Continuous Integration: Aspects in Automation and Configuration Management

Christian Rehn

TU Kaiserslautern, 67663 Kaiserslautern, Germany,
c_rehn@cs.uni-kl.de
Seminar supervisor: Bo Zhang

Abstract. One of the key practices in agile methods is continuous integration (CI). Instead of integrating the software in a separate phase at the end of an iteration CI requires all developers to integrate, build and test at least once a day resulting in several integrations per day. In this way integration problems are detected early in the software development process and thus can be fixed more easily. CI demands a high degree of automation and also has effects on software configuration management (SCM). This paper discusses CI, which issues have to be considered for automation and which effects it has on SCM.

Keywords: Continuous Integration, CI, Automation, Software Configuration Management, SCM, Agile, Extreme Programming, XP

1 Introduction

Writing software is difficult and writing bigger software is even more difficult. So just writing some code is not enough to produce quality software. The "code and fix" approach does not scale and led to what is called the software crisis. In order to solve this problem software engineering and the plan-driven software development approaches arose. They work quite well for some kinds of projects but have problems when requirements remain fuzzy and change frequently. For this reason agile processes came up, which are much more able to respond to changing requirements. [15] [12]

Agile methodologies set much less value on planning and documentation in favor of lean and responsive processes that are able to react to changing requirements quickly. But in order to compensate for the lack of planning and documentation several practices have to be adopted. So problems which normally would have been avoided by planning must be detected early in the development process in order not to have a bad influence on productivity, schedule, and quality. One of these practices is *continuous integration* or CI for short.

Extreme programming explicitly demands these practices to be in place [12] whereas other agile processes like Scrum don't mention them. Nevertheless they are necessary for these processes to work. This raises the question how important these practices are, which effects they have, and in which environments and

projects they should be applied. Apart from agile methodologies some of these practices might also be valuable for traditional plan-driven development.

Although CI is one of the least controversial XP practices the there is the question on how valuable CI is for agile and plan-driven software development and what has to be considered when adopting this technique. Two aspects of CI are discussed in the following. First of all automation is the key-aspect of CI as CI is all about automating the build process and getting rapid feedback whether something went wrong. And the second aspect which is addressed here is configuration management and branching in particular. Adopting CI implies the use of certain strategies for branching and avoiding others. These aspects will be discussed later, but before that the basics of CI are described.

2 Basic Idea of CI

2.1 Traditional Integration

Traditional waterfall-like processes typically have a separate phase "integration and test" after the implementation phase. After all the modules of the software have been implemented, they have to be assembled and tested in order to find out which problems remain despite all planning.

This leads to some problems and these problems even increase in agile processes where planning, i. e. upfront-design, is reduced to a minimum. Having integration late in a project (or in an iteration) means that certain faults will be detected late. There might be technical problems due to misunderstandings of a certain technology, interfaces of independently developed components not fitting together, and defects because of wrong assumptions and misunderstandings between developers. This makes the effort of the integration phase hard to predict and leads to late projects. The later defects are found the more expensive they are.

In plan-driven processes these effects are mitigated through careful planning and thorough inspections. Agile processes on the other hand do integration and testing early and not in a separate phase. In this way faults are detected just after they are introduced which also decreases the time to locate and fix the cause of the problem.

2.2 Continuous Integration

A key principle of agile thinking is doing everything in small and easy steps but continuously. Development is done in small iterations, estimation is done for small amounts of work, refactoring is done in small steps, etc. Martin Fowler calls this principle "frequency reduces difficulty" [19] which essentially is an specialization of the well known "divide and conquer".

Transferred to integration and test this means that integration and test is done very often meaning continuously throughout the development process. Every developer integrates at least once a day which results in several integrations

per day as there are several developers. In order to make this feasible integration is completely automated. A commit to the source code repository triggers compilation, automated tests, and automated feedback on the success of the integration. [13]

Technical problems are detected early on, as integration tests will reveal them. Integration tests will also find design defects and the fact that everyone commits to the same repository improves communication among the developers and thus avoids misunderstandings.

