Composing a cross-platform development environment using Maven

Terry J. Speicher[1,2]✉[0000−0003−3912−8418] and Yoonsik Cheon[2]

[1] Timely Enterprises, Inc. El Paso, TX 79932, USA
http://www.timelyep.com
terry@timelyep.com
[2] Department of Computer Science
The University of Texas at El Paso, El Paso, TX 79968, USA
ycheon@utep.edu

Abstract. It is challenging to develop a cross-platform application, that is, an application that runs on multiple platforms. It requires not only code reuse but also an individual building or compilation for each platform, possibly with different development tools. In this paper, we show a simple approach for creating a cross-platform application using Maven, a build tool. We configure a cross-platform development environment by composing a set of platform-specific tools or integrated development environments (IDE's). The key idea of our approach is to use Maven to immediately propagate changes made using one platform tool, or IDE, to other platforms. For this, we decompose an application into platform-independent and platform-dependent parts and make the individual tools and IDE's share the platform-independent part in the form of a reusable component or library. We explain our approach in detail by creating a sample application that runs on the Java platform and the Android platform. The development environment consists of IntelliJ IDEA (for Java) and Android Studio. Our approach provides a way to set up a custom, cross-platform development environment by letting software developers pick platform-specific tools of their choices. This helps maximize code reuse in a multi-platform application and creates a continuous integration environment.

Keywords: Continuous integration · cross-platform application · software development tool · Maven · Android platform · Java platform

1 Introduction

This paper is about using a popular industry tool, Maven, to simplify tasks on a personal computer with regards to programming in Java. The Apache Maven is a project management and comprehension tool [6] (see Section 2 for a quick introduction to Maven). We show how a small investment of time in learning this tool can help a Java developer with: organizing code for sharing and reuse, utilizing a standard project and file structure, and most importantly, being able to share Java code across multiple platforms or IDE's on the same computer.

For clarification in this paper, we will loosely consider the terms "library", "artifact", and ".jar files" to be interchangeable. All terms refer to the underlying .jar, .ear, or .war files that contain Java code. Technically, a library would be the code that is stored either on a hard drive or in a repository and is brought into a project to add the functionality of previously created code. In addition, an "artifact" is the compiled .jar file that is the result of the project currently being worked on. An "artifact" may be stand alone code, such as a finished project, or it may be used as a "library" for a bigger project. In addition, we will use the term "dependency" to refer to any "library" that is required by a Java program.

1.1 Corporate Environment

Maven can be used in the work sector as an integration tool that will allow completed modules to be shared between computers working on a common project. As each Java developer works on a portion of a larger project, the compiled code, along with its source code and Javadoc, is uploaded to a dedicated repository server set up by that company. Not only can code be uploaded to the repository, but any updates to other modules can be downloaded and integrated into the development environment. Thus, each developer can use their IDE (or other development environment, including the Windows command line or terminal on Linux or Mac) on their own computer. This is commonly known as cross-platform application development and continuous integration.

1.2 Personal Environment

Our problem, however, was on a smaller scale. We mostly develop smaller applications for use only on our own computer. But we still had the need to develop our own libraries or Java .jar files that would provide some list of common functions for use in multiple other projects. These needs even extended to using the libraries we developed across multiple platforms such as Eclipse, IntelliJ IDEA, and Android Studio.

As we mention later in Section 5, we found that we could utilize Maven to automatically download dependencies, along with automatically setting up a correct and standardized project structure. Furthermore, the repository that is installed on the local computer leads to a personal level of "continuous integration" by delivering any artifact changes to any other environment on that computer. The details of this encounter are laid out in Section 5.

Programming in Java creates artifacts. An individual developer will be required to keep track of needed projects and artifacts. The interaction of projects or other dependencies can make future code alterations and artifact sharing difficult. Updating code becomes problematic after a sufficient amount of time has passed. While there exist build and versioning methods in a business programming environment, there is little talk of implementing such a solution on a personal level for a student or full stack developer. In addition, learning these

methodologies at a personal and educational level will help facilitate better programming practices when a student or developer moves into a business environment. Maven is a build tool that can be used to setup such an environment on a personal computer so that changes made in a Java .jar file from one environment can be immediately propagated to other programming environments, including updating and storing the project's source code and Javadocs. This creates a "personal continuous integration" environment for a single developer on a single workstation.

