Remote dependencies in pluto

Remote Abhängigkeiten in pluto
Bachelor-Thesis von André Pacak
Tag der Einreichung:

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. rer. nat. Sebastian Erdweg

TECHNISCHE
UNIVERSITÄT
DARMSTADT

Department Computer Science
Software Technologie Group

Remote dependencies in pluto
Remote Abhängigkeiten in pluto

Vorgelegte Bachelor-Thesis von André Pacak

1. Gutachten: Prof. Dr.-Ing. Mira Mezini
2. Gutachten: Dr. rer. nat. Sebastian Erdweg

Tag der Einreichung:

Erklärung zur Bachelor-Thesis

Hiermit versichere ich, die vorliegende Bachelor-Thesis ohne Hilfe Dritter nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus Quellen entnommen wurden, sind als solche kenntlich gemacht. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Darmstadt, den December 9, 2015

_____

(André Pacak)

Abstract

Incremental build systems are very important in the software development process nowadays, because it is necessary to minimize the compile time. Today, build systems need to be capable of handling dependencies of resources which are not available locally. Larger Software projects are split into sub-project for reusability. Imagine a project that is split into smaller sub-projects which are maintained with the help of Git. It would be useful that the build system could handle a dependency on a Git repository and automatically pull the latest commits added to the remote repository. For a Java-based project, it would be useful when the build system could resolve the dependencies on Maven artifact automatically. We extend the sound and optimal incremental build system pluto by providing a new abstraction which supports the dependency on remote resources. The abstraction gives us the possibility to forbid consistency checks between the cached and the remote resource for a defined interval after the last successful consistency check. With the use of the new dependency, we support the resolution of Maven artifact dependencies, Git repositories and downloading files from HTTP servers. We demonstrate how to use the three different kinds of dependencies on remote resources by implementing a build script for a specific project. The project is split into different Git repositories, depends on libraries provided by Maven Central, and needs a library located on a HTTP Server.

Contents

1 Introduction **4**
- 1.1 Contributions . . . 4
- 1.2 Overview . . . 5

2 Background **6**
- 2.1 pluto . . . 6
- 2.2 Git . . . 8
- 2.3 Maven . . . 9

3 Remote Requirement **11**
- 3.1 Properties of RemoteRequirement . . . 11
- 3.2 Implementation . . . 11
- 3.3 Discussion . . . 13
- 3.4 Example of HTTPDownloader . . . 14
  - 3.4.1 HTTPInput . . . 14
  - 3.4.2 HTTPRequirement . . . 14
  - 3.4.3 Implementation of HTTPDownloader . . . 15

4 Git **16**
- 4.1 Behavior of Builder . . . 16
- 4.2 GitInput . . . 16
  - 4.2.1 Additional Branches To Clone . . . 17
  - 4.2.2 Submodules . . . 17
  - 4.2.3 Fast-Forward Mode . . . 17
  - 4.2.4 Commit After Merging . . . 17
  - 4.2.5 UpdateBound . . . 18
  - 4.2.6 ConsistencyCheckInterval . . . 18
- 4.3 GitRequirement . . . 19
- 4.4 GitRemoteSynchronizer . . . 20

5 Maven **21**
- 5.1 Behavior of Builder . . . 21
- 5.2 MavenInput . . . 21
  - 5.2.1 Dependency . . . 22
  - 5.2.2 Repository . . . 22
- 5.3 MavenRequirement . . . 23
- 5.4 MavenDependencyResolver . . . 24

6 Case Study **26**
- 6.1 ServiceBaseJavaBuilder . . . 27
- 6.2 ServicesJavaBuilder . . . 28
- 6.3 ServicesJavascriptBuilder . . . 29
- 6.4 Advantages of proposed Solution . . . 29

7 Related Work **31**

8 Conclusion and Future Work **32**

References **33**

1 Introduction

In today's software development, it is essential to have automation tools which can build software projects and provide incremental compilation. The projects which are developed today consist of millions of lines of code. Take Microsoft Windows 7 [2] for example, it has over 25 million lines of code and is probably structured in a lot of sub-projects. Additionally, developing software is an iterative process, therefore it is important that the build systems we use in software development are incremental to shorten the compile times of a project. A software product can be build by hundreds of developers which have to share their progress. A lot of software is using third-party libraries which are mostly stored on a server and can only be accessed via other protocols than the file protocol. Today's build systems need to be capable of resolving dependencies on remote resources.

In the paper, published by Erdweg et al. [8], a new build system called pluto was developed which is sound and optimal incremental and is built upon a two-layered dependency graph. In addition, it supports dynamic dependencies, which means that dependencies can be registered during the build process. pluto can only access files located on the machine the build script is running on and assure the sound and optimal incremental conditions are not violated. It would be useful to rely on libraries which are provided by e.g. Maven Central or source files located on a Git repository and let builders resolve the dependencies automatically. We want a build script to be repeatable and reproducible. Therefore, downloading libraries or cloning Git repository by hand is not a good solution. It does not ensure consistency between sub-projects. Currently, pluto only supports dependencies from builders on files and on other builders. Therefore, we can only check for consistency of a builder or a file. Because we want to provide a way of accessing remote resources, we need a way of registering dependencies which points to a remote resource and pluto has to check for its consistency to ensure the systems optimal incremental condition is held.

We introduce a new abstraction for dependencies located on a remote server, called `RemoteRequirement`. We need an abstraction instead of a concrete implementation because every remote resource is different. Thus, the determination if a newer version is remotely available is different for every kind of resource. The new abstraction differs from the currently supported requirements, file and build requirement, because the consistency can be expensive, which is caused by remote accesses. Therefore, each instance of `RemoteRequirement` can be configured to only check for newer versions if a certain amount of time has passed after the last successful consistency check. Because we do not have control over the accessibility of the remote resource but want to make it possible to run a build script if every needed resource is available locally, a remote requirement is considered consistent if a cached version is available but the remote is not accessible. We show how to implement the abstraction `RemoteRequirement` by creating three different builders and the needed requirements. First, we discuss a builder which is capable of downloading a file via the HTTP protocol. Next, we demonstrate how to construct a builder and the requirement, which is used by the builder, that can keep a directory synchronized with a Git repository. We introduce an abstraction, called an `UpdateBound`, to make it possible that the local repository can be kept in sync with a commit, tag or even a branch. The last builder we implement can resolve dependencies stored on a Maven artifact repository. We implement a build script for sub-projects of Monto, to show how the builders, which are introduced in this thesis, can be used to achieve consistency between projects that depend on each other.

1.1 Contributions

The main contributions of this thesis are the following:

- We extended the build system pluto by introducing a new requirement abstraction for remote resources.
- We implemented a new builder for downloading a resource via the HTTP protocol.
- We implemented a new builder for keeping a directory synchronized with a Git repository.
- We implemented a new builder resolving artifacts located on Maven repositories.
- We provided a case study which demonstrates how to use the introduced builder in order to build a project.

1.2 Overview

In section 2, we introduce the build system pluto along with Git and Maven terminology. It provides the needed knowledge to understand the following sections. In section 3 the requirement abstraction `RemoteRequirement` will be introduced and we discuss how it fits into the current build algorithm of pluto. Next, we will introduce the implementations of `GitRequirement` and `GitRemoteSynchronizer` in section 4, which are subtypes of `RemoteRequirement` and `Builder`. The last new builder we will introduce is `MavenDependencyResolver`, which will be discussed in section 5. In section 6 we will take a look at the case study. It will show how to use the three implemented builders to create a build script which resolves dependencies on Maven artifacts, Git repositories and files located on machine that can handle HTTP requests. We will discuss other build systems and how they handle dependencies on remote resources such as Maven artifacts or Git repository in section 7. In addition, we will compare them with the work done in this thesis. At last, we will conclude the thesis by discussing the presented work and show what could be done in future work, in section 8.

2 Background

2.1 pluto

Build systems play a key role in modern software development. They provide an automation for invoking the needed build tools in the correct order and with the wanted resources as input. Imagine we have a project A that is built out of the sub-projects B and C. The build system can define the dependencies between those sub-projects and make sure that the projects B and C are built before project A because it depends on the products of sub-projects B and C. We can see a dependency graph of this example in Figure 1. Thus, the build system has to choose between two build orders:

$$B \rightarrow C \rightarrow A$$

$$C \rightarrow B \rightarrow A$$

A build system pieces all the different components of a software project together and creates the final product. It also ensures that the process of building the software product is repeatable and reproducible.

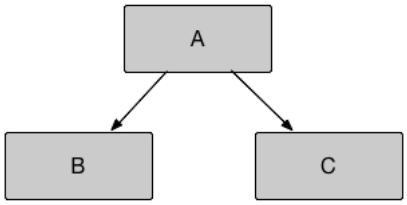

Figure 1: Example of a dependency graph for a project.

In this thesis we focus on the build system pluto which was introduced in [8]. pluto is a sound and optimal incremental build system. A build system is sound if the generated files reflect the latest source files consistently. Let's suppose that we have a Java source file A.java and a unit J which takes Java source files and compiles them down to Java class files. We compile the file A.java with J which produces A.class. The system is sound when the next invocation of unit J produces a new file A.class if the source file A.java has changed. The content of A.class has to represent the latest version of A.java. A sound system is called incremental if it satisfies two additional conditions. First, the number of builder executions is minimized. Secondly, the number of checks for internal consistency is minimized. The optimal incremental condition states that for every input, the build script minimizes the number of builder executions and the number of consistency checks.

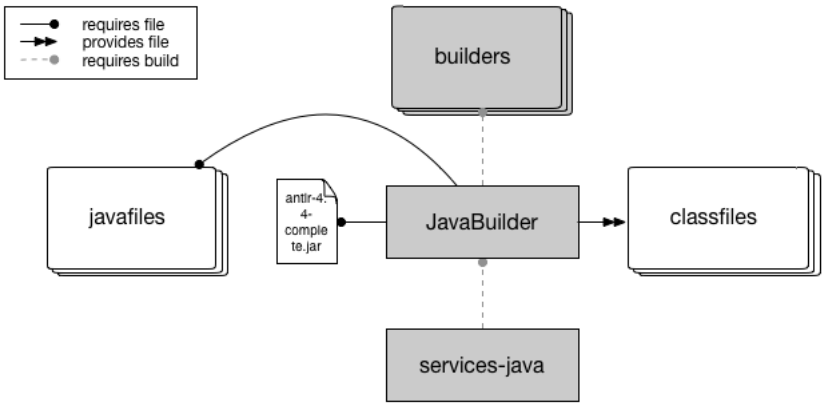

Figure 2: A part of the dependency structure of the services-java build script.

