.

3.3 Building Process

The controlling Groovy script invoked building process for each of the 5,188 target projects. The building process is shown in Figure 2. Every project was given a time limit of 1 hour to avoid builds that would never stop.

In the building process, usage of Gradle wrapper was first checked. Since Gradle syntax varies with its version, we preferred the Gradle wrapper set by the author over the default version.

We used command ‘assemble’ rather than ‘build’ because we only concerned whether the project could be compiled and could produce Java Archive files. The result of unit tests, which may cause build failure, was not in consideration.

The error message along with build status (success, timeout, or failure in which step) of each project were redirected to log file for further classification and analysis.

3.4 Error Classification

We first analyzed the error logs to make clear of which kind of behavior leading to which kind of error message. This set of error logs was the building result of projects collected with the same criteria except a Github star threshold of 10, containing 2,744 projects and 811 error messages.

The behaviors were then classified to several types, each type contains several error message patterns. The idea of our automatic classification scheme is to use regular expression to represent the error message patterns, and to recognize error message with the keyword groups captured by regular expression.

The error types were rearranged into 8 categories, each of which contained several error types storing in JSON format. The classification script loads all error types, recognizes and classifies the input error messages.

4 Findings

4.1 Build Success Rate

Figure 3 shows the overall result of the build-ability test, In total, 3,327 of the 5,188 collected projects passed the whole building process, accounting for 64.1%. In the research of Sulir et al [1] and Hassan et al [4], the build success rates were 61.87% and 50.50%, respectively. Moreover, the build success rates lied between 50% to 55% for projects built by Gradle in both works.

There were 8 projects exceeded the 1 hour time limit due to the endless waiting for response from unreachable maven repository while resolving dependencies.

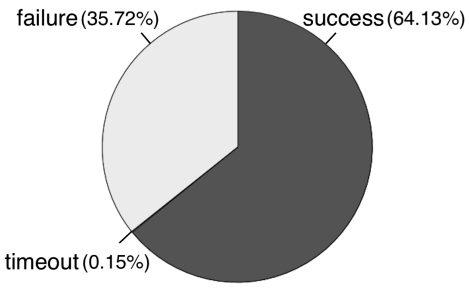

Fig. 3. The build success vs. failure

4.2 Build Error Types

In this section, we will discuss the classification criteria used in this work. After analyzing the 811 error messages in the 10-star testing mentioned in section 3.4, we extracted about 200 error patterns, and arranged them into 45 error types in 8 error categories.

The 8 error categories include: (1) wrapper, (2) plugin, (3) method, (4) property, (5) dependency, (6) Java version, (7) non-Gradle, and (8) others.

4.2.1 Error Category: Wrapper

The wrapper category consists of error types relating to the Gradle wrapper script, caused by the following errors:

- Un-executable script due to lacking the 'gradle-wrapper.jar' and 'gradle-wrapper.properties' file, or containing newline character in Windows.
- The designated Gradle distribution URL cannot be accessed.
- The user-defined wrapper task does not follow the classic Gradle wrapper layout.

4.2.2 Error Category: Plugin

The plugin category consists of error types relating to the plugins or scripts applied, caused by the following errors:

- The plugin applied is deprecated and is removed in current Gradle version.
- The plugin applied is not introduced yet in current Gradle version.
- The plugin applied does not support current Gradle version. This can be a version check, or usage of Gradle features incompatible with current version.
- The plugin applied is defective.
- Improper usage of plugin like 'application' or 'spring-boot', which tries to build an executable Jar on project that does not have a main class.
- The script applied from a URL is not accessible.
- The dependencies of the build script is not properly defined.

4.2.3 Error Category: Method

The method category consists of error types relating to the method used, caused by the following errors:

- The method used is deprecated and is removed in current Gradle version.
- The method used is not introduced yet in current Gradle version.

4.2.4 Error Category: Property

The property category consists of error types relating to the properties used, caused by the following errors:

- The property used is deprecated and is removed in current Gradle version.
- The property used is not introduced yet in current Gradle version.
- The build script lacks necessary properties used in Jar singing task.
- The build script lacks necessary properties used in the authentication to repository, IntelliJ, Github, AWS, and etc.
- The build script requires a local configuration file, like 'gradle.properties' or 'local.properties'.
- The build script requires environment variables be set in advance.
- Usage of dynamic property that has been deprecated.
- Usage of undefined property that needs to be set in local configuration file.

4.2.5 Error Category: Dependency

The dependency category consists of error types relating to the project dependencies, caused by the following errors:

- The dependency cannot be resolved since no maven repository is defined.
- The dependency cannot be resolved since the maven repository defined is unreachable.
- The dependency cannot be resolved since dynamic version syntax used.
- Dependency on local file that does not exist.
- Dependency on remote file that does not exist.
- The dependency cannot be resolved since it cannot be found.

