

UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Tõnis Pool
**Continuous Integration & Feature
Branches**

Bachelor thesis (6 EAP)

Supervisors: Toomas Römer, Rein Raudjärv

Academic Supervisor: Vambola Leping

Author: “.....” May 2013
Supervisor: “.....” May 2013
Supervisor: “.....” May 2013
Supervisor: “.....” May 2013
Professor: “.....” May 2013

TARTU 2013

Contents

Introduction	3
Goals	3
Prerequisites	3
Outline	4
1. Continuous Integration	5
1.1. Definition	5
1.2. Prerequisites	5
1.3. Workflow	6
1.4. Usefulness	6
1.5. Downside	7
1.6. CI servers	7
1.6.1. Jenkins	8
1.6.2. Bamboo	8
2. Feature Branches	9
2.1. Definition	9
2.2. Prerequisites	9
2.3. Workflow	10
2.4. Usefulness	10
2.5. Alternatives to feature branches	11
2.5.1. Feature toggles	11
2.5.2. Incremental changes	11
2.5.3. Branch by abstraction	12
2.5.4. Decoupled components	12
3. Problem Statement	14
3.1. General problem	14
3.2. Artifact propagation problem	16
4. Current Solutions	17
4.1. No component dependencies	17
4.1.1. Plan branches	17
4.1.2. Job cloning	18
4.1.3. Mitigating integration risks	19
4.2. Many component dependencies	20

5. Our Solution	21
5.1. Introduction	21
5.2. Configuration	21
5.3. Running jobs on feature branches	23
5.4. No component dependencies	24
5.5. Many component dependencies	25
5.5.1. Multijob plugin	25
5.5.2. Throttle Concurrent Builds plugin	25
5.5.3. Environment locked multijobs	25
5.5.4. Multijob gatekeepers	27
5.5.5. Accessing built archives from tests	28
5.5.6. Rerunning tests with the same archive	28
6. Future Work	29
6.1. Improvements to our solution	29
6.2. Alternative solutions	30
6.2.1. Generic branch watcher	30
6.2.2. Maven specific solutions	30
Conclusions	32
Resümee	33
Bibliography	37

Introduction

Continuous Integration and feature branches are mutually exclusive approaches to software development. You can get the benefits of one, but not the other. We will look into ways how to reap the fruit of both and which corners we have to cut for that. We will see what the standard Continuous Integration tools offer today and what they could offer tomorrow.

We will concentrate on the open source Jenkins Continuous Integration server but also look at the competition. Platform wise this paper is geared towards Java programming language and more specifically the Apache Maven build tool[1], but the proposed solutions are general and platform agnostic.

Goals

The thesis carries the following goals:

1. Introduce the reader to the terms Continuous Integration (CI) and feature branches with a more in depth look at Jenkins CI server and Mercurial Version Control System (VCS).
2. Enumerate existing solutions of using those two methodologies together.
3. Provide an enhanced solution to use those two methodologies together in Jenkins CI server and Mercurial.

Prerequisites

This work requires the reader to have a basic understanding of Version Control Systems (VCS) and software development in general. Familiarity of build tools, such as Maven[1], and their purpose is expected.

Outline

The work is organised as follows:

1. *Continuous Integration* - Introduces the reader to the term Continuous Integration (CI), describing its contents, usefulness and giving an overview of two existing software solutions - Jenkins and Bamboo.
2. *Feature Branches* - Introduces the reader to the term feature branches, describing its contents, usefulness and providing a short example workflow for Mercurial VCS.
3. *Problem Statement* - Introduces the inherent clash between feature branches and Continuous Integration, explaining why they are considered mutually exclusive.
4. *Current Solutions* - Lists current solutions of using the two methodologies together.
5. *Our Solution* - Introduces our solution to the previously defined problems.
6. *Future Work* - Lists possibilities for future work on our solution and on research in general in this problem domain.

Chapter 1

Continuous Integration

1.1. Definition

Continuous Integration is a software development practice, where members of a team integrate their code frequently with code from other developers. Usually each person integrates at least daily - leading to multiple integrations of project source code per day. Each integration is verified by packaging the application and running it.

To get more benefit out of CI the application should be covered with automatic tests that run every time the application is packaged[2]. This process of automatically packaging and testing is often referred to as an automatic build.

The term 'Continuous Integration' originates from the late 1990's with the Extreme Programming development process, as one of its original twelve practices[3]. Though mission-critical software projects used similar approaches earlier, NASA being one example of this[4].

1.2. Prerequisites

The use of Continuous Integration (CI) within a software project has namely three prerequisites:

1. *Shared Version Control System* - CI demands that the source code of the project be held within a shared code repository accessible to all developers.
2. *Automated build* - In order for CI to work the software must be built and packaged in an reproducible, automated way.
3. *Automated tests* - To ripe the full benefits of CI the project must validate its correctness with automated tests.

1.3. Workflow

The practice of Continuous Integration can be explained by going through the workflow of writing a new feature to a software system. The current state of the system (code in the VCS) is usually referred to as the *mainline*. At any time a developer can make a controlled copy of the mainline onto their own machine, this is called *checking out*[5]. Developer tasked with a new feature first checks out a working copy from the mainline to his local machine, makes the necessary changes and then runs the automated build and tests to verify, whether the project still compiles and works as expected.

When the feature is ready, the changed source code has to be put back into the shared repository. To achieve this the developer has to update his local copy of the mainline and then push the changes to the remote repository. Notice however, that between the initial checkout and the later update, surely other team members have made changes to the code base as well. Nothing can guarantee that changes made by two developers working in parallel are compatible in any way. This is where CI steps in.

After the changes are pushed to the shared code repository the build and test phases are ran again on a dedicated integration machine with updated code. Only when the build succeeds and tests pass on that machine is the build considered good and the feature completed. If two developers have made incompatible changes it will be visible after the next CI build, usually just hours after the clash occurs.

1.4. Usefulness

The usefulness of CI can be derived from the increasing popularity and use of its tools itself[4], but in general CI provides the following benefits:

1. *Bugs are found and fixed sooner* - With a sufficiently large coverage of automated tests, bugs (software faults) in the project will become visible shortly after they appear, indicated by a suddenly broken build. Fixing and finding bugs early is crucial to any project's success, because as time goes on and bugs accumulate, they become much more expensive to fix and remove[6].
2. *Project health is visible and measurable* - Daily integrations and automated tests provide a good insight into the health of the project at any given time. Everyone involved knows how many tests are failing and/or whether or not there are some integration problems.

Not knowing about integration problems often used to become a pitfall for many projects where 'big bang' integration - all features are integrated in one go near the deadline of the project - resulted in large numbers of errors. These errors are hard

to isolate and correct owing to the sudden vast expanse of the program, when every feature is put together to form an application[7]. These large integration errors in turn delay the project indefinitely, with no way of predicting when they would be resolved[2].

3. *Reduced assumptions* - By rebuilding and testing software in a clean environment in an automated fashion the dependence on any specific configuration or setup and chance for human error is minimized (e.g. if someone forgets to add a third party library to the environment).
4. *Reduced time to market* - Because the project health is always visible and (at least) critical functionality is covered by automated tests, a release can happen at any time when all (or sufficient amount of) tests pass which reduces the time to market of newer features.