Figure 2 shows a graph which lays out the dependencies of a build script implemented with the help of pluto. We can see that JavaBuilder depends on the file antlr-4.4-complete.jar and on a list of Java source files that are determined at runtime. ANTLR [1] is a parser generator. It can used to generate a parser which can parse a given grammar. The system makes sure that every builder, required by JavaBuilder, is executed before the JavaBuilder itself. As shown in Figure 2, a unit inside of pluto can provide files it generated at runtime. This is important because the system detects hidden dependencies and throws an error if it can find one. A hidden dependency is a dependency where a unit A requires a File b, which is provided by another unit B, but the requiring unit A does not require the unit B.

```
1  public abstract class Builder<In, Out> {
2      protected final In input;
3
4      protected abstract String description();
5      protected abstract File persistentPath();
6      protected abstract Out build() throws Throwable;
7
8      protected void require(File f) { ... }
9      protected void provide(File f) { ... }
10     protected <In_, Out_> Out_ requireBuild
11         (BuilderFactory<In_, Out_> fact, In_ in) { ... }
12 }
```

Listing 1: A simplified version of the abstract Builder class.

pluto provides a Java API for implementing build scripts. In Listing 1 we can see the outline of an abstract builder. Every builder inside of the system is a subtype of this abstract class. A builder takes an input of type `In` and produces an optional output of type `Out`. A builder has to implement three abstract methods. The first method `description` is used for logging purposes, so we can see which builder is executed at which point in time of the whole execution of the build script. Secondly, we have the `persistentPath` method. It is used to determine the path for the build summary of the builder. The build summary contains information about which files were required and provided in the previous run of this builder. The last abstract method is `build`. This method is the core of the builder. It determines the files and other builders that are required, and which files are provided by this builder and how those provided files are generated. Because the build method is written in Java, it is possible to use existing Java libraries, which will become quite handy within the work of this thesis. By invoking the methods `require` and `requireBuild` in the body of the `build` method, the builder can register dependencies to files and other builders. Therefore, the dependencies can be dynamically added at build time. The method `provide` gives us a way to indicate that a builder has generated a file, and make it possible that another builder can require the generated file.

```
1  public interface Requirement extends Serializable {
2      public boolean isConsistent();
3      public boolean tryMakeConsistent(BuildUnitProvider manager) throws IOException;
4  }
```

Listing 2: The Requirement interface.

Listing 2 shows us the interface that every kind of dependency has to implement. `Requirement` defines two methods that need to be implemented by its concrete subclasses. The method `isConsistent`, like the name states, indicates if the requirement is consistent. The second method, `tryMakeConsistent`, takes a `BuildManager` and checks if the requirement is consistent. In the case of inconsistency, the `BuildManager` tries to make the requirement consistent.

Currently there are two different kinds of dependencies. First, there are file dependencies which are represented by the class `FileRequirement`. A dependency on a file can be seen in Figure 2 where the JavaBuilder requires the file antlr-4.4-complete.jar. Only a builder can register dependencies on files. In case of `FileRequirement`, the method `tryMakeConsistent` just calls `isConsistent`. It is additionally possible to create dependencies between a builder and another builder, even the builder itself, which would form a cyclic dependency. This type of dependency is designed by the class `BuildRequirement`. In Figure 2 we see such a dependency between services-java and JavaBuilder. This results in a two-layered dependency graph. The first layer is the file layer. The file layer represents the required and provided files. Because there is no dependency between two files, the file nodes are not connected. In the second layer, the build layer, a node represents the result of a builder. A node contained in the build layer has a connection to all nodes of files the builder, which is represented by the node, requires and provides, and is connected with the builders it required. It

should be pointed out that a dependency always has only one direction. Therefore, the dependency graph of a build script implemented in pluto is a directed graph.

The requirements have to be consistent at the end of the execution of the build script. It is necessary to achieve the soundness of pluto. A file requirement is consistent, when the content or meta data of the file has not changed, in comparison to the previous run of the builder, which has a file registered as required. The notion of checking if a File has changed in regards to its content or its meta data, is represented and can be customized by creating an implementation of the interface Stamper. This interface is not important for this thesis, but is worth mentioning. A build requirement is consistent when every builder and every file that the builder requires is consistent, which means that every requirement that is contained inside of the transitive hull of the requirement relation has to be consistent. Additionally, the provided file of the builder have to be consistent as well.

2.2 Git

Git is a distributed version control system which was designed to be the version control system for the Linux kernel. A version control system is a tool to keep track of the changes that were made to a file or a set of files. This enables a developer to reset a files content to an older version. For a version control system to be distributed means that every repository, even a local copy, have an equal standing and are as important as a repository that is located on a centralized server. The advantage of a distributed VCS is that creating new versions of files is a quick action because it all happens on the local repository. Therefore, we do not need a connection to the centralized server, where the repository is located, to work on a project and log the progress that has been made in the meantime.

In the following subsections we will look into some Git terminology [7], which is needed to understand section 4, which describes a builder we designed for the synchronization of a local directory with a Git repository.

Repository.
A repository is a set of files which are tracked by Git and stores the version history of each file it tracks. All this information is contained within a directory called .git, which is is located in the root of the directory, Git should track files from, respectively the software project.

Commit.
Git differs from other version control systems by viewing commits not as a set of changes applied to files. Instead, Git views a commit as a snapshot. Every time a commit is created, Git takes a snapshot of the current state of the files we want to include in the commit. Therefore, Git views its data as stream of snapshots instead of a stream of changes. The repository is a directed acyclic graph, where the commits are the nodes, and every commit, but the initial commit, which has no parents, has at least one parent. A commit has two or more parents if it was created by the merge operation, which we will explain later. Therefore, a commit has three informations stored, namely the SHA-1 hash, its parent commits, and the snapshot of the files which the commit represents.

Branch.
A branch is an active line of development and has a most recent commit which gets referred to as HEAD of the branch. If we make a change to a branch, by creating a commit, the HEAD of the branch points to the new commit, and therefore moves forward. A repository can have an arbitrary number of branches, but the current working tree can only represent one branch.

Tag.
If a milestone has been achieved milestone, and it should be marked within the repository, a tag can be used. A tag is a reference which points to an object of arbitrary type. For example, it can point to a commit. The tags of the repository are stored inside of the .git/refs/tags/ directory.

Merge.
It is possible to apply the changes, which were made in another branch, to the current one. This action is called merge. If the branch, which should be merged in to the current branch, is a remote, Git has to fetch the latest not downloaded changes. Next, Git can merge the changes of the remote branch, into the current local one. This operation is called pull. It is possible that changes, which were made to a file in the both branches, conflict each other. If that happens, the user has to resolve those conflicts manually. A successful merge creates a new commit which represents the result of the merge.

Fast-Forward Modes.
A fast-forward update is an update where the local branch history has no additional changes in comparison to the remote branch history. Therefore, every new commit is just fetched and the pointer of the branch has to be moved to the most recent commit of the remote branch. There are 3 different options:

- FF
- NO_FF
- FF_ONLY

The FF option allows a fast-forward merge and thus no merge commit is created. When the merge resolves to a fast-forward update, the NO_FF option forces Git to create a merge commit. The last option, FF_ONLY, forbids every merge that is not a fast-forward update.

Clone.
It is possible to clone a repository and transfer it to another machine. This operation copies every file the repository tracks and downloads meta data of the repository, such as the commit history of those files or the available tags.

Working Tree.
The working tree is the current state of the filesystem. It includes every file the current branch was told to track. When a branch has been checked out and no additional changes were made, the working tree represents the latest commit of the branch, which has been checked out previously. But when a file has been altered, which is tracked by the branch, the working tree differs from the latest commit of the branch.

Checkout.
Git can checkout different objects. For example, it is possible to checkout a branch. This means that the state of the working tree represents the latest commit of the branch which has been checked out. Additionally, a specific file can be the target of a checkout action. By doing so, only the file which was checked out changes its state in the working tree.

2.3 Maven

Apache Maven [9] is a software project management tool which was designed for Java-based projects. Maven is a build system just like pluto. For building a project, it can resolve the dependencies on other Maven projects. We will focus on the dependency management of Maven. Maven has a collection of almost every public Java library, which is called Maven Central. Other dependency management tools like Apache Ivy can handle the resolution of dependencies located on Maven Central. Build systems like SBT, Gradle, Apache Buildr also support fetching dependencies from Maven Central. pluto itself is built with Maven and the artifacts of its ecosystem are published on a public remote repository.

Maven Central is a so called repository. A repository in the Maven world is a location where artifacts and their meta data are stored. There are two different kinds of repositories, local and remote. A remote repository is a collection of artifacts which are located on an external machine. A remote repository can be truly remote, and can be used by third-parties to provide their artifacts or it can be a HTTP or file server which can only be accessed within a local network. A local repository is a local cache of the artifacts that are needed to build the projects on the current machine. But it also can contain projects, that are not deployed to a remote repository yet which are needed to build another project on the local machine. The structure of remote and local repositories are exactly the same.

```
1  <dependencies>
2      <dependency>
3          <groupId>com.googlecode.json-simple</groupId>
4          <artifactId>json-simple</artifactId>
5          <version>1.1.1</version>
6          <type>jar</type>
7      </dependency>
8      <dependency>
9          <groupId>junit</groupId>
10         <artifactId>junit</artifactId>
11         <version>4.12</version>
12     </dependency>
13 </dependencies>
```

Listing 3: A snippet of POM which shows how dependencies are defined.

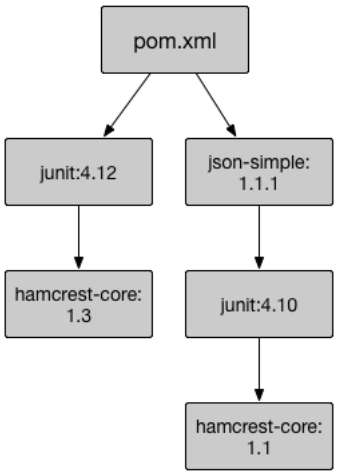

Figure 3: The dependency graph of the example POM.

In Listing 3 we can see a snippet of the dependency management settings of a Maven project, which is specified inside of the POM. Two mandatory informations, Maven needs to resolve a dependency, are the groupId and the artifactId. The version of the dependency has to be specified. Instead of specifying a single version, it is possible to specify a version range. For example, we can set [1.0, 3.0] inside of the version tag and Maven chooses at least version 1.0 and at most 3.0. It is possible to add more than one dependency inside the dependencies tag.

When Maven resolves the dependencies it creates a dependency graph, which includes every transitive dependency of the dependencies that were specified inside of the POM. Such a dependency graph can be seen in Figure 3. It is possible that the dependency graph contains the same artifact twice, but with different versions. In this case, Maven uses the nearest definition, which means it chooses the version of the dependency that is closest to the root of the dependency graph. In the shown example, this would mean that the version of the junit artifact, which Maven will use for building the project, is version 4.12. The type tag defines which type the artifact has to be in. The default setting for type is jar. The classifier makes a distinction between artifacts that were built from the same project. For example, we can specify a classifier javadoc that resolves to an artifact which only contains documentation for the project and nothing else. It is possible to exclude dependencies of artifacts the project depends on. For an exclusion of an artifact, we need to specify the groupId and the artifactId within an exclusions block which is located inside of the dependency block.