1.3 Outline

The rest of this paper is structured as follows. In Section 2, we give a quick overview of the Apache Maven, an open-source build tool. In Section 3, we describe the problem of keeping track of artifacts and libraries that can be used in other projects and the inherent difficulties of having to update a library that is used as a dependency in multiple projects. In Section 4, we give an overview of how Maven uses repositories to download and store libraries. This commentary should help lay the groundwork for Section 5, where we go into detail about how we used the repository to successfully coordinate code and code changes across multiple development platforms.

2 Apache Maven

2.1 Maven Overview

As mentioned previously, Apache Maven is a key component of our approach. It is used as a glue for a set of platform-specific tools, thus creating across-platform development environment.

Apache Maven is a software project management and comprehension tool, and it can manage a project's build, reporting and documentation from a central piece of information [6]. This simple statement from the Maven website [9] is akin to describing the tip of an iceberg. We will, of course, attempt to provide a primer on the aspects of Maven that apply to this paper. As such, we offer this cursory overview of several key features to entice the reader into further consideration of Maven. Full resources on Maven are available at its website [9] (refer to `https://maven.apache.org`).

Archetypes Maven can be used to create a new project using *archetypes*. Archetype is a Maven project template or prototype from which other similar projects are to be generated [6, Chapter 3]. For example, we can generate a new Java project using the "quickstart" archetype, an archetype for generating a sample project[3]. The result of creating a new project with the "quickstart"

[3] In this paper, we use IntelliJ IDEA and the "quickstart" archetype for reference. This procedure in other IDE's and command lines may vary slightly, but not significantly, since one of Maven's objectives is standardization.

archetype is a project with directories already created and marked as "sources root", "resources root", "test sources root". As soon as a project is created from an archetype, the directory structure contains the folder for code along with a sample "Hello World" class and the corresponding JUnit test class for reference.

POM Central to the way that Maven operates is the *Project Object Model* (*POM*) [6, Chapter 9]. It is an XML representation of a Maven project held in a file named `pom.xml` containing all necessary information about a project. It is Maven's main configuration file and is used extensively to direct the project after the initial structure is created. In creating a project, there are three pieces of information that are vital and unique to each project: groupID, artifactID, and version. This information is among the first few lines of the POM file and is used to distinguish each project in the repository. Below is an example of the first half of a sample POM file from a newly created Maven quickstart project in IntelliJ:

```
<?xml version="1.0" encoding="UTF-8"?>

<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="
    http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http:
    //maven.apache.org/xsd/maven-4.0.0.xsd">
  <modelVersion>4.0.0</modelVersion>

  <groupId>com.timelyep.com</groupId>
  <artifactId>mavenTest1</artifactId>
  <version>1.0-SNAPSHOT</version>

  <name>mavenTest1</name>
  <url>http://www.example.com</url>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.
    sourceEncoding>
    <maven.compiler.source>1.7</maven.compiler.source>
    <maven.compiler.target>1.7</maven.compiler.target>
  </properties>

  <dependencies>
    <dependency>
      <groupId>junit</groupId>
      <artifactId>junit</artifactId>
      <version>4.11</version>
      <scope>test</scope>
    </dependency>
  </dependencies>
  <!-- ... -->
```

Listing 1.1. The first part of POM.xml

Notice the "dependencies" section of the `POM.xml` file. This is of particular importance because any library that is needed, such as the "junit", may be automatically imported into the project by listing it here. The three identifiers listed can be used to identify and add any dependency to the project. The quickstart project comes with the junit dependency listed in the POM file because the project structure includes JUnit testing. At the time of this writing, the basic Maven project has included JUnit version 4.11. This version of JUnit has its own dependency, namely, "hamcrest-core, version 1.3". No matter which environment is being used, specifying Junit 4.11 as a dependency will automatically download Hamcrest-core 1.3 from the online Maven repository. This is called a transitive dependency and is also key to Maven's functionality and this paper.

