LAPPEENRANTA UNIVERSITY OF TECHNOLOGY
FACULTY OF TECHNOLOGY MANAGEMENT
DEPARTMENT OF INFORMATION TECHNOLOGY

# Introducing Continuous Integration for C and C++ Software Development Projects on Linux Platform

The topic of the master's thesis has been accepted by the head of the Department of Information Technology on October 21, 2009.

Supervisor: Professor Jari Porras

Examiners: Professor Jari Porras

M.Sc.E.E. Vedran Bartonicek

# ABSTRACT

Lappeenranta University of Technology
Faculty of Technology Management
Department of Information Technology

Jarkko Palviainen

**Introducing Continuous Integration for C and C++ Software Development Projects on Linux Platform**

Thesis for the Degree of Master of Science in Technology

2009

70 pages, 7 figures, 3 tables and 7 appendices

Examiners: Professor Jari Porras
M.Sc.E.E. Vedran Bartonicek

Keywords: continuous integration, build management, software engineering practices

Software integration is a stage in a software development process to assemble separate components to produce a single product. It is important to manage the risks involved and being able to integrate smoothly, because software cannot be released without integrating it first. Furthermore, it has been shown that the integration and testing phase can make up 40 % of the overall project costs. These issues can be mitigated by using a software engineering practice called continuous integration.

This thesis work presents how continuous integration is introduced to the author's employer organisation. This includes studying how the continuous integration process works and creating the technical basis to start using the process on future projects. The implemented system supports software written in C and C++ programming languages on Linux platform, but the general concepts can be applied to any programming language and platform by selecting the appropriate tools.

The results demonstrate in detail what issues need to be solved when the process is acquired in a corporate environment. Additionally, they provide an implementation and process description suitable to the organisation. The results show that continuous integration can reduce the risks involved in a software process and increase the quality of the product as well.

# TIIVISTELMÄ

Lappeenrannan teknillinen yliopisto
Teknistaloudellinen tiedekunta
Tietotekniikan osasto

Jarkko Palviainen

**Jatkuvan integraation käyttöönotto C ja C++ ohjelmistokehitysprojekteille Linux-ympäristössä**

Diplomityö

2009

70 sivua, 7 kuvaa, 3 taulukkoa ja 7 liitettä

Tarkastajat: Professori Jari Porras
M.Sc.E.E. Vedran Bartonicek

Hakusanat: jatkuva integraatio, koontikäännöksen hallinta, ohjelmistotekniikan käytännöt
Keywords: continuous integration, build management, software engineering practices

Integraatio on ohjelmistokehitysprossesin vaihe, jossa erillisistä komponenteista kootaan kokonainen tuote. Kyky integroida sujuvasti ja vaiheeseen sisältyvien riskien hallinta on tärkeää, sillä ohjelmistoa ei voida julkaista ennenkuin se on integroitu. On myös osoitettu, että integraatio- ja testausvaihe voi kattaa jopa 40 % projektin kokonaiskuluista. Näitä ongelmia voidaan pienentää käyttämällä jatkuvaa integraatiota.

Tämä diplomityö esittää, kuinka jatkuva integraatio voidaan ottaa käyttöön kirjoittajan työnantajaorganisaatiossa. Työssä tutkitaan kuinka prosessi toimii, ja luodaan tekniset edellytykset soveltaa sitä tulevissa projekteissa. Toteutettu järjestelmä tukee C ja C++ -ohjelmointikielillä kirjoitettuja ohjelmistoprojekteja Linux-ympäristössä, mutta työssä kuvattuja konsepteja voidaan hyödyntää ohjelmointikielestä ja -ympäristöstä riippumatta valitsemalla niihin soveltuvat työkalut.

Työn tulokset havainnollistavat yksityiskohtaisesti ongelmia, jotka on ratkaistava kun prosessi omaksutaan yrityskäytössä. Integraatiojärjestelmän toteutus ja siihen liittyvä prosessikuvaus ovat mukana tuloksissa. Työn tulokset osoittavat, että jatkuva integraatio voi pienentää ohjelmiston kehitysprojektiin liittyviä riskejä ja parantaa ohjelmiston laatua.

# ACKNOWLEDGMENTS

This master's thesis work has been carried out during 2009 at Sesca Mobile Software Oy. It aims to improve the quality of software development processes at the organisation. Fortunately, I was given the opportunity to work on the subject and write my thesis based on that work.

I want to thank professor Jari Porras for supervising my thesis work and giving valuable feedback. A big thanks goes to my employer and colleagues, who supported me and gave many good advices along the way. Especially, Vedran Bartonicek for reviewing and tutoring my thesis. Katja Karhu's knowledge of scientific research also helped me when I started this work. In addition, I want to thank Tuomas Inkiläinen, whose advices and expertise with Linux were most useful.

Finally, I wish to express my appreciation to my parents and friends. Their support and encouragement throughout my studies have been most important for me. Thank you.

Helsinki, October 24, 2009

Jarkko Palviainen

# Table of Contents

# ABBREVIATIONS AND SYMBOLS

| | |
|---|---|
| **API** | Application Programming Interface |
| **CI** | Continuous Integration |
| **GUI** | Graphical User Interface |
| **HTTP** | Hypertext Transfer Protocol |
| **HTTPS** | Secure Hypertext Transfer Protocol |
| **IDE** | Integrated Development Environment |
| **IRC** | Internet Relay Chat |
| **OS** | Operating System |
| **PKI** | Public-Key Infrastructure |
| **SCM** | Software Configuration Management |
| **SSH** | Secure Shell |
| **VCS** | Version Control System |
| **SVN** | Subversion |
| **XP** | Extreme Programming |

# 1 INTRODUCTION

## 1.1 Background

Software integration is the practice of assembling a set of software components or subsystems to produce a single, unified software system that supports some need of an organization 1. The process of integrating software is not a new problem. Integrating a small piece of software, perhaps involving only one developer, is still fairly easy. Basically, the only thing required is to handle few external system dependencies and see that the features work as expected. But when the project gets larger and there are several members developing different features simultaneously, the complexity raises quickly.

This complexity has been commonly referred as “integration hell” in the software industry [2, 3]. It is a typical problem for linear development models, such as the waterfall model. After implementation phase the components need to be integrated and tested, but unexpected implementation defects and interface mismatches cause delays and rework. The cost can be significant, since it has been estimated that integration and testing work can make up 40 % of the overall project expenditure [2]. In the worst case, the team cannot even predict whether or when the integration can be finished due to the difficulty of locating and fixing the defects on the system level.

Continuous Integration (CI) was developed to solve this problem. CI is a software engineering practice where members of a development team integrate their work frequently. Each member updates their changes at least once a day, thus leading to multiple integrations per day. The integration is verified by an automated build which includes running tests. This way defects are found early which makes it possible to fix them a lot earlier. Fixing bugs early is essential because it has been shown that the cost of the fix is proportional to the age of the bug [4]. It is easy to fix one bug detected two

hours ago, but fixing a hundred bugs written weeks ago can be very difficult. The fundamental benefit of CI is reducing the amount of time fixing bugs when one person's work has affected another person's without either of them realising what happened.

The term “Continuous Integration” emerged in the early 00's from Extreme Programming (XP) which is an agile software development methodology [5]. However, CI is not entirely a new invention, but rather an advance in the evolution of integrating software. The practice of building software nightly has been discussed as a best practice for many years. For example, Microsoft has been using it at least since the time of Windows NT 3.0 and the “daily build and smoke test” practice has been known in the industry since the mid-1990's [6]. In daily build process, every file is compiled, linked, and combined into an executable program every day. The program is put through a “smoke test”, a relatively simple checking that the program runs without a failure. With the advent of agile methodologies, the daily build has become a continuous process involving the whole project team.

Effective usage of CI can mitigate the following common risks in a software development project [3]:

- **Lack of deployable software.** A fully leveraged CI system can automate compilation of individual components, system integration, testing, and generating the package for installation. It is possible to achieve all these steps within a single “integrate” button.
- **Late discovery of defect.** By running builds that include unit tests with every change, defects can be detected as early as possible.
- **Lack of project visibility.** Project's status information is always up-to-date when builds are done continuously. CI practices define a variety of automated feedback methods and levels enabling better visibility for all stakeholders.

- **Low-quality software.** Consistently adding and executing automated tests and code inspections reveal defects and actual quality metrics even for a large code base without laborious manual work.

## 1.2 Objectives and Restrictions

The employer, Sesca Mobile Software Oy, has substantially increased its Linux competence and amount of Linux developers in the last couple of years. This has revealed the lack of common practices for integrating software. Until the start of this thesis work, the organisation had managed system-level integration at the latter part of a project, when most of the features are already done. Many hidden bugs and integration problems were discovered too late causing substantial amount of rework. Thus, it was felt that the benefits of CI practices should be considered to get a better control and visibility over the projects in the future. It is also expected that there will be a growing interest from the customer side towards CI-oriented development process.

This thesis acts as the preliminary work prior to introducing continuous integration into the organisation. The first goal is to understand the fundamentals of continuous integration: what kind of practices are involved and their significance to the development process. The lessons learned are used for applying CI at Sesca Mobile Software Oy. The main objective in this thesis is to define and implement the technical infrastructure for CI. It requires evaluating proper tools for automation and creating an installation of dedicated build server. The building environment is designed for handling multiple independent software projects. Furthermore, a process needs to be established for the build management determining the roles and responsibilities alongside with the instructions how the system is used. The last major goal is to identify what are the most important implications organisations need to consider when introducing the CI process.

The scope of implementation is limited for software written with C/C++ languages on Linux platform. It should be understood that other type of projects with different software languages and platforms should be dealt with their appropriate CI tools. Otherwise, the practices and processes defined in here can be applied to a broader range of development environments. Due to the time and confidentiality constrains, the evaluation of the implemented system had to be done by using an existing open source project as a reference. Continuous database integration is left out of the thesis scope, though its importance cannot be omitted when developing database-centric software [3].

## 1.3 Structure of the Thesis

The remainder of this thesis is the following: chapter two explains the fundamental software engineering practices constituting the CI process. In chapter three the related work is studied and experiments from previous attempts to introduce CI practices are gathered. Chapter four covers selecting the appropriate tools. In chapter five the build management, server implementation and their rationale are described in detail due to the complexity involved. Chapter six concludes the work by explaining the results and implications towards the organisation. Considerations for future work are included in the last chapter.

# 2 PRACTICES OF CONTINUOUS INTEGRATION

Continuous integration consists of many useful software practices, which are discussed in detail in this chapter. The practices presented here help to design for deployable software. The software can be fully built and verified by testing at every change and the build can automatically produce an installation package. This benefit, as well as having greater project visibility by immediate and continuous feedback can greatly reduce risks and false assumptions on a project.

Figure 1 presents a summary of the CI process work flow. Developers send the changes they have done to the centralised storage called repository. The CI server detects changes and executes the automated integration defined by a build script. The results are then published via a feedback mechanism to the project members.