3 Remote Requirement

The build system pluto currently only supports dependencies from builders to other builders and to files. It can be possible that a builder performs an action on a remote resource, e.g downloading a file that is located on a HTTP server. In this scenario, we have no way of checking if the remote resource has a new version available. This would result in violating the optimal incremental condition of pluto because the number of executed builders is not minimal. By checking for the consistency of a remote dependency, we have to make a remote access which can be quite costly. To solve this problem, we need a way of telling the system, that a remote dependency does not need to be checked for consistency every time we want to execute the builder, which has registered the remote dependency. In this thesis, we are exploring a way of adding dependencies which point to resources that are not stored on a local machine and introduce a new notion of being consistent for those requirements.

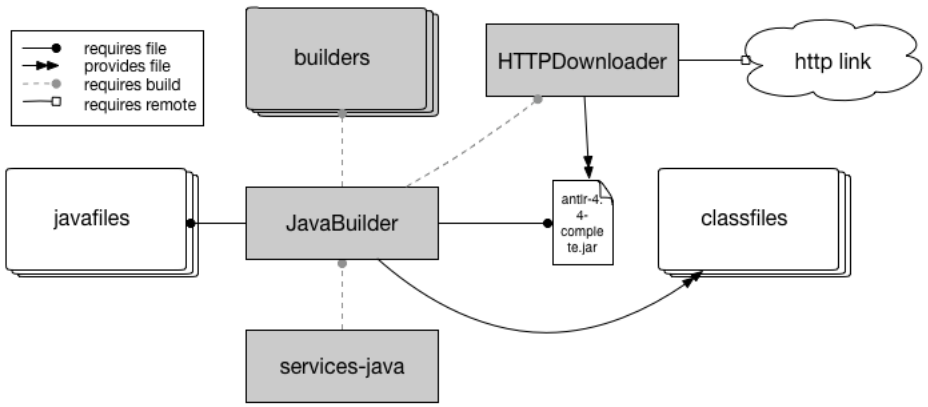

Figure 4: Dependency graph of build script where a file is downloaded by a builder.

Figure 2 displays the dependency graph of the build script we have seen in Figure 4 but with the addition of a builder that handles the task of downloading the antlr-4.4-complete.jar file from a HTTP server. In Figure 2, the file antlr-4.4-complete.jar is stored on the local machine and had to be downloaded manually by the user of the build script. The newly introduced builder `HTTPDownloader` provides the file antlr-4.4-complete.jar, which is stored on the local machine, after the builder has been executed successfully.

3.1 Properties of RemoteRequirement

We need to specify a few properties the `RemoteRequirement` has to have. First, we want to tell the system that we do not need to check the remote resource for consistency every time the builder, which has the requirement registered, is trying to calculate its consistency. Secondly, we want to make it possible that if the remote resource is not accessible, but there is another version acquired by a previous run of the builder stored on the local machine, the requirement should be considered consistent. This would enable the user to run a build script without an internet connection, when the resources are available locally.

Those two properties weaken the notion of a requirement to be consistent because the remote resource may not be the same as the previously cached local resource. Regardless of this fact, we see the requirement as consistent. Thus, the latest generated files do not reflect the latest source files.

We can see the important methods of the new abstract class `RemoteRequirement` in Listing 4. In the following section, we will discuss the methods of `RemoteRequirement` and how we achieved that the class `RemoteRequirement` has the stated properties.

3.2 Implementation

We propose to implement an abstract subclass of the interface `Requirement`, which we discussed in the background section, and extend pluto in such a way that we can register other dependencies than file and build dependencies.

First, we need a way to keep track of the time that the system checked the consistency of a `RemoteRequirement`. It is needed to decide if we want to skip the consistency check entirely, based on the last time the consistency was checked

successfully.

We decided to use a timestamp of the initial requirement of the entire build script, which then executes the builders, to have consistent timestamps for a whole execution of the script. In addition to the timestamp, we need a way of telling the `RemoteRequirement` how long it is not allowed to check for consistency between the local and remote resource. We used an integer to store this information, which we will call interval. The interval tells the system how many milliseconds, starting at the last successful consistency check between the local and the remote resource, the instance of `RemoteRequirement` has to wait for checking the consistency between the two resources again. The JDK offers an easy way of converting hours, minutes etc. into milliseconds.

For the values of interval we assigned the following meanings:

$$behavior(interval) = \begin{cases} \text{invalid} & \text{if interval} < -1 \\ \text{never check} & \text{if interval} = -1 \\ \text{forbid consistency checks for interval ms} & \text{if interval} > -1 \end{cases}$$

We view the values lower than -1 as invalid because when the requirement checks if it wants to perform a consistency check between the local and the remote resource, it makes no sense to forbid consistency checks for a negative amount of time. For remote resources, where the consistency check would take more time than executing the builder, which handles the remote resource, we had to make it possible to forbid consistency check completely. We choose -1 because every negative value is invalid and it was the most obvious choice. Therefore, we have every non-negative value available to choose from for the interval.

```
1  public abstract class RemoteRequirement implements Requirement {
2      private final long consistencyCheckInterval;
3      private final File persistentPath;
4      private long getStartingTimestamp();
5      public boolean isConsistent();
6      private boolean needsConsistencyCheck(long currentTime);
7      protected abstract boolean isConsistentWithRemote();
8      protected abstract boolean isRemoteResourceAccessible();
9      protected abstract boolean isLocalResourceAvailable();
10 }
```

Listing 4: Outline of abstract class RemoteRequirement.

For persisting the timestamp of the last execution, we choose to save the timestamp of the last consistency check inside of an own file. Every `RemoteRequirement` has a file, which stores the information when the last successful consistency check between the local and remote resource was executed, thus the timestamp can be accessed by the next execution of the build script. At first, we wanted to save the timestamp inside of the `RemoteRequirement` as an attribute. That forced us to write out the build summary every time a consistency check is performed. But currently, the system only writes the build summary when the builder is executed. There was the possibility to adapt the system in such a way that after every consistency check of a builder, its build summary gets updated. This would amount into a lot of new filesystem accesses which would slow down the execution of build scripts. The build summary would be written for every builder even if nothing changed and a builder was consistent in the first place. By using a file as an information storage for the timestamp, the system only needs to write the file if the consistency check (isConsistentWithRemote) was successful, therefore we save a lot of time. This behavior allows pluto to achieve the first property we set for the `RemoteRequirement`.

In Listing 5 we can see how pluto determines if it allows a consistency check between the local and remote resources. First, we check if the file, which should contain the timestamp information, exists. If the file is non-existent, we create a new file, store the timestamp inside of the file and allow a consistency check between the local and the remote resource. This has to be done, because if the file does not exist yet, it means that until this nt there has not been a consistency check. Thus, the builder, which registered the requirement, has not been executed once and we need to force an execution. In the next step, we check if the `RemoteRequirement` was instructed to never allow another consistency check and forbid or allow a consistency check accordingly. If we advance further, the next action to take is to read the timestamp from the file and test if the current timestamp is bigger than the interval and the last timestamp added together. If it is bigger, we allow a consistency check by returning true. If it is smaller or equal, we forbid it by returning false. If an overflow while adding the timestamp and the interval occurs, we forbid a new consistency check because the interval has to be very big since both values are non-negative.

Before the method `isConsistent` can check if the local resource is consistent with the remote resource, we have to check a couple of things first. The first condition that has to be met, is that the system is allowed to check for consistency, which is accomplished by checking if `needsConsistencyCheck` yields true. The next step is to check if the remote resource can be accessed. This check has to be implemented by the subclass of `RemoteRequirement` by implementing the abstract method `isRemoteResourceAccessible`. If that is the case, we proceed with the call of `isConsistentWithRemote`. If the check for the accessibility of the remote fails, we have to look up if an older version of the remote resource is available locally. Just like the check for accessibility of the remote resource, we have to implement the abstract method `isLocalResourceAvailable`. At last, it calls `isConsistentWithRemote` and returns its result. Additionally, if `isConsistentWithRemote` yields true, we store the timestamp which was taken at the start of the whole script inside of the file. By this design we achieved the second property of the `RemoteRequirement` we set.

The method getStartingTimestamp is responsible for instantiating the timestamp of the initial requirement of the build script. This is achieved by adding a static attribute to the class BuildManager, which contains a map that points from thread to the initial requirement timestamp. We modified the method requireInitially of the class BuildManager in such a way that it fills map correctly. We fetch the timestamp of the thread the build script is running on. The BuildManager's task is it to execute a given builder and all the builders it depends on if they are not consistent. Currently the system uses one BuildManager per Thread.

By extending the abstract class `RemoteRequirement` and implementing the method `isConsistentWithRemote`, the maintainer of the build script has the possibility of expressing what it means to be consistent with the remote resource and therefore has the power of customizing the notion of consistency for his own purposes. In addition, we need to tell the system when a cached version is available, which can be expressed by implementing `isLocalResourceAvailable`. The method `isRemoteResourceAccessible` determines the accessibility of the remote resource, which is attached to the instance of `RemoteRequirement`.

```
1  private boolean needsConsistencyCheck(long currentTime) {
2      if (!FileCommands.exists(persistentPath)) {
3          writePersistentPath(currentTime);
4          return true;
5      }
6      if (consistencyCheckInterval == NEVER_CHECK) {
7          return false;
8      }
9      long lastConsistencyCheck = readPersistentPath();
10     long afterInterval = lastConsistencyCheck + consistencyCheckInterval;
11     if (afterInterval > 0 && afterInterval < currentTime) {
12       return true;
13     }
14     return false;
15 }
```

Listing 5: Implementation of method needsConsistencyCheck.

3.3 Discussion

In this section we will discuss how `RemoteRequirement`, which we introduced earlier, fits into the current state of the build system and how it effects the soundness and optimal incremental condition of pluto.

If we look at the two-layered dependency graph, which describes the dependencies of a build script implemented with pluto, we have to add a new type of node for the `RemoteRequirement`. We propose to expand the file layer of the dependency graph to a file-remote layer which contains file and remote nodes. The reason for this decision is that the remote and file nodes are very similar. A remote node has no connections to other remote or file nodes and there are only connections coming from build nodes. The only difference is that remote nodes can not be provided but file nodes can.

When the algorithm determines the consistency of a build node, it checks whether all outgoing connections are consistent or not. If there exists an inconsistent connected node, contained in the file-remote layer, the algorithm triggers a rebuild of the build node, which is currently inspected. If there exists an inconsistent build node, which the current build node depends on, the algorithm forces a rebuild of the inconsistent build node. Therefore, every inconsistent build node is forced to rebuild. After the algorithm terminates successfully, all inconsistent builders were forced to rebuild. Because

the consistency of a builder is not only dependent on other builders and files, but now depends on remote resources, the execution of a builder is triggered if a remote, which the builder requires, is inconsistent.

Because the consistency check of a remote node can be forbidden and be marked as consistent, the former soundness definition of pluto does not hold anymore. But the system is still sound because at some point in time, after the last successful execution, which means the remote requirement was consistent, the builder will be executed and the interval will be expired and a new consistency check between the cached version and the remote resource has to be made. Therefore, the generated files will eventually reflect the latest source files

3.4 Example of HTTPDownloader

Now we will take a look at the first example of an implementation of the proposed abstraction `RemoteRequirement`. We can see a dependency graph of a build script implemented with the help of Pluto that uses the `HTTPRequirement` and the `HTTPDownloader` in Figure 4, which we have to implement. Additionally, we need to define an input class for the `HTTPDownloader`.