Furthermore, maintaining an always integrated, always quality assured code base assures that development is on the right track. A demo of the current project state can always be shown to customers and managers which enables feedback from the customer side and makes progress of the development visible. So CI is also a technique to reduce risk. Andrew Hunt and Dave Thomas call this risk reduction perspective of the technique "tracer bullets" [21].

As many defects are found just after they are introduced, they are easier to locate and remove. Developers will remember easier what they have done and they can also use the version control system to show the altered code lines which most likely contain the fault ("diff debugging"). Cumulative defects, i. e. several faults influencing each other and producing more complicated failures, are avoided which decreases the time needed for rework. [16]

In order to achieve these positive effects, there need to be rapid feedback mechanisms in place, which tell the developers if recent changes "broke the build", that means if defects have been introduced. Hence integration must be fast and thus automated. Furthermore, the result of the integration (meaning compilation, tests, etc.) needs to be made visible. This can be done by automatic emails or large monitors but also via lava lamps or ambient orbs (a ball shaped device connected to the network that can glow in different colors) [13]. Research suggests that a combination of informative emails and visible but unobtrusive devices such as lava lamps is well suited for agile teams. [10]

2.3 Integration Strategy Selection

CI has advantages and disadvantages, so it's not a replacement for traditional integration in general. Which integration approach is best, depends on the project, it's characteristics, and it's environment.

Obviously CI fits to agile processes like XP. Moreover, XP demands continuous integration to compensate for the lack of planning. Other agile processes don't mention or demand CI directly. Indirectly though there is the need for finding defects early. And because there is only a minimum of planning, a technique like CI is necessary. So if a project is done in an agile way, CI is the integration approach to choose.

But also for traditional plan-driven projects, CI can be beneficial. Planning can avoid many faults upfront, but there will inevitably be some defects slipping through. CI can be used for finding even more faults early on which reduces the effort later in the project and thus may improve predictability. Nevertheless

there is a tradeoff between the benefit of finding and correcting defects fast and the additional effort setting up and maintaining the CI environment. There also might be some learning effort for the developers to accommodate with the new technique.

Certainly CI only pays off for projects of a certain size. For very small projects the setup effort for a full-fledged CI environment may be considered too high compared to the benefit. So here a more traditional integration approach should be used. Nevertheless automation can be used gradually. So if a complete CI environment is too costly, it is possible to automate a part of the build process resulting in a lower benefit but also a lower effort. The same can be be done for legacy projects. CI can be introduced gradually slowly shifting from a more traditional integration approach to full-fledged CI.

Scaling the integration technique to very large projects is difficult—both with traditional integration and CI. The problems with traditional integration increase with the project size leading to even more difficult integration phases. On the other hand also continuous integration gets harder. More developers are involved and more commits are made which raises the possibility for broken builds. Also build times increase with the code size. This leads to fewer and bigger integrations and the advantage of CI disappears.

Normally agile projects are scaled down instead of the techniques being scaled up. That means instead of employing many developers, only a small team of smart people works on the product. Because of their higher productivity, a less bureaucratic process, and a simpler design the overall productivity is not influenced that badly. [14] Nevertheless if it is necessary to have big agile projects, there are also techniques for scaling CI [23]. One is to have sub-teams on loosely coupled subsystems doing CI only for their subsystem.

As now the basic idea of CI has been explained, in the next two sections two aspects of CI namely automation and configuration management will be discussed in more detail.

3 Automation

3.1 Automated Build

Without automation the integration process is time-consuming and error-prone. Doing integration very often when there is no or only insufficient automation in place is infeasible and would result in poor productivity and bad quality.

Since manual tasks are prone to errors they should be eliminated. Ideally in a CI environment a commit to the source code repository triggers a prior update from the repository, a compilation, database setup, several forms of tests, automatic code analysis, deployment to a production-like system and gives the developer a rapid feedback if there are problems with the commit. All this together is called a *build* in CI, so "build" not only means compile but all automated tasks relevant for finding defects early. [13]

Building daily came up in the 90ies and was practiced at Microsoft for example. Building the software every day was a big deal those days and having the developers commit every day still seamed impractical so the daily build only included new code because several developers integrated every few days. [22]

What was considered hard fifteen years ago is now a common practice. These daily or "nightly" builds are now adopted widely and several integrations per day are not regarded infeasible anymore. Also the number of activities that are included in a build has changed. In the past only a "smoke test" was conducted. Such a smoke test only determines if the system "smokes" when it is run, i. e. if there are obvious problems. As now the computational power of computer systems has increased, more thorough tests can be included in a build. Beside more thorough testing also static and dynamic analysis can be conducted to find further problems or check adherence to coding guidelines.

