Continuous Integration with Jenkins

Coaching of Programming Teams (EDA270)

J. Hembrink and P-G. Stenberg
[dt08jh8 | dt08ps5]@student.lth.se

Faculty of Engineering, Lund Univeristy (LTH)

March 5, 2013

Abstract—Developing software in teams using agile methods often requires new code to be shared within the team and integrated with existing code at a regular basis during development. This can be very time consuming as the integration of new code must be verified by building and testing the system for each integration. *Continuous Integration* is a practice and a part of the agile development method *Extreme Programming* that involves automation of the building and testing of new code and thus reducing the time spent by the developers on integrating their work. In this study we try to implement the practice of continuous integration into an agile student project in the course *Software Development in Teams (EDA260)* at the Faculty of Engineering, Lund University, hoping to improve the efficiency of integration of tasks as well as improving code quality. The implementation has been implemented with the use of the tool *Jenkins*, a popular tool for continuous integration, supporting automation of building, testing and more. Some common terms and theories of both continuous integration and the tool used is explained and is then followed by results and conclusions following the practical implementation.

Index Terms—Continuous Integration, agile development, test-driven development, eXtreme programming, Jenkins, LTH

1 Introduction

When developing software in a team using agile methods a team of developers have collective code ownership, sharing the code in a common repository where each developer checks in new code or changes whenever the developer is done with the task. This requires each check in of new functionality to be integrated in the existing code base—something that involves building and testing the system to be able to verify that the system still works with the new changes. Performing the integrations at the end of a development cycle increases the risk of conflicts and thus increasing to cost to solve the integration problems.

Continuous Integration (CI) is a practice where each integration made by a developer is verified by an automated build, a complete test run and other reports providing feedback about the state of the software or code. This makes errors in the integrations rapidly and easily detectable and thus reducing larger problems later in the development cycle.

Continuous Integration was first minted as a part of the rule of *integrate often* [1] in Extreme Programming (XP), a set of activites, rules and principles for agile development focusing on frequent releases in short development iterations. The practice of CI states that developers should integrate code whenever it is possible, allowing integration problems to be detected as early as possible. Developers should never have unintegrated code for more than a day following the rules of XP.

While a tool is not needed to apply the practice of continuously integrating, an inte-

gration server is often used for this purpose, making the automation easier. In this study we used *Jenkins* [2], a widely used CI server. A description of what Jenkins is will be handled in section 3.

This study is a part of a course at the Faculty of Engineering, Lund University where students in groups of eight to ten develop a project in an agile manner, XP to be exact. We, the authors are acting as coaches to these team of students. We will then try to coach the team of students to practice CI with the support of Jenkins. In section 5 we will compare our team's performance with other team not focusing on CI.

2 Practices of CI in the Course

The practice of CI has a set of sub-practices related to those of XP defined by Fowler and Foemmel [3]:

- *Maintain a Single Source Repository*
- *Automate the Build*
- *Make Your Build Self-Testing*
- *Everyone Commits To the Mainline Every Day*
- *Every Commit Should Build the Mainline on an Integration Machine*
- *Keep the Build Fast*
- *Test in a Clone of the Production Environment*
- *Make it Easy for Anyone to Get the Latest Executable*
- *Everyone can see what's happening*
- *Automate Deployment*

However we adapted some of these practices to suit the course environment used in this study, making some of the practices redundant. For example *Everyone Commits To the Mainline Every Day* becomes redundant since there is only one day of development per week, the developers should instead commit as soon as a story is implemented. See section 4.1 about the student project and the course. In the sections 2.1–2.7 our practices are described.

2.1 Using an Integration Machine

One approach to verify the integration of new code is for the developers to manually build the system and make sure all tests pass on a build server before declaring the change integrated.