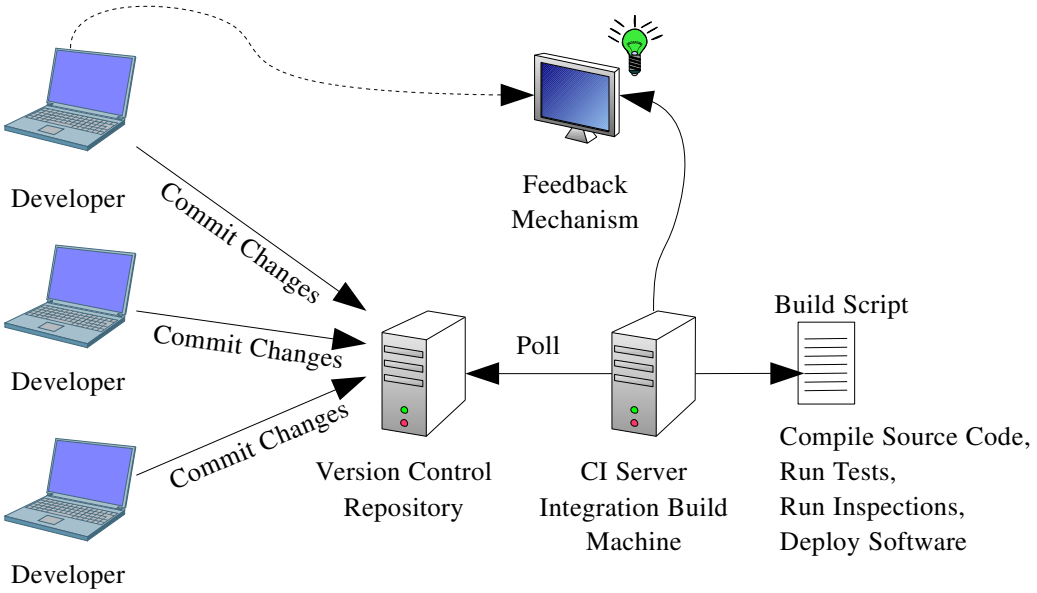


Figure 1: Summary of continuous integration process work flow [3]

## 2.1 Build Automation

"Software build" is nowadays a vague definition and can include various other steps besides a simple compilation. A pure build is a process which takes a defined set of input and transforms it into machine-executable form by compilation [7]. The input typically includes source code, software dependencies, such as system libraries, environment settings, and configurations. In the context of continuous integration, build generally means a larger set of activities: compiling, executing tests in multiple levels (unit/regression/functional testing and code inspections), and deploying the software [3]. The suitable level of included actions should always be determined by the purpose of the build. Build types can be classified into three different categories: individual, team, and user builds. The developer runs a private (individual) build after making changes and verifies it with basic testing, such as the unit tests. The changes are applied to the code base, and an integration build (team build) is executed. This build is usually done on a dedicated machine including larger set of testing. The user build readies the software for deployment in the production environment and runs all tests, including functional and performance testing.

Automating the build has become an essential feature for any software project. It can even mean the difference between success and failure. The reasons are obvious: larger code base, bigger and distributed teams across multiple sites, and interaction with other tools or databases. If the build process is neglected, it will become an endless source of problems and frustration, thus significantly reducing productivity. On the other hand, a well-maintained build keeps the development running smoothly without constant interruptions. The automation should include all build tasks. It may not always be obvious which tasks can be automated. For example, generating documentation for Application Programming Interface (API) or publishing the build results through a Web page can be automated. A bit more advanced example is handling the change logs in a release: the build tool fetches the list of recently fixed bugs from the issue-management database and attaches them within the release. In summary, automation serves three

purposes: it documents the processes, speeds up the tasks, and eliminates human errors and forgotten steps. [8]

The simplest way to automate is by using operating system's shell language or a general purpose scripting language. These scripts can be hand-made, but usually it is better to use a dedicated build tool to generate the complex scripts from the initial configuration. There are numerous tools for different languages and platforms: Ant, Maven, make and its many variants [3, 8]. In essence, the selected build tool should be able to run the build with a single command. It must also be able to handle dependencies for required software packages and be able to detect and re-compile only the changed parts of the software.

Integrated Development Environments (IDE), such as Microsoft Visual Studio and Eclipse, may also generate the build script for the project. However, they are not suitable for handling build automation [8, 9]. The generated script might be launched from the command line, but dependencies to the local application settings and paths can prevent running it outside the developer's machine, or make it very difficult. The build should depend only on the input source and software dependencies, not on a development tool configuration. Any IDE tool can be used for development as long as it does not interfere with the build setup.

## 2.2 Centralising Software Assets

Building software effectively requires that all software assets are centralised [3]. Software projects tend to involve lots of files which are modified by several people, sometimes concurrently, making it difficult to keep track the files and their changes. The solution is to use a management tool, often called Software Configuration Management (SCM), Version Control System (VCS), repository, or similar. Such management tool

provides a single source point and prevents problems where a developer might be asking from another “where is the source file for...” or arguing “but it works on my machine”. Moreover, VCS contains the earlier versions of each file, so it is easy to return to an older, working version in case a file gets corrupted or the latest changes need to be discarded. Centralising makes it also easier to achieve a single command build as the build tool can perform a scripted retrieval from the version control. Some of the most common and open source VCS tools are Subversion (SVN) [10], Git [11], and Concurrent Versions System (CVS) [12].

VCS should store everything required for building the product. Berchuk et al. call this approach as the “Repository pattern” in [13]. The pattern includes a workspace, a collection of all files maintained in an abstract space by a tool. The workspace contains particular version of all artefacts required to accomplish a task. For a building task, the workspace should contain the following items:

- Source code, arranged in the appropriate package structure
- Source code for tests
- Third-party components or library files for interfaces that the project depends (JAR files, DLLs, Unix shared libraries)
- Configuration files and data files to initialise the software
- Build scripts and build environment settings
- Installation scripts for deployment step

The decision to add certain items always depends on the project. A product with a long lifespan may even require to store certain version of compilers and tools. When a bug in an older release requires going back and compiling the exact version, the subsequent version of a compiler or tool may have subtle and undetected differences in behaviour, thus affecting the bug. In general, system wide tools, such as compilers, Java

development environment or database systems, are not kept in the version control. Instead of containing a database, VCS should include all required initialisation methods to create the database locally. Storing or publishing build results or artefacts under version control system is not recommended [5]. It can be seen as a sign of a deeper problem, usually an inability of reliably recreate builds. It will also generate more traffic slowing down the CVS.

Many VCS's provide a feature called branching. Branches are used to handle different streams of development, and are independent of each other. However, as Fowler points out in [5], branches are often overused and causes development to diverge in multiple branches. Making separate branches often diverge the development, until it is very difficult to merge different features from different branches back together. It is important to have a *mainline*, a master branch of the project where current development happens. Branching should me minimised, but there are scenarios where working in a separate branch is useful: bug fixing prior to production release, and temporary experimenting with a new feature. This serves as a protection for the mainline, leaving it to a stable state. When changes in the branch are approved, they can be merged back to the mainline.

The directory structure must be consistent and logical inside the repository. A typical approach is to structure the directories based on development activities, such as requirement, design, implementation, testing and release [3]. A testing directory can be further divided into subdirectories according to testing levels. The aim is to keep the directories distinctive and have a clearly defined content. The build tasks can be kept compact as only the relevant source code and scripts are retrieved from the version control. This approach also minimises the bandwidth usage when required input files can easily be retrieved without taking the whole repository every time.

## 2.3 Committing and Building Frequently

Commit is the “check-in” procedure to apply the developer's local changes in the source code to the centralised repository in VCS. Committing frequently is one the key factors in continuous integration. This ensures that the changes are integrated *early and often*. Beyond the technical details, integration is primarily about communication [5]. It allows developers to tell others about their changes, and frequent communication (integration) allows people to know quickly about the changes as more updates are made.

Frequent commits can be achieved by making changes in small pieces. It is better to split the work into small tasks, write the source code, and unit tests per task compared to modifying many components at once. The tasks should be broken up so that they can be finished within a few hours. Commit is then done after each task. This behaviour is also supported by many agile development methods. For example, Scrum states that the duration of concrete tasks in Sprint Backlog (developers' task list) should range between four and sixteen working hours [14].

It is dangerous to assume that everyone knows that it is forbidden to commit broken or unfinished source code to the VCS [3]. Many times the new code or change is only assumed to work correctly throughout the whole system. This can be eased by having unit tests implemented at the same time when the change is checked in. A well-factored build script runs the tests, thus verifying the code for defects and regression. An additional approach is to establish a practice where developers run a private build locally before committing. The private build tries to resemble the integration build closely, but orchestrating it to be an easy practice depends on the selected tools.

Broken builds should be the exception, not the rule in CI environment, and fixing them is a top priority. A broken build is not just a compilation problem, it can be anything that prevents the build reporting success. As there are more frequently commits, there will

probably be more broken builds. Fortunately, each error is discovered incrementally when the build finishes, and with small commits, it is easy to locate the defect fast.

Introducing the practice of keeping the build healthy into the project culture may take a while [3]. McConnell suggests in [6] that a penalty is created for breaking the build. The penalty should be light-hearted, like contributing few dollars to a morale fund, or handing out a token indicating that one has broken the build [6]. More severe actions have also been used, like having to wear a beeper in the late stages of the project. If they break the build, they get called in, even if the defect was discovered at 3 am. [6]. In contrast, Koroorian et al. in [15] do not recommend using penalties. They believe that providing clear guidelines and emphasising the purpose of keeping the build healthy motivate developers better. While the described penalties may sound controversial, the aim is to keep the build successful and avoid having long periods of broken builds.

By doing daily commits, the team can do daily builds. However, building frequently on developers' machines is not viable for several reasons: the build can require lots of hardware resources and there is most probably environment differences between developer's machines. Therefore, a dedicated build machine should be used. It is possible to use the build script for running manual builds on the server, but the most effective solution is to use a CI server software for this. The server can monitor source code repository for new changes, and launch build automatically when it detects a new change. CI servers can have various other automation features besides plain building. They provide feedback about the build, usually a Web page for the build status and other mechanisms, such as sending an e-mail or Web feed. They can also be configured to provide detailed testing results.

## 2.4 Self-Testing Builds

Testing is an essential part of the CI process [3]. It can be included to the automated build processing, thus making the build *self-testing*. To get self-testing code, a suite of tests covering a large part of the code base needs to be created. Easiest and the most consistent way is to write the testing code in parallel with the source code. Testing framework or tool must be able to run with a simple command and return the results in some suitable form for the build tool and server. When the build is run, any failing test should cause the whole build to fail. It is recommended that 100% of tests must pass, otherwise there is little point having good tests if defects are not fixed consistently [3]. It should be remembered, that tests do not prove the absence of bugs [5]. While the coverage heavily depends on how many and how good tests the developers write, having imperfect tests is always better than not having testing at all. The final testing before public release should still include manual usability tests done by actual users.

Automated testing is usually divided into unit, component, system, and code inspection stages [3]. Unit tests are implemented first. They verify the behaviour of small components, which roughly correspond to a class in object-oriented languages. A single unit test should depend only on the component tested and test only its behaviour. For example, if the tested component requires database connection, the test case should be moved to a higher-level testing category. The component being tested may depend on another object only when it does not have additional dependencies. An alternative choice is to replace the depending part with a *mock* or a *stub,* which is a simple fake replacement for the real-life component. A key part in unit tests is having no dependencies on external components for keeping the execution time as low as possible. Unit tests should be created and run early in the development, i.e., from the first day.

