Platforma průmyslové spolupráce

CZ.1.07/2.4.00/17.0041

Název

Effective Testing of Local Git Branches Using Remote Execution

Popis a využití

- efektivnější spouštění automatizovaných testů (méně vytěžující server) při výuce
- výuka: pokročilá Java

Jazyk textu

- anglický

Autor (autoři)

- Jakub Senko

Oficiální stránka projektu:

- http://lasaris.fi.muni.cz/pps

Dostupnost výukových materiálů a nástrojů online:

- http://lasaris.fi.muni.cz/pps/study-materials-and-tools

Contents

1 Introduction

Most software developers work on multiple tasks at the same time, especially in large companies. Programmer may be a member of more than one project team or multiple new features require his attention. Additionally, one of the tasks may have higher priority than others and must be resolved immediately – if a customer reports a problem, the bug must be fixed as soon as possible. When working on these features, developers should perform intermediate testing to make sure they did not introduce inadvertent errors. Often, changes are published in the source code repository before verifying that everything works. They can contain broken code which may cause complications for other developers before the problem is fixed. Therefore, each change should be tested locally on the developer's machine, before the solution is ready to be shared with other team members or made public. This practice, commonly known as *pre-commit testing,* however consumes not only developer's time, but also his computer's resources, particularly if it is done for every feature he is working on. It is especially noticeable if the projects are large and have an extensive test suite.

Fortunately, most of the tasks that are associated with project development, including compilation and test execution can be *automated* using various tools. The goal of this thesis is to investigate these tools and use them to make pre-commit testing more effective. This can be achieved by offloading the build and test execution from the developer's machine to a remote server using a convenient automated tool. The topic of this thesis has been provided by Red Hat, the world's leading provider of open source solutions[1], so the result must be specific to the processes and tools that are used to develop Java software there. Projects are assumed to use *Apache Maven,* which is a very popular open-source project management tool. Moreover, the source code is stored in a *git* repository and the solution may take advantage of the internal continuous integration server running a popular *Jenkins CI* software.

All these tools are properly presented in the second chapter of this thesis, *Project Automation,* after an introduction to the software configuration management is provided. In the following chapter, *Analysis and Design,* existing solutions are discussed and requirements for the application are listed. Subsequently, they are analyzed to determine the best way to implement the tool. The fourth chapter, *Implementation and Tools* contains

1. `http://www.redhat.com/about`

description of the technologies used during the development, as well as the structure and internals of the program itself. Finally, in the *Conclusion* chapter, short summary and several options for future development are presented.

2 Project Automation

In order to solve the problem presented in the introductory chapter, more detailed understanding of processes and tools that are used to develop software is required. The goal of this thesis is development of an automation tool, this chapter therefore provides an introduction to *project automation,* which is a concept of using automated tools to perform *configuration management.* Additionally, it is important to describe existing automation software that is relevant to the implementation part of the thesis - *git,* Apache Maven and Jenkins CI and to compare them to other similar tools.

2.1 Configuration Management

„Configuration management (CM)[1] *is concerned with the policies, processes, and tools for managing changing software systems.*“[2, chap. 25] The authors of the Configuration Management Best Practices identified several key areas of CM[1]:

- Change control
- Source code management
- Environment configuration
- Build engineering
- Release engineering
- Deployment

Software development is a very dynamic process. At any point, project requirements can change or a bug may be reported. The goal of *change control* (CC) is to *„ensure that the evolution of the system is a managed process and that priority is given to the most urgent and cost-effective changes.*“[2, chap. 25.1] The most common tool to facilitate CC is an issue tracker[2].

Source Code Management (SCM) *„is responsible for keeping track of different versions of software components or configuration items and the*

1. Configuration management is a more general engineering term. In the context of software development, *Software Configuration Management* is usually used. However, to avoid confusion with *Source Code Management,* the general term is used in this thesis.
2. According to a book about a commercial issue tracker: *„Issue in JIRA can be anything in the real world to represent a problem domain. It can be a software bug, a help desk ticket, or a customer request.*“[3, chap. 3]

systems in which these components are used."[2, chap. 25.2] The main purpose of SCM is to monitor each code modification using various metadata. As a result, developers are able to view project history, revert changes or work on isolated lines of development (branches), depending on the specific tool used.

Environment Configuration (EC) *„refers to identifying, modifying, and managing the interface dependencies required for the system to successfully progress from development to QA*[3] *to production."*[1, chap. 3] In other words, every system interacts with various software and hardware. EC makes sure that these dependencies are properly set up. Environment configuration must be coordinated with other areas of configuration management, especially build engineering, release management and deployment to be effective.

Software build can be defined as a process of translating human-readable source code into an executable program (compilation) or a result of such procedure. In a broader sense, it is a series of steps that transforms creative artifacts[4] into a software deliverable[5, chap. 2.1]. The goal of *build engineering* is to reliably perform build in the shortest possible time. The authors of Configuration Management Best Practices also emphasizes importance of the role that build engineers play in software development: „[...] *the build engineering team should consider themselves to be a service function with the development team as their primary customers.* [...] *As build engineers, we provide a service to support the development effort, but our primary goal is to help secure the assets of the firm that are built and released through the build engineering function."*[1, chap. 2]

Purpose of the software development is to deliver programs to customers. The build that is being delivered is called a release. Ian Somerville states that „Release management *involves making decisions on system release dates, preparing all information for distribution, and documenting each system release."*[2, chap. 25].

„The main goal of deployment *is to promote a release into production without any possible problem occurring."*[1, chap. 6] Moreover, it is important that in case of a problem, the release can be rolled back as quickly and easily as possible to ensure unhindered operation of the service[1, chap. 6].

3. Quality Assurance
4. In this context, *„artifact is the specification of a physical piece of information that is used or produced by a software development process, or by deployment and operation of a system"*[4, sec. 10.3.1]. It is not limited to source files, but it can include scripts, documents, or a mail message[4, sec. 10.3.1].

To sum up, all these areas of CM are overlapping and it is hard to categorize all actions that are often performed during the development, such as writing or generating documentation. Additionally, they often depend on each other. For example, to build a program, engineer must have access to the source code managed by an SCM tool. In the following text, several tools that facilitate configuration management are introduced. They cannot be usually classified into a single CM type. For example, Apache Maven can both build and release software artifacts.

2.2 Project Automation Tools

During software development, developers have to perform various tasks related to the configuration management, such as:

- Creating documentation
- Compilation
- Testing
- Verification
- Deployment

In the past, programmers had to execute these steps manually, which prevented them from focusing on creative tasks and problem solving. This often led to frustration and consequently to a decrease in developer productivity. Build automation tools originated as a solution to these problems. By letting the computer do the tedious and repeating work, programmers are able to work more effectively. The author of *Pragmatic Project Automation* states the following criteria of a good build system[5, chap. 2.1]:

- Complete – all data that the build system requires to perform a build must be available, so the build can be automated and self-contained.
- Repeatable – the build system must produce consistent results every time. In addition, it must be able to reproduce builds of previous releases. This requires that the build artifacts are stored in a version control system.
- Informative – developers must be able to receive useful information about the build process. Part of this information should be the results of automated tests.
- Schedulable – the build can be executed automatically, without human intervention and in regular intervals.

- Portable – developers may require that the project is built in a specific hardware and software environment. Build systems must be as platform-independent as possible to enable these builds.

Peter Smith states three additional requirements[6, chap. 1]:

- Correctness – the tool should perform the build according to our requirements, automatically making best decisions, such as choosing correct dependency version.
- Convenience – to fulfill its purpose to save developer time and effort, it must be as easy-to-use as possible.
- Performance – correspondingly to the previous requirement, the tool should be fast so it programmers do not need to wait in order to continue working.

2.2.1 Project automation types

The tools that enable project automation can be divided into three categories, based on how are they executed[5, chap. 1.2]:

- Commanded automation
- Scheduled automation
- Triggered automation

Commanded automation tools run on-demand and automatically perform steps that the programmer would otherwise do manually. Developer selects a high-level task, such as „run integration tests“, optionally providing parameters, and the tool will execute it by following instructions specified in a build file. GNU Make and Apache Maven belong to this category.

Scheduled automation is a practice of performing builds regularly. Most software companies perform *nightly builds*. „*The idea is that a batch process will compile and integrate the codebase every night when everybody goes home.*“[7, chap. 3] Each morning the developers have the latest development release ready for testing, information collected during the build and confidence that everything works as expected. Additionally, this feedback is available even if developers have forgot to use a commanded automation tool. On the other hand the authors of Continuous Delivery state that „*this is a step in the right direction, but it isn't very helpful when the team arrives the next morning only to find that the code didn't compile. The next day they make new changes – but are unable to verify if the system integrates until the next night.* [...] *In addition, this strategy is less than useful* *when you have a geographically dispersed team working on a common codebase from different time zones.*"[7, chap. 3]

Triggered automation is an improvement of the scheduled automation. The tool is waiting for some specified event to commence a build. Note that this is not limited to timer events. It is a very common practice to check a source code repository and wait for a programmer's commit. If a code change is detected, new build is scheduled. *This technique provides faster feedback to developers* and, in case of problems, the automation software can even contact the developer who broke the build. This is the basic idea of the *continuous integration* technique.