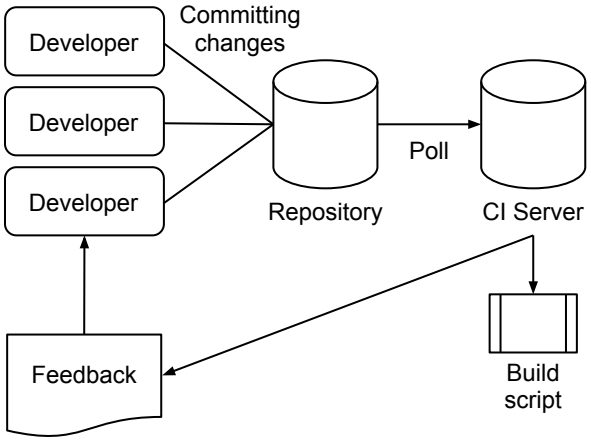

Fig. 1. An overview of the process of CI as defined in the student project

This can be automated with an CI server which monitors the repository for changes and if it detects any builds and run the tests. The CI server then provides the developers with feedback about the status of the build, whether the build was successful or not. In the case of a build failure the developers can quickly solve the problem and commit a fix.

The use of the CI server can be seen in figure 1. The developers commit changes to the repository. The CI server polls the repository on a regular basis, if it detects any changes commited it will execute the build script that automatically builds the system. It will then publish feedback telling the developers if the build and tests was successful or not.

2.2 Maintain a Single Source Repository

When using agile methods to develop software—XP in particular—it is important to have a collective code ownership [4]. Software configuration management (SCM) tools is an important part of this. One kind of SCM tools widely used is version control (VC) tools. The VC tool used in the project, Subversion, helps developers keep all source code files in one place, a repository, with each change saved so that one can go back and trace changes made.

In the practice of CI it is essential that the team has one single master repository [1] [3] where all files needed to build the software is

in place, this also includes build scripts such as Ant [5] (a Java library used mainly to build Java applications described in build files) and test classes such as JUnit [6] test suites (JUnit is a framework for writing unit tests in Java). The developers should always maintain this repository first hand.

The master repository is also where the CI server looks for changes and base the build upon.

2.3 Automate the Build

To be able to verify the integrations commited to the repository by the developers, the CI server should be able to build the software by itself. This can be done with a build script, in our case an Ant-file which builds the Java application being developed. In addition to building, the build script also automatically runs all tests, code auditing tools and generates JavaDoc.

2.4 Test First and Automated Test Runs

Another way to help verify new commits is by running unit tests for all code. In extreme programming, test-driven development (TDD) is a vital part of writing new code and for CI [7]. Luckily this practice makes it easy to automate a complete test run.

The CI server can automatically run all unit tests and create a report for all developers in the same way as the build report, but more detailed as it can show exactly which tests failed and thus which methods failed. It can also provide detailed information about test coverage and warnings in the source code.

2.5 Frequent Commits

As mentioned previously in this paper and as a part of the other practices developers should commit often and as soon as new functionality is done (and of course, tested). Each commit should be done to the master repository so that the CI server can run the verifications and allow other developers to retrieve the latest revisions of the source code and build files.

2.6 Shared Artifacts

When executing the build script the CI server can create several artifacts. These includes executables of the program, test reports, documentation and more. Every developer should have access to these artifacts so that one can test run the compiled program, inspect reports or read up on the API.

2.7 Extreme Feedback

Communication is a corner stone in every agile project, this is especially true when using CI. It is important that all members of the development team know the state of the build of all time.

Every developer can access the build and also see all reports and the current build state through a web browser. It is also common to use one of many plugins for *extreme feedback* available for Jenkins. An example of this can be a monitor which lights up in different colours depending on the build state, for instance red if it fails and green if it's successful. Some teams even go one step further and utilises physical lamps or sounds.

3 A BIT ABOUT JENKINS

Jenkins is a continuous integration tool written in Java. It can execute Ant and Maven scripts and shell scripts both for Windows and Unix/Linux enviroments. Builds can generate JUnit test reports out-of-the-box while other test frameworks are supported via plugins. Supported SCM tools are: *CVS, Subversion, Git, Mercurial, Perforce and Clearcase*. Builds can be triggered by commits, polling the repository, manually triggers, via an API or by other successful builds, i.e. one can setup Jenkins to build one module only after another module has been successfully built.