Component or subsystem testing verifies a portion of the whole system and may have outside dependencies, such as databases, file systems, or network connections. Such a testing is also called integration testing. The purpose of it is to verify the interaction of a

set of components producing the overall behaviour for a feature. Component tests cover larger amount of code and use external dependencies, thus the execution is slower than with unit tests. One important note is that these tests use the component APIs, but the APIs may necessarily not be the public interfaces [3]. For example, when testing a Web service software, a component test can exercise the underlying business layer, while the higher-level system tests should use the actual Web page.

System tests are meant for testing the complete software and requires a full installation of the software. They verify the software and its user interface, for example, a Web page or GUI work end to end as expected. The execution time is longer and requires implementing the automated deployment of the software. Thus, the recommended way is to run system tests on predefined intervals, such as a secondary, nightly build when the previous tests have passed. System tests have a significant difference compared to the highest level of testing category, functional tests (also referred as acceptance tests). As described in the previous example, system tests use the Web page, but they do not yet use it via a Web browser. Instead, they mimic the browser by manipulating the site directly via HTTP. Functional tests make use of the browser and test the software from the client's viewpoint. Thus, they mimic the client.

The CI process can be leveraged even more by adding automated code inspection after testing. The purpose of code inspection is to improve the overall quality of the code. Three common inspection methods exist: code reviews, pair programming, and static code analysis. The two first mentioned are peer-based method. They have been shown to be quite effective when properly conducted, but the problem is that the method needs to be applied rigorously to cover even a considerable amount of the code base. Furthermore, they are conducted by humans, who are error prone and have subjective judgement. [3]

Static analysis is done by tools, which check the source code files for violations against predefined set of rules. They have a couple of benefits compared to the manual methods: very low cost to run, no human intervention after configuration, and repeatable over every change in a file. They are also more accurate by having a prompt set of rules to check. Duvall et al. estimates that automated code analysis tools can cover 80 % of issues, while the rest 20 % is up to manual processing [3]. Especially, recognising the nature of the problem is where these tools make the best use. For example, Java's PMD static analysis tool recognises duplicate (copy – paste) code, and high cyclomatic complexity value. The value describes how many different paths a call can travel through a block of code, such as a function. It is obvious that searching this kind of matters by hand requires a lot of time and expertise.

Finding suitable tools for testing depends on the language and software frameworks or platforms used in the project. The xUnit frameworks are popular for implementing unit testing and are available for most languages [3]. In general, Java and .NET types of platforms tend to have a more complete set of testing tools, like JUnit, DbUnit, StrutsTest, Selenium, Fit and FitNesse [3]. Some frameworks also provide their own tools; Qt has QTestLib tool covering unit tests and to some extent also functional tests [16]. A good starting point for testing discussion is at [17] where over 400 open source testing tools are listed.

## 2.5 Keeping the Build Fast

Keeping the build fast is crucial in CI. If the build is slow and getting the feedback takes long, developers tend to commit less frequently, leading to a larger batches of changes. This has a negative effect on the integration, shifting it from a continuous activity to a more event-like. Keeping the build fast can be challenging, since there is a constantly growing code base with growing numbers of test cases, and perhaps multiple code inspection tools. A good guideline is to keep the commit build under ten minutes. Most

projects should be able to achieve this, as long necessary diagnose and optimisations are done [18]. The following discusses finding the bottlenecks and applying corrective actions.

Table 1 presents a summary of improvement tactics, their priority and impact on scalability, performance and difficulty to carry out the improvement. Scalability describes the capability of handling an increase in the code amount for integration and analysis. Build performance simply refers to the duration of the build. The tactics should be considered according to the priority.

| **Improvement Tactic** | **Priority** | **Scalability** | **Performance** | **Difficulty** |
|---|---|---|---|---|
| Use a dedicated integration build machine | 1 | High | High | Low |
| Increase hardware capacity | 2 | High | Medium | Low |
| Improve test performance | 3 | Low | High | Medium |
| Run staged integration builds | 4 | Low | Medium | Low |
| Optimise infrastructure | 5 | Medium | Medium | High |
| Optimise build process | 6 | Low | Medium | Low |
| Build system components separately | 7 | Low | Medium | Medium |
| Improve software inspection performance | 8 | Low | High | Low |
| Perform distributed integration builds | 9 | High | High | High |

Table 1: Integration build duration improvements [3]

The benefits of using a dedicated build machine are clear, and it should be used by default in all cases. Increasing the hardware resources is another simple way to speed up the build. The initial selection criteria for the machine should take considerations about

the expandability: adding more processors, memory modules, and hard drive space. Network bandwidth can also become a bottleneck with large check-outs.

A large code base will have lots of tests when all levels are implemented. The tests should be organised in a way that it is easy to select what categories are included in a certain build. If the build tool or testing framework does not provide sufficient categorising, tests can be located into separate directories which are pointed out to the build tool according to the build type. Unit tests should be verified that they are true unit tests and work without external dependencies. A quick test would be, for example, removing the network cable, and run the tests to see whether they try to connect a database.

The next approach is separating the build to multiple stages. The first build is a commit build compiling the latest version and running unit tests. This will reveal any obvious build errors. Later, when the commit build is finished successfully, a more extensive build is run with component, system, and functional tests along code inspection and deployment. If the secondary build is heavy taking an hour or more, it can be scheduled to nightly basis, preserving the build resources on office-hours.

System infrastructure may affect the build duration. Small network bandwidth, or problems with virtual private networks are not unusual even in corporate environments. Geographically scattered systems constitute a risk for any continuous process, and should be avoided if possible. Network latencies can have direct impact to build duration, if a large check-out takes minutes to complete. Build execution can be evaluated by measuring the time each step takes. If the compilation step takes long time, an incremental build could be used instead rebuilding the whole project. Building only the changed parts may increase the risk of introducing a defect outside the changes and not noticing it, because other parts are not built again.

If building the whole software still takes a long time, it can be split into separate sub-projects within the CI system. In this case, one project is acting as a master and depends on the other sub-projects. When one of them has changed, the depending master is rebuilt, but not necessarily the other subsystems. Software inspection performance can be increased by decreasing the frequency of heavy inspections, and evaluating whether there are tests that do not provide any tangible value, thus removing them. The last alternative, distributing the builds, can speed up building extremely large code base. It may also be useful technique if there is a multitude of target platforms which may need distinctive hardware and software for execution. Distributed builds should only be devised with a CI server designed for the purpose.

## 2.6 Communication and Feedback

In CI process developers handle development in small changes and commit frequently. Each commit triggers a build, and the feedback indicates whether the developer can move to the next task, or fix the found defects. In order to keep the cycle running, feedback must be provided immediately to the correct audience with the right information; what were the problems. Essentially, the same lack of visibility and communication must be dealt when integrating manually. Within CI process, the rapid feedback becomes even more important as a broken build halts the continuous development process during the time it is being fixed. But when the team is using automated build and testing on a dedicated CI server, the feedback from the process can be automatically collected and distributed via various ways.

The most crucial information is the build status. Most CI servers provide a dashboard-like Web page where everyone can instantly check the results of the last build. There should be an easily identifiable green mark for a success and a red one for a failure. Build logs and other relevant information, such as who made the commit, and what files were changed are usually there. Most CI servers can send an email to the selected

persons when build failed or succeeded. Other common feedback methods are instant messaging and Web feeds. Some teams prefer external feedback mechanisms, such as the Ambient Orb, X10 devices, or audible feedback [3]. These gadgets are placed on a visible location and they change their appearance, usually the colour, according to the build status. The authors in [3] believe that making the development environment more fun and personalised while conducting the CI process improves the spirit and demonstrates how seriously the team takes their work, not the opposite. Similar conclusions have been done in other projects [5]. For example, one team used audio clips from the movie *2001 – A Space Odyssey* with HAL saying phrases like "It can only be attributable to a human error" [9].

Testing and code inspection results and statistics can be added to the feedback as well. They reflect the overall quality of the code base and can be interesting to the testing team and management. Rather than waiting for stakeholders to notice that they want to ask something, the feedback can be devised to send notifications on particular issues directly [3]. However, information overloading should be bewared. Sending the feedback to everyone on the project will most probably lead to everyone ignoring the information. The team should reserve time for planning a proper feedback strategy. For example, sending a "fine" message for every successful build will bury the critical ones, failed builds. Also, informing the whole team about a problem only one or two members can fix leads to unnecessary bulk.

Figure 2 summarises the concept of continuous feedback: getting the right information to the right people at the right time and in the right way. CI is the tool making this feedback automated, targeted, and continuous. The ultimate aim of having accurate feedback is reducing the time between a defect or a risk is introduced, found, and fixed.

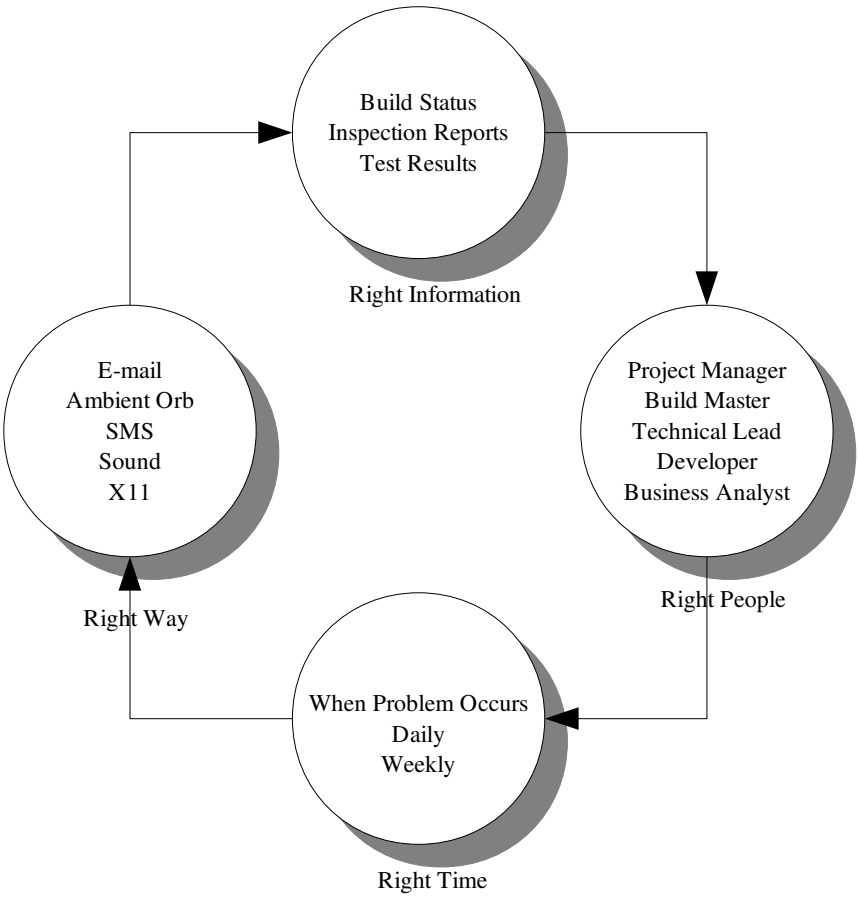


Figure 2: Continuous Feedback [3]

## 2.7 Continuous Deployment

All the previously described CI practices culminate in one key benefit: being able to release *working* software at any time by merely pushing the "integrate" button. Without successful deployment it is almost as if the software never existed. As building and testing, deployment has its own recommended steps that can be applied to any software project. Duvall et al. have gathered the following six high-level steps in [3]:

- Label a repository's assets: identify all files belonging to a certain point in time. This usually means copying a certain revision in the repository to another directory, for example "*release-1.0*". Labels can be used to go back fixing defects found in an earlier release. This approach was taken in [9] where it was

called a release tracking system. Each build name is appended with the unique build number which is used when the customer sends a bug report.