The purpose is really to find as many defects as possible automatically in order to reduce the time between introducing, detecting, locating and correcting the fault. The longer this time gets the more difficult the following tasks will be. And automation is the key to keeping builds fast. And also the feedback mechanism must make sure that "the right information [gets] to the right people at the right time and in the right way" [13, p. 205].

Keeping build time fast is easy for small and simple projects, but it is hard for large projects. Even today it is infeasible to run all tests on a project with a million lines of code every time a developer checks in some code. Thus there are typically several types of builds.

Private builds run on the developer machine prior to a commit on order to reduce the probability that the commit introduces defects influencing other developers. Then a commit-triggered build is done on a separate integration machine possibly followed by some secondary builds that include more time-consuming tests. These secondary builds can also be done time-triggered at night. Finally there might be some tasks which are that costly that they are done even more infrequently, i. e. at the end of a iteration as a kind of release build. By shifting more complex tasks to the more infrequent build types, build time can be kept small. In this way you can trade off efficiency for effectiveness of the build process. [13] [16]

3.2 Tools Used for Automation

Over the past years several tools were developed which help to automate the aforementioned integration tasks. In principle automation could also be done using simple shell scripts, but these tools make automation easier, portable and flexible.

First of all there are test automation tools like JUnit [7] which help automating all kinds of tests. Despite the name JUnit and others are not only able to automate unit tests but various kinds of tests.

In order to invoke compilation and testings there are build automation tools like Ant [1] or Maven [2]. These tools let you declaratively configure what has

to be done in order to integrate a software system. They are independent from integrated development environments and can be run without any user interaction.

Additionally there are specific CI servers like Hudson/Jenkins [6] or CruiseControl [4]. These tools automatically invoke the integration process (by using a build tool) when a commit to the source repository is done. Therefore, they monitor the source code repository for changes. After integration is done, they provide feedback, summaries, and statistics.

3.3 What to Consider When Automating

Software development always comprises a number of tradeoffs. The same holds for automation. What and how to automate depends on characteristics of the project, the organization, the process, and the environment.

A high degree of automation only pays off for projects of certain size and is especially beneficial for team-based development. Nevertheless even single developer projects can take advantage from automation. On the other hand automating a large project bears several challenges one of which is test time. Plan-driven projects can benefit from automation, but agile projects will even require a certain degree of automation in order to ensure a high quality of the product.

When automating tests, it is important to distinguish between unit tests (which only test one module) and component tests[1] (testing a set of modules). Unit tests need to be fast and isolated from other parts of the system. So there need to be proper test dummies ensuring isolation. Architecture also has an influence on that. A good architecture (using for example dependency injection) makes writing fast unit tests easy, a bad one hinders it. So architecture is very important for a proper build automation.

4 Configuration Management

4.1 Branching Models

Version control systems (VCS) like Subversion [3] are widely used in practice. They facilitate collaborative software development, versioning and archiving of source code and other kinds of files. They typically also let you creating branches meaning parallel versions of the same project. A branch can be created for developing certain features in isolation, for some bigger changes, for different platforms, for different releases, for different customers and so on. Development in different branches is parallel and isolated from each other. That means when some changes (e. g. a bug-fix) of branch A also need to be done on branch B, they have to be transferred (i. e. merged) to this branch. In figure 1 for example the branch “check due date” is merged to the mainline in commit C13. Despite

[1] In plan-driven processes component tests are called integration tests.

tool-support for merging, this task is easy for small changes but can be very difficult and error-prone for large ones.

The branching and merging features of VCSs can be used in various ways—some of which are beneficial and others which are hampering. So in order to use branching properly there has to be a sound branching model describing when and how to use branches for which purposes [24].