4.2.6 Error Category: Java Version

The Java version category consists of error types relating to the Java version, caused by the following errors:

- Version check by the build script or plugins.
- Incompatibility between Gradle version and Java version.
- Usage of Java command line flag incompatible with current version.

4.2.7 Error Category: Non-Gradle

The non-Gradle category consists of error types not relating to Gradle, caused by the following errors:

- Failure while compiling source code, generating Javadoc, and conducting unit tests.
- Failure due to the included Ant build script.
- Failure due to the execution of third-party program, including lacking required software and accessing service not exists.

4.2.8 Error Category: Others

The other category consists of the rest of recognizable error types, caused by the following errors:

- Plugin property is not properly configured.

TABLE 2
The most frequent error types.

Type	Description	%
1.1	Un-executable Gradle wrapper	14.62
7.1	Compilation failed	14.09
5.6	Dependency not found	10.20
2.3	Plugin only supported in lower Gradle version	5.94
4.4	Missing authentication properties	5.77
5.2	Unreachable repository defined	4.21
8.5	Need full clone	4.05
4.3	Missing signing properties	3.62
7.5	Error from third-party program	3.35
4.9	Get unknown property	3.02

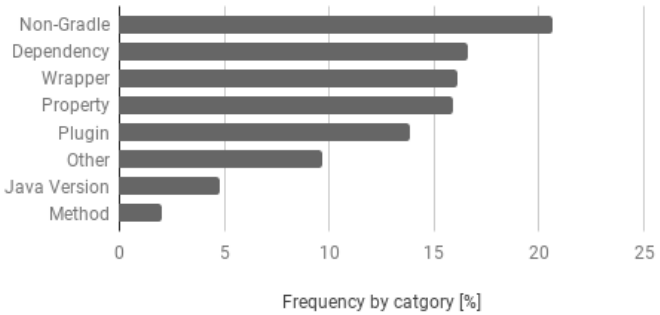

Fig. 4. The most frequent error categories

- The build script contains behavior that is allowed only in lower Gradle version.
- The bundle versions and project configurations conflict with each other.
- The build script contains Gradle syntax error.
- The build script needs full commit record.
- The build script executes shell command without permission.
- The build script includes subproject that does not exist.
- The project is built by customized command, and cannot be built by our default command.
- Other behaviors like defining same-named task, using undefined dependency configuration, asking for user input at runtime, circular dependency, and etc.

4.3 Build Error Classification

In this section, we will discuss in depth of the classification result, including coverage of build errors, most frequent error types, the impact of using Gradle wrapper, and the dependency of local environment settings. (see Appendix A for details of the classification result.)

4.3.1 Coverage of build errors

We applied the classification criteria to the 1,853 error logs of the failed builds, and found that there were 393 errors (21.21%) that could not be recognized by any known error pattern. That is, our coverage of the build errors is 78.79%.

We then checked the 393 unrecognized error logs manually, and found that 384 of them actually fell into some error type. Namely, our error type taxonomy based on root cause analysis is able to cover over 99.5% of build errors.

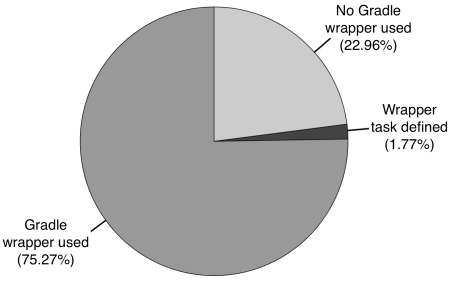

Fig. 5. Three partitions of the usage of Gradle wrapper

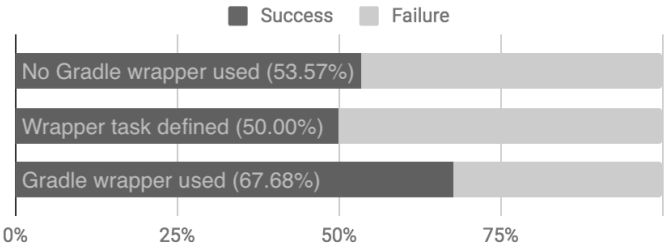

Fig. 6. Build success rate by the usage of Gradle wrapper

4.3.2 Most frequent error types

The top 10 frequent error types are shown in Table 2. There were 271 projects (14.62%) that did not upload required Jar file to Github despite of their usage of Gradle wrapper. There were also 261 projects suffered from compilation error, which may be caused by incompatible JDK version or other exception in Sulir et al [1] and Seo et al [2].