Jenkins uses HTTP in the form of a web page as interface making both the builds, feedback and configuration available through a web browser as mentioned in section 2.7.

4 IMPLEMENTATION

In this section we describe how we, as coaches, went to introduce and implement CI into the student project.

The goal with impementing CI in this project was to see if team could have a release ready fast, at all time. Release management in this course is known, from own experience, to be a stressful moment. We think the reason for this is simply that CI has the least focus compared to all the other practices of XP during the theory part of the courses.

Antoher goal with the introduction of CI and Jenkins in particular was to give be able to monitor and give feedback to the team in the aspect of testing. TDD and unit testing has in the opposite of CI a large focus. By letting the team know if a test fail or if the test coverage is low after each commit, our hypothesis was that we could coach the team to be more aware of TDD and thus improving the validation the integrations.

4.1 The student project

The course *Software Development in Teams (EDA260)* is a compulsory course for computer science students at LTH. The course is divided into two parts—-one theory part and one project part.

The project part is—as brifely mentioned in the introduction—a part of a course and therefore a simulation of a real project where the students are divided into extreme programming teams developing a Java software for a fictional customer, usually played by an employee at the department of computer science. All twelve teams develops the same software.

The development is taking part in a time period of six weeks (iterations), each containing an eight hour "work day" on Mondays and a two hour planning meeting on Wednesdays. The work day—the programming sessions—is done with XP rules in mind, including (but not limited to) *Pair Programming, TDD and on-site customer*. The Planning meetings is a session where all developers, the coaches and the customer takes part, discussing and most of all estimating the development time of a set of predefined stories given to the team by the customer. After the estimation done by the team the customer decides what stories to go into the next release. Each student also gets a so called *spike*, a four hour task to do outside of the programming sessions, these spikes are not allowed to be implementations of any kind, but can constist of reasearch or refactoring.

During the theory part of the course the students have been taking laborations in using SCM tools like CVS or Subversion. They have also had lectures in topics such as SCM, Test-driven Development, Simple Design, Refactoring and XP.

The course *Coaching of Programming Teams (EDA270)* is a parallel course to EDA260 where previous students of that course play the role of coaching the team through the project.

4.2 Introducing CI in the Student Project

The first step of our implementation of CI was to give the developers a basic, bare minimum skeleton of the system, including some equally simple unit tests. This is given by all coaches to their respective teams.

As we knew from our own experiance the knowledge on build scripts such as ANT is very limited we also wrote a basic script which could compile the software and run the tests that we wrote. Just enough to get our Jenkins server running and actually performing the task of giving the team feedback on the builds and tests.

On the first programming session the Jenkins server was started and the students were given a short introduction on how to access and use Jenkins. Since no release was planned during the first iteration the Jenkins server was mainly used to monitor the build and test reports to make sure the integrations of implemented stories were successful.

The first release was planned to the second iteration, therefore the students were told to read up on Jenkins. They were also shown where to access the artifacts generated by the build script and published on the Jenkins server, making sure every team member knew how to get make a release.

On later iterations the team (with the coaches assisting) added more functionality to the build script and Jenkins. Some of the following Jenkins plugins were used, in no particular order of importance:

Ant Plugin. A plugin for building Java applications with Ant. This is installed out-of-the-box in Jenkins.

Javadoc Plugin. A plugin for publishing the Javadoc generated by the build script on the Jenkins web interface. Comes installed with Jenkins.

Subversion Plugin. This plugin enables Jenkins to poll a SVN repository for changes and trigger builds if changes are detected. Comes installed with Jenkins.

Green Balls. A simple plugin to replace the blue balls to green balls indicating successful builds.

Jenkins Emma Plugin. Emma is a tool for measuring test coverage in Java. The plugin publishes reports and graphs on the web interface.