1.5. Downside

Though not much talked about, one of the downsides of CI is that the mainline may not always be in a releasable state[8]. This happens when all development is done on the mainline, which means larger changes to the application have to be done sequentially on the tip of the repository. Often the intermediate steps in those changes leave the project in a broken state[8]. (How to keep the mainline in a releasable state is further discussed section 2.5).

Also by the rules of CI, developers encountering a broken build, must not check in their changes until the build has been fixed[8]. This can potentially halt development, forcing the entire team to turn their attention towards fixing the build and not developing new features.

1.6. CI servers

The integration build doesn't require any specialized tooling and can happen in any clean environment. One option would be to use a spare development machine and trigger a remote build manually. As this is again a task that should be automated to reduce assumptions, it would be better to use a dedicated Continuous Integration server.

CI server is a dedicated server that monitors your shared code repository for changes pushed by team members and then triggers the automatic build. Modern CI servers are much more flexible and extensible than this and can be used for any automated task. Releasing the project is one example of such a task. In this work we'll look at two available CI servers.

1.6.1. Jenkins

Jenkins is an open source CI server created by Kohsuke Kawaguchi in 2004. The project changed its name in 2011 (previously called Hudson) after a trademark dispute with Oracle[9]. It's currently the most popular CI server, with over 49% of the market share[4]. It's popularity can be explained because of its low costs (free to use) and extensibility (it has over 600 plugins)[10].

In Jenkins the main concept is a configurable *job*, that can in turn trigger other jobs to run. Each run of a job is referred to as a build, with a unique numeral identifier. Each job consists of zero or more pre-build, build and post-build steps. Pre-build and post-build steps are mostly meant for setting up and tearing down the environment to execute build steps. A build step can be any desired automated task, from a shell script execution to some artifact deployment.

Jenkins also has a notion of distributed builds, where the workload of building projects is delegated to multiple "slave" nodes, allowing a single Jenkins installation to host a large number of projects, or to provide different environments needed for builds/tests[11]. It means that at the start of every build Jenkins looks for a free node and executes the build on that machine. This is especially useful for speeding up testing phase, because different tests could be executed in parallel on different nodes. Quicker test results means faster feedback cycle and integration errors are reported sooner.

1.6.2. Bamboo

Bamboo is a commercial product developed by Atlassian, Inc. In Bamboo the focus is not so much on a specific job, as on a much more abstract concept of a *plan*. A plan can contain one or more sequential phases, which in turn can contain any number of sequential jobs. Each job can contain any number of sequential tasks. Bamboo also supports distributed builds.

Each plan is by default meant to work on a single code repository (with an option to configure multiple repositories)[12]. Bamboo is also extendable via plugins similarly to Jenkins, though there are significantly fewer add-ons (around 130)[13].

Chapter 2

Feature Branches

2.1. Definition

Most Version Control Systems have the ability to create a *branch* from the mainline. In broader terms branching means the duplication of objects under version control so that modifications on those objects could happen in parallel versions. Thus when a developer creates a branch from the mainline he can then check out and work on that branch instead of the mainline. Integrating the changes done in mainline and a branch is called merging. After a merge objects in the branch and mainline are back in the same state.

Feature branches is a software development practice where a member of a team tasked with creating a feature (or a bug fix) first creates a branch of the code repository and checks it out instead of the mainline. This allows him to work on that feature in isolation from his team members, before merging his changes back into mainline development. This allows development teams to partition work and prevent interruptions from external sources (other developers)[14, 5].

2.2. Prerequisites

The only prerequisite of using a feature branch is that the VCS in use for the project supports creation of branches. Though feature branches are more popular with Distributed Version Control Systems (DVCS) - like Mercurial and Git - namely because branches are considered as first class citizens and a variety of branch specific operations are supported[15].

2.3. Workflow

The usual workflow with the example of Mercurial is the following[16]:

1. *Create a new branch for the feature* - To implement a feature first create a new branch for it.

```
hg branch feature-x
```

2. *Commit and publish the changes* - Then continue developing in the branch pushing changes to make them available to other developers.

```
hg commit -m "Started implemented feature-x"
hg push
```

3. *Merge mainline into feature branch* - As often as possible (at least once a day) merge mainline into the branch

```
hg merge default
hg commit -m "merged default into feature-x"
```

4. *Merge completed feature branch into mainline* - When the feature is completed and stable merge it into default

```
hg update default
hg merge feature-x
hg commit -m "merged feature-x"
```

5. *Close feature branch* - After that the feature branch should be closed

```
hg commit --close-branch -m "finished feature X"
```

2.4. Usefulness

The usefulness of feature branches lies in their simplicity. It's easy to create a branch to isolate oneself from others and keep the mainline clean until the feature is complete[8]. Then it may be merged back into the mainline, enriching it with one complete and stable feature. Indeed this is a commonly used pattern for riskier changes to the code[15], such as refactoring or experimenting, where developers are cautious about potentially halting the teams' work by breaking the application.

Although, this simplicity comes with a cost of increasing the risk of lengthy integrations when the feature is thought to be complete. We'll discuss this topic in more depth in chapter 3.

Feature branches have a one more unique advantage in open source projects using or migrating to Distributed VCS-s (like Mercurial or Git). In such projects branch usage has increased significantly[17] due to better code sharing capabilities of Distributed VCS-s.

In the days of Centralized VCS-s, changes in open source projects were often in the form of emailed attachments containing code patches. The bigger the patch, the less likely it was to get accepted by the reviewer. This was mainly due to the trouble and effort needed to go through a one monolithic patch[18].

With distributed VCS-s however, monolithic patches can be replaced by simple branches, preserving individual changes to the code, thus making it easier to review changes[17]. To apply a patch the reviewer simply has to merge a branch from the repository of the committer.

2.5. Alternatives to feature branches

Feature branches isn't by far the only technique to keep the mainline in a releasable state. Because of the problem between CI and feature branches (discussed further in chapter 3), many other ways have been thought and proposed that allow developing in the mainline and keeping the application releasable at all times.

2.5.1. Feature toggles

The idea behind feature toggles is to have a set of configurable switches or toggles that can be turned on or off which in turn decide whether some feature is shown to the user or not. Thus the incomplete feature may be present in a release, but as it's disabled internally, users don't know it exists. Configuration can be given either through command-line or runtime options[8].

Runtime toggles provide the additional benefit of being able to switch off broken features in production instead of redeploying an older version or degrade the service under heavy load simply by disabling features[19].

2.5.2. Incremental changes

Another (and complementary to the previous) solution is to roll out changes incrementally in small patches. Often this can seem unproductive and hard, but this just remedies the pain of integrating large changes when they are finishing[8]. Dividing a feature into several smaller patches forces the developer to keep the application in a working state through the whole process. One example of this could be a complete overhaul of a user interface, which is done either per view or per widget basis instead of starting to update

every view at once.

A second benefit of this approach is avoiding wasted time and resources involved in getting halfway through a large change and then abandoning it[8]. Meaning if the business objectives change during an implementation of a large feature then it's easier to rollback the completed smaller steps than trying to rollback a change that has taken everything to a broken state.