- Using clean environment resembling the production environment as much as possible. This is the only way to make sure that no leftover files or existing configurations fail the software, or produce a false positive. Ideally, removing and adding back all layers should be automated. The layers consist of operating system, its configurations (network, users, etc.), server components (application and database server, messaging servers, etc.), their configuration, third party tools, and finally the product being developed. One approach is to create an image of the operating system used with a virtual machine.
- Label each build to create a unique identifier for a build. While the repository label marks a group of source files in a repository, a build label points to the binary output of a certain build. Build labels are used to identify binaries for different platforms, as they are generated from the same source assets. Quality assurance and testing teams can also use the build labels for reporting the results.
- Run all tests. Other stages of development may require running only a subset of tests, but prior to packaging a deployment build, *all tests must run and pass*. Automated tests should be complemented by manual functionality tests done by real users.
- Create build feedback results. This includes listing the changes which were done in the most recent build. Quality assurance and testing teams are interested to know which of the reported bugs were addressed since the last build. A similar automated feedback was created in [8], where a list of recently fixed bugs from the issue-management database was attached to the release.
- Posses capability to roll back release. Sometimes, a situation might occur where a defective code must be reverted rapidly. The release-tracking system described in [9] was also reported to work as a safety net in case customer wants or needs to use another version. The deployment packages were kept in a separate storage space.

# 3 RELATED WORK

Studying the results of previous experiences and studies can provide insightful background information. This chapter discusses about three different aspects concerning the CI process. The first topic is evaluating the cost of realising the process including a case study with actual data for comparison. The second topic is about a model developed for evaluating organisation's readiness to implement the process is presented. The third study included here examines how the duration of build can affect the team behaviour. During gathering these notes from the literature, an observation was made that the terms “continuous integration” and “daily build” are used in a rather interchangeable manner. They both refer to the practice of building and testing software frequently.

## 3.1 Estimating the Cost of CI

So far, there has been lots of discussion about the benefits of CI, but less about the actual effort required for maintaining the CI server and trying to keep the build intact at all times. Estimation is rather difficult, since getting a reference data for comparison would require doing the exact same development effort again without CI process. Ade Miller has studied the cost in [19] by gathering data over a period of hundred working days with CI. The study compares the results to a hypothetical heavyweight check-in process of the team, and also discusses the most common reasons which caused the build to break.

The data was collected from the Team Foundation Server (SCM tool) and CI servers used on the *Web Service Software Factory: Modelling Edition (Service Factory)* project [http://www.msdn.com/servicefactory]. The project was relatively small including about a dozen of people distributed in three different geographic location. On average,

developers checked in once a day, although offshore developers had issues with network latencies and batched the work into single change sets, thus reducing the check-in frequency. The CI server compiled the code, run unit tests and static analysis using .NET specific tool (FxCop). A secondary build during night run the heavier tests and test coverage analysis tools.

During the analysis period, there were 551 check-ins which resulted in 515 builds and 69 failures, thus giving a failure rate of 13 %. The slightly larger number of check-ins is explained by the server configuration; rather than queuing and building each change set separately, the server retrieved only the latest when multiple new check-ins were detected. A break is defined to be the interval between a failure is introduced and the next successful build.

Four major causes of build breaks were found, as shown in Figure 3. Compile errors and changes which broke the unit tests caused a bit over half of the failures. Static analysis found relative large amount of code quality issues, 40 % of failures were caused by changes which failed the given threshold. The rest 6 % of breaks was caused by the server infrastructure; running Windows Server 2003 on a virtual machine eventually consumed all memory and disk space resources as the builds became larger, thus causing the CI server to fail. The author also notes that some of the team members were not particularly familiar with CI practices, and coaching offshore members was quite challenging. Therefore the build failure rate could well be improved on a subsequent project.

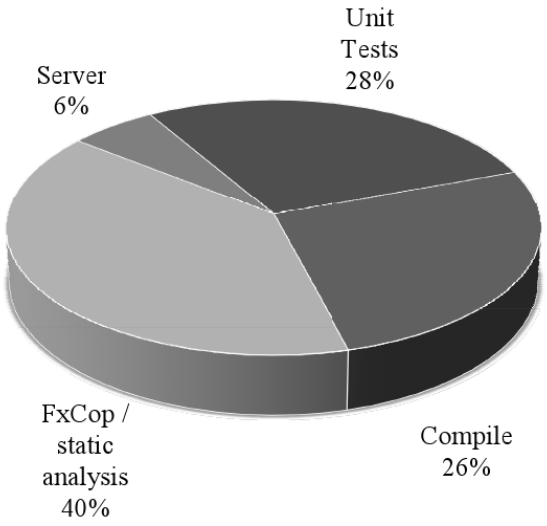

Figure 3: Causes of build breaks [19]

The length of the build break is important because once the build is broken, other developers cannot integrate their local changes to the latest, broken version in VCS. Thus, the build break affects the whole team. In the project, the average time to fix a CI issue was less than one hour, including the time to submit the fix and have it verified by the build. This was considered short enough not to block other developers significantly. Seven lengthy (overnight) breaks occurred, several of them related to server issues. The statistics are presented in Figure 4.

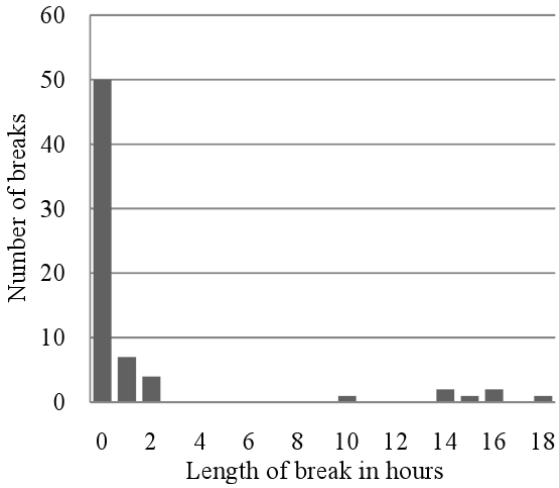

Figure 4: Length of build breaks [19]

To estimate the cost of CI overhead, the following assumptions were made: developer compiles the code, runs appropriate unit test, and possibly some static analysis prior to check-in. Code reviews are not considered here because they would be required to be done with or without adopting the CI process. Based on the team's experience, time spent doing these tasks is 20 minutes for each check-in: 10 minutes for a full compilation and running tests, and 10 minutes for reviewing the changes, writing check-in notes and e-mails, etc. Calculating from the CI server logs, fixing a build break took about 45 minutes and there were 69 build breaks. Table 2 summarises the overhead calculation. The total number is approximately 267 hours, which is 7 % of the total effort spent (4000 hours).

| | |
|---|---|
| CI Server setup and maintenance time and build script authoring (est.) | 50 |
| Total time spent checking in 20 minutes (est.) × 551 | 165.3 |
| Total time spent fixing build breaks 45 minutes × 69 | 51.75 |
| **Total overhead (hours)** | **267** |

Table 2: CI process overhead [19]

Understanding whether this overhead is still worth the benefits, a hypothetical alternative check-in process was defined for this team. Prior to each check-in, the developer has to compile, run all unit tests, installation tests, and static analysis tools on a clean build machine. The developer is now executing all the work that CI server was doing. This is justified for the comparison, since it is the only equivalent alternative when ensuring the same level of quality of the code base and the product. This approach would take at least 50 minutes: 10 minutes for a full compilation and running tests on developer's machine, 10 minutes to create the change set and unpack it on the clean build machine, 20 minutes for compilation, running all tests and static analysis (based on the execution time taken from CI server), plus 10 minutes reviewing the changes, writing check-in notes, etc. By assuming that no build breaks ever occurs, the projected overhead would be approximately 464 hours, as summarised in table 3.

| | |
|---|---|
| Clean build machine setup and maintenance time (est.) | 5 |
| Total time spent checking in 50 minutes (est.) × 551 | 459 |
| Total time spent fixing build breaks | 0 |
| **Total projected overhead (hours)** | **464** |

Table 3: Alternative process overhead [19]

Comparison to the CI overhead indicates another 200 hours (5 %) effort spent on integrating the changes. The approximation presented is the most optimistic case. In reality, build breaks will occur when developer is running the full suite on the clean build machine, and manual execution indicates more human errors. Effort allocated for managing the build machine is also minimal. Miller concludes that the actual cost using the CI approach on this particular project was at least 40 % less than the hypothetical cost of a check-in process that does not leverage CI but still maintains the same level of quality. As the size of the developed product was relatively small and the cost of doing full build was low, this number actually presents the smallest saving likely from deploying a CI process.

## 3.2 Evaluating prerequisites for CI

Kajko-Mattsson et al. have developed a methodology for evaluating the readiness of the organisations to introduce a daily build process in [20]. This work was later continued in [15] where the results of the previous study were compared to another large Swedish software company, Ericsson. The evaluated organisation in the original study was not revealed, thus it is referred to as Anonymous Organisation. Both organisations studied were in the process of improving their development processes with a daily build. The goal of the comparison was to recognise the underlying reasons for success and failure when introducing a daily build process into an organisation.

The evaluation is started by studying the prescribed development and evolution processes in the organisation. This can be done by reviewing the internal process documentation, or following a development project in case of a lacking documentation. Issuing an open-ended questionnaire done with all project personnels can reveal more details from the current state of development. Such a questionnaire can be found in [20]. The following explains the evaluation criterion of the model and gathers the lessons learned from the studied organisations.

**A new build is created frequently**

Ability to create builds frequently promotes any time integration approach. The goal is to decrease the feedback loop time so that the changes are communicated faster among developers. Koroorian et al. state also that the frequency does not need to be constant during the project, but can vary from once a day to once a week depending on the progress made. The original study [20] states that, in reality, it was not possible to build on daily basis due causing too much stress and pressure, and lack of meaningful code in such a short period. In the later study this claim has been removed, and the suggestion for build frequency is more similar with other literature [3]. In addition, the authors recommend that the organisation should provide guidelines stating how and when developers should submit their code. This helps to create more frequent and successful builds.

In both organisations, new build was created from the repository once a week, and in the later stages, the system could be built on daily basis. There were no guidelines available, only vague statements that the code should be submitted as soon as its ready. At Ericsson, developers checked in their code when they were ready with it. However, at the Anonymous Organisation, developers submitted the code almost on daily basis but regardless of the code quality or readiness.