Figure 4 shows the frequency by error categories. We ignore the non-Gradle category and the wrapper category here since both of which are basically not caused by Gradle build script. We can then find that the most frequent error category is dependency, which fits the result in Sulir et al [1], following by the property category, which is mainly caused by lacking manual configuration, and the plugin category, which is mainly caused by Gradle version.

4.3.3 Usage of Gradle Wrapper

In this section, we will discuss how the usage of Gradle wrapper impacts the build success rate.

Since features and syntax in Gradle vary with the executing Gradle version, Gradle developer team provides the feature of Gradle wrapper to deal with this issue, which is not officially provided in other Java build tools like Maven and Ant.

We divided the 5,188 projects into three parts: projects using Gradle wrapper, which contained a gradlew script in their project directory; projects with a wrapper task defined, which contained no gradlew script but defined a wrapper task in their build script; and the rest were projects not using Gradle wrapper. Figure 5 shows the partition of the 5,188 projects by the usage of Gradle wrapper.

We can see in Figure 6 that projects using Gradle wrapper have a build success rate of 67.68%, higher than the

overall build success rate, while projects belong to the other parts have only about 50%. In addition, if we exclude the 271 projects with un-executable Gradle wrapper, the build success rate of projects using Gradle wrapper is 72.73%.

If we focus on error types relating to Gradle version, including type 2.1, 2.2, 2.3, 2.4, 3.1, 3.2, 4.1, 4.2, 4.8, and 8.2, 215 projects not using Gradle wrapper and 28 projects with wrapper task defined failed their build because of incompatible Gradle version, accounting for 18.05% and 30.43%, respectively. As for projects using Gradle wrapper, the number is 23, accounting for only 0.59%.

Note that using Gradle wrapper can effectively decrease the build failure due to Gradle version-related error types. Though some users think to include Jar in Github repository is not recommended and exclude the 'gradle-wrapper.jar' from their repository, our findings indicate that this kind of projects leads to a less success rate than not using Gradle wrapper. As a result, we recommend that developers should make the best of Gradle wrapper.

4.3.4 Dependency of local environment settings

During the manual classification of build errors, we noticed that some projects failed because of their dependency on local environment settings. For example, some projects configured their authentication property from environment variables instead of local configuration file like 'gradle.properties'; some projects included subproject or depended on file from outside of their project directories; some projects checked the operating system or directory names and terminated the build. Though these kinds of projects account for a relatively small part in our work (about 2% of build errors), a high environment dependency can surely damage the build-ability of open source projects. Therefore, we recommend that developers should always keep an eye on whether their projects depend on their local developing environment too much.

5 Conclusion

With usage of continuous integration tool, Jenkins CI, and a standardized build process, we tested the build-ability of over 5,000 Java projects built by Gradle from Github in an automatic manner. The result of the testing shows that about 36% of these projects have failed the build.

By conducting root cause analysis on the build error logs of 2,744 projects, we proposed our build error type taxonomy and build error classification criteria. The result of applying our classification criteria to the above build-ability test shows that our taxonomy covers over 99% of build errors and our classification criteria recognizes 78.79% of build errors.

We came up with two recommendations to improve the build-ability of open source projects: to include an executable Gradle wrapper script, and to decrease the dependency on local development environment.

Among the listed 45 error types in our build error taxonomy, over 30 error types are able to be fixed with a simple step. For example, Gradle version related error types can be fixed by giving a proper Gradle version with Gradle wrapper; perfecting local configuration file can prevent property-missing error types. Based on the automatic build error classification scheme we proposed, we are planning to build an automatic build error fixing scheme, which will start a continuous iterating process of error classification and fixing until meeting a non-Gradle error type, like a compilation error or a Javadoc generation error.

Our research focused on Java projects using Gradle as their build tool only. For the sake of generality, other build tools like Maven and Ant will be taken into consideration in our future research. Similar research can also be conducted on other programming languages like C and C++.

References

[1] M. Sulir, and J. Poruban. *A Quantitative Study of Java Software Buildability*. In PLATEAU'16, November 1, 2016, Amsterdam, Netherlands ACM. ISBN 978-1-4503-4638-2.

[2] H. Seo, C. Sadowski, S. Elbaum, E. Aftandilian, and R. Bowdidge. *Programmers' build errors: A case study (at Google)*. In Proceedings of the 36th International Conference on Software Engineering, ICSE 2014, pages 724-734, New York, NY, USA, 2014. ACM. ISBN 978-1-4503-2756-5. doi: 10.1145/2568225.2568255.