In this section we have provided theoretical introduction to project automation tools. In the next, we will explore a few commonly-used build tools and introduce Apache Maven and Jenkins CI.

2.3 GNU Make

GNU Make[5] was created by Dr. Stuart I. Feldman in 1977. Since then it has become one of the most famous and influential build tools[8]. It has introduced important concepts that were used to develop other build automation software, therefore it is useful to provide a quick overview.

Despite its age it is still widely used and is included in most Linux distributions so we can rely on manual pages[6] for a brief introduction: „*The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them.* [...] *make is not limited to programs. You can use it to describe any task where some files must be updated automatically from others whenever the others change.*"[9] Make accomplishes this goal by using a file with a description of the build process, called *makefile* which is written using a special-purpose scripting language. From a high-level point of view the makefile consists of rules[10]:

```
targets : prerequisites
    recipe # this is a comment,
           # rule steps must be tab-indented
```

Figure 2.1: GNU Make rules format.

5. `http://www.gnu.org/software/make`
6. They can be accessed on most UNIX-type systems using `man make` command. In addition, citation of an online version is provided.

Each target is an action that needs to be performed. Commonly, the name of the action is a file being updated. Prerequisites are rule names that have to be performed before the current rule, effectively defining a dependency graph. The rules are comprised of a sequence of steps, such as running a compiler, and are executed when the rule is invoked. To do this, user specifies the name of the rule and additional parameters as arguments to the `make` executable in the command line. The tool then resolves rule dependencies and executes the steps. It no target is specified, the first defined in the file is used.

The make tool and language provides more features than simple rule definition[10]:

- Variables
- Implicit rules – are predefined rules included in the tool, such as updating a `.o` file from a correspondingly named `.c` file using a `cc -c` (C compiler) command. These rules are applied automatically if make determines they are needed[10, sec. 10.1] and can be configured by modifying specific predefined variables.
- Functions – enable the definition of more complex rules. They provide control flow, string manipulation capabilities and file management.
- Inclusion of another makefile
- Comments

Despite being popular, make has been criticized for inconsistent language design resulting in a more difficult learning process and challenging debugging[6, chap. 6].

To sum up, GNU Make is a widespread build system providing the user with rich functionality using build files written in an imperative language.

2.4 Apache Ant

Ant[7] (Another Build Tool, formerly Another Neat Tool) is open source commanded automation utility inspired by GNU Make. While written in Java and primarily aimed to build software for the same platform, it can also be used for other types of projects. Analogously to Make, the build steps are defined in a `build.xml` file. Although the chosen language is based on XML, it uses the same concept of *targets*, which are comprised of *tasks*.

7. `http://ant.apache.org/`

These are again sequences of instructions, so the language can be categorized as procedural[6, chap. 7]. The tasks themselves are implemented as Java classes and the developers are encouraged to create their own in addition to the built-in ones, therefore tasks for most purposes are already available for other users[11].

Considering the clear inspiration by make, it is important to state the reasons that led to its creation as described in one of Ant's mailing list posts: *„The core functionality* [of Make] *is not optimized for any language in particular, and make often makes developers uncomfortable because it is unlike most languages and therefore carries its own learning curve.* [...] *The language of task abstraction for ant is Java. The core functionality implements the bare basics needed for Java-based projects, and the whole system is very appealing to developers of Java/Web-based projects because it works within the same framework (Java/XML) that those developers use daily.“*[12] In addition to convenience, Ant is able to manage project dependencies using integration with Apache Ivy[8][13] and because it is written in Java, it is portable.

While Ant is a powerful build tool, it still uses concepts of GNU Make and as a result inherits some of its problems. In the past, many projects lacked common approach to compilation, distribution, and web site generation and their build systems gradually become complex and unmaintainable [15, chap. 1.1.2]. Eventually, other approaches to project management led to development of Apache Maven.

2.5 Apache Maven

The official web page[9] contains a concise description of what Apache Maven is – *„a software project management and comprehension tool. Based on the concept of a project object model (POM), Maven can manage a project's build, reporting and documentation from a central piece of information.“* However, this definition is not very clear. As a result of a requirement (1 on page 18) for the implementation part of this thesis, a more comprehensive introduction to Maven and its core ideas is provided in the following sections.

8. `http://ant.apache.org/ivy`
9. `http://maven.apache.org/`

2.5.1 Basic principles

The most prominent idea that its authors incorporated into Maven is *convention over configuration* (CoC). It states that an application should use reasonable default values for its configuration options whenever possible. *„Without requiring unnecessary configuration, systems should 'just work'."* [14, chap. 1.2].

The best way to illustrate this principle is to use an example. Maven provides an option to generate the initial project structure from a predefined template, called an *archetype*. This is done using Maven Archetype Plugin, which can be used to both create and apply the template[16]. Maven is a command line tool, therefore, provided we have successfully installed and configured it, following command has to be executed to create a simple „Hello World" Java project in the current directory:

```
mvn archetype:generate
    -DarchetypeGroupId=org.apache.maven.archetypes
    -DarchetypeArtifactId=maven-archetype-quickstart
    -DgroupId=net.jsenko -DartifactId=hello-project
    -DinteractiveMode=false
```

Figure 2.2: Maven Archetype Plugin execution.

The command consists of the Maven executable name, `mvn`, followed by the plugin name, goal and configuration parameters. Usually Apache Maven must run in the context of some existing project. For some use cases, such as this, plugins may choose to allow to be executed without the `pom.xml` present. As a result, its name must be specified directly.

Plugin goal, `generate` is a name of the specific functionality. As mentioned, archetype plugin can also create new templates, which can be accomplished using `archetype:create-from-project`[17].

The parameters are prefixed with `-D` and are used to identify which template to use and to specify the coordinates of the resulting project. Additionally, the goal execution is interactive by default, but to avoid the need to confirm default choices, the last parameter is used. The ability of Apache Maven to create a working sample project using just a single command is a great example of Maven's adherence to the CoC principle.

Figure 2.3 shows the resulting project directory tree. This basic directory structure is the same for most Maven projects, therefore it is very easy for other developers to familiarize with the project and start contributing.

```
hello-project
|-- pom.xml
`-- src
    |-- main
    |   `-- java
    |       `-- net
    |           `-- jsenko
    |               `-- App.java
    `-- test
        `-- java
            `-- net
                `-- jsenko
                    `-- AppTest.java
```

Figure 2.3: Standard directory structure for Java-based Maven projects.

The sources are in the `src` directory and `net/jsenko` corresponds to the package name. Maven version of a build file, `pom.xml`, in located in the project's root directory and contains the *Project Object Model*.

2.5.2 Project Object Model

Project Object Model (POM) is an *„XML file that contains information about the project and configuration details used by Maven to build the project.“*[18] This is done in a declarative way, so the developers do not define instructions on *how* to perform a task, they express *what* the task is. Additionally, the POM contains *description* of the project in the sense that it contains comprehensive information, that can be used for all areas of configuration engineering – SCM, release engineering, deployment, even the list of project contributors and means to contact them. This notion of *conceptual model of a project* is explained by the authors of Maven – *"Maven is more than just a build tool, it is more than just an improvement on tools like make and Ant, it is a platform that encompasses a new semantics related to software projects and software development.*"[14, chap. 1.5]

Each Maven project and the artifact it produces has an *unique identifier*. This is a consequence of the release management features of Maven, that requires identifiable releases, such as dependency management and publishing of the released artifact. This identifier consists of five text strings, however only four are commonly used[19]:

- groupId – represents namespace in which the artifact resides. It is conventionally a reversed domain name that belongs to the organization or individual that produced the release.
- artifactId – meaningful name for the artifact in the group namespace. Commonly it is a compact version of the project name.
- version – to distinguish different releases of the same project. Usually it is a one or more numbers concatenated by a period, but additional information, such as development stage (alpha, beta, release candidate) or nightly build number can be also added.
- packaging – this piece of information represent a form in which the artifact is stored. The default value of this attribute is „jar" (java archive).

This identifier is used to specify in the `pom.xml` that the project generated by the archetype plugin depends on `junit:junit:3.8.1`[10], which is a framework for testing. Moreover it is used to identify plugins. The full identifier of the archetype plugin is `org.apache.maven.plugins:maven-archetype-plugin:2.2`[11] but in this case, an alias is defined by default, so full coordinates are not required.