**Developers should test their code prior to delivery to the build system**

The lack of informal guidelines also lead to insufficient testing of changes, and many defects were detected during the build and integration testing processes. At Ericsson, developers still took more responsibility for their code. They had an informal agreement that simple code was entirely handled by its owners, while more complex code was to be reviewed by peers, and critical code should undergo a formal inspection. Ericsson also greatly emphasised the importance of interface specification and documentation, which decreased the amount of interface defects during the daily build testing.

The Anonymous Organisation suffered the same lack of guidelines, but were not as successful as the other company. Developers submitted code almost every day, because the company policy was to record all the latest changes centrally. However, the repository had no distinction between the daily build (finished code) and storage (backup purposes). Thus, the submitted code was far from ready for executing a daily build. The organisation also had to struggle with other quality problems, such as high degree of coupling components and interface mismatches. These issues led to many unnecessary build breaks.

Both organisations' developers tended to use the build process as a verification method, rather than testing the code before the daily build process is performed with the changes. Both also had the possibility to create their own private builds for testing their components prior to submit, but it was utilised only at Ericsson. In Anonymous Organisation, developers used rather an older version of the system as a substitute for a private build. Three reasons was discovered to cause this behaviour: (1) there were no guidelines which version of the build to use; (2) developers preferred to use stable version which may not always correspond to the latest produced system build; and (3) installation of the latest system after each build would require too much work and effort. This clearly indicates that creating a private build and deployment of the build need to be automated.

**Staff is committed to produce a successful system build**

Developers at Anonymous Organisation were not always committed to produce a successful build. Instead of having any guidelines concerning the quality aspects of the code, they had strict deadlines when the code should be delivered. They also did not have any explicit penalty regarding to breaking the build. Punishments would have anyway been rather controversial in an environment having such a poorly defined process and low product quality. Either at Ericsson there was no penalties for build breaks. However, the importance of keeping the build healthy was thoroughly explained to the developers. They discovered that the daily build process helps them find problems early and were highly committed. Koroorian et al. do not believe that drastic penalties, as discussed previously in chapter 2.6, are the best way to improve commitment or the spirit of fellowship. Convincing developers about the significance of a successful build is an effective substitute for punishment, as was observed at Ericsson.

**Developers are organised into feature teams**

There are two ways to organise developers into teams [21]: based on the end user functionality where a team is responsible for one or more features, or based on the system modules where a team implements requirements from all features in their module. When using daily build, developers should be organised into feature teams [19]. Modular teams require more co-operation and communication between teams when multiple features depends on a module. The module designer has several features impacting his module, and usually each feature affects other modules as well. The team responsible for the module must analyse each feature change and distribute it to each affected module. This makes delivering consistent updates on daily basis across multiple modules very difficult, thus leading to build failures. In a feature team the coordination and external changes on modules are handled inside the team without impacting other teams.

However, feature teams do not come without a cost. The problem is concurrent updates of the same module by different teams. This can be mitigated by two alternative approaches: opportunistic or planned [21]. In the former, anyone can start making their changes, and all updates made in parallel must be merged with the latest updates before checking in to VCS. In the planned handling, the feature teams negotiate a time window when a certain feature owns the module completely.

Both organisations used modular teams at the time of the study. Anonymous Organisation had over eighty developers responsible for the whole system. They were divided into nine separate module teams, and several teams had to work on the same customer feature. During a build, all subsystems were integrated simultaneously using the “big bang” approach. The build included all tests (integration, regression, and new functionality) at once which lead to huge problems to isolate defects. On the other hand, the department studied at Ericsson was responsible only for one subsystem and possessed well-defined features. Although being formally defined as a module team, its nature could be compared to a feature team. They had more controlled build process: the system was not built at once, but a subsystem at a time, and then integrating it on the system level and running only integration tests there.

**The role of the build engineer is established**

There must exist a role responsible for the build process, which is referred here as the build engineer. The engineer should be full-time committed, and a large project may have several build engineers. The role should be visible, so that the developers know who to turn to when seeking for help, or providing/receiving feedback. Anonymous Organisation had a team of four build engineers, from which none was fully committed to the role having other responsibilities aside. Several interviewees, mainly developers, failed to identify the build engineer and urged the need for a more visible and collaborative build engineer role. At Ericsson, the department had one dedicated build

engineer, who was easily identified due to the small department size of 10 – 12 developers.

**Daily build process is supported by a software configuration management process and tool**

Both organisations had a configuration management tool in use: Anonymous Organisation used Microsoft Visual Source Safe, and Ericsson used IBM Rational ClearCase. While Ericsson had rigorous guidelines and instructions on corporate level for the process, the other organisation had only vague rules. The primary requirement was that all the latest changes are saved to the build repository. Thus, the repository was used as a back-up storage. At Ericsson, there was a separation between the check-in code storage space and the in-progress space.

**The build process is automated**

In the Anonymous Organisation, the process was semi-automated with computerised tools and needed supervising by a skilled personnel. The average time of executing the build was 1,5 - 2 days, thus far from optimal time. The main cause were compilation and linking errors that had to be dealt with manually. Usually, solving the errors required interaction with the developer responsible for the halted component in question. Fixing the build problems was reported to cause several days of delay. The actual processing time for build was estimated to take 6 – 8 hours, which is still far too long for building more than once a day. The author of this thesis has also come across a similar situation in a project, where the build engineer had to run often to discuss with the developers. A solution would be to include the build personnel *into the project*, rather than having him as an outside entity. Alternatively, developers could have access to the build process directly for fixing the problems more flexibly. Ericsson had just one subsystem on their

daily build system, thus the build duration took about 20 minutes. No major problems were encountered during the process.

**Testing is automated**

At the beginning of the daily build process implementation, Anonymous Organisation had established a smoke test which was always followed by integration tests. All tests were run manually, the technical infrastructure for executing tests in an automated manner was under construction. On the other hand, Ericsson had already a well-established infrastructure for automated testing. They executed stepwise integration of their subsystem with other stable subsystem using only stable regression tests. Anonymous Organisation tried to test all subsystems against regression and new functionality tests. In practice, it made testing the daily build too excessive.

The authors conclude that the majority of the daily build components must be in place before attempting to implement it. Anonymous Organisation had established partially some practices in advance, but the piecemeal introduction of daily build turned out to be too prohibitive to the overall success of the project. Therefore, it is suggested that most of the prerequisites, both technical and human, are implemented before taking the daily build process in use. These prerequisites include committed staff, enforcement of developers' testing, establishment of a configuration management process and automation of daily build and testing. Otherwise, too much resources are spent resolving the immaturity issues of daily build process. Furthermore, this study implicates that it is better to start with a smaller department rather than trying to change the whole organisation's development practices at once.

## 3.3 The Effect of Build Duration to Team Behaviour

Keeping the build fast is a primary concern for the team. However, build duration can have much broader impact on the CI process and team behaviour than just having to wait a few minutes longer to get the feedback. Graham Brooks has studied these effects in [22] at first hand when working with two different agile teams. Both teams had similar starting points: developing Java applications using the same tools for version control (Subversion), build execution (Ant), and CI server (CruiseControl). The environment dependencies and requirements were also similar, both projects required relational database for application data, had automated testing run by developers prior to check-in and afterwards on the CI server.

The difference in build durations was caused by design choices: the first team, called "Bubble" was obligated to follow the given policy as they were part of a larger development program. The developer build (commit build) took approximately 20 minutes due to more extensive testing which required creating database and deploying application to the application server. Integration tests including functional testing with FitNesse took approximately 40 minutes. The second team, "Squeak", had free hands to design and manage their build process. They used an in-memory database and were able to execute all tests without deploying the application to an application server. Thus, the developer build took only about 2 minutes. The integration build took approximately 5 minutes requiring to deploy the database and application, and publish the build artefact. This is the basis for the following discussion how the build duration can affect the team behaviour.

### Checking-in code

Running tests, or a private build, is a standard practice prior to submitting code change to the repository. However, when the build takes more than 20 minutes, developers may submit more changes at once and running the local tests after that. Even if local tests are

not run, the developer needs to wait for the feedback from the CI server. This leads to batching the changes into large check-ins. Developers in Bubble team struggled with this issue. In some extreme cases, developers did weekly check-ins and paid the price of merging and rework cost. The longer build duration also affected when developers were willing to commit. They avoided committing before lunch time, or at the end of the day, because a defect would have been especially painful to fix at that point. In contrast, Squeak team's developers were commonly able to check-in once an hour, leading to multiple builds per hour.

**Willingness to refactor**

Code refactoring refers to improving the internal implementation without affecting the external behaviour. Refactoring is usually done with small changes as chances for improvement are discovered. However, long build times cause developers to avoid single, small changes, and proper refactoring. The Bubble team was also less likely to carry out architectural refactoring. Architectural changes needed to be discussed beforehand in detail, and were postponed later when time and priority allowed. Eventually, this led to an accumulation of “development debt”, as the author describes the situation. The study does not give other reasons for inability to refactor, but the important finding is that the other team was much more willing to apply refactoring to the code base. The low build overhead allowed to commit in small pieces. When an architectural change broke the build, it was easy to either locate the defect and fix, or revert the changed files and redo the changes properly.

**Work flow and broken builds**

The team may experience a work outage, where new tasks are available, but cannot be started as the latest version of code is broken. Developers can still do a few changes in their local code, but merging them after a long build break causes extra work. The

Bubble team suffered from fragile and complex build with automated application server deployment, database creation and module dependencies. They were required to use a more general build infrastructure defined by a larger development program, which was not always suitable for their work. Therefore the team had frequent build system breaks. They were unable to access their colleagues' latest change, and having a broken build caused additional strain and complexity. It was observed that once the build went to green, developers rush to check-in their own changes before it gets broken again.

There were many reasons for Squeak team's successful building. The collective ownership of the build and build process was as a big difference maker in the overall success of the project. Each team member kept on eye on the build time, and necessary steps were taken when the developer build would slip over two minutes. Simplicity was another key factor in both application and the build. They used the same language and technologies for writing application and tests, and the tests were organised into developer and integration builds. The significant difference to the other team's developer build tests was that the Squeak team managed to reduce the dependencies low enough. They used in-memory database, and were able to run tests without deploying application to the application server.

# 4 CONTINUOUS INTEGRATION TOOLS

Choosing the automation software for CI is a matter of finding the tools which fit best to the development environment and processes. The automation software considered here includes two different types: build tools and build servers (sometimes referred to as build schedulers). Before diving into the details of these tools, the requirements for functionality will be discussed for both types. They provide the basis for considerations when evaluating CI tools. In addition to the technical details, reliability and longevity aspects are also addressed.

## 4.1 Build Tool Functionality

Functionality can be divided into essential and extended functionality as described in [3]. The following functionalities must be implemented in the build tool:

- **Code compilation:** the core feature of any building software. Compilation should support compiling only the changed parts of the code to speed up the fast build after each check-in. External software dependencies must be detected before starting the compilation.
- **Component packaging:** After compilation, the build artefacts (e.g. application binary, library, or documentation) need to be bundled into deployable components. For example, a JAR file for Java, EXE for Windows or a .deb package for Debian-based Linux. The build tools should have this feature integrated. Otherwise, the build tool must be able to call appropriate system tools for the package creation.
- **Program execution:** the build tool should have a good support for invoking programs on the target platform and also invoking any program with a command line interface. It is obvious that a build tool cannot implement all custom features one might need, so support for calling a script or a system tool is desired.