3.4.1 HTTPInput

For the `HTTPDownloader` we need to define an input. An input is a class that implements the interface Serializable and contains attributes that the builder can use to execute the task it was build for. Thus, the input serves as an data access object for the corresponding subclass of `Builder`. The builder needs the following information to run:

- Where is the remote resource located
- Where should the local resource be located
- How long is a consistency check forbidden from the point of the last execution.

We turned the information into attributes of the `HTTPInput` and thus we came up with the class presented in Listing 6.

```
1 public class HTTPInput implements Serializable {
2     ...
3     public final String remoteLocation;
4     public final File localLocation;
5     public final long consistencyCheckInterval;
6     ...
7 }
```

Listing 6: Input class for the builder HTTPDownloader.

The attribute remoteLocation defines the URL of the resource, which we want to download via a HTTP_GET request. Next, we have to set the path where the downloaded resource should be stored, which is encoded within the attribute localLocation. Just like we have seen in the definition of the abstract class `RemoteRequirement`, the user of `HTTPDownloader` should have the freedom of setting the interval in which a consistency check between the remote and the local resource is forbidden.

3.4.2 HTTPRequirement

The next step is to define the class `HTTPRequirement`, which is a concrete subclass of `RemoteRequirement`. As we discussed previously, we need to implement the following abstract methods:

- `isConsistentWithRemote()`
- `isRemoteResourceAccessible()`
- `isLocalResourceAvailable()`

Those three methods define how the requirement determines its consistency. The first problem we encounter with the requirement, is that we have no way of checking if a new version is available at the remote location. To compare the hash between the local and the remote file, we have to download the file first, thus we do not need a consistency check because the execution of the builder and the consistency check take roughly the same amount of time. Therefore, we propose to choose a very large interval or use `RemoteRequirement.NEVER_CHECK` to speed up the execution of the build

script where the HTTPDownloader is used.

To check for the availability of the remote resource, we need to establish a HTTP connection and observe the response code. If we receive a HTTP_OK as status code, we can access the remote resource and return true.

The concrete implementation of isLocalResourceAvailable is straight forward,, because we check for the existence of the file with the path of localLocation, which is contained in HTTPInput and is passed to the HTTPRequirement via the constructor as an argument.

3.4.3 Implementation of HTTPDownloader

Listing 7 shows the build method of the Builder subclass HTTPDownloader, which takes HTTPInput as an input and produces no output. The HTTPDownloader builder downloads a file via a HTTP_GET request and stores it at the path, which is specified inside of HTTPInput. The first step of build is to register an instance of HTTPRequirement. The registration happens by calling the method requireOther, which is defined inside of the superclass Builder. Next, the builder has to establish a HTTP connection and check if the status code of the response message is 200 which maps to HTTP_OK. If this is the case, the builder sequentially reads the data from the httpConnection and writes it into a file by using Java's InputStream and FileOutputStream, which can be found within the JDK. After all data was read by the InputStream, which was provided by the HTTP connection, we close both streams and disconnect the HTTP connection. Now, the method build can terminate. The exceptions that can occur while calling methods such as openConnection of an instance of HttpURLConnection are not handled within the build method. We choose to do this because any exception which occurs while the connection is trying to be established or reading the data from the HTTP server indicates that the builder has failed. Thus, the build method itself has to throw an exception. If the status code of the response message was not HTTP_OK we throw an exception because the builder could not establish a connection successfully.

```
1  @Override
2  protected None build(HTTPInput input) throws Throwable {
3      File localFile = input.locationOnLocal.getAbsoluteFile();
4      URL remoteURL = new URL(input.remoteLocation);
5      RemoteRequirement httpRequirement = new HTTPRequirement(
6              new File(input.locationOnLocal.getAbsolutePath() + "http.dep.time"),
7              input.consistencyCheckInterval,
8              localFile,
9              remoteURL);
10     requireOther(httpRequirement);
11
12     //get file
13     HttpURLConnection httpConnection =
14         (HttpURLConnection) remoteURL.openConnection();
15     if (httpConnection.getResponseCode() == HttpURLConnection.HTTP_OK) {
16         FileOutputStream outputStream = new FileOutputStream(localFile);
17         InputStream inputStream = httpConnection.getInputStream();
18         int readBytes = -1;
19         int BUFFER_SIZE = 4096;
20         byte[] buffer = new byte[BUFFER_SIZE];
21         while((readBytes = inputStream.read(buffer)) != -1) {
22             outputStream.write(buffer, 0, readBytes);
23         }
24         inputStream.close();
25         outputStream.close();
26     } else {
27         throw new IllegalArgumentException("HTTP request could not be sent");
28     }
29     httpConnection.disconnect();
30     return None.val;
31 }
```

Listing 7: Implementation of method build of the class HTTPDownloader.

4 Git

Some projects are build out of sub-projects that we want to separate in different Git repositories. The reason for this could be that different projects need the same sub-project and we do not want to have redundancies across the whole ecosystem of a project. In the development stage of a project, the source code can change quite frequently. We want to provide a way of getting the latest changes of the sub-projects every time the code is build and tested where the sub-projects are maintained with the help of the version control system Git. In the following section, we will take a look into what the specialized builder has to achieve and how we have to implement the concrete subclass of `RemoteRequirement` to fit the general needs.

4.1 Behavior of Builder

The first step is to identify the scenarios that can occur when the builder is executed. The following list shows every interesting scenario:

(S1) The remote repository is not accessible and no local repository is available

(S2) No local repository is available and the specified directory is not empty

(S3) No local repository is available and the specified directory is empty

(S4) There is at least one new commit available on the remote repository that we are interested in

(S5) The local repository is up-to-date

The next step is to define the behavior of the builder in every scenario. In this section we will look at every scenario listed above and discuss the behavior we defined for the builder.

In the case of scenario (S1), the builder has to fail because it cannot provide the files which are tracked by the specified repository. The reason behind it is that we are not able to clone the remote repository to the local directory and the directory does not contain an older version of the remote repository.

The second scenario (S2) also has to trigger a failure of the builder because we do not want to clone a repository into a directory that is not empty. The clone command of Git itself does not allow to target a directory which contains other files. The possibility of deleting the content of the directory is out of question because we don't want to make it possible that the builder deletes files the user needs. Thus, losing progression he has made when working on a project.

When scenario (S3) arises, we can clone the repository into the specified directory and provide every file that the repository tracks.

The scenarios (S4) and (S5) can only occur when the behavior of the builder was executed, which was defined for scenario (S3), beforehand. For the scenario (S4), the builder should fetch the latest commits from the origin and merge them into the specified branch. If no error occurred during the fetch and merge actions, the builder should terminate successfully. If there is a fatal error during the pull, we need to let the builder fail. This way the user can check why the builder failed and resolve the problem by hand, e.g. resolving a merge conflict.

The last scenario we need to define the behavior for is (S5). In this case, we don't need to do anything. Thus, we only provide the files that the repository tracks and terminate successfully. For the optimal incremental condition of the build system pluto, the execution of the builder should be skipped because we want to minimize the number of executed builders.

4.2 GitInput

In this section, we will define the settings that can be configured which the builder `GitRemoteSynchronizer` uses in order to execute the defined behavior. The settings will be represented by the class `GitInput`. As we will see, the `GitInput` has a lot of settings which can be configured. To avoid a lot of different constructors and guarantee safety, we will use the builder pattern which is described in [6]. The mandatory attributes have to be passed to the constructor as arguments and the optional attributes can be set via setter methods which the builder provides. Each setter method returns an instance of the builder itself, which is an altered state of the builder object. This allows us to chain method invocations. By calling a separate build method of the builder it returns the final object with the wanted configuration. In the following list we can see every option we can set for the execution of the builder:

- directory the Git repository is stored in on local machine
- URL of the remote repository
- update bound(branch, commit, tag)
- branches that should be downloaded when cloning the repository
- if submodules should be cloned
- fast-forward mode to use
- if a commit should be made after a successful merge
- how long should a consistency check be forbidden

There are only two attributes that are mandatory and therefore have no default value. First, the directory where the repository has to be stored on the local machine. Secondly, the URL of the remote repository. Our builder does support the same protocol that the git command line tool does support, which are ssh, git, http and https. Git also supports ftp, ftps and rsync but those are inefficient and deprecated. We avoided reimplementing the command line tool for Git by using JGit[1]. JGit is a Java library with very little dependencies. It is maintained by the Eclipse Foundation and provides us with porcelain commands. Porcelain commands are user-friendly commands which can be used in day-to-day work while using Git. By using JGit, the user who executes the builder does not have to install the git command line tool. Thus, we have very little dependencies and the builder can be used right out of the box. The resulting attributes of `GitInput` can be seen in Listing 8. In the following subsections we will discuss the optional parameters in detail.

4.2.1 Additional Branches To Clone

The builder has to clone the remote repository into the specified directory when it has never been executed before. By default, the clone action only keeps track of the master branch. By adding the attribute `branchesToClone`, we provide a way to clone more than just the master branch. This is important because sometimes we want to build the project and use the development branch which has experimental changes that are not present in the master branch. The master is mostly used for stable versions that are set and do not change in the near future.

4.2.2 Submodules

Some repositories contain other repositories. This is needed if a repository depends on another repository and it is important what version of the other repository is used. To clone the repositories which are a part of the repository which are cloned, the flag `cloneSubmodules` can be set. When fetching and merging the latest commits of the remote repository, the commits of the submodules are also fetched and merged.

4.2.3 Fast-Forward Mode

As discussed in paragraph 2.2, there are three different options we can set for the fast-forward mode that should be used when merging the latest commits of the remote repository into the local one. Those options are encoded with the help of an enumeration. The fast-forward mode is important because by deciding which mode to use, we implicitly decide how we want to use the repository. Suppose we choose FF_ONLY and local changes were applied to the local repository. The next execution of the builder would not merge the latest commits of the remote. This would result in an inconsistency between the remote and local repository. Thus, the local repository should only be altered by the builder when using the fast-forward mode FF_ONLY.

4.2.4 Commit After Merging

Whenever a merge was successfully performed, the changes that are made to the current branch by the merge action are committed. If additional changes should be made after the merge, the setting `createMergeCommit` can be set to false. The default setting is to create a commit after a successful merge. Use this setting with caution. The reason for this is that every time the consistency between the local and the remote repository is checked, we check if the latest commit hashes of the local and remote branch differentiate. If this is the case, we see the requirement as inconsistent and indicate that the builder, which has registered the requirement, has to be executed. If no commit is made after the successful execution of a builder the hash of the latest commit does not change and therefore it is viewed as inconsistent every time we check for consistency.