The identificator is only one of the prerequisites for the automatic *dependency management*. The second is a storage of artifacts accessible to Maven called a *repository*[12]. This concept is very similar to software repositories used to distribute software packages in Linux. There is a main public repository, called Maven Central[13], which is by default available to all projects. The ability to define additional repository locations has important positive consequences for *release management* - development team can store internal builds in a private repository and make releases available to customers via a public repository.

Finally, the POM usually contains property definitions. They are the most important tool for making the project build customizable and flexible[14], and can be defined in several ways:

- automatically by Maven – these predefined properties include project root directory (where `pom.xml` is located), and a place where data

10. The parts of the identifier are connected by `:` when representing it as a simple string. In this format – `groupId:artifactId:version`.
11. Latest version in the time of writing.
12. Not to confuse with SCM repository.
13. `http://search.maven.org`
14. The other being profiles – `http://maven.apache.org/guides/introduction/introduction-to-profiles.html`

generated during the build are stored.

- via command line interface (CLI) – used in the example.

2.5.3 Build Lifecycle

In figure 2.2 on page 10, the `generate` goal of the Archetype plugin is executed directly, and it represents a single task. On the other hand, build process involves execution of many different steps. In case of GNU Make or Ant, these steps are grouped together to form high-level targets defined and named by the programmer, based on the specific project requirements. However, most projects share *common actions* that developers want to perform (as mentioned in 2.2 on page 5). Consequently, the authors of Maven introduced concept of a *build lifecycle*.

„*A* build lifecycle *is an organized sequence of phases that exist to give order to a set of goals.*"[14, chap. 10.1] Each phase represents an action that should be performed during the build process. The phases are ordered and executed sequentially. For example, in order to run the *test* phase, previous phases, including *validate*, *generate-resources* and *compile* have to be executed first. „*Lifecycle phases are intentionally vague,* [...] *and they may mean different things to different projects.*"[14] As a result, the lifecycle defines a process that has to be followed in order to accomplish some high-level objective. The previously mentioned phases belong to the *default* (or *build*) lifecycle, but there are others available[15].

If user wants to execute a phase, Apache Maven by itself does not know what to do. The functionality is provided by the plugins that can assign their goals to some specific lifecycle phase. For many of these phases, Maven provides default plugins that work for most use cases and require minimal configuration. As a result, in order to create a „jar" file of the example `hello-project`, the user just has to execute `mvn package` command. During the appropriate phases, `maven-compiler-plugin` and `maven-jar-plugin` will automatically run and perform the required tasks.

2.6 Jenkins CI

Jenkins CI[16] is an open source continuous integration server software, written in Java and with built-in support for maven-based projects. The project

15. For more details, see `http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html#Lifecycle_Reference`
16. `http://jenkins-ci.org`

is in active development and has a large number of contributors[25], including it's original author, Kohsuke Kawaguchi. Among the most important features are easy installation and configuration, intuitive user interface, powerful machine interface, extensibility using over 700 official plugins[27], and ability to perform distributed builds.

2.6.1 Jenkins Job

Job represents a task that can be executed and managed by the CI server in response to some event, usually performing an action on a project's source code. This may include *„running your integration tests, measuring code coverage or code quality metrics, generating technical documentation, or even deploying your application to a web server. A real project usually requires many separate but related build jobs.*"[26, chap. 2] There are several basic types of jobs available:

- Free-style project
- Maven 2/3 project
- Multi-configuration project
- Monitor external job

The first is a general-purpose job type that enables the user to select from almost all available configuration options and thus provides the most flexibility.

The second type of job is specifically designed to build maven-based projects. Because the Jenkins understands the Project Object Model, it can provide maven-specific build options and retrieve build results and other information.

Multi-configuration project provides the feature of "matrix builds". It is very common that a project needs to be tested under various conditions, such as multiple versions of the JDK, different database versions, operating systems and so on. Without the matrix build, user has to create new job for each possible combination of the parameters, but with otherwise identical configuration. This can result in a large number of jobs which quickly become very difficult to manage. This can be solved by adding a build matrix to a job. It consists of axes which are parameters with corresponding possible values. These parameters that can be used as variables in the job configuration, and Jenkins will take care of running the job for each of the possible parameter combinations.

Monitor external job feature enables Jenkins to monitor a process that was not originally executed from Jenkins. For example, this can be a legacy bash script that executes nightly builds.

2.6.2 Job configuration

Before a Jenkins job can be executed, user must provide it with some configuration options. In the following list we provide general types of settings:

- General options
- Parameters
- Source Code Management
- Triggers
- Build steps
- Post build actions

General options include the name of the job and its description.

Parameters provide a way to pass information to the job before it starts and are subsequently available as named variables. Among default types of parameters are string, boolean, choice and a file.

„In its most basic role, a Continuous Integration server monitors your version control system, and checks out the latest changes as they occur. The server then compiles and tests the most recent version of the code."[26, chap. 5] As a result, Jenkins must be able to work with various SCM tools. By default, CVS[17], Subversion[18] and git are available.

Triggers are events that cause the job to be scheduled for execution. There are several basic types of events – the job may run at periodical intervals, wait for another one to finish, or check the project's source code repository for changes. Moreover, thanks to the plugin ecosystem, new triggers can be easily created, such as an ability to perform builds using interactive IRC[19] bot[20].

Builds steps do the actual work. They can be used to execute a shell command, invoke a build tool or trigger another build.

Main reason for using the continuous integration technique is to receive feedback as soon as a problem occurs. This is provided by the post-build

17. `http://www.nongnu.org/cvs`
18. `http://subversion.apache.org`
19. Internet Relay Chat
20. `https://wiki.jenkins-ci.org/display/JENKINS/IRC+Plugin`

actions, which can range from a simple email message to the developer who broke the build or a more aggressive physical response[28].

2.7 Git

Git[21] is a free and open source distributed version control system (VCS)[22] tool[20]. It is used to manage most projects at Red Hat, therefore the pre-commit testing solution must be able to interact with it. In the following text, an introduction to the important concepts of git are presented.

Git was created by Linus Torvalds in 2005 after a decision to stop using a proprietary source code management system, for the Linux kernel project. Torvalds started the development of Git because none of the existing tools at the time satisfied his requirements for a good version control system[21]:

- Distributed repositories
- Effective branching
- Good performance
- Reliability

Git is a *distributed version control system*. This is in contrast to a more traditional approach that uses a central server to keep track of the changes, which clients access over a network. This architecture has several advantages. Author of Pro Git states that *„everyone knows to a certain degree what everyone else on the project is doing. Administrators have fine-grained control over who can do what; and it's far easier to administer a* [centralized VCS] *than it is to deal with local databases on every client.*"[24, chap. 1.1.2] On the other hand, it is not suitable for large open-source projects because they are inherently decentralized. Most contributors do not know each other and they might be untrustworthy. Moreover, the central server must be able to handle every collaborator and as a result represents a single point of failure.

In case of the distributed VCSs, every repository contains all data about the underlying project. There is no central place and each repository acts as a peer. The owners can subsequently transfer changes to and from other *remote repositories* (performing *push* and *pull*), provided they have been given access. Moreover, the distributed nature of git allow many different

21. `http://git-scm.com`
22. *„Version control combines procedures and tools to manage different versions of configuration objects that are created during the software process.*"[22, chap. 22.3.2]. This term is used interchangeably with SCM in practice.[23]

workflows[24, chap. 5.1], which allows for various organizational structures.

Another of Linus's requirements, *performance* and *reliability* comes from the amount of data that users contribute to the Linux Kernel Project, so git was designed to be extremely fast.

Generally, VCS repository contains revisions, representing a point in the history of the project, and additional metadata. Because a new revision is created by making a change upon some existing state and then *committing* (saving) the changes, the revisions (or *commits*) constitute a directed graph structure. Each commit points to the previous state, usually represented by a single commit, but in some cases, multiple lines of development (*branches*) may be *merged* together. In case of traditional version control systems, such as the Apache Subversion, revisions are stored as *differences* containing only changes made since the previous commit. On the other hand, git stores *snapshots* of the project at the time of commit in a very efficient key-value data store[24, chap. 1.3.1]. Keys are generated by applying SHA-1[23] on the change and the related metadata, including a pointer to the previous commit(s). This hash is then used as an unique revision identifier. As a result of the chaining, user just has to know the hash of the latest commit to verify integrity of the entire project history.

Open source projects are by nature very dynamic, at any point a group of developers may be working on a separate feature that may or may not be eventually accepted. This places requirements for *easy branching*. Because of the way revisions are stored, branches in git are just named pointers to the top commit. This results in a very inexpensive and, more importantly, local branching and merging. Additionally, in order to work with changes in the remote repositories, there is a special type of branches – *„Remote branches are references to the state of branches on your remote repositories. They're local branches that you can't move; they're moved automatically whenever you do any network communication.*"[24, chap. 3.5]

23. Secure Hash Algorithm version 1

3 Analysis and Design

The goal of this chapter is to identify and analyze requirements for a solution to the problem presented in the *Introduction* and choose the best way to implement it.