FindBugs plugin. FindBug is static analysis tool, looking for bugs in Java programs. The plugin publishes reports and graphs on the web interface.

While some of these plugins is necessary to be able to use Jenkins with Java, plugins such as the Emma and FindBugs plugin was installed to give additional feedback about code quality et cetera.

4.3 Using Extreme Feedback

A way for developers to quickly see whether a build is successful or not is to display clear, simple feedback to the developers. In our environment we had a computer screen set up so each developer could see it clearly at all time, running a full screen application simply making the whole screen turn red for builds failing, yellow for unstable builds—meaning the build was completed but with failing tests and finally green for successful builds.

Having this external screen allowed the developers to see the outcome from Jenkins without having to browse to the web interface on their own workstation making them aware of failing or unstable builds even while in their normal workflow.

4.4 Improving Test Coverage

Covering as much code as possible with the tests is essential to make sure all modules are integrated.

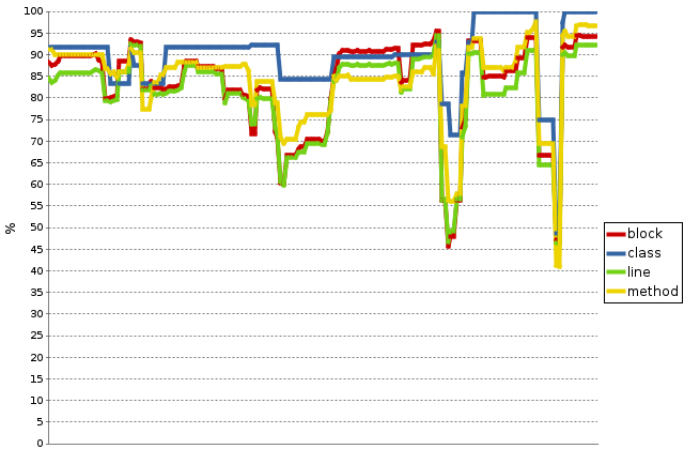

Fig. 2. Test coverage for each build for the project

We used the tool Emma to measure the test coverage of the code together with the plugin for Jenkins to publish the reports making them available to developers. A threshold can be configured to make the build considered unstable or failed if the coverage percentage is too low. This makes the team aware of untested code which can be important when making large refactorings where new classes might go untested (it should of course not happen, but it does).

Figure 2 shows a graph over the test coverage in the project with percentage on the Y-axis and builds on the X-axis.

4.5 Improving Code Quality

Detecting bugs can be a hard task for any developer and some slip by the tests. Many of them can be found with static code analysis, such as comparing strings in Java without calling the equals-method.

We introduced FindBug as a static code analysis tool to be integrated into our build script and in the Jenkins server as a plugin. Many of the faults found by the tool is merely Java specific warnings about unused imports or wrongfully named methods according to the standard coding conventions. But some critical bugs as the one mentioned earlier can

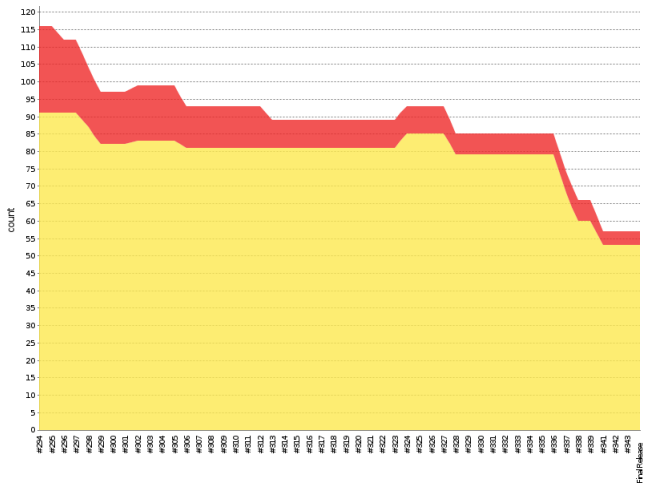

Fig. 3. Bugs and warnings found by FindBugs for each build in the project

be detected fast after each commit using the feedback published on the Jenkins server.

