The Dispersion of Build Maintenance Activity across Maven Lifecycle Phases

Casimir Désarmeaux, Andrea Pecatikov, and Shane McIntosh
McGill University, Montréal, Canada
{andrea.pecatikov,casimir.desarmeaux}@mail.mcgill.ca, shane.mcintosh@mcgill.ca

ABSTRACT

Build systems describe how source code is translated into deliverables. Developers use build management tools like MAVEN to specify their build systems. Past work has shown that while MAVEN provides invaluable features (e.g., incremental building), it introduces an overhead on software development. Indeed, MAVEN build systems require maintenance. However, MAVEN build systems follow the build lifecycle, which is comprised of `validate`, `compile`, `test`, `packaging`, `install`, and `deploy` phases. Little is known about how build maintenance activity is dispersed among these lifecycle phases. To bridge this gap, in this paper, we analyze the dispersion of build maintenance activity across build lifecycle phases. Through analysis of 1,181 GitHub repositories that use MAVEN, we find that: (1) the compile phase accounts for 24% more of the build maintenance activity than the other phases; and (2) while the compile phase generates a consistent amount of maintenance activity over time, the other phases tend to generate peaks and valleys of maintenance activity. Software teams that use MAVEN should plan for these shifts in the characteristics of build maintenance activity.

1. INTRODUCTION

In order to produce official releases of a software system, the build system must be executed to translate its source code into deliverables. Build systems automate *build processes*, which (among other tasks) are often responsible for (1) configuring the build environment, (2) executing only the necessary order-dependent build commands to synchronize source code with deliverables, and (3) running test suites to ensure that the system is ready for delivery to users.

Build systems are typically specified and executed using build management tools like MAKE, ANT, or MAVEN [7]. First, stakeholders specify their build processes in build specifications (`Makefile`, `build.xml`, and `pom.xml` files in MAKE, ANT, and MAVEN, respectively). To execute the build process, the build management tool is executed, which will analyze the build specifications to determine what commands needs to be executed. Build management tools differentiate themselves from naïve build scripts by supporting *incremental builds*, which analyze the current state of the source code and previously built deliverables to calculate and execute the minimal set of commands necessary to synchronize the deliverables with any changes made to the source code.

While build systems provide features like incremental building, they come at a cost—build systems require maintenance. Indeed, just as source code tends to grow unless explicit effort is invested in refactoring it [2], so too does the build system [5]. This growth of the build system introduces overhead on software development [4, 6].

While past work has made important discoveries about build maintenance activity, it has generally focused on file- or line-level activity. Little is known about how build maintenance activity is dispersed among the subtasks of the build process. For example, we do not know if maintenance effort is mainly spent on build configuration details, preserving the order-dependencies among build commands, or updating test suite automation.

In this paper, we analyze how build maintenance affects MAVEN build phases. To do so, we analyze 1,181 repositories in the MSR challenge dataset [3] that have MAVEN build specifications. We map the line-level changes that occur in these repositories to the MAVEN build phases that they impact in order to address the following research questions:

(RQ1) How much maintenance does each build lifecycle phase require?
Although the median of the amount of maintenance activity per project is relatively similar for most phases, the `compile` phase accounts for 24% more of the total maintenance activity on average.

(RQ2) How does the maintenance of each build lifecycle phase change over time?
While the maintenance activity of the `compile` phase remains consistent over time, the `validate` phase peaks within the 10 first commits. On the other hand, the `packaging`, `install`, and `deploy` phases peak much later in the development history (between the 50th and 80th commits).

Our results indicate that build maintenance activity is not evenly dispersed among build lifecycle phases, with the `compile` phase dominating maintenance activity, but with other phases peaking throughout a project's lifetime. Software teams should plan for these shifts in the characteristics of build maintenance activity.