If there are multiple development branches, the integration process comprises merging these branches so that there is a revision containing all features that should be included in the end product. This means merging is an integral part of integration. In order to figure out whether two features interfere with each other, CI demands that every feature is merged to the mainline (i. e. the main branch) every day. This means that two features which should finally go into the same product may not reside in different branches.

4.2 Feature Branches

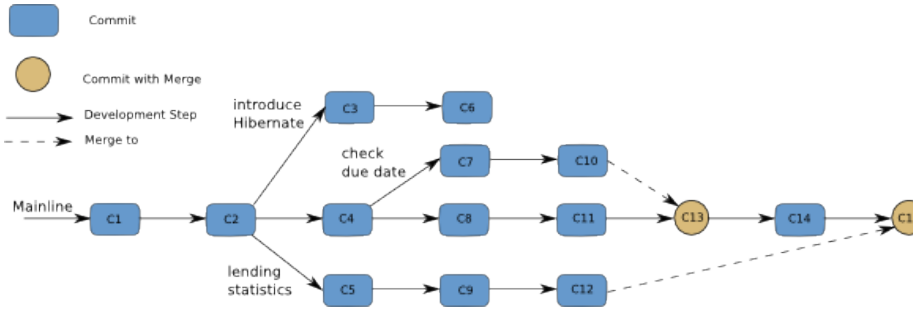

Fig. 1. A Repository containing one mainline and three feature branches.

A commonly used branching model is called *feature branch* [17] [11]. In this model for every new feature a branch is created and later merged into the mainline. There can also be commits to the mainline, but these are preferably minor changes and bug-fixes. This has the advantage of being able to easily select the features that should be included in the final release. Only these features are merged to the mainline. Incomplete and not thoroughly tested features remain in their feature branch and can be included in later releases.

Figure 1 shows an example of a repository of a library system containing a mainline and three feature branches. First the feature "check due date" is merged into the mainline (Commit C13) and later also the feature "lending statistics"—which obviously took longer to develop—is merged (Commit C15). The long-term migration task "introduce Hibernate" is still under development and not merged, yet.

Due to the private feature branch each developer works isolated from the others. There are no other commits on the feature branch disturbing the development of the feature. On the other hand this also means that integration is deferred till the feature branch is merged into the mainline. Over time the branches diverge and the integration becomes more and more complex.

In the example above merging of "check due dates" just had to consider C8 and C11. For merging "lending statistics" not only the new mainline commits C4, C8, and C11 have to be included but also C13 which contains C7 and C10. Even worse with the long-running task "introduce Hibernate" which is still not merged. When this feature will be merged into the mainline somewhen in the future, all other features which have been merged till then have to be considered. Inevitably some of these will interfere with the changes which had been developed unaware of the others.

The more feature branches there are, the more complex the integration gets. And due to the isolated development of the features, potential problems remain undetected until integration is done. So in order to make the feature branches model work, there has to be a considerable amount of upfront planning to avoid these problems. Obviously this does not fit to agile development. For this reason CI demands that every developer commits to the mainline and not to a private feature branch.

4.3 Feature Toggles

Normally it's better not to have pending features when the software is about to be released. Often this can be achieved by splitting up features into smaller features. Nevertheless there might be features which cannot be subdivided that easily so they take longer than one iteration to be developed. Hence there is the need for isolating features. As an alternative to feature branches *feature toggles* can be used for that purpose. [18]

Instead of separating incomplete features using version control systems they can be isolated by means of programming. In the simplest case this can be done by not including a button in the user interface so the feature cannot be invoked. There are also more sophisticated feature toggles such as compiler switches and switches in configuration files or command-line options. In this way incomplete features can be switched on and off.

In the example above the features "check due date" and "lending statistics" could be implemented right in the mainline. And a simple compiler switch could be used to disable these features by default as long as they are incomplete. Or even simpler: the GUI part could be developed last, so there is even no need for an explicit toggle.

Of course this means that incomplete and immature features will end up in the delivered system. Obviously this increases the risk of unintended side-effects and hidden defects. In order to keep the product quality high, excessive testing is used as common in agile processes. Furthermore, it is necessary to keep track

of the feature toggles and which features they control. After the feature is ready and has proven it's stability, the feature toggle has to be removed.