Plugins After setup of the Maven directory structure and dependency listings in the `POM.xml` file, Maven provides for compilation, testing, reporting, and deployment of Java code, among other tasks. Each of these steps are specified by "Plugins" [6, Chapter 17] and each Plugin contains "goals", or specific subtasks. There are two plugins that we wish to explain for the purpose of this introduction and case study. The first is the "clean" plugin which simply deletes the folder called "target" along with any reports, artifacts, and test results that were created during a previous build. The second plugin is the "install" plugin which first executes all the other plugins to compile, run tests, create reports, package the artifact, and finally, place the artifact into the local Maven repository on the local computer. This plugin provides the local repository with the three identifiers that will allow this compiled artifact to be used in other projects in the future. The fact that this plugin puts the artifact into the repository is a key to the case study outlined in Section 5.

2.2 Maven Platforms

For its overall functionality, Maven is a build tool that has been developed and maintained by Apache. Maven can be run from a command line or is available with extensive support in the three most popular Java IDE's: Eclipse, Netbeans, and IntelliJ IDEA, as well as others. The Maven repository is available to other build tools such as Gradle; Android Studio uses Gradle. As a build tool, Maven provides organization and standardization of project structures as previously outlined. Maven provides versioning of artifacts in the repository and provides testing and reporting tasks as part of the constant development process. The Maven repository is useful not only for organizing artifacts, but can also store source code and Javadoc for each artifact. Since Maven is based on plugins, or tasks, it has extensive support for development of custom plugins [6, Chapter 17]. As will be seen in this paper, Maven is scalable. While Maven provides support for large organizations and programming teams, Maven is also powerful on an individual scale.

3 The Problem

Since there are multiple platforms which a developer can choose from, there is not one standard. In fact, a developer often needs to use more than one development environment. For example, a coder may have a preference for writing Java code in Eclipse or IntelliJ IDEA, but may prefer the enhanced GUI developing capabilities of Netbeans. This would require artifacts from one IDE to be imported as libraries into another IDE.

As programs or modules are developed, they often need to be used as building blocks, or libraries, and integrated into larger projects. Even on a small, individual scale, there is a need for continuous integration [1] [4] [11] as the case study in this paper indicates (see Section 5). Our term for this is "Personal Continuous Integration". Our experience came from developing code in one IDE (IntelliJ IDEA) and using it in a different IDE (Android Studio). We then needed to be able to change and recompile code in one IDE and have the change automatically imported into the other IDE.

There are also other situations that create code that needs to be maintained. One situation is in developing a library, and another is when a program developer wants to extract a library, say as part of a Model-View-Control software development/design pattern [5]. In both of these situations, an artifact is created that needs to be distributed, maintained, and redistributed. The usual process for developing such code on an individual basis might be demonstrated by looking at the development of a game of "Battleship". If the game was to be developed for multiple platforms, one could decide to pull out the "core of the solution" into a model that would perform the basic handling of the "behind the scenes" work of keeping track of players, ships, boards, and scores. A typical development cycle might include the following scenario. A developer creates Java code and compiles it into a .jar file (artifact). This artifact can now be added to other Java projects. This artifact is usually located in a local directory, which is deeply embedded in the directory structure of the development project. The artifact must then be copied to a location so that it can be loaded into the main project. This has to be done manually for every project that uses this artifact as an external library. Having to go back to update, debug, fix, or further develop the artifact means that the .jar file must be recompiled and recopied to all projects that utilize the updated artifact. Furthermore, any dependencies of the artifact must be loaded into projects that use the artifact. Consider also that if any of the artifact's dependencies should need to be updated, then those dependencies must also be tracked and updated in any project that uses the artifact.

In summary, keeping track of Java archive (.jar, .war, or .ear) files becomes a task that must be handled by a Java programmer at the intermediate level and above. These Java artifacts become modules that work together, or they become the libraries that other modules use. After a developer has several projects on a local computer, it becomes tedious to keep track of artifact locations and versions. Should any artifact need to be updated, it must then be copied to any other project that used the artifact as a dependency.

4 Our Approach

The core of our approach to configuring a cross-platform development environment is to decompose an application into two parts: a *platform-independent part* and a *platform-dependent part*.

- Platform-independent part (PIP): the part of an application that doesn't depend on a specific implementation platform such as application programming interfaces (API's). In MVC, the model is a good candidate for the PIP.
- Platform-dependent part (PDP): the part of an application that does depend on a specific platform, e.g., an API available only in a specific execution environment. In MVC, the view and the view-specific control correspond to the PDP.