MSR'16, May 14-15 2016, Austin, TX, USA

ISBN 978-1-4503-4186-8/16/05…$15.00

DOI: http://dx.doi.org/10.1145/2901739.2903498

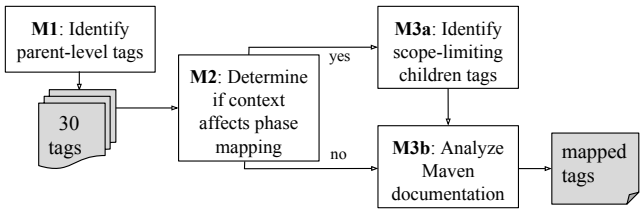


Figure 1: An overview of our MAVEN tag-to-phase mapping process.

Paper organization. The rest of the paper is organized as follows. Section 2 describes the MAVEN build tool. Sections 3 and 4 discuss the design and results of our case study, respectively. Finally, Section 5 draws conclusions.

2. MAVEN

When a project is built with MAVEN, a series of six build phases are executed, each depending on the previous one. These phases constitute the default MAVEN lifecycle,[1] and are described as follows:

Validate checks if all of the settings that are required to build the project are specified in a valid way.

Compile translates the source code of the project into raw deliverables (e.g., `.class` files).

Test compiles and executes test suites to check for errors.

Packaging bundles the raw deliverables into the target deliverable format (e.g., `.jar`, `.war`, `.ear`).

Install places the deliverables in their target location(s) on the local system.

Deploy transmits the local installation to production environments.

2.1 Mapping Maven tags to Lifecycle Phases

In order to lift the build maintenance activity from the line-level to the lifecycle phase-level, we establish a mapping between each potential `XML` tag in MAVEN build specifications and the phase of the build lifecycle that they impact. To do so, we follow the procedure outlined in Figure 1. We describe each step in the mapping process below.

M1: Identify parent-level tags. This consists of inspecting the MAVEN documentation to find all elements of depth 1 in the MAVEN `pom.xml` tree structure. For instance, the tags on lines 2-8, 14, and 23 of Figure 2 are the depth 1 parent tags of this example `pom.xml` file.

M2: Determine if context affects phase mapping. While phase mapping at the tag-level is usually sufficient, some tags are dependent on their context to provide a one-to-one phase mapping. In this step, we identify the context-dependent depth 1 parent tags. For example, the `dependencies` tag on line 23 of Figure 2 is context-dependent.

M3a: Identify phase-impacting children tags. In order to determine the phase of context-dependent depth 1 parent tags, we analyze their immediate children tags. We analyze these children tags in search of the tag that will bind the parent to a phase. For example, in Figure 2, the `scope` tag on line 28 binds a `dependency` to a phase (in this case, to the `test` phase).

M3b: Analyze online documentation. We read the MAVEN documentation in order to finalize our mapping of identified tags to lifecycle phases. To assist in replication studies, we have made our mapping available online.[2]

```
1  <project>
2    <groupId>org.sandag</groupId>
3    <artifactId>SANDAG-ABM</artifactId>
4    <version>13.2.4</version>
5    <name>SANDAG CT-RAMP Activity Based Model</name>
PACKAGING
6    <packaging>jar</packaging>
DEPLOY
7    <url>http://svn.sandag.org:8081/nexus/repositories/releases/</url>
VALIDATE
8    <repositories>
9      <repository>
10       <id>nexus-sandag-public</id>
11       <url>http://svn.sandag.org:8081/nexus/groups/public/</url>
12     </repository>
13   </repositories>
14   <build>
COMPILE
15     <plugins>
16       <plugin>
17         <groupId>org.apache.maven.plugins</groupId>
18         <artifactId>maven-compiler-plugin</artifactId>
19         <version>3.0<version>
20       </plugin>
21     </plugins>
22   </build>
TEST
23   <dependencies>
24     <dependency>
25       <groupId>junit</groupId>
26       <artifactId>junit</artifactId>
27       <version>4.10</version>
28       <scope>test</scope>
29     </dependency>
30   </dependencies>
31 </project>
```

Figure 2: Example `pom.xml` file with phase mappings.

3. CASE STUDY DESIGN

The mining challenge dataset provides 7,830,023 repositories, and the full development histories with parsed abstract syntax trees for JAVA repositories. Figure 3 provides an overview of the 3-step approach that we use to address our research questions using the mining challenge dataset. Step 1 is performed using the Boa tool [3], while steps 2 and 3 are performed on the studied Git repositories. Below, we describe each step in the process.