In figure 3 a graph is shown with different types of fualts found by FindBugs. The Y-axis is the number of faults found and the X-axis is the builds.

4.6 Managing the releases

Jenkins together with a build script can automatically package releases as soon as a build is considered successful. Anyone in the team can then get the latest package—artifacts—from the Jenkins server. The package can include binaries (runnable jar-files in our case), generated JavaDoc and any documentation residing in the repository. Everything that should be needed for a release in this course.

The process of a new release, including preparing for a release in our team is described below:

1. **Make sure everything is integrated.** While a release could be made by just taking the latest successful build, the team generally wants a release with as much new stories implemented as possible. Therefore a developer responsible for the release makes sure that the latest commits is integrated—and if not tells the team about it so they can fix it. This usaully takes place about an hour before the planned release making sure the team have time to fix problems.
2. **Check documentation.** Something that is required for every release in the course is the documentation, including technical documentations and user manuals. These are artifacts that cannot be checked automatically, there this is validated manually by a pair of developers. This is usually done in parallel with the point described above.
3. **Manual acceptance testing.** Automatic acceptance testing is done as a part of building but to ensure it works on an isolated environment the release is downloaded from Jenkins and tested manually with the binaries. This is especially important since the development environment is different from the one where the product is being used. This also ensures the now up-to-date manual is correct. This is done just about 30 minutes before the release to make sure there is time to correct any errors.
4. **Sending the release.** The actual release is done by downloading the release package from the Jenkins server and mailing it to the customer. This is done as soon as the release is verified to work and can be done in minutes.

5 Results

The result of implementing CI with Jenkins in the course project shows that there were a relatively small amount of failed builds and tests during the entire development. The releases were generally good and fast with most problems being missing documentations or “usage bugs” meaning undetected bugs caused by the user, for example providing wrong parameters in the configuration file.

Jenkins was used continuously used by all members to validate their commits during the development sessions. In the small amount of occurances where the build failed it was clearly communicated to the rest of the team both via the feedback monitor and vocally by the developer responsible for the commit.

Looking at figure 4 a large drop in the number of tests can be seen after about halfway

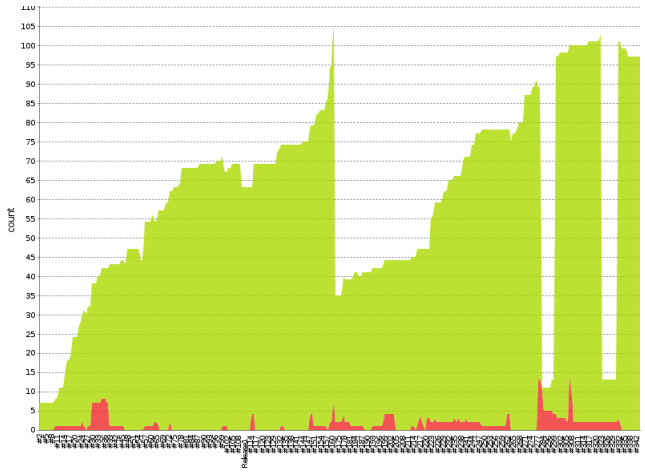

Fig. 4. Graph showing the trend of both successful and failed builds

through the list of builds. This was just after a large refactoring. The drop can be explained both by the reduced amount of classes but also by the team "cheating" with the practice of TDD in the refactoring. We can observe that more tests were written gradually after discovering this. Figure 2 also confirms this as in the same period the test coverage of classes is reduced a bit while the method, line and block coverage drops significantly more than the class coverage, meaning there was some removed classes, but the new methods went untested.

5.1 Other Teams

A survey was sent to all other coaches asking them if they introduced CI to their team, how their release process looked like and how long it took.