The purpose of this distinction is to share the PIP across platforms while developing a specific PDP on each target platform. Thus, the key criterion on determining the PIP of an application is whether the code can be shared on all the target platforms of the application. Note that our notions of PIP and PDP are similar to those of platform-independent models (PIM) and platform-specific models (PSM) in model-driven software development [2] [10]. As stated earlier, Maven plays a central role in our approach by immediately propagating changes on the PIP made using a platform tool, or IDE, to other platforms. In a sense, PIP is a reusable component or library.

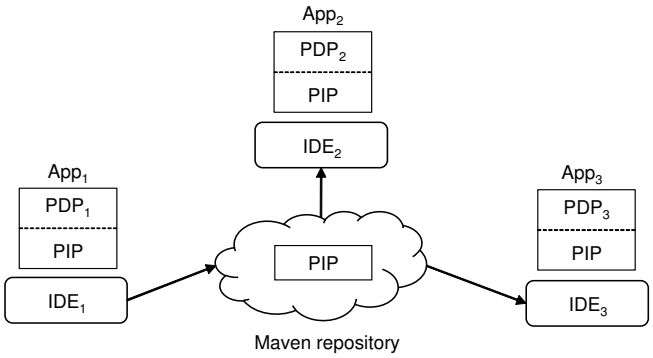

Fig. 1. Our approach.

Figure 1 depicts our approach. The PIP of an application, say App_1, is developed in IDE_1 and is added to a Maven repository for sharing. Other IDEs like IDE_2 and IDE_3, possibly for different platforms, use the shared PIP to create platform-specific applications, say App_2 and App_3. Once a project is configured, the sharing of PIP is done automatically through Maven. It is of course possible

to have multiple PIPs for an application, each developed using a different IDE. Below we summarize some of the Maven features that enable our approach.

4.1 Maven Repository

Maven is a powerful and multifaceted build tool. Our approach uses two of Maven's powerful aspects: the repository and dependency resolution. The first of these two items is the use of repositories. Maven has an online global repository and a local repository for storing and versioning artifacts. The process for adding artifacts to the global online repository is beyond the scope of this paper (refer to [6]). Once an artifact is in the global online repository, it can automatically be identified and added to a project by specifying three pieces of information: groupId (usually domain name), artifactId (program name), and version. Adding this information in the dependency section of the Maven `POM.xml` file (see Listing 1.1, lines 23–25) will automatically download the artifact from the online repository into the repository on the local workstation (see Figure 2). The artifact is then loaded into the Maven project and is now available in the local Maven repository for future use in other projects.

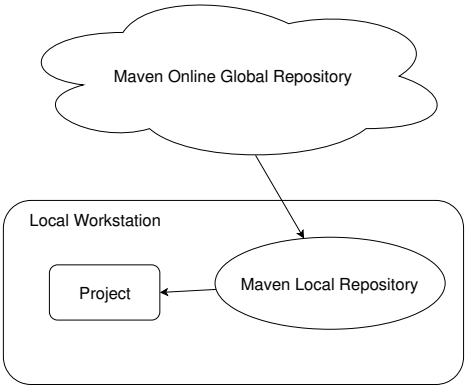

Fig. 2. Maven repository structure depicting the flow of dependencies.

4.2 Local Repository

The local Maven repository is a folder on a local workstation that can be browsed and viewed as any other folder. The default location is found in the `${user.home}/.m2/repository/` folder. There are three ways that artifacts can be added to the local repository. As previously mentioned, an artifact included in a Maven project (as a dependency) is downloaded from the online repository and is loaded into the local repository for current and future use. The second

method is to execute the "install" goal of the Maven "Install" plugin. The "Install" plugin, or task, is used to compile the Java code, run any tests, generate the artifact, and then place the artifact, be it a .jar, .ear, or .war file, into the local Maven repository. Using the groupID, artifactID, and version specified in the build from the `POM.xml` file, the artifact can now be loaded as a dependency by any other project on the local machine. The third way that an artifact can be loaded into the local repository is by using the "install-file" goal of the Maven "install" plugin. This is usually used from the command line to specify that an existing externally created artifact is to be manually added to the local repository.