3.1 Data Filtering

Prior to analyzing the data, we first filter the challenge dataset to select suitable repositories for analysis. After the filtering, the remainder of the case study design will be performed independently of Boa.

DF1: Select Java projects. Since MAVEN is primarily used for JAVA projects, the initial step was to select the projects that are labeled as JAVA projects in the challenge dataset. 554,864 repositories survive this filter.

DF2: Select Maven Projects. After selecting the JAVA repositories, we need to select those repositories that are using the MAVEN build management tool for our analysis. We do so by selecting only those repositories that have at least one `pom.xml` file. 75,958 repositories survive this filter.

DF3: Select Representative Sample. Due to computational restraints, we cannot analyze all 75,958 repositories that use MAVEN. Hence, we randomly select a statistically representative sample of 1,181 repositories, which yields a 99% confidence level that the reported results are within ±4% bounds.

3.2 Data Extraction

[1] http://is.gd/OVqR19

[2] https://figshare.com/s/6a80f8153e66f76615e5

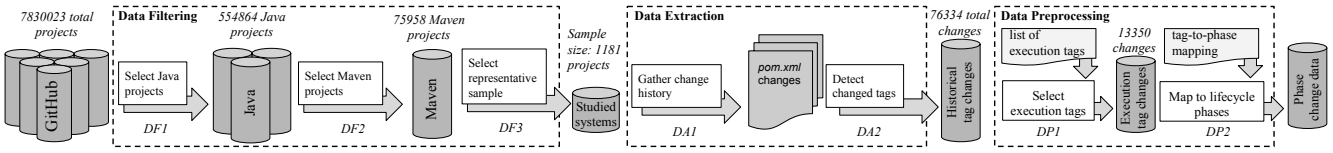


Figure 3: Our approach to study the dispersion of build maintenance activity across MAVEN lifecycle phases.

After filtering the challenge dataset, we extract raw data from the 1,181 sampled repositories. Our data extraction approach is comprised of two steps, which we describe below.

DA1: Gather change history. First, we collect the total line-level historical changes of the `pom.xml` files at every commit in the studied MAVEN repositories.

DA2: Detect changed tags. Next, we lift the line-level `pom.xml` change data to the tag-level by identifying the `XML` tags that are impacted by each line-level change. We consider that a tag is impacted by a line-level change if it has been added, removed, or modified by a commit. We also associate the depth of the changed line in the XML tree structure to identify the enclosing tags.

3.3 Data Preprocessing

After producing the tag-level change data, we preprocess it before using it to address our research questions. Our preprocessing procedure is composed of two steps.

DP1: Select execution tags. MAVEN build specifications contain tags that may not impact the execution of the build process. For example, the project metadata tags on lines 1-5 of the example `pom.xml` file in Figure 2 do not impact the execution of that build process. Since these non-execution tags may introduce noise in our analysis, we preprocess the tag-level change data to select only the execution tags.

DP2: Map to lifecycle phases. After selecting the execution-impacting tags, we use our tag-to-phase mapping (see Section 2.1) to lift the tag-level change data to the lifecycle phase-level. We consider that a change c impacts a phase p if any of the tags changed by c are mapped to p. Since a change can modify several tags that span multiple phases, in our analysis, a change can impact several phases.

4. CASE STUDY RESULTS

In this section, we present the results of our case study with respect to our two research questions. For each research question, we discuss our approach and our observations.

(RQ1) How much maintenance does each build lifecycle phase require?

While recent studies have analyzed build maintenance [1, 5, 6], the impact that changes have on phases of the build process remains largely unexplored. Since it may be the case that some phases are more prone to maintenance than others, we first set out to study the quantity of change with respect to each phase of the MAVEN build lifecycle.

Approach. In order to address RQ1, we compute the relative amount of phase-level change for each phase of the build lifecycle in each project. Figure 4 provides an overview of the results using boxplots.