- **File Manipulation:** creating, copying, and deleting files and directories are common actions which take place when build is done.

The following functionalities are recommended for the build tool and further extend the build automation and CI process:

- **Unit test execution:** after compiling the software, a common task is to run a suite of automated unit tests. The best approach would be having a tool which integrates well with testing tools. If this is not possible, the testing tool can still be integrated via the command-line interface.
- **Documentation generation:** many programming languages have support for embedded documentation from which API reference documentation can be generated automatically. For example, Java has Javadoc [23], and Doxygen [24] can be used for C, C++ or Python code.
- **Deployment functionality:** for running functional or system tests the build should first deploy the software into testing environment. In the simplest case, it could mean a file copy to another location or FTP file transfer to the test server.
- **Code quality analysis:** code inspection tools can be integrated to the build tool. The most benefit from inspection tools is usually gained when both the build tool and build server have in-built support for them and can generate resulting statistics as a web page. Otherwise, handling and automating the feedback of inspection results by hand can be difficult.
- **Extensibility:** possible to add new functionality to the build tool. For example, when a new testing or reporting tool is to be integrated into the build. A well-documented extensibility API is needed for adding a plug-in or making modifications to the existing behaviour.
- **Multi-platform builds:** certain type of products are targeted on running various operating systems or platforms. This requires a build server which can orchestrate build processes to multiple servers running different platforms.

## 4.2 Build Server Functionality

Build server functionality has its own set of requirements. The following list of functionalities should be considered as a minimum when choosing a build server:

- **Build execution:** build scheduler must enable execution of automated build on a periodic basis. There are three different approaches:
    - Polling-driven where the build server queries the VCS for new changes. It is usually done every few minutes.
    - Event-driven scheduling triggers the build automatically when the repository receives a new change. Enabling the event triggering requires a modification to the VCS behaviour, i.e., installing a script which generates and sends the event to the build server after a successful check-in.
    - Predetermined schedule. As stated by Duvall et al in [3], schedule-driven tools are not following continuous integration by its definition, since the point of automated build is to detect defects immediately after the change. However, a scheduled build may still be acceptable if it helps to do the job most effectively.
- **VCS integration:** seamlessly integrating with the VCS is a clear benefit and majority of the build servers support the most popular VCS's. The attention should be focused on how the build server interacts with the VCS. Does it always fetch a complete set for each change? This may be impractical for a very large project with continuous building. Another issue to look for is how well the tool identifies the changes that triggered the build. Is it possible to ignore certain changes or trigger build conditionally?
- **Feedback:** the essential part of CI. After a build finishes, the server must be able to give out the results. The most common feedback methods are a Web page for detailed information and sending an email for summary. The latter is important

for reaching immediately the person who made the last check-in when build fails. Other feedback options include instant messaging and IRC, text message, and other visual and audible devices.

- **Build labelling:** marking all source artefacts that were used on the build. Labelling ensures that the set of source artefacts are unambiguously identified and the build is repeatable. When a defect is found on any version of the software, it is easy to locate and rebuilt it. A common labelling method is providing an ascending counter which can be appended to the build name.

The following issues should also be considered as a complementary functionality for the build server:

- **User interface:** an optional feature to ease working with the build server. Commonly, the server runs as a daemon process and is configured via a configuration file. Some people prefer editing these XML, script or similar configuration files by hand, while others like doing it via a Web application interface.
- **Artefact publication:** a successful build results in a deployable components which need to be transferred into a commonly accessible location, for example a Web page. Generated documentation can be copied to a Web page as well as test results, code quality analysis and other quality metrics. It makes easier to see the overall state of the project when all results and metrics are directly accessible for easy review.
- **Security:** in a corporate environment it may be required to limit the visibility of the project source code, results and configurations. The CI process is rather collaborative by nature and does not endorse the security restrictions, but in some cases they are required. The security measures include authentication and access control to the build directories, configurations and results.

## 4.3 Reliability and Longevity Concerns

Reliability of the build tool or server can also be seen as the maturity of the tool. Maturity can be estimated before actually testing the tool by checking various matters: is the current release version rather something like “3.0” than “Beta”, and what is the size of the user and development base. The latter concerns more open source tools, which are the scope of this work. One could say that the more larger and active the community is, the easier it is to get answers, bug fixes and support. The activity of mailing lists and forums is one indication of the community base. A tool with long and documented history is very probably going to be more reliable than another one with no publicity.

While reliability gives information about the tool's past and present state, longevity tries to predict the tool's future. Also longevity can be estimated by the size of the user base. According to [3], longevity is actually a compelling argument for choosing an open source tool. This is due to the fact that the tool or software is kept alive and developed on by the community. Good tools with unique values will eventually take over those tools with nothing special to offer and their development resources eventually moves to support the better ones.

Commercial products have other factors influencing the life cycle, such as the profitability of the product. It could happen that the software house declares a bankruptcy or be acquired by another company, and the product might disappear or otherwise change its essence. On the other hand, commercial products usually come with a guaranteed technical support for its life cycle.

## 4.4 Build Tool and Server Evaluations

Build tools are used to build the source code and CI servers to automate the process. Automated build tool should be selected according to the language and framework the

project is going to use. Two of the most commonly used build tools [8], *make* and *Ant,* are discussed in this chapter.

The build server utilises the build tool and automates various tasks, such as testing, scheduling, publishing results and deployment package, and so on. Today, there is a wide range of different build servers available for selection. One of the widest and most up-to-date collection of information for comparison can be found at [25]. It is advisable to narrow down the selection first by including only the tools directly targeted for your development platform or environment, for example, Java, .NET, or C/C++. Next, the critical technical requirements should be identified, like cross-platform development, which may not be available on every server. In the scope of this thesis, only two open source CI servers are evaluated: CruiseControl and Buildbot.

### 4.4.1 Make

`Make` is the grandfather of all build tools. It was developed in 1977 at Bell Labs and is still widely used in Unix-like systems for developing C/C++ based software [3, 26, 27]. The original technique consisted of fully hand-written `Makefile` files which define the rules what and how to compile. Eventually, managing the `Makefiles` became quite complex as projects grew in size. Thus, they are usually generated with tools like GNU build system, which has become the de facto standard today [28]. GNU build tools are provided with all common Linux distributions and used for compiling, for example, the Linux kernel which is one of the largest and widely spread open source project containing nearly 9,000,000 lines of source code [29].

During the past 25 years, a numerous amount of implementations of `make` has been created, and almost every platform has its own variant [28]. The variants are not fully compatible with each other, although being inspired by the original version of `make`.

Since the GNU version is still used most commonly, it has been chosen here for further discussion. For clarity, the term "`make`" will refer to the GNU version of `make` throughout later parts of this thesis.

When comparing to the functionality described in the previous chapters, `make` provides practically all of them. Source file dependencies are declared with its own language and tools are called via Bourne-compatible shell [30]. The commands are directly embedded into the `Makefiles`, thus making the invocation transparent and fairly flexible. In general, `make` is only used for the compilation of required build artefacts, while packaging and deployment is done with system tools and shell scripts. Linux developers are usually familiar with the system tools and shell scripts and the behaviour of these tools is well established.

`Make` has also a couple of disadvantages. The recursive use of `make` with `Makefiles` calling other `Makefiles` (in subdirectories) can lead to incomplete dependency graphs or circular dependencies [27]. Traditional fixes are to change the order of recursion by hand, run `make` multiple times or run a clean build always. The latter choices naturally lead to longer build times. Peter Miller has elaborated these fairly complex issues in [31] and summaries that "*they are not an inherent limitations of make, as is commonly believed, but are the result of presenting incorrect information to make.*" The extensive usage of system or external tools and shell scripts has also been criticised [27]. These tools may have slight differences on different platforms, or some of them may not be available at all, which hinders portability. This issue depends on the context: software developed for a specific platform, such as embedded or mobile device will rarely be used outside the original target platform.

There are a few `make`-based alternatives worth mentioning. They are shortly described here, since there is no need to lock on a certain tool or version for all projects. On the

contrary, each project should consider whether some other tool suits their development environment characteristics better. These alternatives usually describe themselves as the next-generation build tools, providing improved cross-platform support and easier configuration of the build. The downside is that they have smaller user base, and developers need to be trained first. These tools may also require understanding some underlying software platform, like Python.

- SCons is a substitute for the `make` and GNU Auto tools combination [32]. It utilises the Python library which makes it widely portable and ensures a versatile range of tools and utilities to be used within the build script. According to [27], SCons is sufficient for most common development projects.
- Higher-level frameworks for generating standard `Makefiles`. The architecture of the GNU build system has been considered too difficult to understand and manage for very large projects [33]. To solve this complexity, a few tools have been developed which are not meant to replace `make`, but rather generate the standard `Makefiles` by using a higher-level description language. Examples of such tools are CMake for standard C++ projects [34], and QMake for Qt framework based projects [35].

### 4.4.2 Apache Ant

Ant is the most commonly used tool in Java projects [3, 8]. Ant was originally released by the Apache project in the year 2000 to overcome the portability issues related to `make` [36]. It is written with Java and the builds are defined using an XML description file. In contrast to `make`, which uses many system or external tools for the job, Ant has built-in support for all tasks, meaning that the build behaves identically on all platforms where Java is available. Ant has been well documented [36], there is a large world-wide user base, and it is been recognised as a reliable and robust tool [3, 8].

Ant has many strengths, but it does not seem to be spreading outside Java world so much [28]. The lack of proper support for different compilers is one issue. Compiling for C/C++ languages is possible only via Ant-Contrib project [http://ant-contrib.sourceforge.net] and few similar commercial products. These add-ons are not officially supported by Ant project. Ant-Contrib is mentioned, but described rather as a development playground [36]. Besides that, Ant has the a general task for executing command-line-style commands. The task makes possible to call any other build tool, but there is no mechanism to handle the build configuration or detect software dependencies. They need to be done with the underlying native tools, such as `make`.

Mecklenburg et al. present other criticism against Ant in [28]. The build description language, XML, is not very practical for humans to read or write. More important, when writing build files with Ant, one must struggle with another layer of indirection. For example, the Ant `<mkdir>` task does not invoke `mkdir` command from the system, but the Java function `mkdir()` from `Java.io.File` class. Thus, Ant's behaviour is opaque and difficult to debug. Individual commands cannot be tested directly from a shell as when working with `make`. Another issue is implementing custom build tasks, which requires Java knowledge.

### 4.4.3 CruiseControl

The incentive to use Ant is that it is used by one of the most popular and well-known build servers on, CruiseControl [3, 5, 37]. It is also commonly deployed in research projects, such as [38, 39, 40]. The author has also seen its popularity in the software industry. It is briefly studied here to see what makes it so successful and where it is best used. Technical details, such as configuring the server, are out of the scope.

CruiseControl is one of the first developed build servers and was inspired directly by the practices of continuous integrating and automated testing introduced in Extreme Programming (XP) [5, 18]. CruiseControl is written with Java and uses Ant as the build tool. As both tools are configured by using XML, they integrate seamlessly together. Later on CruiseControl has gained separate variants for .NET and Ruby which provide similar level of integration for the languages by using tools targeted for .NET or Ruby platforms.