4.3 Dependency Resolution

Of specific importance is how Maven uses the local repository to resolve dependencies. When a project tells Maven that it needs a specific library or artifact, it is done by adding the artifact's information to the dependency section of Maven's `POM.xml` file. Almost immediately, Maven loads the dependent artifact from the local Maven repository into the "External Library" section of the local project. If Maven does not find the artifact in the local repository, it will look in the global online repository, find the artifact, load the artifact into the local repository, and then into the project's "External Library" section. If Maven is unable to find the artifact in the local or global repository, then it will display an error.

In addition to resolving dependencies specified in the `POM.xml` file, Maven also resolves transitive dependencies. A transitive dependency is when an artifact has its own dependency. Maven loads the artifact from the repositories and then searches the repositories for the dependencies listed by the artifact. Maven then also loads those transitive dependencies into the local repository and/or the project.

4.4 Summary

With these key features of using the Maven repository system to store artifacts and resolve dependencies, we can see several key results in practice. If Maven is used to build project A, then the resulting artifact of project A will be added to the local repository. Now that artifact A is in the repository, it can be used concurrently in any other local Maven project (e.g. Maven projects B, C, and D) on any development platform. Once project B, C, or D specify a dependency of artifact A, any further changes or updates to project A will automatically be deployed to the parent project. In addition, any dependencies that Maven project A specified will also be included in projects B, C, and D as transitive dependencies.

5 Application

We performed a small case study by applying our approach to the development of an Android app. Our case study involves a development of a library based on the MVC architecture. We were writing an app for Android that used a library .jar file provided by a manufacturer. We did not like the functionality of the manufacturer's library and decided to make our own library to more easily expose the functionality that we wanted to use. Instead of doing this work in Android Studio, we decided to create a "model" of the functionality in a more familiar environment (IntelliJ IDEA), then access that library in Android Studio and then later in other platforms. We also wanted to perform coverage and mutation testing for our Java code, and there were several open source tools available for Java IDEs such as Eclipse and IntelliJ IDE. Below we summarize the steps we took in our case study along with our findings. We first developed our own library in IntelliJ IDE and used the library in Android Studio. Of course, we had to switch back and forth between the two IDEs.

POM Dependencies We first had to learn how to add the manufacturer's .jar file into our IntelliJ IDEA project as an external library using Maven. Our research showed that we could specify the library as a static link to the folder location by adding it to the Maven `POM.xml` file. This, however, was discouraged in every help forum that presented a solution to this dependency problem. The better solution was to use a Maven command line to add the library (.jar file) to the Maven local repository, specify the library as a dependency in the Maven `POM.xml` file of our project, and then allow Maven to load the library into our project.

POM Plugins Once we started learning how to add dependencies into our project, it became trivial to load the correct versions of testing and test coverage tools such as Junit, JaCoCo [7] and PITest [3]. We had to fool around with the parameters of the JaCoCo and PITest Maven plugins in order to get the test results in the format that we were looking for, but overall, it was simple to use things in Maven. For example, at one point, we wanted to compile our project into a .jar file to test from the command line. We had to search and find which plugin option to set in the `POM.xml` file that would create the manifest file and include it in our .jar artifact. In addition, since our project had a dependency on the manufacturer's external library, our .jar file would not run from the command line. A little bit of research showed us the Maven plugin to add to the `POM.xml` file to instruct the build to include all of the dependencies into our artifact. The next build then resulted in the normal .jar file, along with a separate .jar file appropriately named to show that it included all of the dependencies. This second file ran from the command line with no additional configuration changes or classpath requirements.

Local Maven Repository We learned about the location of the local Maven repository out of curiosity. The default location on a local PC for the local repository is `${user.home}/.m2/repository/`. Browsing this folder shows all of the artifacts that are in the repository. We recommend glancing at the organizational structure of the repository to get a better understanding of how Maven functions. For example, this is where we realized that every time we hit the "install" button on the Maven tasks, the build process actually placed a copy of our artifact (.jar file) in the local repository. Not only did we have our artifact embedded in the target directory of our project folder, but the artifact was also continuously updated in the local repository. Since we were almost finished with our Java library by this time, we knew that we would be starting on the Android Studio portion of our project. This meant that we would be working with Gradle, which is a different build tool that is used in Android Studio.