Observations. **The `compile` phase generates the most build maintenance activity**. We observe that many of these `compile` phase changes are due to churn of `dependencies` and `plugins`. After applying the tag-to-phase mapping of Section 2.1, we find that the 90% of `dependencies` were bound to the `compile` phase.

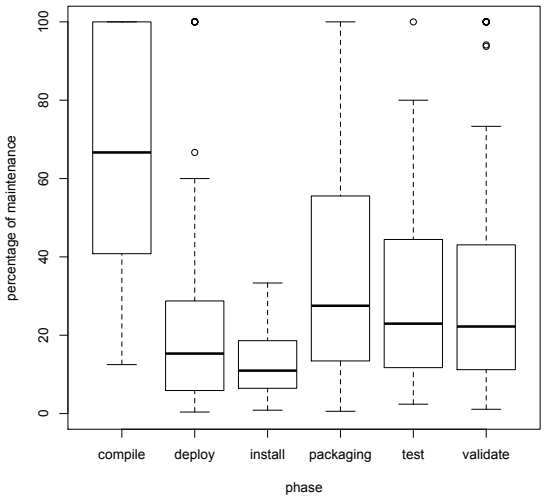


Figure 4: The quantity of change per phase.

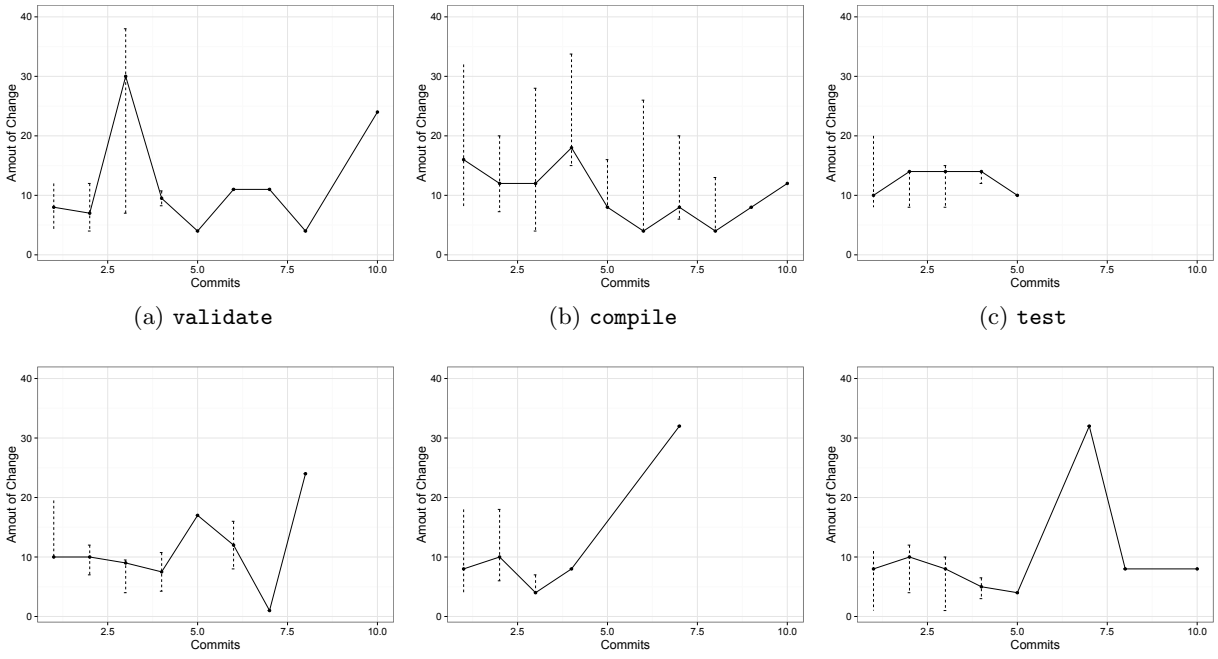
Turning to the `validate`, `packaging`, `deploy`, `install`, and `test` phases, we observe that the medians of the `test`, `validate` and `packaging` phases are higher than the others. This is likely because 74% and 20% of the `dependencies` that are not bound to the `compile` phase are bound to the `test` and `validate` phase respectively. As for `packaging`, there is no built-in feature in MAVEN to allow one to specify the multiple `packaging` types. Consequently, projects that need to produce packages of multiple types rely on `plugins`, which generate maintenance activity in the `packaging` phase. This is highlighted by the collected data—54% of the `plugins` are bound to the `packaging` phase.