3.1 Requirements

There are two types of requirements. „Functional requirements *are statements of services the system should provide* [...] *and how the system should behave in particular situations. In some cases, the functional requirements may also explicitly state what the system should not do.*"[2, chap. 4.1] *Non-functional requirements* impose constraints on the *overall* functionality of the system[2, 4.1]. Meticulously composed list of requirements is essential to avoid problems during implementation and evolution of the software.

Following list contains initial functional requirements for the solution:

1. The tool must support Maven projects.
2. It should be used locally from the developer's computer.
3. The tool should be invoked by an user from the command line.
4. Source code of the target project[1] must be stored in a git repository.
5. The tool must gather local code changes not yet pushed to a remote repository.
6. Project with these local changes must be built on a remote machine.
7. Application must be able to retrieve results of the build and present them to the user.
8. User must provide necessary configuration options for the application.
9. In order to archieve speed and efficiency, the program will reuse existing resources when possible.
10. The tool must provide a way to clean up unneeded data and resources.

Additionally, the program is designed to make testing of small local changes in the code faster so one of the most important non-functional requirements is that it must be *easy to use and reasonably fast*. Consequently, the author

1. Project on which the tool is used.

decided to follow the convention over configuration principle. In addition, users must have access to help information and documentation and the resulting solution must be released under an open source license, which is in accordance with Red Hat's principles and values.

3.2 Existing solutions

There are several existing options for doing pre-commit testing and all use a continuous integration server to perform the build.

Team City[2], a CI software from JetBrains, can commit the changes after they were succesfully tested[32]. However, it requires a support from an IDE[3] to communicate with the server and send the code to be tested. This solution is inacceptable for two reasons:

- Requires IDE support
- It is specific to Team City, while at RedHat, Jenkins is used.

As a result, I have investigated an existence of similar solution for Jenkins. There is a article on Jenkins Wiki that mentions the Team City functionality and considers possible solutions for Jenkins[29]. It was created in december 2009, however no final method is currently[4] described. In addition, there is an unresolved item in Jenkins's issue tracker and the related comments express that there is a demand for such feature[30]. However, author of the latest comment (september 2013) states that he implemented a partial solution in a form of a Jenkins plugin called the *Pretested Integration Plugin*. It works by implementing one of the approaches described in the wiki page. It assumes there exists a special branch for each developer, where they commit their code, and an integration branch where the code is merged by Jenkins after it is verified[31]. This approach requires that developers agree on a special branching model, and prevents the untested changes to appear only in the integration branch so it is not very user friendly. Moreover, the current version does not support git.

Another approach that I have found combines git, Jenkins and Gerrit[5], a code review tool[33]. Gerrit works as a wrapper around a git repository to which the developers push their changes. These changes can be optionally submitted for code review, which means that they must be aproved before

2. `http://www.jetbrains.com/teamcity`
3. Integrated Development Environment
4. December 2013
5. `http://code.google.com/p/gerrit`

they can be accepted into the repository. The reviewer does not have to be a human. There is a Jenkins plugin available, called Gerrit Trigger that can execute a build and verify the submitted change, effectively performing a pre-commit testing[34]. While this existing solution works, it is not always easy to use. If a change is approved and other developer has updated the same parts of the code, merge conflict arise that require user to manually resolve them[35]. Moreover, this method requires additional configuration before it can be used which takes time.

Consequently, I have decided to implement a tool that does not require any additional setup besides a functional Jenkins server and works with plain git repositories. It is inspired by the Team City solution, but does not depend on the IDE and is in form of a simple command line tool. As a result of it using Jenkins CI, following requirements have been added:

11. The local changes must be transfered to a remote Jenkins server.
12. The resulting code must be built and tested on Jenkins.

3.3 Maven Plugin Goals

The requirements do not specify how the tool should be implemented, only that it must support Maven (1) and be executed locally (2), preferably using CLI (3). In the previous chapter it was established that core Maven is in fact a lightweight framework for executing plugins, which do the actual build tasks. They are easy to implement and use, therefore it is beneficial to develop the application as a plugin for Apache Maven.

The requirements contain two basic tasks, which can be assigned to the following corresponding goals:

- `run` represents the main tool functionality.
- `clean` to fulfill requirement 10. The purpose of the plugin is to quickly test features during development, so it is reasonable to assume that it is not desirable to keep the Jenkins jobs and test results for a long time.

3.4 Local Code Changes

In this section, we analyze how the plugin fits in with a developer's usage of git and examine the notion of *local code changes* that the tool must be able to work with according to the requirement 5.

There are several model git workflows[36], but most of them incorporate following steps:

1. A copy of code of the target project is available in one or more shared *remote repositories*.
2. Contributor creates a *local repository* by cloning the remote to work with the code.
3. User makes changes on top of the *working branch*. This can be one of the existing branches, or he may create a new local *feature branch*.
4. User transfers the changes back to the shared repository after (a portion of) the work is done.

For our purposes, the *local changes* are represented by a series of commits that are on the top of a working branch, but are not present in the remote repository. In order to create an algorithm to identify these commits, it is useful to visualize the aforementioned concepts:

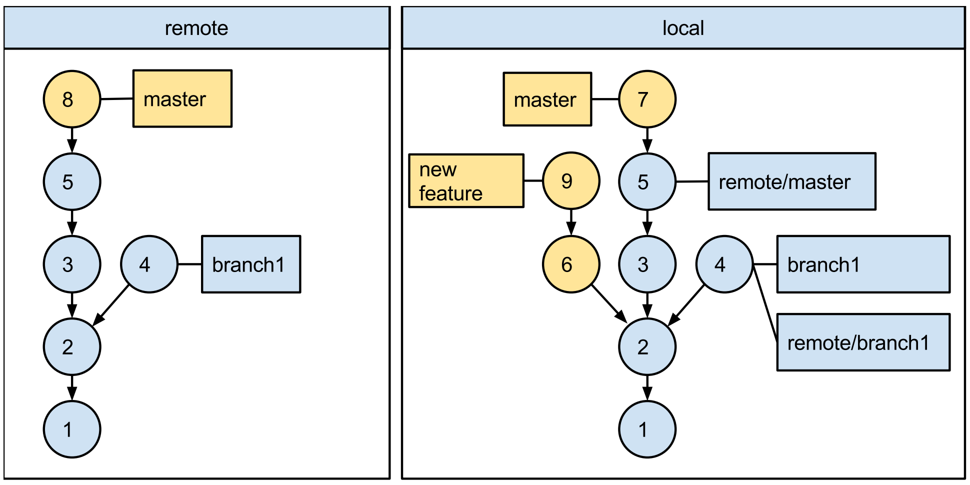

Figure 3.1: General git workflow.

In the figure we see a graph of commits in the *remote* an the *local* repository. Each commit identifier also represents order in which the commits have been created (starting with 1). Blue color represent commits that were present in both repositories at the moment of cloning and orange color denotes consequent changes:

1. Developer has an idea for a new feature and creates `new feature` branch, and makes first commit (6).

2. Then he decides to make some unrelated changes on top of the `master` branch (7).
3. Meanwhile, other developer updates the `master` branch in the remote repository (8). The local tracking branch `remote/master` points to the old commit (local repository does not know about it yet).
4. Developer continues work on the new idea (9).

From this example scenario, several observations can be made:

Firstly, there may be multiple sets of changes in the local repository that the developer might want to test. Every feature has its own branch (`new feature`, `master`) so the developer just needs to *specify the branch containing the changes* to be tested. User can specify the working branch directly, but is is very probable that the changes are in the active (checked out) branch. Git provides a special `HEAD` reference, that points to this branch. This is great way for the plugin to determine default configuration and follow convention over configuration principle.

Secondly, the remote repository can be updated by other users but as mentioned, the goal of the plugin is not to merge the changes, so it does not present a problem.

Thirdly, the remote repository contains commits (2 and 5) on top of which the local changes (6, 9 and 7, respectively) were made. To test code in the local `new feature` branch for example, the problem of transfering modified source code to Jenkins can be easily solved by letting the Jenkins *download the unchanged original code from the remote repository (commits, 1 and 2) and then only the changes (commits 6 and 9) need to be transfered and applied to get the entire source code for testing*. To do this, the plugin must be able to identify the *base commit*, in this case 2, which is the latest commit available in both `new feature` branch and remote repository. To determine if the commit exists in a specific remote repository, it is sufficient that it is available from one of the remote tracking branches (either `remote/master` or `remote/branch1`).