An example would be when managers decide the new user interface is not so important as feature X. When only a handful of views have been already edited and completed, then they can either remain with the new look or be rolled back to the previous version. Opposed to a scenario where the developer has started to change every view at once and completed none of them.

2.5.3. Branch by abstraction

Branch by abstraction is a term coined by Paul Hammant[20] that can be used when a large scale change needs to be done, that cannot be broken down into a series of smaller changes. It's a practice of finding entry points to the part of the application that needs changing, abstracting those entry points to a common proxy, which delegates to the current implementation and finally replacing the current implementation with the new one[8]. The steps can be defined as follows:

1. Create an abstraction over the part of the system that needs changing.
2. Refactor rest of the system to use the abstraction layer.
3. Start creating the new implementation.
4. When the implementation is complete, update the abstraction layer to delegate to it.
5. Remove the old implementation.
6. Remove the abstraction layer if needed.

Two most difficult steps of branching by abstraction are the identification and abstraction of all entry points[8]. Also managing any changes made to the old implementation in an earlier release, as part of a bug fix for example, can become problematic.

2.5.4. Decoupled components

This technique can be considered to be an extension of the previous, where the abstracted part is moved into a separate component and added as a dependency[8]. A component in this sense is a reasonably large-scale code structure, with a well-defined API, that could

be swapped out for a new implementation.

The new component can be used through the abstraction layer, which starts to serve as the API to that component. Thus development on the separate component can continue in its own pace and not affect the release cycle of the whole application, because a suitable stable version of the component can be used to release the application.

Chapter 3

Problem Statement

3.1. General problem

From the sections 1.1 and 2.1 the inherent clash between Continuous Integration and feature branches is immediately visible - CI is about integrating code often, enabled by developing in a **single** mainline to avoid lengthy integrations phase at the end of the project. Opposed to feature branches, which are about moving development to **separate** branches in order to keep the mainline releasable and isolate developers, thus preventing integration.

Feature branches tend to be long-lived and merges between them are done infrequently, or not at all, until the completion of the feature[15]. This is the main reason why the use of feature branches is strongly discouraged along with Continuous Integration[5]. It creates an opportunity to fall into the problem that was tried to remove by using CI in the first place - the lengthy integration phase at the end of the project[8].

In section 2.5 we listed numerous alternatives to feature branches that can, and often should be used instead. Though one can note that all the alternatives include some sort of extra effort for the developers or other pitfalls.

Implementing large changes in a series of small ones or branching by abstraction can be a lot more time consuming than creating a branch. Feature toggles can get so numerous that they're hard to maintain and in extreme cases more than the underlying operating system can handle[8]. All the alternatives are common in that they require some extra code to implement, increasing the application complexity and the needed development effort.

The world isn't black and white and there are situations in which feature branches would be the most practical solution for parallel development. One of such are open source projects with distributed teams[8] where the integration risk is justified by the comfort

of applying patches and improvements via branches from developers. Another example would be a small team of experienced developers working with DVCS-s, where everyone is aware of the integration risks associated with feature branches and merge between branches as often as possible.

In any case the use of feature branches would be justified for practical reasons of not introducing extra complexity in order to keep the mainline clean and developers isolated. Though the integration risk remains, merging branches often can mitigate it. But currently, for example, the open source CI server Jenkins offers little or no support to be used alongside feature branches development.

In the following chapters 4 and 5 we'll look into ways of using CI alongside feature branches in the Jenkins and Bamboo CI servers, but before that we'll define the scope of the problem in terms of component dependencies.

The term component was already defined in subsection 2.5.4, but here we'd like to further narrow it to mean a code-structure not located in the same code repository as the main application. Thus a component dependency means a dependency to a code-structure, managed by the team, located in another code repository.

1. *No component dependencies* - In this scenario the whole application resides in a one shared code repository. It may be modular, but all the components are located together with the main application. This is the simplest scenario in which to use CI and feature branches together, because the only problem here is, how to force merges between branches to avoid integrating only on feature completion.
2. *Many component dependencies* - In this scenario at least one external component is used, where we would also like to use feature branches. An example situation is where main repository contains feature branches *branch1*, *branch2* and the external component contains feature branches *branch1* and *branch3*. Any number of branches could be shared (meaning the feature involves both repositories) or unique to one or the other code base.

This scenario is more complex, because besides the integration problem it introduces the artifact propagation problem, discussed in the following section 3.2.

When distributed CI builds are also a requirement, this problem gets even more complex. This is due to the fact that now we need to solve the artifact propagation problem across multiple machines.

3.2. Artifact propagation problem

When one or more of the components is an external dependency then we also need to deal with artifact propagation problem when using feature branches and CI. The component dependency can be either build-time or runtime, but from now on, we'll only be concerned with build-time dependencies, because this is how Maven dependencies usually work[21].

The problem itself lies in that every project has to resolve its dependencies somehow, in order to package or run the application. How it's often done is that the dependencies are searched from the machine's local environment or some remote location. In case of Maven, local repositories are mostly used to look for dependencies, if they are not found then the search begins on remote repositories[22].

The problem arises when some component dependency also uses feature branches. As Maven local repository contains only the latest snapshot of each artifact it gets overwritten every time a build is made on some branch. Let's assume we have a Maven project *projectMain* that has a component dependency on *projectAPI*. The *projectMain* contains feature branches *branch1*, *branch2* and *projectAPI* feature branches *branch1*, *branch3*.

When building *projectMain* Maven will look for *projectAPI* in the local repository. Assuming it has been built before on that machine (e.g. via CI), it will be successfully retrieved. But we have no data or guarantees about which branch the dependency was last built on. When we're building *projectMain* on *branch1* we could unknowingly package it with *branch3* build of *projectAPI*. This is illustrated on figure 3.1

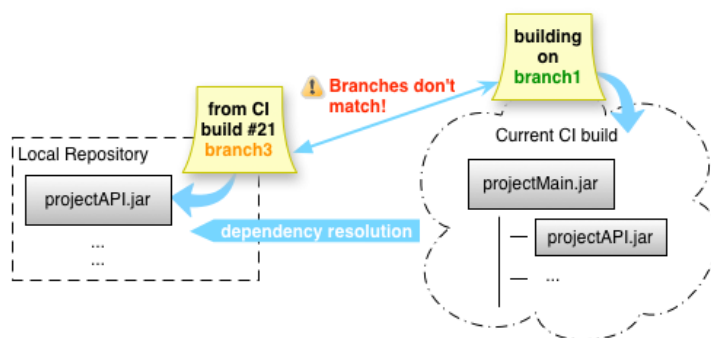


Figure 3.1: Illustration of the artifact origin problem

This uncertainty or lack of guarantee for the origin of the dependency can produce unexpected results when mixed with using multiple feature branches. When working on a certain branch one would expect the dependencies to be from the same branch (i.e. propagate artifacts through branches) or at least from the mainline. Thus to solve this problem we need to guarantee the origin (branch) of the artifact in the local repository.