[1] https://eclipse.org/jgit/

4.2.5 UpdateBound

Instead of only keeping the current working tree in sync with the master branch, it possible to keep it in sync with another branch, commit or even a tag. Because the hash of the latest commit of a branch and a tag can change, we need a way of expressing the synchronization with changing hashes. We propose to define an interface `UpdateBound` which can be seen in Figure 5. The interface defines three methods which have to be implemented by the subtypes of the interface. First, there is the method `getBound` which returns the textual bound which is tracked, like the name of a branch or a tag. The next method `getBoundHash` returns the newest commit hash of the bound. The last method compares the provided hash with the hash that gets determined by `getBoundHash` and should return true if they are the same.

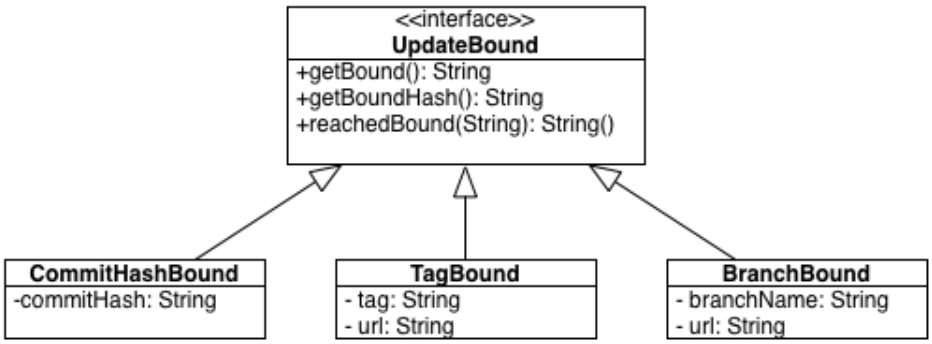

Figure 5: Structure of the UpdateBound abstraction and its concrete subclasses.

The simplest subtype of `UpdateBound` is `CommitHashBound`. It has an attribute `commitHash` which contains the hash of the commit the local repository has to be in sync with. The concrete methods `getBound` and `getBoundHash` both return `commitHash`. This results in a fixed state of the working tree, because it is not possible that the commit hash changes. The only way of modifying the changes that the commit represents is to rebase the commit history of the remote repository. However, this should happen very rarely.

The next subtype is `TagBound`. The concrete subclass needs an attribute which indicates what tag we want to keep the working tree in sync with. Additionally, it has to know the `url` of the remote repository. The URL is needed because we need to check what the tag of the remote points to. The method `getBound` returns the name of tag. The second method `getBoundHash` yields the hash of the ref the tag points to.

`BranchBound` is the last subtype of `UpdateBound` we want to introduce. The subclass has two attributes. The first attribute `branchName` states which branch we want the local repository to be in sync with. The second attribute is the `url` of the remote repository. The method `getBound` returns the name of branch which is represented by the attribute `branchName`. The second method `getBoundHash` has to calculate the hash of the latest commit that is stored on the branch of the remote repository.

```
1  public class GitInput implements Serializable {
2      ...
3      public final File directory;
4      public final String url;
5      public final List<String> branchesToClone;
6      public final boolean cloneSubmodules;
7      public final FastForwardMode ffMode;
8      public final boolean createMergeCommit;
9      public final UpdateBound bound;
10     public final long consistencyCheckInterval;
11     ...
12 }
```

Listing 8: Attributes of input class which is used for the builder GitRemoteSynchronizer.

4.2.6 ConsistencyCheckInterval

As we have seen in the `HTTPInput`, we need a way of setting the interval that indicates how long a consistency check is forbidden. This information is stored within `consistencyCheckInterval`.

4.3 GitRequirement

When defining a concrete subclass of `RemoteRequirement`, which calculates the consistency between the local and remote repository, we need to have an idea what it means to be consistent. By being inconsistent, the requirement will trigger the execution of the builder which has the requirement registered. Therefore, we have to think of the possible scenarios that can occur and what the requirement has to consider consistent and what not.

In the following list we can see what scenarios are possible and additionally are interesting for the requirement:

(R1) specified path of directory does not exist

(R2) specified path of directory does exist and is not a repository

(R3) hash of local update bound can not be calculated

(R4) update bound has not been reached by the local repository

(R5) update bound has been reached by the local repository

Now we will take a look at every scenario listed above and discuss if it has to trigger an execution of the builder and therefore has to indicate inconsistency or not.

For scenario (R1), we know that the builder has not been executed or has failed while executing, thus we have to consider the requirement as inconsistent and need to tell the build system that an execution of the builder is necessary. The second scenario (R2) is the corresponding scenario to the scenario (S2), which we defined earlier as we looked into the behavior of the builder. In this case, we have to consider the requirement as inconsistent because we have to let the builder fail while executing. The inconsistency of the requirement triggers an execution and the builder then fails to tell the user that the specified path is not valid and he has to change the attribute of the `GitInput` which he is using.

That an local update bound can not be calculated (R3) could be caused by a damaged repository, which does not know where the branch or tag, depending on the update bound the requirement is using, points to. In this scenario, we need to let the builder fail because the repository is damaged and can not be repaired automatically.

If the update bound of the requirement has not been reached by the local repository (R4), we need to force an execution of the builder, which synchronizes the local repository with the remote. It is important that the builder is executed because we want to ensure that the build system is sound. If we would not force the builder to execute, the target files would not represent the latest source files.

The last scenario (R5) would be that the local repository has reached the update bound. Therefore, the requirement has to be marked as consistent because we want to make sure that the optimal incremental condition holds, which means that the number of executions of builders has to be kept minimal.

Now we will take a look at the implementation of the three abstract methods of `RemoteRequirement`. `GitRequirement` needs three different informations to determine the consistency between the local repository and the remote. First, it needs to know where the local repository is located. Additionally, it has to know about the update bound it has to check against. As we discussed earlier, this could be an commit hash, tag or branch. The last information is the URL of the remote, which the local repository needs to be in sync with. They result in three fields as we can see in Listing 9

```
1 public class GitRequirement extends RemoteRequirement {
2     ...
3     private File directory;
4     private UpdateBound bound;
5     private String url;
6     ...
7 }
```

Listing 9: An outline of the attributes of GitRequirement.

With those three attributes, it is possible to check for the consistency between the local repository, with the path stored inside of the attribute `directory`, and the remote, which can be accessed via the attribute `url`.

The first method we will discuss in detail is `isRemoteResourceAccessible`. To find out if `url` is a repository and is accessible, Git provides us with the command ls-remote which returns all the refs the repository has. If the command does

not terminate successfully `url` can not be accessed or `url` does not reference a Git repository. The command ls-remote is a porcelain command. Therefore, JGit offers a Java implementation for it. We create a `LsRemoteCommand` object and try to call it. If the call did not throw an exception, we can assume that the repository is accessible and return true. Otherwise, the method has to yield false.

The next method is `isLocalResourceAvailable`. It has to tell the method `isConsistent` whether the directory does contain the correct repository. A correct repository, is a repository which has a remote set that has the URL set as `url`.

The last method we have to implement is `isConsistentWithRemote`. Its job is to check if the local repository has reached `bound` in comparison to the remote repository. The first step is to determine the hash of the commit, which is called HEAD, that represents the working tree. Then, `bound` calculates its own hash and compares it to the hash of the working tree. If they are different the requirement is marked as inconsistent. In the other case, the requirement is consistent.

4.4 GitRemoteSynchronizer

At last we will look at the builder `GitRemoteSynchronizer` that uses the newly introduced `GitRequirement`. To implement a concrete subclass of `Builder` we have to implement the following methods:

- `String description(GitInput input)`
- `File persistentPath(GitInput input)`
- `None build(GitInput input)`

The path of the build summary, which is returned by the `persistentPath` method, needs to be invisible to the repository. This stems from the fact that the build summary is dependent on the input and therefore can be different for every user who runs a build script including the `GitRemoteSynchronizer`. Files which are contained in the .git directory can not be tracked by the repository and therefore can not appear as uncommitted changes. Thus, we chose to let the path of the build summary be .git/git.dep which is contained in the directory `input.directory`.

Now the will discuss the method `build`. The first condition which has to be checked is if the directory, defined within the input, is empty. In this case, we have to clone the remote repository into the directory. We looked in the reason for this action in (S3). Thus, we will clone the remote repository into the local directory. Then, we have to check the set update bound and change the HEAD ref to the most recent ref of the update bound. In the case that the directory is not empty, we can force a pull. The remote requirement was not consistent, because the method `build` is executed, thus there are new commits available that have to be fetched and merged. It could be possible that the user changed the HEAD of the local repository manually. Therefore, we have to checkout the ref of the old update bound before we can pull the new commits. The checkout itself could fail and in that case `build` will throw an exception.

After cloning or pulling the latest commits we have to provide the files the repository tracks. But we will only include the files that are not ignored within the repository and no files contained inside the .git directory.

5 Maven

As we will see in section 6, most projects which are implemented in the programming language Java are dependent on a number of other libraries. In this regard, Maven Central and the dependency resolution algorithm of Maven are very useful. Therefore it would be useful for a build script of the pluto build system to resolve dependencies which reference artifacts on Maven repositories. In the following section, we will discuss a concrete builder which can resolve Maven dependencies, the needed input and remote requirement subclass.

5.1 Behavior of Builder

The first step is to identify the scenarios that can occur when the builder is executed. The following list shows every interesting scenario:

(S1) Artifact is not available and repositories are not accessible

(S2) Artifact is not available and repositories are accessible

(S3) Artifact is outdated and repositories are accessible

(S4) Artifact is outdated and repositories are not accessible

(S5) Artifact is up-to-date

The next step is to define the behavior of the builder in every scenario. In this subsection we will look at every scenario listed above and discuss the behavior we defined for the builder.

In the case of scenario (S1), the builder has to fail because it cannot provide the files attached to the requested artifact. The reason is that the resolution algorithm is not able to access the remote repositories. Additionally, the artifact is not available on the local repository. Therefore, builders which require files provided by the builder that should resolve the Maven dependencies are not able to proceed because the dependencies could not be resolved.

When the second scenario (S2) occurs, the resolution algorithm can resolve the artifact and create the resulting dependency graph. Thus, the execution of the builder has to be triggered, which would enable a successful run of the whole build script.

When scenario (S3) arises, a newer version of the artifact is available which is contained in the defined version range of the dependency. Additionally, the repositories which contain the defined dependency are accessible and therefore the dependency resolution algorithm can be executed. Therefore, the builder has to be executed.

In the case of an outdated artifact and an inaccessible repository (S4), we do not want to execute the builder because the resolution would fail, but it should not be considered as a failure because it should be possible to run a build script without an internet connection. Thus, the execution of the builder should not be triggered.

If the artifact is up-to-date (S5), the execution of the builder should be forbidden to ensure the optimal incremental condition of pluto because we have to minimize the number of executed builders.