As a result, the following informal algorithm to find a *base commit* is provided:

1. Input is the name of the working branch and a list of tracking branches for some specific remote repository.
2. Create a set `S` of all commits in the working branch.
3. For each of these branches, traverse the commit graph until a commit that is also present in `S` is found. Add it to a `base commit candidates` list and continue to the next branch.

4. Find the latest commit in this list to minimize the size of changes that must be tranfered. This is the result.
5. Output is the ID of the base commit (2 in the example).

This way we have identified commits that contain the local changes.

3.5 Patch file

Given a of sequence commits, the changes they represent can be conveniently stored using a *patch file*. In general, it is used to represent a difference between two (sets of) files. In other words, given two files, or two versions of a file, it constains instructions that can be used to transform one version of a file into the other. Git supports both generation and application of patch files. Once the base commit is identified, this provides a convenient way to transfer the changes to the Jenkins server, as described in requirement 11 on page 20.

3.6 Jenkins Job Configuration

In accordance with the requirement 12, the tool must be able to create and execute Jenkins jobs. In section 2.6.1, we have described four basic types of Jobs that Jenkins supports. The plugin is designed to test maven-based projects, so the *Maven 2/3 project* job type is the most suitable. In order to determine how the job will be configured it is useful to list how its execution on Jenkins should look like:

- Checkout *base commit* from the *remote repository.*
- Apply *patch file* to obtain local version of the code.
- Execute user-defined pre-build steps if available.
- Execute maven *goals* provided by the user.
- Run optional post-build steps.

Specific details are provided in the *Implementation and Tools* chapter.

3.7 Retrieving results

After the job execution, the plugin must retrieve the results and present them to the user (7). Jenkins provides following basic types of information about the build:

- Console output of the build process
- Simple test summary (number of passed/failed tests)
- Individual tests reports

3.8 Remote Jenkins API

The plugin must be able to communicate with Jenkins in order to create, execute and delete jobs. Moreover, various information, including list of jobs, list of builds and their results must be accessed. There are currently two options for interacting with Jenkins remotely (without the using web based interface):

- Jenkins CLI tool
- Jenkins RESTful[6] API[7]

Jenkins CLI tool comes as a part of Jenkins distribution and in a form of a standalone java „jar“ application [8]. It has a basic set of features and is suitable for small utilities because it is easily invoked from shell scripts.

Jenkins RESTful API offers comprehensive access to Jenkins features for machine clients. Every important page of the web interface that represents some Jenkins object or functionality has a corresponding URL where the resource can be accessed in a RESTful way.

A decision to use RESTful interface to communicate with Jenkins was made because it provides all required features and the alternative is a standalone application intended to be executed from the command line.

In the next section, a brief introduction to the REST is provided so we can consequently present the Jenkins API itself.

3.9 Representational State Transfer

Representational State Transfer (REST) is an architectural style for distributed systems. It is a set of constraints that are imposed on design of the these systems. It was first introduced and described by Roy Fielding in his doctoral dissertation[37]. It is now widely used to design APIs for web services. The reason for this is that although it is independent of the underlying protocol, it can be easily used over HTTP[9], to the development of which Fielding also

6. Conforming to the *Representational State Transfer* architectural style.
7. Application Programming Interface
8. `https://wiki.jenkins-ci.org/display/JENKINS/Jenkins+CLI`
9. Hypertext Transfer Protocol

contributed. It is stateless, which correspond to the stateless nature of HTTP. Additionally, it is used for client-server communication, where server holds a resource accessed by a client.

Resource is a key REST concept. It is an object that the service exposes via the API. To be able to manipulate this object, each resource must have an unique identifier, such as the URL in case of a web resource. Using the identifier, the server presents the client with a representation of the resource. It can take any form, however a structured data, such as an XML document, is used when machine readability is required.

The client then invokes operations on the resource. In case of the HTTP, accessing the representation is a matter of invoking a HTTP GET request on the resource URL. Basic operations that can be executed on data stored in a database are known under the CRUD acronym: create, read (retrieve), update and delete. These operations have their counterparts as HTTP methods which are used to work with web-based RESTful resources:

Operation	HTTP	SQL
Create	POST	CREATE
Read	GET	SELECT
Update	PUT	UPDATE
Delete	DELETE	DELETE

Table 3.1: Operations on RESTful resources and their HTTP and SQL counterparts.

There are also other methods that can be used for specialized cases, such as PATCH used for partial updates[38], but these cases can be solved by good API design - the representation of a resource must have adequate granularity (details about book author should not be part of the book representation).

Finally, in case of web-based resources, only GET and POST methods are often used, because the delete operation, for example, can be simulated by sending a GET (or POST) request with a special parameter (e.g. `...url/to/a/book/42?action=delete`).

3.10 Jenkins RESTful API

Jenkins provides a simple way to discover URL of a REST resource and access documentation about it. Each page of the Jenkins web interface can be used to access to a particular object or feature. For example, the dashboard, which provides general overview and list of the Jobs, is a page that is located at the root of the Jenkins server. From this address we can get the related documentation by concatenating "/api" to the URL. This is available for all resource access points. It states that there are three types of representation available for each resource:

- *Python*[10] representation consists of a python code containing information in a form of python data structures.
- *JSON*[11] is a lightweight data-interchange format inspired by JavaScript.
- *XML* is a popular markup language that can be used to represent information in a structured way that is suitable for machine-processing, while being relatively human-readable.

Finally, I have decided to use the third option because XML documents .

3.11 Job Reuse

We can assume that the plugin will be often invoked multiple times for a specific branch of the project, after the developer makes another code change he wants to test. Therefore it is reasonable to support reuse of Jenkins jobs, provided that the plugin is able to check that the job configuration has not changed.

To archieve this, we first consider situations that do not require creation of another job:

- User commits additional changes locally, resulting in a different patch file. However, it can be provided as build parameter and configuration stays the same.
- Invocation of different Maven goals. This can be parametrized as well.

On the other hand, job should not be reused if:

- Some part of the configuration that cannot be parametrized changes. User can add a new parameter or entire build step.
- User invokes the plugin for a different working branch. This results in a different patch and a base commit, which can be sent as a build argument, but it would result in unrelated builds grouped in the same job.

10. `http://www.python.org`
11. `http://www.json.org`

As a result, exactly one job for each local working branch is a good compromise and its default name can be the same as the name of the branch. However the job with that name might already exist, created either by the previous plugin invocation or by some unrelated user for completely different purposes. Therefore, the tool must determine if the reuse is possible, and if not, permit the user to provide alternative job name.

4 Implementation and Tools

The Jenkins pre-commit test maven plugin implementation consists of several components:

- *Maven Mojos* – represent Maven goals and are responsible for providing main functionality. These are `RunMojo` and `CleanMojo` classes.
- *Git API* – is used to interact with git repository of the target project. This task is implemented in the `GitTools` class.
- *Job Configurator* – consists of several classes responsible for generating Jenkins job configuration file.
- *Jenkins RESTful Client* – is used to communicate with Jenkins[1] via its RESTful interface.
- *Data storage* – is responsible for saving several pieces of data for each job to facilitate easy job reuse.
- *Result processors* – are a collection of classes that retrieve various information from Jenkins after the build is finished and present them to the user.

In this chapter, implementation details of these parts are described.

4.1 GitTools

Git stores data in a `.git` directory located in the repository root. `GitTools` must have a reference to this file, but Maven only provides location of the `pom.xml` file, which is not necessary located there. As a result, constructor of this class is private, and instances are created by a factory method `lookup` instead. It will search for the `.git` not only in the specified directory, but also in its parents.

User provides a reference to the working branch by specifying its name as a string argument. The `getRef` method is used to parse this string and return its object representation (containing an identifier of the tip commit), which is then used to work with the branch in other methods.

Another argument that is provided by the user is name of the remote repository to be used in the base commit algorithm. This repository contains project code on top of which the patch with the local changes is applied. Jenkins must know its URL so it can download the code before it can

1. Current version of Jenkins CI is 1.544, but the plugin should work for older versions.

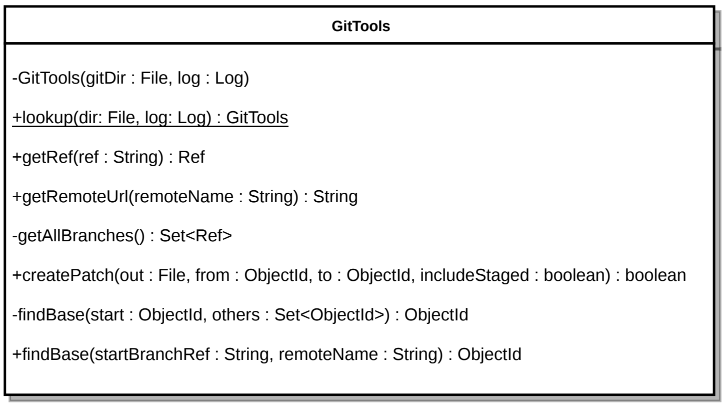
GitTools
-GitTools(gitDir : File, log : Log)
+lookup(dir: File, log: Log) : GitTools
+getRef(ref : String) : Ref
+getRemoteUrl(remoteName : String) : String
-getAllBranches() : Set<Ref>
+createPatch(out : File, from : ObjectId, to : ObjectId, includeStaged : boolean) : boolean
-findBase(start : ObjectId, others : Set<ObjectId>) : ObjectId
+findBase(startBranchRef : String, remoteName : String) : ObjectId

Figure 4.1: GitTools class diagram.

be patched. Method `getRemoteUrl` is used to retrieve this URL so it can be used to properly configure the Jenkins job.