Chapter 4

Current Solutions

As was mentioned earlier, there is little or no support for feature branches development in Jenkins, though other CI servers have some features worth mentioning. In this chapter we'll investigate how to use feature branches in the scope defined in chapter 3.

4.1. No component dependencies

In this scope the main problem to solve is to first make CI jobs run on feature branches and then enforce merges between branches to mitigate integration errors on feature completion.

4.1.1. Plan branches

Since version 4 Bamboo CI server has a feature called *Plan Branches*, which exactly tries to tackle the "No component dependencies" problem stated in chapter 3. Plan branches are Bamboo plan configurations that represent a branch in your Version Control System. They inherit all of the configuration defined by the parent plan, except that instead of building against the repository's mainline, they build against a specified branch[14].

When plan branching is activated, Bamboo automatically creates plan branches when it detects new branches in the repository. This can be configured to a specific regular expression, e.g. to only create plan branches for branches starting with the prefix "feature-" [23].

Automatically starting CI jobs also on feature branches was only one part of the problem. To mitigate the integration risks, Bamboo introduced the notions of a *gatekeeper* and a *branch updater*. In principle they are different sides of the same coin. Gatekeeper merges the feature branch into mainline before each build and branch updater does the opposite, by merging the mainline into the feature branch[14]. The principles of a gatekeeper are illustrated on figure 4.1

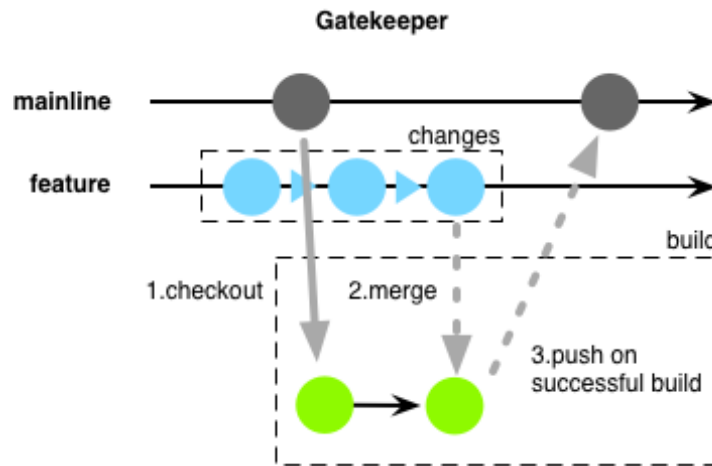


Figure 4.1: Illustration of gatekeeper

Both the gatekeeper and branch updater push the merge only after a successful build, meaning all of the plan's jobs have successfully finished[14]. This gives us the opportunity to run all of the tests on the merged code and be certain that the build is not broken if the merge is pushed to the shared code repository.

By enabling either gatekeeper or branch updater the developers working on a feature branch will immediately know when they can expect to run into integration errors, because either the merge or the build after the merge will fail.

We would recommend using a gatekeeper, when it's desirable to make the changed code immediately visible to other developers, without braking the build for them (the merge will be pushed only if the build succeeds). Branch updater could be used when it's desirable to keep any part of the feature code from reaching the mainline (and on to a release) until the feature is ready. The decision is largely a matter of taste, as the desired effect of learning about integration errors is reached by both approaches.

4.1.2. Job cloning

One way to support feature branches in Jenkins CI server is to clone each job running on a repository and configure them to run on the required feature branch. As this would require much manual work, developers would rarely be willing to create corresponding CI jobs for their branches. To mitigate this problem a couple of solutions exist that monitor the code repository and clone a defined set of jobs for each branch.

1. *Jenkins autojobs* - An open source project started a year ago (April 2012) by Georgi Valkov, written in the Python programming language. Intended to work as a script that on each run, given either the configuration file or command-line options, lists the branches in the repository and creates a clone of a template job running on that branch. A template job is just an existing and working Jenkins job, that gets

cloned and renamed with only its VCS branch value changed[24].

The project supports Git, Subversion and Mercurial Version Control Systems, though the Mercurial support is currently (version 0.6.0) a bit buggy, not allowing feature branch names to contain spaces. Meaning if we name a branch *feature x* the cloned job will be configured to run on branch *feature*.

2. *Jenkins Build Per Branch* - An open source project also started in April 2012 by a company called Entagen, LLC. Written in the Groovy programming language. In principle, a very similar solution to Jenkins autojobs, intended to be used as a script that on each run clones Jenkins jobs, configuring them to work on a feature branch[25].

The upside of this project is that it takes a set of jobs to clone for each branch, not just one template. The set is defined by a common prefix. For example every job starting with "projectAPI-" prefix and running on the defined repository, would get cloned for each detected feature branch.

One big downside of this solution is that it currently only supports Git VCS, with no apparent intention of extending that support to Mercurial.

Job cloning would solve the problem of automatically starting CI jobs on feature branches, though creating a new job for each branch can clutter the jobs view in Jenkins and be hard to navigate. This could be mitigated by the use of the Nested View plugin for Jenkins[26], which, as the name hints, allows to nest similar views. Using this plugin we could group all feature branch jobs under one view. The Jenkins Build Per Branch project has the feature of automatically nesting the cloned feature branch jobs to a separate view, but again that project only supports Git VCS.

4.1.3. Mitigating integration risks

After the jobs are running on feature branches, lending the gatekeeper and branch updater notions from Bamboo can solve the matter of potential integration risks. In Jenkins executing shell scripts could simply mimic this functionality. One script that does the merge as the first build step and another that pushes the merge as the last build step, building and testing in the steps between.

This ensures that the merge would be pushed only if the build succeeds, because if any build step fails beforehand, Jenkins doesn't execute the remaining build steps. As a side note, the committed merge should be rolled back when the build fails, to clean up the local repository.

4.2. Many component dependencies

This problem scope is mostly unexplored and currently no solutions have been proposed that we are aware of. Though the artifact propagation problem can be solved by cleverly combining multiple existing plugins in a few complex job configurations, which we'll propose as part of our solution in chapter 5.

Those complex jobs could be automatically cloned for each branch with the existing projects, but we can't get around the problem of having different repositories where we want to use feature branches. The complex jobs would have to have so called "sub-jobs" (discussed in chapter 5), that work on different repositories. Current cloning projects aren't smart enough to clone the complex jobs and reconfigure just the "sub-jobs" to a certain branch. Similarly Bamboo's plan branches only work on one repository at a time.

Chapter 5

Our Solution

Our solution comes in a form of a new Jenkins plugin that integrates deeply with the Mercurial VCS plugin and Jenkins's core itself. In combination with some of the existing plugins we are now capable of solving all of the problems defined in chapter 3.

5.1. Introduction

We created a plugin called "Feature Branch Notifier" that can be enabled for any job. Its main functionality lies in detecting new commits in feature branches and launching new builds on those updated branches. It works as a combination of a patched Mercurial plugin, new trigger plugin and a pre-build step, enabling us to detect updated branches with polling and change the branch the build will run on. Currently only Mercurial VCS is supported.

Builds on feature branches will be specially marked and they can be scheduled for a rerun with a special menu action. The plugin's source code is available at <https://bitbucket.org/poolik/feature-branch-notifier>.