5.2 MavenInput

In this subsection, we have to define an input class for the builder which resolves Maven dependencies. As we did for Git, we will use the builder pattern. Now we will discuss the possible settings in detail. First, we have to define the path where the local repository is located. The second setting is the list of dependencies which the builder has to resolve. The input needs at least one dependency. The last option we can set within the input for the builder are repositories the Maven resolution algorithm has to check for the given dependencies in addition to Maven Central.

```
1 public class MavenInput implements Serializable {
2     ...
3     public final File localRepoLocation;
4     public final List<Dependency> dependencyList;
5     public final List<Repository> repositoryList;
6     ...
7 }
```

Listing 10: An outline of the attributes of MavenInput.

5.2.1 Dependency

First, we have to take a look at the definition of a dependency. As we have seen in subsection 2.3, we need the following information to define a dependency:

- groupId
- artifactId
- version
- classifier (optional)
- type (optional)
- exclusions (optional)

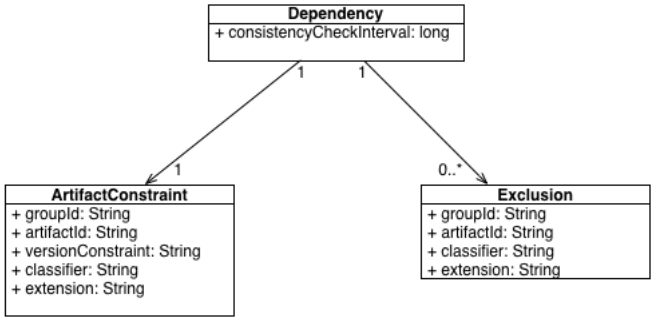

Figure 6: Structure of Dependency class for input of Maven builder.

The class structure that follows, which provides the needed information for a dependency, can be seen in Figure 6. A `Dependency` is split into an object of type `ArtifactConstraint` and a list of `Exclusion` objects.

The `ArtifactConstraint` object contains the information Maven needs to reference the artifact. Because `groupId`, `artifactId` and `version` are mandatory, the corresponding attributes of an object of type `ArtifactConstraint` have to be not null. For the optional attributes, `classifier` and `type`, the attributes can be null and in the case of a null reference the resolution algorithm assumes they have the default values, jar and null respectively.

Next, we can specify artifacts that do not need to be added to the dependency graph, which the resolution algorithm of Maven creates while building the resulting dependency graph. Those artifacts can be added to the list of `Exclusion` objects the dependency has. For an `Exclusion`, you only need to specify the `groupId`, `artifactId`. The optional attributes, `classifier` and `type`, can be set for the `Exclusion` object. They have the same meaning as the optional attributes defined in an `ArtifactConstraint` object. The last information an `Dependency` object needs, is the interval in which rechecking consistency is forbidden. The definition is the same as discussed in subsubsection 4.2.6.

It is important to note that every dependency has its own interval for consistency checks. We chose to do this because every library has its own release cycles. Therefore, we want to check for the availability of a new version in different intervals.

5.2.2 Repository

The next setting that can be set are repositories for the resolution of the defined dependencies. If no repository is set, the requirements and builder will only look for artifacts on Maven Central.

The most important attribute of a repository is `url`. Additionally, the `id` of the repository can be set. The layout of a repository has changed since the version 1.x. If you want to reference a repository, which was created during the usage of a version lower than 2.0, the layout needs to be configured as legacy. In the other case, when referencing a repository with version greater or equal than 2.0, the layout has to be set to default. If the layout of the repository is not configured, the builder assumes that the repository has the default layout.

At last, the policies for snapshot and release artifacts can be chosen. A policy determines how the resolution algorithm acts when checking for new artifacts and what happens if the meta data attached to an artifact is missing or damaged. In our case, a policy can have two settings. First, it can be marked as enabled. Secondly, the checksum policy can be set. The checksum policy determines what happens if a checksum file of an artifact is missing or incorrect. In the following list, the possible settings for the checksum policy are shown: checksum policy:

- ignore
- warn
- fail

Normally, it is possible to set an update policy which states how many times the repository has to be checked for new artifacts when building the program. Because we already have the `consistencyCheckInterval` to tell the system how often the remote repositories have to be checked for new versions of the needed artifacts, the update policy is set to always.

5.3 MavenRequirement

We need to look at the possible scenarios which can occur while checking for the consistency between the locally cached versions of the artifacts and versions stored on the remote repositories.

In the following list we can see what scenarios are possible and additionally are interesting for the requirement:
In the following list we can see what scenarios are possible and additionally are interesting for the requirement:

(R1) Local version of artifact is not available

(R2) Local version of artifact is older than the highest version inside of version range

(R3) Local version of artifact is equal to the highest version inside of version range

Now we will take a look at every scenario listed above and discuss if it has to trigger an execution of the builder and therefore has to indicate inconsistency.

When the first scenario (R1) arises, the builder has never been executed before or has failed while executing. The reason for a failing builder earlier could be that a declared dependency does not exist. Because we do not know the exact reason why an artifact is not available locally, we have to trigger the execution of the builder which has registered the dependency of the specialized remote requirement. Thus, the requirement has to be marked as inconsistent.

In the case of scenario (R2), there exists an artifact, which is contained inside of the specified version range, on one of the defined remote repositories that has a higher version number than the locally available artifact. For providing the higher version, the builder has to be executed. Therefore, we have to consider the requirement as inconsistent.

(R3) is the last possible scenario. In this case, we do not want to execute the builder because the optimal incrementality has to be maintained. Therefore, we need the requirement to be consistent.

Now we will take a look at the implementation of the `MavenRequirement` and its implementations of the abstract methods provided by `RemoteRequirement`. `MavenRequirement` needs three different informations to determine the consistency between the locally available versions of the artifacts and the remotely stored versions. First, it needs to know where the local repository is located. The next information are the artifacts and which versions are allowed to be resolved by the builder. This is encoded within a list of `ArtifactConstraint` objects which we have defined in Figure 6. At last, we need the repositories, which are checked for higher versions of the artifact we are interested in, in a addition to the repository Maven Central. They result in three fields as we can see in Listing 11.

```
1 public class MavenRequirement extends RemoteRequirement {
2     ...
3     private File localRepository;
4     private List<ArtifactConstraint> artifactConstraints;
5     private List<Repository> repositories;
6     ...
7 }
```

Listing 11: An outline of the attributes of MavenRequirement.

With those three attributes, it is possible to check for the consistency between the local artifacts, which are contained within the local repository and the artifacts that are stored on the remote repositories.

For `isRemoteResourceAccessible`, the library Aether does not provide a fast way of checking whether a repository is accessible or not. It forces us to let the method simply return true. Therefore, even if the repositories are not accessible, the builder is executed. If a cached version of every artifact is available locally, the builder will not fail because it uses the cached version.

For the method `isLocalResourceAvailable`, we traverse the directory structure of the local repository. First, we check if the directory of the local repository exists. If this is not the case, we have to return false because no local repository exists. Therefore, no cached version of the artifact is available. Next, we are interested in the sub-directory `groupdId/artifactId` while every "." within the `groupId` and `artifactId` are replaced with "/". In this sub-directory, there are possibly multiple other sub-directories for each version that is available in the local repository. If the directory does not contain any sub-directory, the method returns false. If there exists a version directory which contains a file with the correct extension, defined via the setting type of the `ArtifactConstraint`, the method has to check the next defined artifact. This has to be checked for every artifact that is contained in the list `artifactConstraints`. If every artifact has a locally available version, `isLocalResourceAvailable` returns true.

Now we will discuss the last method `isConsistentWithRemote` and its implementation. At first, the method has to determinate the highest version of the artifacts which are available locally and are allowed by the version range defined by the `ArtifactConstraint` objects. The collection of the highest version per artifact is almost the same as described for the method `isLocalResourceAvailable`. In addition, the method has to collect every version and sort them by version in order to determine the highest possible version. Secondly, the method calculates the highest version of the artifacts in question, which are still contained within the version range, stored on the defined remote repositories. Now the method can compare the highest version of the local artifact with the highest version of the artifact on a remote repository. If there exists a artifact which is not locally stored or has a lower version than located on a remote repository, the method has to yield false. If no higher version was found on the remote repositories, the method has to return false because the builder which registered the requirement should not be executed again.

5.4 MavenDependencyResolver

At last, we will look at the builder `MavenDependencyResolver` that uses the newly introduced `MavenRequirement`. A builder has to define an input class and an optional output. We discussed the input class `MavenInput` in subsection 5.2. The builder `MavenDependencyResolver` also defines an output. The output is a list of `File objects` which point to the paths of the artifacts that are resolved by the dependency resolution algorithm of Maven.

To implement a concrete subclass of `Builder` we have to implement the methods:

- `String description(MavenInput input)`
- `File persistentPath(MavenInput input)`
- `Out<List<File>> build(MavenInput input)`

The description method returns a `String` object which indicates that the current builder resolves maven dependencies.
The path of the build summary, which is returned by the `persistentPath` method, is stored in the root of the local repository. Thus, the returned path is `input.localRepository/maven.dep`.

The method `build` has to register an own `MavenRequirement` for every group of dependencies which have the same `consistencyCheckInterval`. We need a file which is storing the timestamp of the last successful consistency check between the locally available artifacts of the group and the corresponding remote repositories for every `MavenRequirement`. Therefore, for every group of Maven dependencies with the same `consistencyCheckInterval`. By grouping the dependencies by the interval we minimize the number of registered remote requirements but also provide the feature of defining different intervals for different dependencies. Now we have to resolve dependencies with the help of the Java library Aether[2], maintained by the Eclipse Foundation. The next step is to add the dependencies to a collect request the resolution algorithm acts on. In this step, we have to consider the exclusions defined for a dependency and add them to the `Dependency` object of the Aether library. Now the repositories, which we want the defined dependencies to be resolved from, have to be set. Next, the dependency resolution algorithm, which is implemented within Aether, can be executed. If

[2] http://www.eclipse.org/aether/

an exception occurs during the execution of the algorithm, the builder itself forwards the exception. Thus, the execution of the builder fails. After the successful resolution of the dependencies the builder collects the paths of every artifact, which was included in the resulting dependency graph. Therefore, every transitive dependency that was not specifically excluded is contained within the list of paths which point to the artifacts. Because the builder MavenDependencyResolver has an output defined, we have to return the collected list of paths.

6 Case Study

In this case study we will implement a build script for some of the sub-projects that are part of the ecosystem of Monto [12] and use the builders `HTTPDownloader`, `GitRemoteSynchronizer` and `MavenDependencyResolver`, which were introduced earlier in this thesis. Monto is a disintegrated development environment. It was introduced by Sloane et al. and extended by Keidel [10] and Pfeiffer [11]. Because Monto is not an integrated development environment, instead of having a plug-in for every pair of editor and programming language for supporting the language within the editor, Monto provides an abstraction between the editor and the programming language. If you want to use Monto within your editor, you need to implement an interface provided by Monto which enables the editor to communicate with Monto and therefore provide Monto with the needed content of the source files and receive the language support implemented by a specific service. This enables the editor to support every programming language which has implemented the interface provided by Monto. Thus, the number of plug-ins for language support shrinks.