We observe that the `compile` *phase dominates when we analyze the build maintenance activity per build lifecycle phase, likely due to large proportion of dependencies (90%) that are bound to it.*

(RQ2) How does the maintenance of each build lifecycle phase change over time?

To gain more insight into MAVEN maintenance, we analyze how the phase-level maintenance activity changes over time.

Approach. To address RQ2, we analyze how the quantity of phase-level build maintenance activity changes over time. We compute the amount of build maintenance activity. Since the studied systems vary in age, we analyze the first 100 commits to establish a common timeframe. We split these 100 commits into 10 quantiles of 10 commits each



(a) `validate` (b) `compile` (c) `test`

(d) `packaging` (e) `install` (f) `deploy`

Figure 5: The quantity of change per phase over time.

and plot the results using line plots in Figure 5. The trends show the medians, while the error bars show the spread of values across 95% of the studied systems.

Observations. **Each phase has a time period when it becomes the most actively maintained.** Figure 5a shows that `validate` is the most actively maintained phase in quantile 3. Similarly, Figures 5b and 5c show that `compile` and `test` are the most actively maintained phases of quantiles 1 and 4, respectively. Figures 5d, 5e, and 5f show that `packaging`, `install`, and `deploy` are the most actively maintained phases in quantiles 5, 7, and 8, respectively.

Although the `compile` and `test` phases do not show clear peaks, the order in which lifecycle phases become the most actively maintained roughly corresponds to the order in which phases are executed in the build lifecycle. Figure 5 shows that the `compile` phase peaks first in quantile 1. This is to be expected because this is likely the first concern for developers (getting the code to compile). After this, the order in which phases peak in Figure 5 corresponds to the order in which lifecycle phases are executed (see Section 2), i.e., `validate` (Q3), `test` (Q4), `packaging` (Q5), `install` (Q7), and `deploy` (Q8).

While the `compile` phase is the most actively maintained phase overall, each phase has a time period when it becomes the most actively maintained. The order in which phase activities peak mirrors the order of execution.

5. CONCLUSIONS

Past work has shown that Maven build systems require maintenance [5, 6]. However, these studies have largely focused on maintenance activity at the file- and line-levels. Since build systems are composed of phases, in this paper, we lift build maintenance activity to the phase-level and find that the `compile` phase is typically the most actively maintained build phase. However, at one point or another, each phase of the build lifecycle peaks, briefly becoming the most actively maintained build phase. The order in which phases peak mirrors to the order in which they are executed during the build process. Software teams should plan for these shifts in the characteristics of build maintenance activity, possibly be defining the required modular components as well as testing and target environments ahead of time.

6. REFERENCES

[1] B. Adams, K. D. Schutter, H. Tromp, and W. D. Meuter. The Evolution of the Linux Build System. *Electronic Communications of the ECEASST*, 8, 2008.

[2] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976.

[3] R. Dyer, H. A. Nguyen, H. Rajan, and T. N. Nguyen. Boa: A Language and Infrastructure for Analyzing Ultra-Large-Scale Software Repositories. In *Proc. of the 35th Int'l Conf. on Software Engineering (ICSE)*, pages 422–431, 2013.

[4] L. Hochstein and Y. Jiao. The cost of the build tax in scientific software. In *Proc. of the 5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 384–387, 2011.

[5] S. McIntosh, B. Adams, and A. E. Hassan. The evolution of Java build systems. *Empirical Software Engineering*, 17(4-5):578–608, August 2012.

[6] S. McIntosh, B. Adams, T. H. D. Nguyen, Y. Kamei, and A. E. Hassan. An Empirical Study of Build Maintenance Effort. In *Proc. of the 33rd Int'l Conf. on Software Engineering (ICSE)*, pages 141–150, 2011.

[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.