The algorithm itself is implemented in the private `findBase(ObjectId, Set<ObjectId>): ObjectId` method. It takes a commit id of the working branch tip and a set of commits on top of the remote branches. The algorithm then finds suitable base commit and returns its ID. This method is private, because a wrapper method that takes string arguments received from the user, `findBase(String, String): ObjectId` is used instead. Internally, it uses the previously mentioned method, `getRef` and a private `getAllBranches` to parse argument strings and call the underlying method.

Finally, the patch is created using `createPatch` function. It takes the base commit and a commit on top of the working branch to create the patch. The last argument enables the user to test changes even *before committing locally*. If it is set to true, changes *staged*[2] for commit are also included in the patch.

These functions are implemented using JGit[3], a git client written entirely in Java. It is lightweight and provides all services that the application needs:

- Working with references.
- Determining URL of remote repositories.
- Ability to *iterate over commits* in selected branches.

2. This is a git feature. User must first select changes to be committed by adding them to an intermediate staging area (also called „index“).
3. `http://www.eclipse.org/jgit`

- performing a `git diff` analog to create a patch file.

4.2 Job Configurator

Jenkins uses XML files to store job configuration. To create a job using the RESTful interface, client must provide this configuration file using HTTP POST method. Therefore, the plugin must be able to generate such file from parameters provided by the user. This is the task of *Job Configurator* component.

Job Model

To create this configuration, Jenkins pre-commit test maven plugin must work with a representation of the Jenkins job.

Core part of the Jenkins architecture are model classes, located in the `hudson.model` package[4]. They represent core concepts such as Project, Job, Build or Result and contain their state. Most of them have an associated URL and are viewable via HTML web interface are exposed via RESTful API. They are serialized into XML using XStream[5], a tool that enables a conversion of an object graph into a human-readable XML document. It used in Jenkins to persist the state of these classes and generate XML representation for the RESTful API.

Similar solution is used to generate this configuration by the plugin. Job is represented by a collection of model classes located in `net.jsenko.jpct.configurator.model` package.

`JobModel` class represents a maven-based job. In addition to the `name`, `description` and `goal` fields, it contains a list of `ParameterModel` which enables the plugin to define job parameters. Additionally, `GitModel` is used to configure the job to retrieve base commit from the correct remote repository. It contains values computed by `GitTools`. Finally, `BuildStepModel` can be used to define additional build steps either before or after the main maven goal is executed. This is important, because one of the „before“ build steps is tasked with applying the patch.

JobConfigurator

`JobConfigurator` provides methods that work with the model and implement its conversion to the XML file. Because the `JobModel` has many

4. `http://javadoc.jenkins-ci.org/hudson/model/package-summary.html`
5. `http://xstream.codehaus.org`

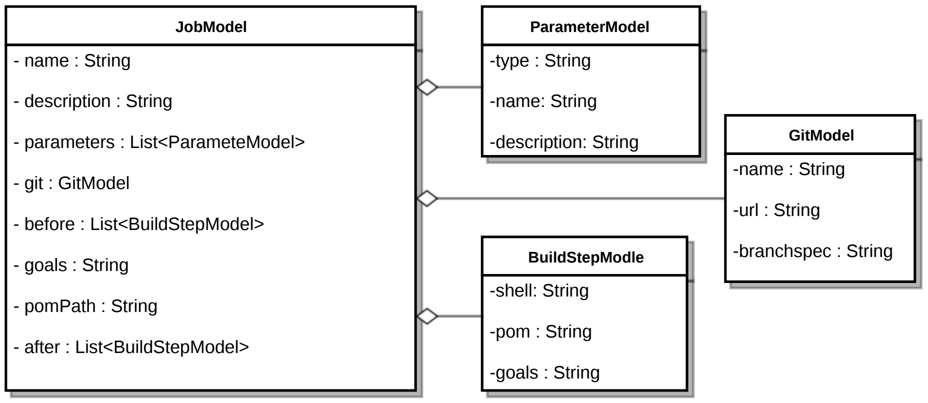

Figure 4.2: Job Model classes, fields only (getters, setters, `hashCode` and `toString` are omitted).

fields, it is not convenient to set them directly in the `RunMojo` and then pass it to `JobConfigurator` to be converted. As a result, a builder design pattern is used to provide configuration received from the user or provided by other components. This includes the name, description, git remote URL and the builder will set the remaining configuration options. Note that the builder does not have methods to set the maven goals or patch file. This is because in order for the job to be as reusable as possible, these values are provided as parameters when the job is executed on Jenkins. Therefore, the builder is also responsible for automatically adding these parameters to the model:

- `path` – file parameter containing the changes.
- `commitID` – ID of the base commit. Because this value might change, it it provided as a parameter.
- `nonce` – pseudorandom string to uniquely identify the build so we can retrieve the correct results.
- `goals` – maven goals to be executed.

In addition, the builder adds another `BuildStepModel` to the `JobModel::before` list. It is responsible for applying the patch using following shell commands:

- `git reset -hard` – Jenkins keeps sources of the project being built in a *workspace* directory. If multiple builds are executed and the git settings have not changed (`commitID` stays the same) its content is reused. Therefore the changes made by previous patch applications are preserved. This may cause another patching to fail. As a result, this command resets the changes made by the previous patching.

- `git apply $patch` – apply the path file given as a parameter. The location of the file is stored in the `$patch` variable.

Note that the `getBuilder(JobModel, Log) : Builder` method already takes a `JobModel` instance as an argument. This is because user can set complex object properties by specifying them in `pom.xml` as a part of plugin configuration. Maven is then able to create instances of these objects with the specified data and provide them to the plugin as another argument. Therefore, in addition to the `-Dgoals` CLI property, user can add custom build steps and other settings via POM. The resulting model objects are used as template by the builder and are combined with other configuration.

After the job model is set up, `createJobConfig(File) : boolean` is called to marshal the data into an XML file. This serialization is also implemented using XStream. The result must have a specific format to be understood by Jenkins , which can not be produced from the `JobModel` without additional configuration. To solve these types of issues, XStream provides custom serialization strategies using *converters*. As a result, the `net.jsenko.jpct.configurator.converter` package contains converters that transform the model classes into the XML with required format.

4.3 Jenkins RESTful API Client

The Jenkins pre-commit test maven plugin communicates with the Jenkins server using a Jenkins RESTful Client. It a collection of interfaces and their implementations located in the `net.jsenko.jpct.Jenkins.client` package and developed specifically for the use in this plugin.

Most of them represent a RESTful resource and define methods that work with them. The operations are implemented using Jersey by sending HTTP requests and processing responses. *„Jersey RESTful Web Services framework is open source, production quality, framework for developing RESTful Web Services in Java that provides support for JAX-RS APIs and serves as a JAX-RS (JSR 311 & JSR 339) Reference Implementation.“*[6]. In