5.2. Configuration

The plugin's configuration itself is straightforward. First the user has to install the plugin. After that in order to start launching builds on feature branches the user has to choose a new Source Code Management option called *Mercurial (feature branch aware)* (figure 5.1).

Its configuration is similar to the default Mercurial plugin, with only one additional field called *Match branch names with* under advanced options. This field can be used to filter branches that get monitored. The value of that textbox should be a regular expression that is evaluated against the branch name. Thus if we would only like to launch builds on branches starting with "feature -" we would input "feature -.*" into the *Match branch names with* textbox.

Source Code Management

☐ CVS
☐ Mercurial
☒ Mercurial (feature branch aware)

Repository URL:

Branch:

Advanced...

Repository browser:

Figure 5.1: Configuring Feature Branch Notifier plugin

Additionally, the user has to set the Mercurial branch in Source Code Management section to be a special environment variable named `$BRANCH` as can be seen also on figure 5.1. This is required in order to dynamically set the branch the build is running on.

`$BRANCH` is an environment variable that get's its value from the Feature Branch Notifier plugin. It's set to "default" when the build is not marked with a special feature branch tag or when the marked feature branch doesn't exist in that particular repository. Otherwise it will be set to the marked feature branch.

As many users use either polling the repository or sending push notifications (which in turn fire a poll) to Jenkins to trigger a build, we had to create a custom trigger in order to dynamically detect changes on feature branches as well. The new trigger checks for changes in all of the branches and adds a special tag to builds scheduled on feature branches to indicate it to users and record the branch.

To enable the trigger, users have to select the new *Feature branch aware Poll SCM* option under Build Triggers as can be seen on figure 5.2. The configuration of this trigger is the same as the usual Poll SCM trigger and users can just copy their old configuration from there and then deactivate the old polling.

Build Triggers

☐ Build after other projects are built
☐ Build periodically
☒ Feature branch aware Poll SCM

Schedule:

Figure 5.2: Configuring Feature Branch Notifier trigger

Finally to actually dynamically set the `$BRANCH` environment variable, we had to

implement a pre-build step that checks the build for a special tag. The pre build step can be activated by checking the *Check and mark builds with feature branches* checkbox, as can be seen on figure 5.3.

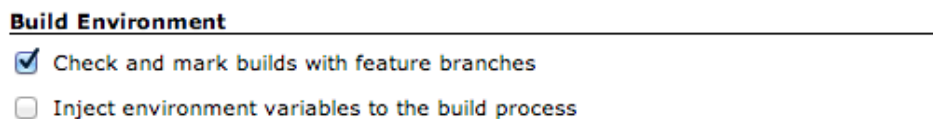


Figure 5.3: Configuring Feature Branch Notifier pre-build step

The pre-build step can and should be activated on jobs that don't necessarily itself run on a feature branch, but might have downstream projects that do. When the pre-build step detects that this builds' upstream build (the build that triggered the current build) was tagged with a feature branch, it adds the same tag to the current build as well. This is in order to propagate the tag through complex hierarchies that may have some jobs running on some feature branches in between.

5.3. Running jobs on feature branches

Builds that run on a feature branch are marked by adding the branch name to the end of the build in the build history view, as can be seen on figure 5.4. This allows users to quickly see what build was run on which branch.

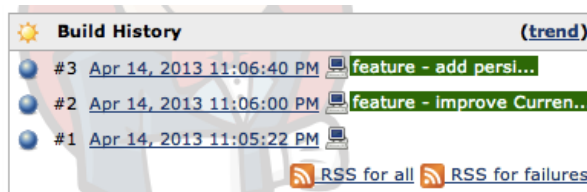


Figure 5.4: Marked builds in build history view

To allow users to quickly rerun a build on a specific branch a custom menu item (as seen on figure 5.5) is added to schedule a new build marked with the same branch as the specified build. This menu item is only visible for builds that were marked to run on a specific feature branch.

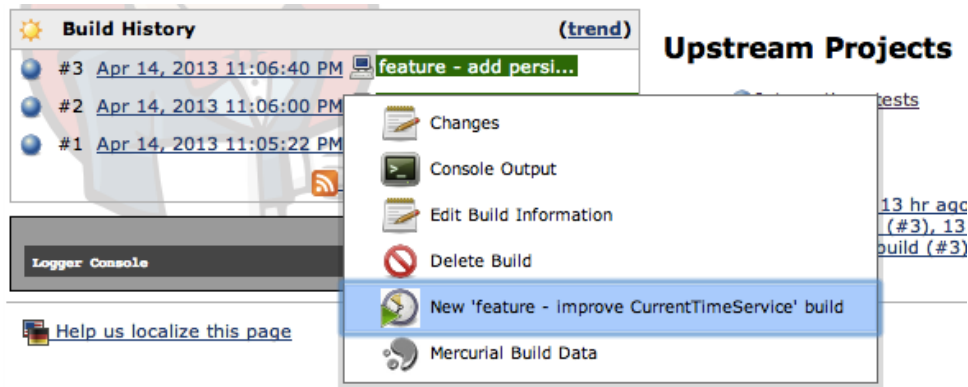


Figure 5.5: Feature branch build custom menu item

When updates on a feature branch are detected from multiple repositories (further discussed in section 5.5) they are aggregated together and scheduled as a one build to avoid filling Jenkins' job queue with duplicate builds and wasting unnecessary resources.

5.4. No component dependencies

Our plugin provides an easy alternative to the existing solutions in terms of the "No component dependencies" problem, attacking it from a different angle. We aren't cloning jobs for feature branches, rather than just running the one job on multiple branches. This can be argued to be semantically more correct, where a job should refer to a set of actions run on a specified code repository. By cloning jobs for each branch this does not apply anymore, as we would have a set of jobs doing the same actions on the same code repository.

When the jobs are configured to be triggered by polling changes from the VCS, then another benefit is that when cloning jobs for each feature branch, there would be a job for each branch that sends periodic requests to the machine hosting the VCS. By using our plugin, only the single job polls for changes and builds are fired only when some changes are detected in feature branches, thus it's more efficient.

The plugin itself solves the automatically starting CI jobs problem and the integration risk problem can be solved by using the same approach as described in subsection 4.1.2 - by adding the first build step for merging and the last one for pushing. An additional check for the current value of `$BRANCH` should be added to only run the gatekeeper or branch updater, when not running on the default branch i.e. when the `$BRANCH` variable is not "default".

5.5. Many component dependencies

As mentioned earlier the "Many component dependencies" problem currently has no known solution, but with the combination of our Feature Branch Notifier plugin and the Multijob, Throttle Concurrent Builds plugins a configuration can be made that solves it.

5.5.1. Multijob plugin

The Multijob plugin was created by the Tikal ALM team. It can be used to create a *multijob*, which in turn can contain so called *sub-jobs*. The multijob enables the user to divide a bigger job into multiple phases. One phase can contain one or more jobs and all jobs belonging to one phase will be executed in parallel. All jobs in phase 2 will be executed only after jobs in phase 1 are completed. Thus a sub-job refers to regular Jenkins job that is configured as part of one phase of a bigger multijob. Multijobs can contain other multijobs as their sub-jobs[27].