CruiseControl provides a very comprehensive level of integration on every aspect of CI. With Java, one would use JUnit framework for unit testing and there is no need to parse the results of separate tests, but CruiseControl outputs them automatically in table form as a Web page. Similar integration is available for functional testing, e.g., automating an in-browser testing with a Web application. Furthermore, several code inspection tools are integrated in the same manner. CruiseControl has been built on top of Ant from the beginning, so the interaction with the build tool is very natural. There is little work that needs to be done by hand when using the server. It supports multiple projects on a single server, but on the other hand there is no access control and all projects are visible for everyone.

However, it all boils down to the fact that CruiseControl and Ant depend too much on Java platform to suit our purposes best. CruiseControl may still be considered for its maturity and other benefits if it fits well with the used build tool. As the author has noticed, CruiseControl has gained popularity also among Symbian/S60 development industry. In this particular case, the development environment is provided in a separate *toolchain* including the compiler, libraries, build tool, and editors. Building is a lot easier when the development environment is separated from the common operating system software and has only minimal set of external dependencies. Thus, it may become feasible to use Ant for calling the actual build tool or compiler.

### 4.4.4 Buildbot

Buildbot [41] consists of a single build master component, and one or more build slave components, connected in a star topology, as shown in Figure 5. The slaves can be located on the same machine as the master or connected over a network, communicating over a TCP connection. The slaves open connection to master and to VCS when retrieving the source code, making it possible to operate behind a Network Address Translation (NAT) setup or firewall.

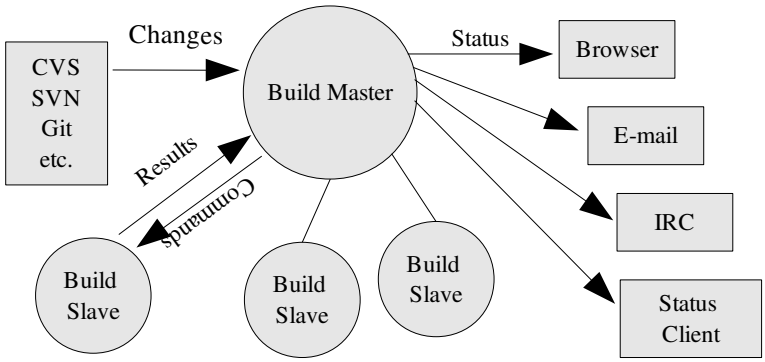

Figure 5: Buildbot Architecture [41]

The master contains all logic (the build configuration), and the slaves run processes called builders, which execute tasks received from the master. Each builder knows how to execute a specific build, such as “make clean build” or “make release package”. When the master detects a new change in VCS, it sends build requests to builders, which execute the build steps in the slave. Below is a typical set of build steps:

1. Update source code from CVS
2. Configure the updated source tree
3. Build the source code
4. Execute unit tests
5. Create a deployment package from the build
6. Upload package to build master for publishing

Build master detects changes in VCS either by polling or by an event sent by the repository. A scheduler component decides, according to the build configuration and the received changes, which builders to invoke. The build status delivery is managed by the server and automatically published on a Web page, by sending e-mail, via Internet Relay Chat (IRC), or API for custom-made status clients.

Buildbot is a cross-platform CI server supporting a wide range of Linux/Unix distributions and Windows platform. The master – slave design makes it possible to run slaves in different target environments, for example i386, PowerPC and ARM, to build the same software without having to setup and maintain separate build systems for each target. The external software dependencies in the host machine are minimal, Python and Twisted libraries are required. Both are directly available from all major Linux distributions. Other notable features in Buildbot are:

- Handling arbitrary build processes, including building C/C++ software.
- Tracking builds in progress and providing estimated completion time.
- Flexible configuration by sub-classing generic build process classes.
- Debug tools to force a new build and submit fake changes.
- Scheduling a private build for changes which are not yet checked in to the VCS. This feature can essentially replace the need to devise a private build in developer's local machine.

Figure 6 is an example of Buildbot's Waterfall status Web page. There are three different builders in the project: building the Twisted library version 2.4 against Python version 2.4, Twisted 2.5 against Python 2.3, and Twisted 2.5 against Python 2.5. Each build is a set of events, starting with the yellow box on the bottom. These builds consist of a check-out from VCS (update step), compiling, and running tests. Orange colour means warnings and red colour would indicate that the step has failed. The “changes” column

contains link to a log page showing what triggered the build and the changed files. The most recent activities are always on the top.

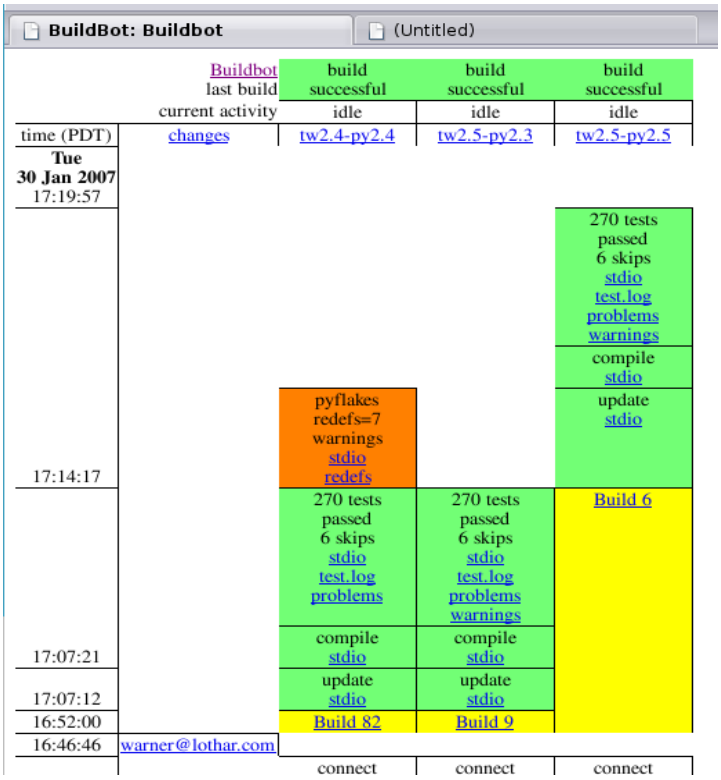

Figure 6: Waterfall display [41]

Buildbot is written in Python and the build configuration is also a Python script, called `master.cfg`. In essence, the script simply defines a list of variables which control the build, but Python can be fully leveraged here to customise build behaviour with any Python module. An example configuration file is presented in Appendix I. The example build fetches the latest version of Check project from its version control system, builds it, and runs a set of unit tests. Check is a common unit testing tool for C/C++. It was chosen because it contains the most common activities for a C project: using `make` to build, compiling a library and an application, and implementing a set of unit tests. The example configuration works without any modification, as the source repository is public.

### 4.4.5 Conclusions

Selecting the build tool is not a straightforward task. Each tool has its own strengths and disadvantages. Even when there are established requirements for the tool, it is not possible to anticipate every issue that a project may encounter. It is also possible to change the tool if the build becomes too cumbersome. A typical example of build tool selection process is the KDE project which provides one of the most common Linux desktop environments [33]. Originally, KDE was using the GNU build system, which was replaced by SCons in 2005-2006. After realising that SCons was requiring too many patches and fixes (i.e. maintaining a fork) for their purposes, KDE developers decided to switch using CMake starting from version 4 of KDE.

After evaluating and testing both `make` and Ant, the former was selected as the recommended build tool. Although Ant would provide a more consistent and unified framework, the lack of support for C/C++ compilation was too big risk from the organisation's point of view. Ant-Contrib proved to be able to build a basic C applications, but even some of the examples within the package needed fixing; apparently they were targeted for an obsolete version of Ant. Alternatively, using Ant's generic `<exec>` task would lead fast to situation, where Ant is just another wrapper layer calling `make`. This would basically double the complexity in build management without gaining any benefit from using Ant.

Buildbot version 0.7.8 was chosen as the build server for the CI system. It fulfils quite well the general requirements as well as organisation's requirements. Unlike CruiseControl, it is designed for a generic build system without making any assumptions about the build tool. It can also build projects managed with Ant, because it relies on the command line interface of tools. The obvious limitation is that the tools can only return whether the call succeeded, but not any specific data or information. They must be acquired either from the log files or files generated by the build task.

# 5 BUILD MANAGEMENT AND PROCESS

This chapter describes the implemented build system, its management and the process of setting up and maintaining the build in a project. The description is started by defining the roles involved in the build process and a more general discussion how the build system components can be deployed in the organisation. The implementation part consist of the server installation procedure and methods to produce separate building environments for each project. The last two chapters explain how new projects are created in the build system. This includes preparing the VCS connectivity and specifying the build configuration in the CI server.

The system is deployed on a dedicated server and is expected to handle multiple projects simultaneously. This affects directly to the system design: the roles involved in the build process must be clearly defined and the server installation must support separating different projects and their building environments. The foremost benefit of having dedicated service is manageability. Once the process is well established, it is easier to take in use for a new project or when responsible persons change.

## 5.1 Build Management Roles

The responsibilities of system maintenance and projects' build management need to be delegated in a reasonable fashion. They cannot depend on a single person's presence or knowledge about the system. Thus, managing the build environment is separated into two distinctive roles.

*Build Server Manager* is responsible for setting up and maintaining the server environment. The responsibilities consist of user and access management handling and

providing system-level tools which different projects may require. Build Server Manager does not handle individual projects but rather ensures that each project is capable of doing it by themselves. Working as a Build Server Manager requires general Linux-server expertise.

*Build Manager* handles setting up the build configuration for the project. He is part of the development team and familiar with the software and requirements the project has for the development environment. To keep the system robust, only one user account is created for the Build Manager. Team members can share the account among multiple persons when needed.

## 5.2 Build System Deployment

There are several ways to setup the build deployment. Separate server machines can be used for the build and VCS. Building workload can be further divided to multiple machines if necessary. In the organisation's case, VCS has been centrally managed on a remote site, while the build server is on the local site. However, this increases the complexity and maintenance cost for building. Latencies and network breaks can slow or stop continuous building when remote servers are involved. Separating the maintenance to different sites makes updating the system more difficult. Due to these reasons, it is suggested that all components are located on the same site if possible.

Figure 7 presents the build deployment with Buildbot. The project has a build master which contains the configuration and status information. The build slaves are separate processes executing the build tasks. Example tasks include making a full build and a deployment package for ARM-based mobile device. The project also has own source code repository in a version control system, SVN. These three entities are drawn separately, as this deployment is rather agnostic about the underlying parts. The communication is transparent and all three components can be on the same or different

servers. It is possible to later move any component to another location with minor changes in the build configuration. Fault tolerance is increased by replicating the repository to a secure, remote storage. Even if the build server containing all components suddenly goes out of service, the build configuration and repository can be restored to another server hardware from the backup storage.