6. `http://jersey.java.net`

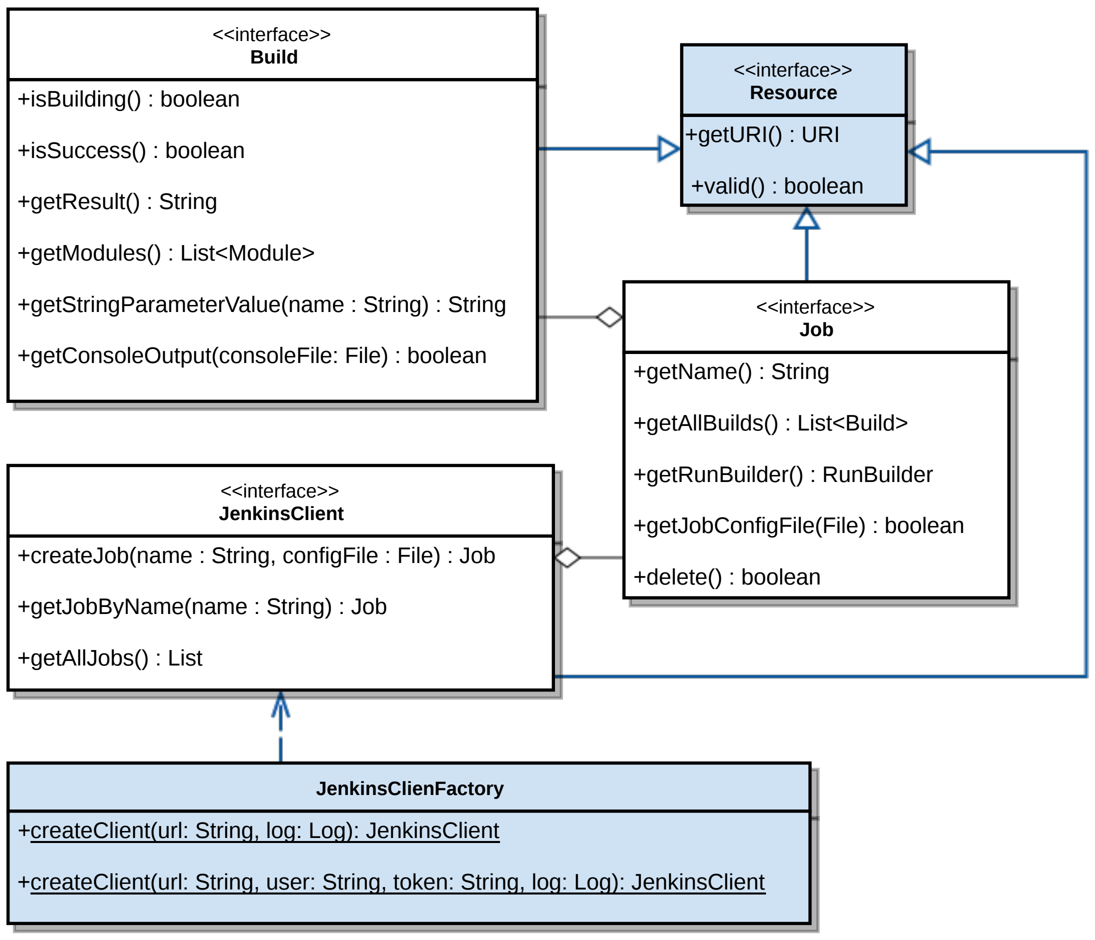
<<interface>>
Build
+isBuilding() : boolean
+isSuccess() : boolean
+getResult() : String
+getModules() : List<Module>
+getStringParameterValue(name : String) : String
+getConsoleOutput(consoleFile: File) : boolean
<<interface>>
Resource
+getURI() : URI
+valid() : boolean
<<interface>>
Job
+getName() : String
+getAllBuilds() : List<Build>
+getRunBuilder() : RunBuilder
+getJobConfigFile(File) : boolean
+delete() : boolean
<<interface>>
JenkinsClient
+createJob(name : String, configFile : File) : Job
+getJobByName(name : String) : Job
+getAllJobs() : List
JenkinsClienFactory
+createClient(url: String, log: Log): JenkinsClient
+createClient(url: String, user: String, token: String, log: Log): JenkinsClient

Figure 4.3: Several of Jenkins RESTful Client interfaces (some methods and classes omitted for brevity).

addition to the server-side part of the framework, Jersey provides a „*fluent Java based API for communication with RESTful Web services.*“[39] It is simple to use yet powerful, so it was chosen to implement this client. Common methods are extracted to a generic `Resource` interface. It contains a method to get the resource URI and an operation to check that the class represents a valid (accessible) resource. The implementation of this interface, `AbstractResource` contains a reference to `WebResource` class which is a part of Jersey and enables invocation of HTTP methods on it. There are also several *protected* utility methods, most important being `apiGetRequest(xpath: String, depth: int, wrapper: String, type: Class<T>): T`. Jenkins supports filtering of the resource representation by executing an XPath expression provided as an URL parameter. Another argument, `depth` is also a Jenkins feature, enabling user to limit the amount of provided information. Finally, `type` is passed to the Jersey which will attempt to convert the receive data and return an object of the requested type. This is usually the `java.lang.String` class in order to read, or the `java.io.File` to download the resource. However, it can also be used to unmarshal the XML into any suitable object using JAXB[7].

The other interfaces form a hierarchy, a consequence of relationships between the resources. For example, `Build` represent an execution of a specific `Job` . The root of this structure is a `JenkinsClient` interface which acts as an entry point representing the entire Jenkins server. Its implementation instance is provided by one of two possible methods in a `JenkinsClientFactory` class, depending on whether authentication is required to access the server.

The most important task for the client is to create and execute a new job, so description other functions is omitted. The creation is performed by invoking the `JenkinsClient::createJob` method. The XML configuration file is supplied as the second argument. To invoke the job, a builder design pattern is used for conveniently providing job arguments. As an example, following code is a snippet from the `RunMojo`:

```
boolean result = job.getRunBuilder()
                .setParameter("commitID", patchCommitId)
                .setParameter("nonce", nonce)
                .setParameter("goals", getGoals())
                .setParameter("patch", patchFile)
                .run();
```

Figure 4.4: Job execution example using a convenient fluent interface.

4.4 Result processors

As described in the analysis section 3.7, *Retrieving results*, there are several pieces of information available about the build that has been executed. To fulfill the requirements, each of them has an associated *result processor*, a class responsible for their retrieval and presentation. After a job is

7. Java Architecture for XML Binding, `http://www.oracle.com/technetwork/articles/javase/index-140168.html`

started using the Jenkins RESTful client, `RunMojo` retrieves the associated build with the correct nonce value. Subsequently, the client is used to periodically poll Jenkins until the build finishes, marking the plugin execution successful or not according to the build result.

Each result processor extends an abstract class, `net.jsenko.jpct.result.ResultProcessor` which defines methods called in the `RunMojo` during the build execution, ensuring that the processors have common interface so new ones can be easily created. The `start(Build, Config, Log)` method is called once at the beginning of the build so the processor can be properly initialized and has access to the available information. Method `run()` is invoked periodically and `finish()` once the building is done.

Currently the Jenkins pre-commit test maven plugin contains three implementations of the `ResultProcessor`, but more can be added in the future:

- `ProgressRP` – print „`.`“ character regularly to indicate that the plugin is waiting. Uses `run()`.
- `TestSummaryRP` – display test statistics such as the number of passed and failed tests for each maven module. Uses `finish()`.
- `ConsoleOutputFileRP` – Jenkins provides an option to view console output of the build process. This processor saves it to a file for detailed examination in case of problems. This happens in `finish()` method.

4.5 Data storage

In order to make the Jenkins pre-commit test maven pluginmore convenient to use and facilitate job reuse, as described in 3.7, there must be a way to store some data on a per-job basis. This is implemented by `net.jsenko.jpct.Config` class. The data are stored in a private `Map` with `String` keys and values. This map is then serialized using XStream and stored in `data.xml` file, located in each job's data directory. These directories are also used by other parts of this plugin, to store generate job configuration or patch files. The following pieces of information are saved to the file:

- `jenkinsUrl`, `jenkinsUser`, `jenkinsToken` – so the user does not have to provide them multiple times as a command line arguments, an so the `CleanMojo` can delete the job on the Jenkins server without asking for its URL.

- `goals` – similarly, this property is automatically reused and does not have to be specified again.
- `jobModelHashCode` – the plugin must be able to verify that it can reuse an existing job. To make sure that the job configuration represented by the model has not changed, the hash code of the previously used `JobModel` is stored and compared to the currently generated one.

4.6 Maven Mojos

„A Maven Plugin is a Maven artifact which contains a plugin descriptor and one or more Mojos.“[14, chap. 17.2.4] Mojo[8] is a Java class that represents a single Maven goal. It implements a `org.apache.maven.plugin.Mojo` interface, which defines methods required for interaction with Maven infrastructure. The most important are `execute()`, containing the main code, and `getLog(): Log`, providing access to the logger object. The descriptor can be generated automatically by another plugin, `maven-plugin-plugin` from annotations that can be used directly in the Mojo class. In addition, this plugin is able to generate standard `HelpMojo` to display simple documentation, which is also present in the Jenkins pre-commit test maven plugin. The properties can be injected into a specific field of the Mojo marked with a `org.apache.maven.plugins.annotations.Parameter` annotation.

8. *„The word mojo is defined as 'a magic charm or spell'*[...]. *Maven uses the term Mojo because it is a play on the word Pojo (Plain-old Java Object).“*[14, chap. 17.2.4]

5 Conclusion

The goal of this thesis was to create a tool that helps developers to conveniently perform pre-commit testing of maven-based projects. The key idea is to perform the build on a remote continuous integration server, rather than locally, to decrease load on the developer's machine, especially in case of large projects with extensive test suite.

In order to properly approach this task, I had to become familiar with project automation tools, including Apache Ant, Apache Maven, Jenkins CI and git, and study software configuration management. As a result, I have gained valuable knowledge and experience not only to complete this thesis, but also to use for development of my future projects.