For the build script, we will take a look at three different sub-projects services-base-java [3] (SBJ), services-java [4] (SJ) and services-javascript [5] (SJS). The project SBJ contains the base classes for implementing new services which provide support for a specific programming language. The concrete implementations can be found within the sub-projects SJ and SJS. Each of the projects listed above has its own Git repository. The concrete dependencies on JARs can be seen in Figure 7. The compilation of the SBJ Git repository results in the JAR services-base-java.jar which is required for building the other repositories SJ and SJS.

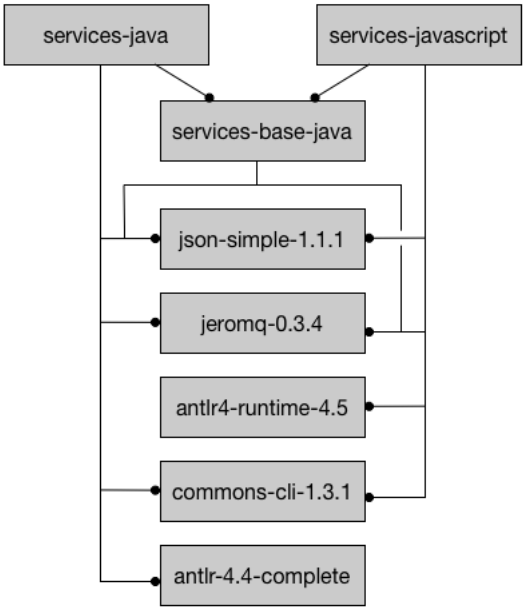

Figure 7: Structure of dependencies between JAR for the sub-projects of Monto.

Currently there does not exist a script for building any of the three projects. The structure of the projects is very similar. The source files are stored within the directory src. The library JARs are located in the subdirectory lib. The exact libraries contained within the directory lib can be seen in Figure 7.

The new build scripts will be able to clone and maintain the Git repository of SBJ and will resolve the Maven dependencies automatically. For building the three different projects, we have to implement three different builders and connect them with the help of the builders which were introduced in subsection 3.4, section 4 and section 5. The new builders will be called `ServicesBaseJavaBuilder`, `ServicesJavaBuilder` and `ServicesJavascriptBuilder`. The name already suggests which projects the respective builder has to build. We will now discuss the implemented builders and

the corresponding input classes.

At first, we will define a class which contains the dependencies for libraries which can be resolved from a Maven repository, which in this case is only Maven Central. The dependencies are encoded by static attributes which are publicly available and can not be altered. The reason we collect them in a single class is that the projects have a common subset of libraries which are needed to compile and execute the resulting JAR files. Additionally, it is safer to reduce redundancy between the input classes. When a dependency of SBJ changes, the dependency of SJ is also likely to change because of the shown dependency between the repositories.

The builders `ServicesJavaBuilder` and `ServicesJavascriptBuilder` will synchronize a directory with the Git repository which contains the project SBJ and compile the source files which are tracked by the repository. The resulting JAR can then be used by the builders to compile the source files of SJ and SJS. Therefore, we need to let the user of the build script decide which settings he likes to use while executing the build script. The settings regarding the execution of the `GitRemoteSynchronizer` are grouped together in the class `GitSettings`. Every setting listed and described in subsection 4.2 can be customized within this class. As for the class which holds information about the Maven dependencies the fields of `GitSettings` are static and publicly available. Additionally, the class provides the user with a method which transforms the given settings into an object of the type `GitInput` which can be used for the execution of `GitRemoteSynchronizer`.

When assembling an executable JAR, we have to provide a manifest file. Such a file contains information about the following:

- Entry point of the program, if needed
- Version of the JAR
- Location of libraries which are needed to run the JAR, called the classpath
- Whether the JAR's packages are sealed or not

We implemented a simple builder ManifestFileGenerator, which can take this information and assemble the content of the manifest file and provide the resulting manifest file.

6.1 ServiceBaseJavaBuilder

The first input class which we will discuss is `ServicesBaseJavaInput`. It is the input class for the builder which can produce a JAR from the source files contained within the repository SBJ. The builder has to know which directory contains the Java source files. The path for the directory which contains the source files is stored in the attribute `src`. Next, the builder needs a directory which will be used to save the resulting class files after compiling the source files. The directory can be referenced with the `File` object of the name `target`. After compiling the source files, it will take the class files and pack them together in a JAR file. The path of the JAR file, `jarLocation`, which will be produced and provided by the builder, has to be set in the input class. Because pluto will raise an error if there is a hidden dependency detected while executing the build script, the builder which compiles Java source files has to know which builder has provided the Java source files. The injected dependencies can be set with the help of the attribute `requiredUnits`. As we will see in subsection 6.3 and subsection 6.2, the builder `ServicesBaseJavaBuilder` will be required by other builders which will call the builder `GitRemoteSynchronizer`. The `GitRemoteSynchronizer` builder will provide the source files for the `ServiceBaseJavaBuilder`. To avoid the detection of hidden dependency, we have to inject the dependency of `GitRemoteSynchronizer`. The attributes which are needed to run the `ServicesBaseJavaBuilder` can be seen in Listing 12.

```
1 public class ServicesBaseJavaInput implements Serializable {
2     ...
3     public final File src;
4     public final File target;
5     public final File jarLocation;
6     public List<BuildRequest<?, ?, ?, ?>> requiredUnits;
7     ...
8 }
```

Listing 12: An outline of the attributes of ServicesBaseJavaInput.

The first action `ServiceBaseJavaBuilder` has to take is to resolve the Maven dependencies by requiring the builder `MavenDependencyResolver`. As we can see in Figure 7, SBJ only depends on json-simple-1.1.1 and jeromq-0.3.4. We have to construct an object of `MavenInput` which has those two Maven dependencies set. As we discussed earlier, every dependency on a Maven artifact has been declared within the class `MavenDependencies`. After the method `build` has assembled all the Java source files contained in `input.src`, the builder has to require the `JavaBuilder` which compiles the source files and creates the resulting class files. If a new version of an artifact is resolved by `MavenDependencyResolver`, the manifest file has to change. Therefore, we will require an instance of `ManifestFileGenerator`. After creating or updating the manifest file, the builder has to collect the class files stored inside the directory `input.target` and archive them in a JAR file. The resulting dependency graph can be seen in Figure 8.

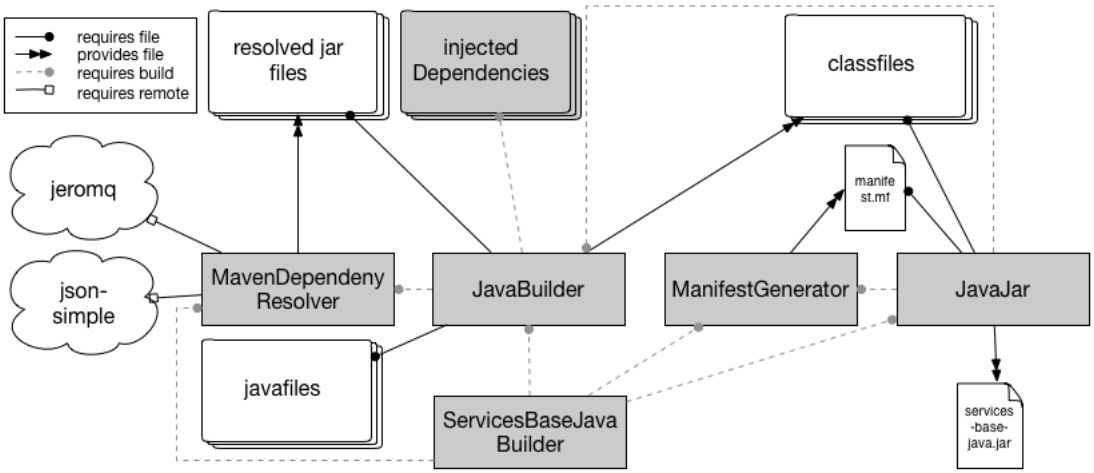

Figure 8: The complete dependency graph of the builder ServicesBaseJavaBuilder.

6.2 ServicesJavaBuilder

For the input, we have to set the directory of the source files, the directory, which will contain the class files, and the path where the JAR has to be stored. Additionally, the path of the class file directory and the path of the JAR of SBJ have to be set.

The `ServicesJavaBuilder` has to take the following step during the execution of its `build` method:

1. keep baseSrc directory up-to-date with Git repository
2. build source files of SBJ and generate JAR
3. resolve Maven dependencies
4. download antlr4-4.4-complete.jar
5. build source files of SJ
6. write out manifest file
7. generate JAR

At first, we have to require `GitRemoteSynchronizer` and construct the input. As we mentioned earlier, the class `GitSettings` provides us with a method to construct an object of type `GitInput`. This ensures us that we have the latest version of SBJ we are interested in locally available. The next step is to require the previously described builder ServicesBaseJavaBuilder. The source directory for this builder is the local repository managed by the required `GitRemoteSynchronizer` builder. Now the builder has to resolve the needed Maven dependencies by requiring an instance of `MavenDependencyResolver`. Because there does not exist an Maven artifact for antlr-4.4-complete, the builder has to require the `HTTPDownloader`. After we assembled every library which is required for building SJ, we can require the

builder `JavaBuilder`, which takes the Java source files and generates the class files for it. Next, the builder has to generate the manifest file and finally take the generated class files and create a JAR which will be provided by the builder. A part of the resulting dependency graph can be seen in Figure 9. We left out the part which is not very different from Figure 8. It would show us requirements of the builders `JavaBuilder`, `MavenDependencyResolver`, `ManifestFileGenerator` and `JavaJar`, which have their own file and build dependencies.