4.4 Branch by Abstraction

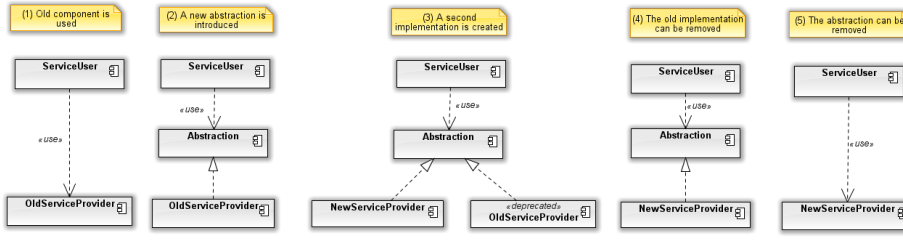

Fig. 2. A component is substituted with a new implementation through branch by abstraction.

Another alternative, which is especially helpful for long-running changes, is *branch by abstraction* (BBA). Instead of creating a second development branch in the VCS, doing the changes there and merging back again, the changes are separated by architectural means.

Long running changes like the migration from directly using JDBC to Hibernate typically have the problem that the system is not working for a long time. This can be avoided by creating a branch in the VCS, but a separate branch means by definition that these changes are not yet integrated into the mainline. This violates the principles of CI and imposes merging problems that would otherwise be avoided by CI.

Thus an abstraction of the functionality to be changed is created (see figure 2 (2)). After that the system is refactored such that it depends on the abstraction. In the next step a new implementation of the abstraction is created (3). The automated tests make sure that the new implementation can substitute the old one. Furthermore, there can be feature toggles that switch between the old and the new implementation. When the new implementation works as expected, the old one can be removed (4) and also the abstraction might be removed if it has no advantages (5). In this way there is a gradual change to a new implementation and the product remains working all the time. Furthermore, error-prone merging is avoided. [20]

In the "introduce Hibernate" example the `OldServiceProvider` would be the data layer using JDBC directly and `NewServiceProvider` would use Hibernate. `Abstraction` is then an artificial abstraction layer above the data layer. As such a further abstraction is uncommon and has no advantages, it is removed after the new implementation is in place.

Of course there are situations where such abstractions already exist. So they can be used without some artificial extra abstraction component and they also do not need to be removed in the end.

Architecture also plays a major role here. If there is a component-based architecture with clear interfaces maybe also using dependency injection frameworks like Spring [9], BBA is easy to use. But if coupling is tight, such changes become really hard.

4.5 Branching Strategy Selection

When CI is practiced, feature branches should be completely avoided as they contradict to the principles of CI. The key for doing so is a flexible architecture. The need for branches can often be avoided by separating concerns in to different modules. Dependency injection frameworks are helpful for doing so. For example platform-specific variants of a software do not have to be developed in separate branches. Instead there can be separate components for each platform. A configuration file then specifies which components to take for which platform.

If there are bigger changes that are not architecturally separated yet, BBA can be used and smaller changes can be separated by feature toggles. Furthermore, if the GUI integration of a feature is developed last, even feature toggles are not necessary as these feature remain implicitly toggled off until they are done.

4.6 Distributed Version Control Systems

Currently there is a change in version control systems. Apache Subversion [3] is still widely used, but it is foreseeable that distributed version control systems (DVCS) like git [5] and Mercurial [8] will gain more and more popularity. These systems are based on a different versioning model and are much more flexible than centralized systems.

This raises the question of how to use their new capabilities in CI and non-CI environments. Much of the flexibility comes from using directed acyclic graphs as a repository model instead of a linear history and more sophisticated merging capabilities. With git there is even the possibility to change versioning history. So the choice of the VCS directly has an influence on how to select the branching strategy. On the other hand the influence of the rise of DVCSs is still not completely understood and more research in this area is needed.

5 Conclusion

The traditional integration approach with a separate integration phase at the end of an iteration has problems and does not fit to agile software development. Continuous integration (CI) is the proper alternative to be used when developing agile. Setting up and maintaining a CI environment imposes some effort but

reduces risk and makes integration easier by integrating in small but frequent steps.