I believe that the resulting tool provides software developers with a good solution to the problem and fulfills all functional and non-functional requirements that have been identified in the *Analysis and Design* chapter.

The *Jenkins pre-commit test maven plugin* has been released under open source license together with documentation and usage examples. I hope that people will find it useful and contribute to its further improvement, particularly in the following areas:

- Add support for additional build tools in addition to Maven, including Apache Ant.
- Currently, user does not have many options to configure Jenkins jobs that are being generated in more detail. Therefore, job model should be extended to support more customization in the POM.
- Efficiency of some parts of the plugin, such as the base commit finding algorithm, could be further improved.

Bibliography

[1] AIELLO, Bob a Leslie A SACHS. Configuration Management Best Practices: Practical Methods that Work in the Real World. Upper Saddle River, NJ: Addison-Wesley, c2011, xxxvii, 229 p. ISBN 03-216-8586-5.

[2] SOMMERVILLE, Ian. Software engineering. 9th ed. Boston: Addison-Wesley, c2011. ISBN 978-0-13-703515-1.

[3] LI, Patrick. JIRA 5. 2 Essentials. Birmingham: Packt Publishing, Limited, 2013. ISBN 978-178-2179-993.

[4] OMG Unified Modeling Language (OMG UML), Superstructure, V2.1.2. 2007. [visited 2014-01-02] Available at: `http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF`

[5] CLARK, Mike. Pragmatic project automation: how to build, deploy, and monitor Java applications. Raleigh: The pragmatic Bookshelf, c2004, xiv, 161 s. ISBN 09-745-1403-9.

[6] SMITH, Peter. Software Build Systems: Principles and Experience. Upper Saddle River, NJ: Addison Wesley, c2011, xxxv, 583 p. ISBN 03-217-1728-7.

[7] HUMBLE, Jez a David FARLEY. Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation. Upper Saddle River, NJ: Addison-Wesley, 2010, xxxiii, 463 p. ISBN 978-032-1601-919.

[8] ACM HONORS CREATOR OF LANDMARK SOFTWARE TOOL. In: ACM - Association for Computing Machinery [online]. New York, 2004 [visited 2013-12-09]. Available at: `http://www.acm.org/announcements/softwaresystemaward.3-24-04.html`

[9] Make(1). The FreeBSD Project: FreeBSD Man Pages [online]. 2013 [visited 2013-12-09]. Available at: `http://www.freebsd.org/cgi/man.cgi?query=make&sektion=1&manpath=Red+Hat+Linux%2fi386+9`

[10] RICHARD M. STALLMAN, Richard M.Roland McGrath. GNU Make: A Program for Directing Recompliation; GNU Make Version 4.0. Boston, MA: Free Software Foundation, 2013. ISBN 18-821-1483-3.

[11] Overview of Apache Ant Tasks. The Apache Ant Project [online]. 2013 [visited 2013-12-09]. Available at: `http://ant.apache.org/manual/tasksoverview.html`

[12] Peter Vogel. MARC: Mailing list ARChives [online]. 2001 [visited 2013-12-09]. Available at: `http://marc.info/?l=ant-user&m=98152914819125&w=2`

[13] Features | Apache Ivy. The Apache Ant Project [online]. 2013 [visited 2013-12-09]. Available at: `http://ant.apache.org/ivy/features.html`

[14] SONATYPE COMPANY. Maven: The Definitive Guide. 1st ed. Sebastopol, California: O'Reilly, c2008. ISBN 978-0-596-51733-5.

[15] CASEY, John, Vincent MASSOL, Brett PORTER, Carlos SANCHEZ a Jason VAN ZYL. Better Builds with Maven: The How-to Guide for Maven 2.0. 2008. Available at: `http://www.maestrodev.com/wp-content/uploads/2012/03/betterbuildswithmaven-2008.pdf`

[16] Maven Archetype Plugin. Apache Maven Project [online]. 2011 [visited 2013-12-16]. Available at: `http://maven.apache.org/archetype/maven-archetype-plugin`

[17] Maven Archetype Plugin Documentation. Apache Maven Project [online]. 2011 [visited 2013-12-16]. Available at: `http://maven.apache.org/archetype/maven-archetype-plugin/plugin-info.html`

[18] Introduction to the POM. Apache Maven Project [online]. 2013 [visited 2013-12-12]. Available at: `http://maven.apache.org/guides/introduction/introduction-to-the-pom.html`

[19] POM Reference. Apache Maven Project [online]. 2013 [visited 2013-12-12]. Available at: `http://maven.apache.org/pom.html#Maven_Coordinates`

[20] Git [online]. 2013 [visited 2014-01-02]. Available at: `http://git-scm.com`

[21] TORVALDS, Linus. Linus Torvalds on git. 2007. [visited 2014-01-02] Available at: `http://git.wiki.kernel.org/index.php/LinusTalk200705Transcript`

[22] PRESSMAN, Roger S. Software engineering: a practitioner's approach. 7th ed. New York: McGraw-Hill Higher Education, c2010, xxviii, 895 p. ISBN 00-733-7597-7.

[23] What's the difference between VCS and SCM?. Stack Overflow [online]. 2010 [visited 2013-12-28]. Available at: http://stackoverflow.com/questions/4127425/whats-the-difference-between-vcs-and-scm

[24] CHACON, Scott. Pro Git. New York: Apress, c2009, xxi, 265 s. ISBN 978-1-4302-1833-3.

[25] GitHub. Contributors to jenkinsci/jenkins [online]. 2013 [visited 2014-01-02]. Available at: http://github.com/jenkinsci/jenkins/graphs/contributors

[26] SMART, John Ferguson. Jenkins: the definitive guide. Beijing: O'Reilly Media, 2011, xxii, 380 pages. ISBN 14-493-0535-0.

[27] Plugins. Jenkins Wiki [online]. 2013 [visited 2014-01-02]. Available at: http://wiki.jenkins-ci.org/display/JENKINS/Plugins

[28] Who broke the build?. PaperCut Blog / News [online]. 2011 [visited 2014-01-02]. Available at: http://www.papercut.com/blog/chris/2011/08/19/who-broke-the-build

[29] Designing pre-tested commit. Jenkins Wiki [online]. 2009 [visited 2013-12-28]. Available at: http://wiki.jenkins-ci.org/display/JENKINS/Designing+pre-tested+commit

[30] Pre-tested commit feature. Jenkins JIRA [online]. 2013 [visited 2013-12-28]. Available at: https://issues.jenkins-ci.org/browse/JENKINS-1682

[31] Pretested Integration Plugin. Jenkins Wiki [online]. 2013 [visited 2013-12-28]. Available at: https://wiki.jenkins-ci.org/display/JENKINS/Pretested+Integration+Plugin

[32] Pre-Tested Commit: No broken code in your version control. Ever. TeamCity [online]. 2013 [visited 2013-12-28]. Available at: http://www.jetbrains.com/teamcity/features/delayed_commit.html

[33] Git, Jenkins, Gerrit, Code Review and pre-tested commits. Knowledge Is Everything [online]. 2013 [visited 2013-12-28]. Available at: http://www.halyph.com/2013/08/git-jenkins-gerrit-code-review-and-pre.html

[34] Gerrit Trigger. Jenkins Wiki [online]. 2013 [visited 2013-12-28]. Available at: http://wiki.jenkins-ci.org/display/JENKINS/Gerrit+Trigger

[35] Resolving a merge conflict on gerrit. Blog of Jeroen De Dauw [online]. 2013 [visited 2013-12-28]. Available at: http://www.bn2vs.com/blog/2013/07/01/resolving-a-merge-conflict-on-gerrit

[36] Git Workflows and Tutorials. Atlassian [online]. 2013 [visited 2013-12-28]. Available at: http://www.atlassian.com/git/workflows

[37] FIELDING, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. Irvine, 2000. Doctoral dissertation. University of California.

[38] Why PATCH is Good for Your HTTP API. In: Mnot's blog [online]. 2012 [visited 2013-12-28]. Available at: http://www.mnot.net/blog/2012/09/05/patch

[39] Chapter 5. Client API. Jersey 2.5 User Guide [online]. 2013 [visited 2013-12-28]. Available at: http://jersey.java.net/documentation/latest/client.html

Appendix A
Archive Content

This archive contains contents of the project's git repository, currently hosted on GitHub:

- Name: *jpct-maven-plugin*
- The repository is accessible via web browser or `git clone` at `http://github.com/jsenko/jpct-maven-plugin.git`.
- Released under Apache License v. 2.0, text included in the `LICENSE` file.
- Documentation available in `README.md` (markdown file).

In addition, compiled plugin (`target/jpct-maven-plugin-1.0.jar`) and documentation (`README.html`) are included.