[3] T. Rausch, W. Hummer, P. Leitner, E. Aftandilian, and S. Schulte. *An Empirical Analysis of Build Failures in the Continuous Integration Workflows of Java-Based Open-Source Software*. In Proceedings of the 14th International Conference on Mining Software Repositories (MSR), IEEE/ACM 2017. ISBN 978-1-5386-1544-7. doi: 10.1109/MSR.2017.54.

[4] F. Hassan, S. Mostafa, E. Leitner, E. S.L. Lam, and X. Wang. *Automatic Building of Java Projects in Software Repositories: A Study on Feasibility and Challenges*. In International Symposium on Empirical Software Engineering and Measurement (ESEM), IEEE/ACM 2017. ISBN 978-1-5090-4040-7. doi: 10.1109/ESEM.2017.11.

[5] C. Macho , S. McIntosh , and M. Pinzger. *Automatically repairing dependency-related build breakage*. In 25th International Conference on Software Analysis, Evolution and Reengineering (SANER), IEEE 2018. ISBN 978-1-5386-4970-1. doi: 10.1109/SANER.2018.8330201.

[6] N. Kerzazi, F. Khomh, and B. Adams. *Why do automated builds break? An empirical study*. In Software Maintenance and Evolution (ICSME), 2014 IEEE International Conference on, pages 41-50, Sept 2014. doi: 10.1109/ICSME.2014.26.

[7] S. McIntosh, M. Nagappan, B. Adams, A. Mockus, and A. E. Hassan. *A large-scale empirical study of the relationship between build technology and build maintenance*. Empirical Software Engineering, 20(6):1587-1633, 2015. ISSN 1382- 3256. doi: 10.1007/s10664-014-9324-x.

[8] "Gradle Build Tool" https://gradle.org/

[9] "Maven - Welcome to Apache Maven" https://maven.apache.org/

[10] "Apache Ant - Welcome" https://ant.apache.org/

[11] "Githut 2.0 - A small place to discover languages in Github" https://madnight.github.io/githut

Appendix A
The classification result of Gradle build errors

Category	Type	Description	Count	Percentage
Wrapper	1.1	Un-executable Gradle wrapper	271	14.62%
	1.2	Gradle wrapper distribution not found	26	1.40%
	1.3	Non-classic Gradle wrapper layout	1	0.05%
Plugin	2.1	Plugin only available in lower Gradle version	48	2.59%
	2.2	Plugin only available in higher Gradle version	6	0.32%
	2.3	Plugin only supported in lower Gradle version	110	5.94%
	2.4	Plugin only supported in higher Gradle version	15	0.81%
	2.5	Plugin with exception used	17	0.92%
	2.6	Plugin 'application' lack of mainClassName	24	1.30%
	2.7	Plugin 'sprint-boot' packing error	23	1.24%
	2.8	Script applied disappeared	2	0.11%
	2.9	BuildScript dependencies not defined	12	0.65%
Method	3.1	Method only available in lower Gradle version	32	1.73%
	3.2	Method only available in higher Gradle version	5	0.27%
Property	4.1	Property only available in lower Gradle version	15	0.81%
	4.2	Property only available in higher Gradle version	1	0.05%
	4.3	Missing signing properties	67	3.62%
	4.4	Missing authentication properties	107	5.77%
	4.5	Missing AWS properties	4	0.22%
	4.6	Need local config file	20	1.08%
	4.7	Need environment variable	7	0.38%
	4.8	Set unknown property	18	0.97%
	4.9	Get unknown property	56	3.02%
Dependency	5.1	No repository defined	6	0.32%
	5.2	Unreachable repository defined	78	4.21%
	5.3	Usage of ivy syntax	9	0.49%
	5.4	Local file dependency	13	0.70%
	5.5	Download file from dead source	13	0.70%
	5.6	Dependency not found	189	10.20%
Java Version	6.1	Need lower Java version	36	1.94%
	6.2	Need higher Java version	52	2.81%
Non-Gradle	7.1	Compilation failed	261	14.09%
	7.2	Test case failed	6	0.32%
	7.3	Javadoc failed	50	2.70%
	7.4	Ant script error	3	0.16%
	7.5	Error from third-party program	62	3.35%
Others	8.1	Plugin property misusage	20	1.08%
	8.2	Illegal behavior in higher Gradle version	16	0.86%
	8.3	Config error	1	0.05%
	8.4	Syntax error	7	0.38%
	8.5	Need full clone	75	4.05%
	8.6	Permission denied	2	0.11%
	8.7	Subproject not exist	16	0.86%
	8.8	Non-typical build command	33	1.78%
	8.9	Bad behavior	9	0.49%
Unclassified	9	-	9	0.49%