Thus the plugins' main plus side is the ability to create abstractions of a job (or workflow) consisting of other smaller jobs. This can directly be related to the way Bamboo abstracts plans, phases and jobs. A multijob would correspond to a Bamboo plan, a Bamboo phase would correspond to a multijob phase and Bamboo jobs correspond to jobs in a multijob phase.








One of its other useful features for our problem, is the ability to force some sub-jobs to run on the same node as the parent multijob and the fact that if any of the jobs in a phase fails the sequential phases are not executed.

5.5.2. Throttle Concurrent Builds plugin

The Throttle Concurrent Builds plugin was created by Andrew Bayer. It can be used for throttling the number of concurrent builds of a project running per node or globally. Users can create categories to throttle the parallel execution of multiple jobs[28]. In our problem we are interested in disabling parallel executions on one node for specific jobs.

5.5.3. Environment locked multijobs

The way we propose to solve the "Many component dependencies" problem is by creating so called environment locked multijobs. In short this means the creation of multijobs similar to one shown on figure 5.6. That sample multijob is divided into 3 phases - "Build the project", "Test the archive" and "Run gatekeepers". The first phase is in itself a multijob consisting of phases "Build dependencies" and "Build main application".

Job	S	W	Last Success
<u>Integrations tests</u>			N/A
<i>Build the project</i>			
<u>build-all</u>			N/A
<i>build dependencies</i>			
<u>build-projectAPI</u>			N/A
<i>build main application</i>			
<u>build-projectMain</u>			N/A
<i>Test the archive</i>			
<u>test-common</u>			N/A
<i>Run gatekeepers</i>			
<u>projectAPI-gatekeeper</u>			N/A
<u>projectMain-gatekeeper</u>			N/A

Icon: [S](#) [M](#) [L](#)

Figure 5.6: Environment locked multijobs

The whole "Integration tests" multijob will run on the default or the tagged feature branch. This is achieved again via our Feature Branch Notifier plugin, that has a unique relationship with multijobs. Before scheduling a new build of a project it will check whether or not the immediate upstream project is a Multijob project, if yes then this process is repeated until the topmost multijob of the hierarchy is found and a build of that project is scheduled and tagged with a feature branch instead.

In our sample project the "build-projectAPI" and "build-projectMain" jobs are configured with our custom trigger and Mercurial plugin. When they detect any change in any branch they will see that the immediate upstream project is a Multijob project and the upstream of that is again a Multijob project. Thus the "Integration tests" job would get scheduled with a specific feature branch tag or none for "default" branch. This tag will be propagated through the hierarchy because all other jobs are configured with the pre-build step. This solves the problem of automatically running CI jobs on feature branches.

The artifact propagation problem is solved by the fact, that the "Build dependencies" and "Build main application" phases are configured to be run on the same node. This can easily be done via the Multijob phase configuration, by specifying the condition under "Advanced" » "Add Parameters" menu, as can be seen on figure 5.7. This option will force the job to run on the same node as the multijob itself is running. If in addition to this, we configure the "Integration tests" multijob to be throttled to one execution per node, it solves the artifact propagation problem.

MultiJob Phase

Phase name:

Phase jobs:

Job name:

Add Parameters ▼

- Boolean parameters
- Build on the same node
- Current build parameters
- Parameters from properties file
- Predefined parameters

Figure 5.7: Configuring multijob sub-job to run on the same node

If all of the component dependencies are freshly built before the main application in the same environment and concurrently the component dependencies are not being built in that environment, then we can guarantee the origin of the dependent artifact in the environment to be from the same build that just created it. Because the whole "Integration tests" multijob runs with a single branch tag, the main application and its component dependencies are guaranteed to be built on the same branch.

5.5.4. Multijob gatekeepers

Now we are left with only the fundamental integration risks of having code in separate branches. We tackle this with the same approach as beforehand - gatekeepers and branch updaters - but with this complex setup the configurations of these have to be put into a separate phase and separate jobs for each build. This is necessary because the contract of a gatekeeper or a branch updater was that it merges and pushes only if the build is good after that merge.

As we are now executing tests separately from building jobs and even in a separate phase, we cannot implement the gatekeeper approach the same way as before, where the first build step did the merge and the last build step pushed it. Though we still have to push the same merge that was tested, meaning we have to access the same repository state later on and push the merge. To do so the "build-all" and gatekeeper jobs are also configured to run on the same node as the whole multijob.

Because no concurrent builds of the whole multijob are allowed on one node, we have the guarantee that on the same node the build job repositories are in the same state when we access them later in gatekeeper jobs. This way we can push the merge at the very end of this multijob, when all builds and tests have successfully ran.

5.5.5. Accessing built archives from tests

We have implemented such a complex setup of jobs that a unique problem arises of how the tests receive the built archive. Currently we didn't want to restrict the running of the tests to a single node, but allow them to be executed in parallel on all nodes (only restricting building of the application and gatekeepers to one node). Test jobs cannot simply copy the built archive from the last successful run of the "build-projectMain", as some other node may have already executed the job in between as well, so the test would get the wrong artifact, from the wrong build.

To remedy this situation, the build number of the "Integration tests" job can uniquely identify artifacts. Thus we recommend to simply copy the artifact with the build number prefix to a simple file server or some dedicated artifact repositories such as Nexus[29] or Artifactory[30]. The current build number of the multijob can be propagated to tests and the build job by a parameterized build, meaning the current value of the Jenkins' provided environment variable `$BUILD_NUMBER` is set as a parameter to the test and build jobs, which use it to either copy or retrieve the correct artifact.

Copying to a special artifact repository has the added bonus of being able to later mark or promote the builds that successfully reach the gatekeepers. This way developers can quickly differentiate between good and bad artifacts for a potential release or further manual testing.

5.5.6. Rerunning tests with the same archive

A frequent requirement with CI jobs is the ability to simply rerun tests with the same artifact. This can be easily achieved with the proposed setup by manually starting the test with the same build parameter as the artifact was built on. Thus the test will retrieve the same artifact and run the tests again.

The process can be simplified by installing the Rebuild plugin[31] to Jenkins. The Rebuild plugin adds a menu action to the build to schedule a new build with the same parameters. So when a test has to be rerun, the developers can open the previous build view and just click the rebuild button.

With the introduction of environment locked multijobs we have successfully created a configuration that solves the "Many component dependencies" problem.

Chapter 6

Future Work

6.1. Improvements to our solution

Our current solution isn't perfect right now and could benefit from the following improvements:

1. *Merge with Mercurial SCM plugin* - We needed to deeply patch the existing Mercurial SCM plugin, which means we have duplicated a small portion of its code into our code base. As we have simply enhanced the functionality of the default plugin, it would be reasonable to simply merge our changes into the default plugin and make feature branch aware behavior a configurable option.
2. *Show trends per branch* - Jenkins by default illustrates trends of a build with a weather icon. Meaning if the build has been stable and successful for the last several builds it shows a sun icon, but if the last build failed, dark clouds gather.

By launching builds on different branches for the job this information is not useful anymore as builds on one branch may continuously fail, but on another branch they are stable and successful. Currently the trend showing would show unreliable information on such behavior. It would be useful if trends would be shown on a per branch basis when using our plugin.