Even for plan-driven software development CI can be valuable as it helps finding defects early. This comprises a considerable amount of automation. On the other hand using CI or not is not a binary decision. There are plenty of possibilities to introduce CI stepwise. Especially automation can be applied gradually starting from just automatic compilation to full-fledged CI comprising automatic compilation, several forms of automated tests, automated static and dynamic code analysis, automated feedback, and even automatic deployment. There are also many valuable tools that help automating all that.

For configuration management it is important to think about a proper branching strategy. Feature branches are popular but don't fit well to CI. As a substitute, feature toggles and branch by abstraction may be used. But beside all that, it is important to notice that capabilities of modern VCSs are not a substitute for a good architecture. If the architecture is designed properly many branching problems are avoided. Additionally a good architecture also makes code testable and is, therefore, important for automation and CI in general.

References

1. Apache Ant, `http://ant.apache.org/`, [accessed 2011-12-03]
2. Apache Maven, `http://maven.apache.org/`, [accessed 2011-12-03]
3. Apache Subversion, `http://subversion.apache.org/`, [accessed 2011-12-03]
4. CruiseControl, `http://cruisecontrol.sourceforge.net/`, [accessed 2011-12-03]
5. git, `http://git-scm.com/`, [accessed 2011-12-03]
6. Jenkins, `http://jenkins-ci.org/`, [accessed 2011-12-03]
7. JUnit, `http://www.junit.org/`, [accessed 2011-12-03]
8. Mercurial, `http://mercurial.selenic.com/`, [accessed 2011-12-03]
9. Spring, `http://www.springsource.org/`, [accessed 2011-12-03]
10. Ablett, R., Maurer, F., Sharlin, E., Denzinger, J., Schock, C.: Build notifications in agile environments. Tech. Rep. 2008-888-01, Department of Computer Science, University of Calgary (2008), `http://pages.cpsc.ucalgary.ca/~denzinge/bib.html#reports`, [accessed 2011-12-03]
11. Appleton, B., Berczuk, S.P., Cabrera, R., Orenstein, R.: Streamed lines: Branching patterns for parallel software development (1998), `http://www.cmcrossroads.com/bradapp/acme/branching/line-elems.html#FunctionalBranch`
12. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley Professional (1999)
13. Duvall, P.M., Matyas, S., Glover, A.: Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional (7 2007)
14. Fowler, M.: Large agile projects (2003), `http://martinfowler.com/bliki/LargeAgileProjects.html`, [accessed 2011-12-03]
15. Fowler, M.: The new methodology (2005), `http://martinfowler.com/articles/newMethodology.html`, [accessed 2011-12-03]
16. Fowler, M.: Continuous integration (2006), `http://martinfowler.com/articles/continuousIntegration.html`, [accessed 2011-12-03]

17. Fowler, M.: Feature branch (2009), http://martinfowler.com/bliki/FeatureBranch.html, [accessed 2011-12-03]
18. Fowler, M.: Feature toggle (2010), http://martinfowler.com/bliki/FeatureToggle.html, [accessed 2011-12-03]
19. Fowler, M.: Frequency reduces difficulty (2011), http://martinfowler.com/bliki/FrequencyReducesDifficulty.html, [accessed 2011-12-03]
20. Hammant, P.: Branch by abstraction (2007), http://paulhammant.com/blog/branch_by_abstraction.html, [accessed 2011-12-03]
21. Hunt, A., Thomas, D.: The Pragmatic Programmer: From Journeyman to Master. Addison-Wesley Professional (1999)
22. McConnell, S.: Best practices: Daily build and smoke test. IEEE Software 13(4), 144, 143 (1996), http://www.stevemcconnell.com/ieeesoftware/DailyBuild.pdf, [accessed 2011-12-03]
23. Rogers, R.O.: Scaling continuous integration. LNCS 3092, 68–76 (2004)
24. Walrad, C., Strom, D.: The importance of branching models in SCM. IEEE Computer 35(9), 31–38 (sep 2002)