Cross Platform Developing As we finished our library, we moved to Android Studio to develop the Android app. We had worked with the Gradle build tool portion of Android Studio in the past. But after what we had recently learned about Maven, we wondered if we would be able to utilize the local Maven repository from Gradle so that Android Studio would automatically load our custom library as a dependency into our Android Studio project. As it turns out, Gradle has the ability to use the local and the remote Maven repository. We added the lines to the Gradle configuration file for our local Maven repository and then added the dependency line for our library project. Almost immediately, Gradle loaded our library .jar file, which we had created in IntelliJ IDEA, into the "External Libraries" section of Android Studio. There were already a great number of external libraries in the list, due to the Android libraries that needed to be loaded by the IDE. As we looked through the list to find our library, we were amazed to come across the manufacturer's library already in the list, even though we had not listed it as a dependency yet. It had been loaded into our project as a transitive dependency. Our original thought was that we were going to have to manually load our library that had the dependencies built into it, or we were going to have to manually include our library and the manufacturer's library as dependencies in the Gradle build file. However, since we had added the manufacturer's library to our local Maven repository and specified the manufacture's library as a dependency in our library, Gradle and Maven knew that our library had a dependency and loaded both of them just by specifying our library as a dependency.

However, one caveat of our approach is that a developer is responsible for making sure that a PIP is indeed platform independent. For example, our library should not use any Java language or API features that are not available on Android[4]. This checking has to be done manually; it might be possible to automate some of it using languages or tools such as AspectJ (e.g., static crosscutting) [8] by performing a static check on the source code.

[4] There is some difference between Java and Android Java in terms of supported language features and APIs.

Debugging a Library or Dependency While we were impressed with the way this setup worked, we did not give it much more thought as we started our Android app. But we soon realized that our custom library had a bug and that we were going to have to go back and work on the original code in IntelliJ IDEA. Without closing Android Studio, we opened IntelliJ IDEA and pulled up our library code and fixed the bug. We ran the Maven "clean" and "install" goals as we had done many times before. We then switched to the Android Studio window to continue to work. To our amazement, the change that we had made in our other IDE had already been updated in Android Studio. In a flash, Android Studio had seen that there was an update in our library and had loaded the new version from the local Maven repository.

Immediate Code Distribution The above finding would prove to be an incredible time saver as we had to make many changes to our library. For example, we would go back to our library code and add a field with getters and setters to one of our Java classes and recompile. Then we would flip back over to our Android code and type the classname, hit the period "." and the new getters and setters would show up in the auto-completion list. This setup has allowed us to utilize the idea of continuous integration on a personal level.

6 Conclusion

We have come up with a method to develop a program component in one IDE and immediately have it published and deployed for access from a completely different IDE or environment. Our approach allows an individual developer to configure a cross-platform development environment by composing a set of platform-specific tools or integrated development environments (IDE's). We don't think that this is a totally new concept. Large projects are developed in this manner. But individual framework development can also benefit from this type of functionality even in the smallest environment. The Maven Central Repository is an incredible tool that allows production code to be configured and used in various projects. But our contribution was to discover how Maven can be used on a single computer as an agile framework development tool in simultaneous programming environments.

References

1. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (1999)
2. Brown, A.W.: Model driven architecture: Principles and practice. Software and Systems Modeling **3**(4), 314–327 (Dec 2004)
3. Cole, H.: Pitest: Maven quick start, http://pitest.org/quickstart/maven/, Last accessed 14 May 2018
4. Duvall, P.M.: Continuous Integration. Improving Software Quality and Reducing Risk. Addison-Wesley (2007)

5. Gast, H.: How to Use Objects. Addison-Wesley (2016)
6. Jackson, B.R.: Maven: The Definitive Guide. O'Reilly, second edn. (2015)
7. JaCoCo: Jacoco plugin for maven, https://www.jacoco.org/jacoco/trunk/doc/maven.html , Last accessed 14 May 2018
8. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In: Knudsen, J.L. (ed.) ECOOP 2001 — Object-Oriented Programming 15th European Conference, Budapest Hungary, Lecture Notes in Computer Science, vol. 2072, pp. 327–353. Springer-Verlag (Jun 2001)
9. Maven: Apache maven project, https://maven.apache.org/, Last accessed 5 July 2018
10. Meservy, T.O., Fenstermacher, K.D.: Transforming software development: an mda road map. Computer **38**(9), 52–58 (Sep 2005)
11. Meyer, M.: Continuous integration and its tools. IEEE Software **31**(3), 14–16 (May 2014)