3. *Number of failed tests per branch* - Jenkins has the option to show number of failed tests in the build and an indication of how many more tests are failing or have been fixed compared to the last build. Similarly to the above problem, when launching builds on different branches this information becomes unreliable, because all tests may pass on one branch and fail in another.
4. *Out of box solution* - The current solution comes in a form of complex configurations achieved through the use of multiple plugins. This setup is not very user friendly, because currently the users are responsible for installing the plugins and making sure the configurations are right.

The usability would be better if the plugins would come preinstalled so that users could simply configure the jobs. To achieve this the used plugins could be bundled with the Jenkins archive so that they would be present without installation.

6.2. Alternative solutions

Our solution isn't by far the only conceivable one, rather the one that was most aligned with our goals and opportunities. Here we will list a couple of other potential solutions that may go into work by the authors themselves or ignite interest in some reader and therefore enrich this domain with another solution.

6.2.1. Generic branch watcher

As stated in chapter 3, the fundamental problem about using CI and feature branches together, is the opportunity to fall into the problem that was tried to remove by using CI in the first place - the lengthy integration phase at the end of the project[8]. We tackled this issue with the notions of a gatekeeper and a branch updater, that in principle publish the changes done in feature branches immediately. Developers can get the information about some integration errors when the merge doesn't succeed, but it gives little or no information about the root cause of the integration error.

To improve this situation, instead of a gatekeeper or a branch updater, a generic branch watcher tool could be constructed, that monitors changes done in all the branches and immediately notifies when two developers start making changes to the same part of the application.

The tool can specifically notify the developers involved after which the developers can communicate with each other and avoid making incompatible changes. The branch watcher tool could also do automatic merges between the specific branches that work on the same part of the application.

6.2.2. Maven specific solutions

When taking a Maven specific approach the two following ideas could be used to construct a solution to the stated problems:

1. *Per branch local repositories* - The artifact propagation problem could be solved by dynamically changing the local repository for each branch. This means that before the maven build we specify a custom local repository indicated by the branch name. To allow for building on different nodes the remote repository must be changed as well, otherwise if the dependency is not found in the local environment an artifact

with unknown origin would be retrieved from the remote Maven repository.

Besides being Maven specific, this approach has the downside of wasting resources in terms of disk space and build time. Maven uses the local repository to avoid retrieving dependencies from remote locations, which is a lengthy operation. When using custom local repositories for each branch, they would have to be filled with artifacts from the remote location on first run, which is an infamously slow operation[32].

The result would be that we have duplicate dependency archives in each repository, which take up disk space and significantly slow down the first build. If the build is ran on different nodes, the impact quickly escalates, because the branch local repository has to be constructed on each node.

2. *Timestamp dependency resolution* - Currently when Maven is deploying a snapshot archive version to the remote location, it saves the build timestamp along with the snapshot version. Meaning a version defined as "1.0-SNAPSHOT" could actually be retrieved as "1.0-(buildTimestamp)", where (buildTimestamp) is for example "20130424.145342-1".

This functionality could be used by always deploying the component dependencies after building them and remembering the build timestamps. When building the main application the version of each component dependency could then be replaced with the build timestamp and thus Maven's own dependency resolution mechanisms will guarantee the propagation of the right artifacts.

Conclusions

Continuous Integration and feature branches are often seen as mutually exclusive software development methodologies, but they are both useful in their own ways. Because of the inherent clash between the methodologies and the fact that feature branches can often be replaced by a different approach, little work has gone into investigating how to work with the two methodologies together.

In this work, we've introduced the reader to the practices of Continuous Integration and feature branches, giving a short introduction to both and proving their usefulness. We've clearly defined the problems arising when using the two methodologies together in terms of component dependencies and we've provided a unique solution to the stated problems using the Jenkins CI server.

Though feature branches can be replaced with other techniques, it's not always the most practical decision. In many cases using feature branches is a much simpler and quicker way of developing than the alternatives. Because of the small amount of interest in this domain, there's currently little support for using feature branches with the open source Jenkins CI server.

Our solution to use those methodologies together comes in a form of a new plugin in combination of some existing plugins. It's as flexible as possible, being platform and programming language agnostic while still allowing for parallel execution of tests and supporting multiple repositories with any number of overlapping branches.

The proposed solution successfully solves the problems between using Continuous Integration and feature branches together with the Jenkins CI server, but it's not perfect. Mainly the solution's usability aspects could be improved. Some of the possible improvements are purely technical challenges, while others would require collaboration and consent from developers maintaining the Jenkins project. We also provided some ideas for possible alternative solutions, which we hope will be investigated in our (or the reader's) future works.

Lähtekoodi pidev integratsioon funktsionaalsete harudega arendusmudel

Tõnis Pool

Bakalaureusetöö

Resümee

Lähtekoodi pidev integratsioon ja funktsionaalsete harudega arendusmudel on pealtnäha teineteist välistavad tarkvara arenduse metodoloogiad. Ometi on nad mõlemad omal moel kasulikud. Selles töös uurime, kuidas siiski kasutada neid kahte metodoloogiat koos nii, et mõlemad säilitaksid oma kasulikkuse. Töö keskendub avatud lähtekoodiga pideva integratsiooni serverile Jenkins ning Mercuriali versioonihaldussüsteemile.

Töö eeldab lugejalt tarkvaraarenduse baasteadmiste, sealhulgas versioonihaldus- ja tarkvara ehitus/pakendamise süsteemide põhimõtete, tundmist. Töö üldosa ning pakutud lahendus on platvormist ja programmeerimiskeelest sõltumatu, kuid põhineb kohati Java programmeerimiskeelel ning täpsemalt Apache Maveni ehitussüsteemil.

Lähtekoodi pidev integratsioon tähendab arendajate kirjutatud lähtekoodi igapäevast omavahelist integreerimist, mille aluseks on ühes versioonihaldussüsteemi harus arendamine. Funktsionaalsete harude arendusmudel seisneb aga selles, et iga süsteemi funktsiooni arendamiseks luuakse uus ja eraldiseisev versioonihaldussüsteemi haru. Selles olev lähtekood hoitakse eraldi kuni funktsiooni valmimiseni.

Sellest tulenevalt on näha nende metodoloogiate fundamentaalne ebakõla, kus üks seisneb lähtekoodi isoleerimises eraldi harudesse, teine aga eeldab ühe ühise haru kasutamist. Leitud probleemi ulatuse defineerime tarkvara komponentsõltuvuste põhjal. Komponent sõltuvus selles töös tähendab projekti sõltuvust koodibaasile, mis paikneb eraldi seisvas lähtekoodi hoidlas, kuid on siiski projekti osa.

Käesolevas töös kirjeldame, kuidas antud probleemi on üritanud lahendada Bamboo nime-

line pideva integratsiooni server ning mis meetoditega on seda võimalik hetkel lahendada Jenkinsis. Ometi on olemasolevad lahendused mitmel moel ebasobilikud ning ükski neist ei lahenda ühte meie defineeritud probleemi. See probleem ilmneb funktsionaalsete harudega arendusmudeli ja lähtekoodi pideva integreerimise kasutamisel projektides, kus on vähemalt üks komponentsõltuvus.