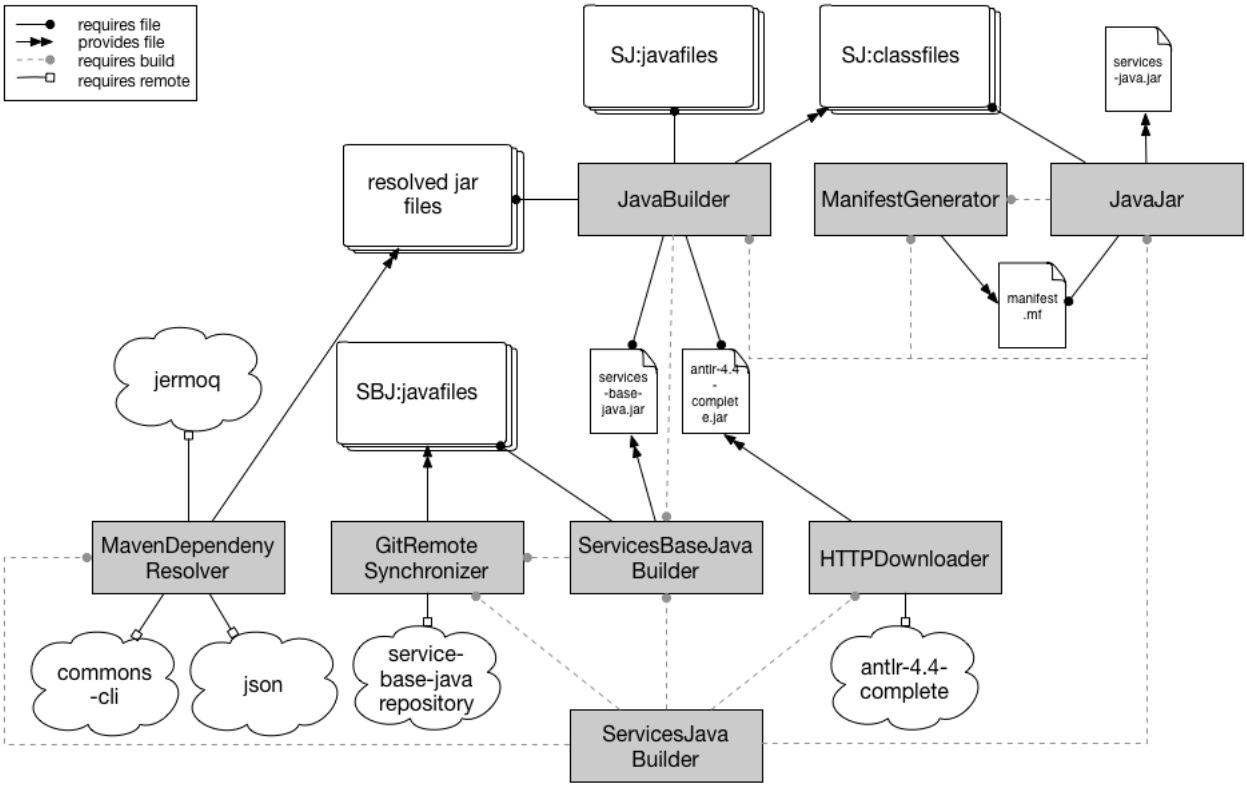

Figure 9: The partial dependency graph of the builder ServicesJavaBuilder.

6.3 ServicesJavascriptBuilder

The input of `ServicesJavascriptBuilder` is structured exactly the same as the input of `ServicesJavaBuilder`. The structure of the method `build` defined by the `ServicesJavascriptBuilder` class is almost the same as `build` of `ServicesJavaBuilder` which was previously discussed. The difference is that we do not need to require the `HTTPDownloader` because the project depends on antlr4-runtime-4.5, which is available as an artifact on Maven Central, instead of antlr-4.4-complete.

6.4 Advantages of proposed Solution

In this section, we will identify the problems which arise because there is currently no build automation available for the three projects and show how the proposed build script will solve those problems.

(P1) inconsistency between the interface provided by SBJ and projects depending on it

(P2) inconsistency between the libraries used by SBJ and libraries which are contained within the classpath of SJ and SJS

Those problems are present because when the repository SBJ changes, we have to take multiple steps to generate the JAR and include it within the compilation of the projects SJ and SJS.

The first issue (P1) is solved by the presented build script because when the content of the repository changes, which results in a new commit, the `GitRequirement` would be marked as inconsistent as soon as the system allows the requirement to calculate a new consistency state. Therefore, the `GitRemoteSynchronizer` builder has to be executed. Because the builder, responsible for keeping the local repository in sync with the remote repository of SBJ, has been executed, the content of the directory, which stores the Java source files, will change. SBJ has to be executed because some old file requirements are inconsistent or new file requirements have been added. Thus, the build requirement is inconsistent and will trigger the execution of the SJ builder. This results in the execution of the builder for SBJ and SJ in a consistent manner.

The problem (P2) can be solved by the proposed solution. Let's assume the projects SBJ and SJ have the same Maven dependencies at all times. Imagine the scenario where a newer version is available for the json-simple library and we want to use this version for the projects SBJ and SJ. The developer would need to replace the library in two places. First, in the directory of SBJ and secondly, in the directory of SJ. In addition, it is likely that the parameter of the calls of javac and jar would have to change. In the proposed build script, the dependency on json-simple is handled by `MavenDependencyResolver`. The `MavenRequirement` would have to be checked, but because there is a new version of the artifact available on Maven Central and the defined version range allows the new version, the requirement is inconsistent and `MavenDependencyResolver` is executed. Thus, the builders of SBJ and SJ, which both require `MavenDependencyResolver`, have to be executed and therefore the libraries which are used to compile the Java source files, will change. The libraries and manifest files for both projects would change consistently.

7 Related Work

There are a lot of different build systems available. Therefore, we cannot discuss every difference between the proposed solution and how other build systems support the resolution of Maven dependencies and synchronization of a directory and a given Git remote repository. We will look at build systems which can build projects that can be run with the help of the Java Virtual Machine because Maven dependencies reference mostly libraries in the form of JAR files.

It is obvious that Maven is capable of resolving Maven dependencies, but Maven does not support resolving the dependency on a Git repository. To use a Git repository as a library, we would have to clone the repository manually and have to install the artifact locally. JitPack[3] is a third-party solution for automatically resolving GitHub[4] dependencies. The JitPack Maven repository has to be added, which will contain a published artifact of the needed Git repository, and the dependency needs to be included in the POM of the project. JitPack does support Release tags, branches and commit hashes. A problem with this approach is that the it forces the dependency on a third-party repository. Another problem are the build times which can take up to 15 minutes.

Gradle[5] can resolve Maven dependencies and it is possible to define how long all artifacts have to be cached. But with the use of the `MavenDependencyResolver` builder, it is possible to set an interval for every Maven dependency the project depends on. Every library has a different release cycle and therefore it is useful to have different dependency resolution intervals for different libraries. Gradle does not have built-in support for resolving dependencies located on a Git repository. Gradle can handle Maven dependencies, therefore we can use the third-party solution JitPack mentioned above. Thus, Gradle has the same shortcomings as Maven when using a Git repository as a dependency.

Bazel[6], a build system developed by Google, has a dependency graph that looks like the dependency graph of pluto. Builders can depend on other builders, and builder require and provide files. However, Bazel is not capable of dynamically discovering dependencies. In this thesis, we proposed to add a new kind of dependency, the remote requirement, to pluto. Therefore, pluto has a way of checking if new commits were added to a Git repository. Bazel can clone a Git repository and checkout a tag or commit hash, but Bazel has no way of pulling new commits when a branch is updated. Bazel does support resolving Maven Dependencies, but when adding repositories, it is not possible to add an update policy and therefore the interval for consistency checks, between the local artifact and the artifact located on the repository, can not be specified.

SBT[7] is a tool for building Scala and Java based projects. Currently, SBT only supports the clone and checkout action performed an dependency in form of a Git repository. It is possible to configure the commit hash or branch of the dependency, but SBT cannot pull the latest changes. To use a newer version after the initial compilation, we have to delete the locally stored dependency manually. Like any other mayor build tool, SBT is capable of resolving Maven dependencies. SBT can declare additional repositories. The advantage of pluto over SBT, with regards to Maven dependency management, is that pluto can set an interval for every dependency, which defines how often the dependency will be checked for updates.

[3] https://jitpack.io
[4] https://github.com
[5] https://www.gradle.org
[6] https://www.bazel.io
[7] http://www.scala-sbt.org

8 Conclusion and Future Work

We extended pluto, which was developed by Sebastian Erdweg et al. [8]. pluto is a sound and optimal incremental build system which supports dynamic dependencies. One main contribution was that pluto does have two different kinds of dependencies, build dependencies and file dependencies.

In this thesis, we proposed to add a new kind of dependency, a remote dependency, which is capable of checking whether a remote resource is consistent with the locally cached version of it. Because the remote resource is not always available, we added a way to check if a cached version is available and whether the remote resource is available. In addition, we added a way to forbid a consistency check for a certain amount of time because a remote access can be a very costly action.

To demonstrate how to use the `RemoteRequirement` in context, we described three different subtypes `HTTPRequirement`, `GitRequirement` and `MavenRequirement` and the corresponding builders which use the new subtypes. Because `HTTPRequirement` would have to download the resource first for checking if it is a new version, it can not check the consistency between the cached and the remote resource. Therefore, the requirement can only provide us with the functionality of forbidding the execution of the corresponding builder for a certain amount of time. `GitRequirement` can check whether new commits have been added to a remote repository or not, which no other build system we looked at does. pluto can not only clone a remote repository and use it as a dependency but also pulls the new commits and provides the new changes locally automatically. Other build systems need manual work to get a newer version of a Git repository or rely on third-party solutions with build times which can take up to 15 minutes. For the `MavenRequirement`, we group the dependencies which have the same consistencyCheckInterval to minimize the number of registered requirements but also provide a way of setting different intervals.

In the case study, we have shown how the new builders can be used to create a build script which involves files located on a HTTP server, Maven dependencies and a dependency on Java source files located on a Git repository. It has shown that we can create a build script which can track a Git repository to ensure consistency between the projects we want to build and the projects they depend on.

In future work, we want to investigate how other systems such as Subversion, Perforce or Mercurial can be supported by pluto with the use of the RemoteRequirement abstraction to enable a sound and optimal incremental building process. pluto's ecosystem is split into multiple Git repositories and every project has a Maven artifact published on a remote Maven repository. It would be interesting to replace the current build script which is using Maven and build pluto with pluto itself, but still publish the artifacts on a Maven repository.

References

[1] Antlr (another tool for language recognition). http://www.antlr.org. Accessed: 2015-11-26.

[2] Infographic about sloc of software. http://www.informationisbeautiful.net/visualizations/million-lines-of-code. Accessed: 2015-11-26.

[3] Monto service base library for java. https://github.com/monto-editor/services-base-java. Accessed: 2015-11-26.

[4] Monto service for java. https://github.com/monto-editor/services-java. Accessed: 2015-11-26.

[5] Monto service for javascript. https://github.com/monto-editor/services-javascript. Accessed: 2015-11-26.

[6] Joshua Bloch. *Effective Java (2nd Edition)*. Addison-Wesley, 2008.

[7] Scott Chacon and Ben Straub. *Pro Git*. Apress, 2014.

[8] Sebastian Erdweg, Moritz Lichter, and Manuel Weiel. A sound and optimal incremental build system with dynamic dependencies. In Jonathan Aldrich and Patrick Eugster, editors, *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA 2015, part of SLASH 2015, Pittsburgh, PA, USA, October 25-30, 2015*, pages 89–106. ACM, 2015.

[9] Brian R Jackson. *Maven: The Definitive Guide*. O'Reilly Media, 2015.

[10] Sven Keidel. A disintegrated development enviornent. Master thesis, Technische Universität Darmstadt, 2015.

[11] Wulf Pfeiffer. A web-based code editor using the Monto framework. Bachelor thesis, Technische Universität Darmstadt, 2015.

[12] Anthony M. Sloane, Matthew Roberts, Scott Buckley, and Shaun Muscat. Monto: A disintegrated development environment. In Benoît Combemale, David J. Pearce, Olivier Barais, and Jurgen J. Vinju, editors, *Software Language Engineering - 7th International Conference, SLE 2014, Västerås, Sweden, September 15-16, 2014. Proceedings*, volume 8706 of *Lecture Notes in Computer Science*, pages 211–220. Springer, 2014.