The result showed that most of them had some kind of CI practice but none of them used a CI server. Instead they manually invoked an Ant script that built the system, one team did semi-automate this by having a shell script run periodically. As this method works, a shared, automated feedback from the build is missing.

On the question about release processes we found out that the teams having build scripts had a process very similar to ours, but with the difference of having Jenkins available to provide easy access to the builds. Some teams had instead automatic emailing of the builds to a team mailinglist.

One team answered that they did not automate the build, but instead produced the build through the IDE. This method works but requires that the developer making the release follows a checklist with procedures such as testing and packaging the release correctly. Feedback for the rest of the team is also missing as they have to rely on the responsible person to communicate with them.

The time taking to do the release varied from one minute to up to an hour. Not surprisingly the team with Continuous Deployment could release in a minutes with the more manual methods taking longer. We assume however that these numbers might not be all fair as it did not state whether to take the preparations in consideration or not. Our team for example could release in minutes but up to an hour was dedicated to the release to manually validate for example documentation as explained in section 4.6.

5.2 Problems

The first noticable problem we found was lack of knowledge of testing. Some developers did not know how to test exceptions in JUnit and automating the acceptance testing was also a problem. The problems were resolved by letting team members educate each other after spikes.

In figure 4 and 2 we can se two dramatic drops in the graphs. These were caused tests invoking a system exit. This was detected fast thanks to the feedback from Jenkins but could have been avoided all together with better knowledge on how to test these types of code.

Other problems was mainly caused by the team not following the practices of XP. Committing broken code occured at some points even though it is a bad practice and showed up on Jenkins. In some cases the commits were due to human erros in the use of SVN, but sometimes intentionally before the end of the day. Going home with broken code should of course never happen, but could be fixed the week after quickly as Jenkins showed it to the

team. Since it was a course project with only the ten developers developing the software one day per week the harm was not major, but could potentionally be devastating in a larger project.

6 Conclusions

Continuous integration is an important practice in agile projects, XP in particular. It can help increasing the frequency of releases, in making releases more managable and improve the quality of the product and the code behind it. However it requires that the team of developers follow not only the practice of CI, but all it prerequisite practices aswell, as described in section 2.

Jenkins is an excellent, highly customisable CI server with many plugins to tailor the tool for specific needs. The web interface makes it accessible to all developers, a prerequisite of CI.

Continuous integration with Jenkins is highly suitable for this course as the result showed that it improved test coverage and kept the number of failed builds down during the whole development cycle. It does however require the coaches to actively support the team in maintaining the tools, for example we decided early to develop a build script for the team at the start, something that could have been done by the team, but we think giving them a jump start gave the team more focus on using the tools rather than spending time configuring the tools by themselves.

Acknowledgment

We would like to thank our team of students in EDA260, Team 09 VT13, our fellow coaches in EDA270 and the lecturers.

References

[1] J. Humble and D. Farley, *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education, 2010.

[2] K. Kawaguchi, "Jenkins CI," http://jenkins-ci.org/, [Online; accessed 03-Mar-2013].

[3] M. Fowler and M. Foemmel, *Continuous Integration*. ThoughtWorks, 2006.

[4] K. Beck and C. Andres, *Extreme Programming Explained: Embrace Change*. Addison-Wesley Professional, 2004.

[5] "Apache Ant," http://ant.apache.org/, [Online; accessed 03-Mar-2013].

[6] K. Beck and D. Saff, "JUnit," http://junit.org/, [Online; accessed 03-Mar-2013].

[7] M. Karlesky, G. Williams, W. Bereza, and M. Fletcher, "Mocking the embedded world: Test-driven development, continuous integration, and design patterns," in *Proc. Emb. Systems Conf, CA, USA*, 2007.

[8] M. Polo, S. Tendero, and M. Piattini, "Integrating techniques and tools for testing automation," *Softw. Test., Verif. Reliab.*, vol. 17, no. 1, pp. 3–39, 2007.