Meie lahendus tõstatatud probleemile seisneb uue Jenkinsi plugina loomises, mis töötab koos paari eksisteeriva pluginaga. Loodud plugin paikab olemasolevat Mercuriali pluginat. Samuti loob plugin uue triggeri, mis otsib uuendusi versioonihalduse kõikidest harudest ning lisab uue järgmise sammu, mille abil saab dünaamiliselt muuta töös olevat versioonihalduse haru.

Kasutades meie enda ja olemasolevaid pluginaid, loome konfiguratsiooni, mis lahendab kõik töös väljatoodud probleemid. Lahendus on nii paindlik kui võimalik, lubades piiramatut arvu funktsionaalsete harude kasutamist eri komponentides. Sealjuures ükskõik kui paljud neist võivad hõlmata mitut komponenti korraga.

Kuna lahenduseks on üsna keerukate seadistuste loomine, siis peaaesjalikult võiks edaspidi lihtsustada seadistuste tegemist ning parandada kasutusmugavust. Töö lõpus pakume välja ka paar alternatiivset lahendust, mida loodetavasti uuritakse kas autorite endi või mõne lugeja käe läbi.

Bibliography

- [1] “Apache Maven.” <http://maven.apache.org/>, April 2013.
- [2] M. Fowler, “Continuous Integration.” <http://martinfowler.com/articles/continuousIntegration.html>, May 2006.
- [3] K. Beck, *Extreme Programming Explained: Embrace Change*. O’Reilly Media, Inc., first ed., October 1999.
- [4] T. Romer, J. Timosin, P. Grigorenko, K. Kapelonis, R. S. James, and O. White, “Why developers love CI: A guide to loving Continuous Integration,” tech. rep., ZeroTurnaround, January 2013.
- [5] M. Fowler, “FeatureBranch.” <http://martinfowler.com/bliki/FeatureBranch.html>, September 2009.
- [6] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, “What we have learned about fighting defects,” in *Proceedings of the 8th International Symposium on Software Metrics*, METRICS ’02, (Washington, DC, USA), pp. 249–, IEEE Computer Society, 2002.
- [7] J. Holck and N. Jørgensen, “Continuous Integration and Quality Assurance: a case study of two open source projects,” *Australasian J. of Inf. Systems*, vol. 11, no. 1, 2003.
- [8] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, first ed., August 2010.
- [9] A. Bayer, “Hudson’s future.” <http://jenkins-ci.org/content/hudsons-future/>, January 2011.
- [10] “Jenkins.” <http://jenkins-ci.org/>, April 2013.
- [11] “Distributed builds.” <https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds/>, April 2013.
- [12] “Configuring plans.” <https://confluence.atlassian.com/display/BAMBOO/Configuring+plans/>, April 2013.

- [13] “Bamboo.” <http://www.atlassian.com/software/bamboo/overview/>, April 2013.
- [14] J. Dumay, “Making Feature Branches Effective with Continuous Integration.” <http://blogs.atlassian.com/2012/04/bamboofeature-branch-continuous-integration-hg-git/>, April 2012.
- [15] S. Phillips, J. Sillito, and R. Walker, “Branching and merging: an investigation into current version control practices,” in *Proceedings of the 4th International Workshop on Cooperative and Human Aspects of Software Engineering*, CHASE ’11, (New York, NY, USA), pp. 9–15, ACM, 2011.
- [16] A. Babenhausenheide, “Mercurial Workflow: Feature seperation via named branches.” <http://draketo.de/light/english/mercurial/feature-seperation-via-named-branches/>, May 2011.
- [17] E. T. Barr, C. Bird, P. C. Rigby, A. Hindle, D. M. German, and P. Devanbu, “Cohesive and isolated development with branches,” in *Proceedings of the 15th international conference on Fundamental Approaches to Software Engineering*, FASE’12, (Berlin, Heidelberg), pp. 316–331, Springer-Verlag, 2012.
- [18] P. Weissgerber, D. Neu, and S. Diehl, “Small patches get in,” in *Proceedings of the 2008 international working conference on Mining software repositories*, MSR ’08, (New York, NY, USA), pp. 67–76, ACM, 2008.
- [19] J. Humble, “On DVCS, continuous integration, and feature branches.” <http://continuousdelivery.com/2011/07/on-dvcs-continuous-integration-and-feature-branches/>, July 2011.
- [20] P. Hammant, “Introducing Branch By Abstraction.” http://paulhammant.com/blog/branch_by_abstraction.html/, April 2007.
- [21] “Introduction to the Dependency Mechanism.” <http://maven.apache.org/guides/introduction/introduction-to-dependency-mechanism.html>, April 2013.
- [22] “Introduction to Repositories.” <http://maven.apache.org/guides/introduction/introduction-to-repositories.html>, April 2013.
- [23] “Using plan branches.” <https://confluence.atlassian.com/display/BAMBOO/Using+plan+branches/>, April 2013.
- [24] G. Valkov, “Jenkins-Autojobs documentation.” <http://gvalkov.github.io/jenkins-autojobs/>, April 2013.
- [25] “Jenkins Build Per Branch.” <http://entagen.github.io/jenkins-build-per-branch/>, April 2013.

- [26] “Nested View Plugin.” <https://wiki.jenkins-ci.org/display/JENKINS/Nested+View+Plugin>, May 2013.
- [27] R. Licht, “Create Multijob in Jenkins using Tikal Multijob Plugin.” <http://www.tikalk.com/java/forums/create-multijob-jenkins-using-tikal-multijob-plugin/>, September 2011.
- [28] “Throttle Concurrent Builds Plugin.” <https://wiki.jenkins-ci.org/display/JENKINS/Throttle+Concurrent+Builds+Plugin>, April 2013.
- [29] “Sonatype Nexus.” <http://www.sonatype.org/nexus/>, May 2013.
- [30] “Artifactory.” http://www.jfrog.com/home/v_artifactory_opensource_overview, May 2013.
- [31] “Rebuild Plugin.” <https://wiki.jenkins-ci.org/display/JENKINS/Rebuild+Plugin>, May 2013.
- [32] T. O’Brien, “How not to download the internet.” <http://java.dzone.com/articles/how-not-download-internet>, April 2011.

Non-exclusive licence to reproduce thesis and make thesis public

I, Tõnis Pool
(author's name)
(date of birth: 07.02.1991),

1. herewith grant the University of Tartu a free permit (non-exclusive licence) to:
 - 1.1. reproduce, for the purpose of preservation and making available to the public, including for addition to the DSpace digital archives until expiry of the term of validity of the copyright, and
 - 1.2. make available to the public via the web environment of the University of Tartu, including via the DSpace digital archives until expiry of the term of validity of the copyright,

Continuous Integration & Feature Branches,
(title of thesis)

supervised by Toomas Römer, Rein Raudjärv and Vambola Leping,
(supervisor's name)

2. I am aware of the fact that the author retains these rights.
3. I certify that granting the non-exclusive licence does not infringe the intellectual property rights or rights arising from the Personal Data Protection Act.

Tartu, **13.05.2013**