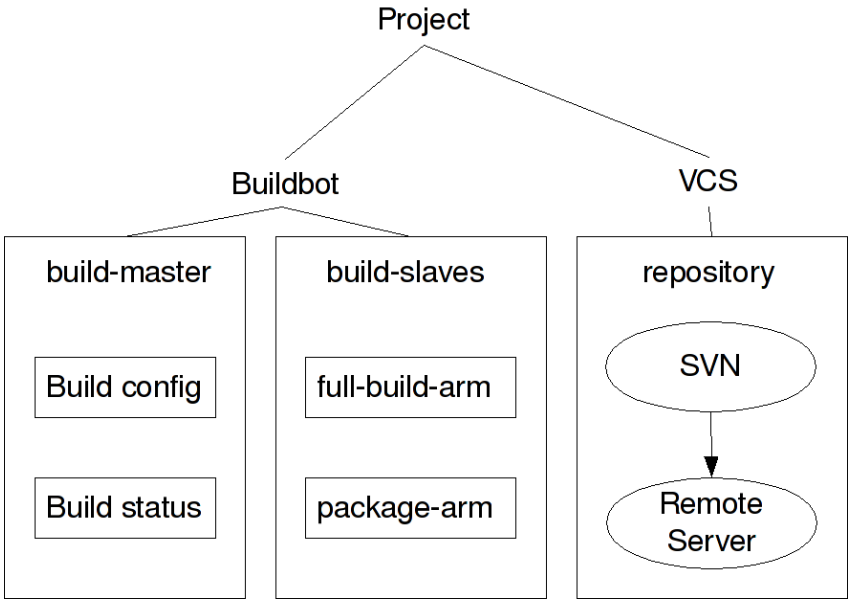

Figure 7: Build Deployment

## 5.3 Preparing the Server Installation

A Debian-based Linux distribution is used as the build server operating system. The tools used are also common in other major distributions, but commands may vary, configuration files can be located in different paths, and software package management differs between Linux distributions. Thus, it is recommended to use Debian server edition. The installation has been verified on Debian 5. The installation procedure assumes having access to the organisation Intranet services as well as Internet for installing software tools provided by Debian package repository.

Team members in a project should not have access to other projects. All project directories need to be protected from unauthorised access. Access to project files is

restricted by the build slave processes, which are started with `umask` (user mask) command set to value of 077 by default. The mask sets default permission modes for newly created files and directories, meaning that all files created by the build slave or its child processes will be unreadable by any user other than the owner of the process.

Access control is provided via normal Unix user accounts. Build Server Manager must add the Build Manager account for the project. Build Server Manager will also create empty project directories with correct access privileges. Appendix II lists a shell script which handles this job conveniently. It was noticed during the implementation that this type of steps involving multiple commands tend to be very prone to user errors and should be automated as much as possible.

## 5.4 Separating Build Environment

Chroot is a system tool which runs a command or interactive shell with special root directory. A "chrooted" program cannot access any files outside the given root directory. By combining it with another tool, debootstrap, it is possible to create a guest operating system (OS) on the system [42]. The guest OS still uses the same kernel and has no GUI components, thus adding minimal overhead. It must be binary compatible with the host OS, meaning that the architecture must be the same. 32-bit guest OS can be used inside 64-bit host, but not vice versa. The guest OS is used as a sandbox environment which does not interfere with the host OS installation. There can be arbitrary number of guest systems, usually one per project.

The installation and usage of the sandbox environment is explained later in chapter 5.6. The Debian package for fakechroot version 2.8-1 contains an intractable bug preventing installing a 32-bit guest into a 64-bit host which we have in our server. The bug is reported at [http://bugs.debian.org/cgi-bin/bugreport.cgi?bug=511296] and includes a

patch which eventually solved the problem. However, identifying the problem cause and trying alternative solutions took easily a couple of weeks of work.

Creating and using chroot environment would normally require administrative privileges. By using the fakechroot variant for debootstrap it is possible to avoid this and entirely pass the creation process to the project team. However, chroot should be used with caution. It does not provide any real security measures. In fact, it can compromise the whole host system when used carelessly. Escaping from the chroot jail is easy, as proven in [42]. When using fakeroot tool to login into a chroot environment, the user can install software and still escape the jail, but never gains any additional privileges. Even when the users are trusted, this security measure prevents accidental changes on the host operating system.

Virtualisation with VirtualBox software [43] was considered as a more complete and consistent way to achieve independent build environment, but it quickly proved to cause large overhead. The mere execution time for a build raised over 30 % which was not tolerated. Running several virtual machines would require even more memory and CPU resources.

## 5.5 Version Control System Integration

As mentioned earlier, VCS can be handled in a centralised manner by organisation's IT support, or by the project team. Regardless of which party will eventually manage the VCS, making it work with the build system is Build Manager's task. There are a few VCS-specific issues which need to be addressed. This chapter points out the issues for SVN and Git, which are the two version control systems used by the organisation. It is highly recommended to create a separate user account in VCS configuration for the build process. First reason is to avoid any accidental check-ins by the build processes by

granting the account only read privileges. Some version control systems, including SVN, cache the account credentials locally on the machine after the first successful check-out. This removes the need for interactive authentication during check-out step in the build process, but may also leave the VCS credentials available to anyone having access to the Build Manager account.

**Preparing Build for SVN and Git**

Buildbot relies to the SVN credentials cache when doing a check-out. Initially, the cache is empty and SVN requests the credentials in interactive mode, halting the build. This can be easily resolved by making the first check-out manually with the same account as the Buildbot processes are run. This works with access methods based on the SVN custom communication protocol or HTTP(S). If SSH (Secure Shell) is the only available access method, then Public Key Infrastructure (PKI) [44] must be used to avoid the interactive query for user name and password during the check-out operation. Using PKI is explained in Appendix III.

Interactive queries during the check-out are problematic since they tend to halt the build process. Another potential problem is an outdated public key certificate on the VCS server side which causes similar query to appear. These problems can be identified from the build log available in the Buildbot status page and solved by simply running the halting command manually and accepting the query permanently.

Current version of Buildbot does not provide polling-based change detection support for Git. Thus, an event-based detection mechanism must be used. It requires installing a post-receive hook on the central Git repository. The hook is a script which is run each time a successful check-in is done. The script gathers a list of changes and sends them to Buildbot which takes appropriate actions based on the changes. Buildbot provides a

ready-made hook script for Git which is simply copied to the appropriate location in the repository [11].

SSH protocol is commonly used access method for checking out source code from a Git repository. It is the only available method in the organisation, hence other methods were not verified. Setting up the SSH-based check-out method requires the steps described in Appendix III.

## 5.6 Project Build Setup and Configuration

Setting up the build consists of three parts: creating a chroot sandbox, creating a slave, and creating the master. While the technical details are presented in the appropriate appendices, the selected design and its significance to the build setup is discussed here. The setup is then finished by preparing a method for managing all configuration files under version control system.

### Creating Build Setup

Appendix IV presents necessary steps to create a new chroot environment. It is important to note that the sandbox contains only minimal Debian OS and requires installing build tools and software dependencies required by the project. This step must be verified by checking out the source code into the sandbox and ensuring that the code compiles by hand. The build server cannot build the project if any dependency is missing in the environment and debugging from a live build system is always more difficult, thus the sandbox must be ready for the build. When the sandbox is ready, the slave can be installed by using the utility script listed in Appendix V. It merely adds scheduled tasks to *crontab* for cleaning out old log files and restarting the daemon in case of server machine reboot.

The build master is similarly created with the utility script listed in Appendix VI. The master still needs to be configured in the `master.cfg` file. The configuration depends highly on the project preferences and cannot be elaborated here very deeply. Appendix I presents a simple configuration example and the user's manual [41] contains a more comprehensive documentation and examples.

There are two different ways to handle the configuration when building inside chroot. The first one is to launch the slave process *inside* the guest system, which does not affect the configuration. The project can be build as demonstrated in Appendix I. The downside is that Buildbot, VCS and other tools must be separately installed inside every guest system. The second option is to launch the slave processes on the host. This approach requires explicit changes to the build configuration. Each build step needs to execute the chroot login and run the appropriate command from there. We use the latter approach to keep the build environment as minimal as possible.

To understand how build configuration is adapted to the chroot environment, a build step from the Appendix I example can be used for demonstration. The following step defines a command to build the project:

```
f1.addStep(shell.ShellCommand(command=["make", "all"]))
```

When using chroot, a login to the chroot environment is done, the current working directory is changed to the build directory, and only after that, the command is executed. For this sequence of tasks, a Python script called `chroot_exec.py` was created. The script is listed in Appendix VII.

Below is a modified version of the build step given above. Instead of running "`make all`" directly, it executes the script which uses the rest of the parameters to execute the given command inside the sandbox. It should be noted that the value of `build_path` is relative to the file system structure inside the sandbox. Other steps can be written in

the same manner.

```
chroot_exec = "/work/scripts/chroot_exec.py"
chroot = "/work/builds/example-check/slaves/chroot"
build_path = "/buildbot/full/build"

f1.addStep(shell.ShellCommand(command=["fakechroot",
chroot_exec, chroot, build_path, "make", "all"]))
```

**Handling Build Configuration Files with VCS**

The build configuration files should be kept in the repository among all the other software assets. The problem is that all files managed by the VCS must be under the same root directory, which is not the case with build configuration: source tree is in the build directory, but the configuration is in the build master's directory. Furthermore, the build directory is often completely removed when executing a clean build, so configuration files cannot be directly used from there.

The solution is creating a separate subdirectory for build configuration and always checking in the configuration updates there. This directory is checked out under the appropriate Buildbot component, the master or slave. Since Buildbot expects to find certain configuration files (the most important being `master.cfg`) in the build master and slave directories, a symbolic links can be used to point to the actual files. The setup is transparent to both Buildbot and the VCS and requires very few changes in them. The configuration files can be edited in place and checked in, or edited elsewhere and checked out in the master and slaves. Automatic reloading of configuration files can be done via a post-commit hook. The hook is a script located in the repository which is executed when a certain criteria matches in the received set of changes, e.g., a configuration file was updated.

# 6 DISCUSSION AND CONCLUSIONS

This chapter presents the results of implementation and further discusses their implications in a broader view. The implementation discussion is divided into two parts: tools and the build system created using the tools. The latter part includes evaluations for the CI process as well. The implications for organisations considering introduction of CI are discussed next. Finally, the work done in this thesis is summarised and suggestions for future work are given.

## 6.1 Build System Implementation

**Continuous Integration Tools**

Software tools for CI were divided into two categories: build tools and build servers. The primary goal was to identify the most suitable tools for C/C++ language-based projects. It was not possible to find a single build tool which would fit to all anticipated projects. The main reason discovered was that some software frameworks may require using a specific build tool not available outside the framework. For example, applications based on Qt framework are easiest built with QMake. The choice has to be made case by case.

The build tool selection was narrowed down to `make`-based tools and Apache Ant. Even though Ant is mainly limited to Java world [8], it could not be directly put aside. Many build servers provide a high level of integration with Ant, which could outweigh the inconveniences caused by managing C/C++ builds with Ant. This has been done successfully with CruiseControl CI server in [45]. However, it was considered requiring too much manual work and far from easy to implement. When longevity and reliability issues were considered, it was noticed that the approach is still used only marginally and

there is no official support available. Thus, `make`-based tools were seen as the recommended choice. GNU build system provides the end-to-end build functionality and is the most widespread and portable `make`-based tool, although it may not be the most convenient to use.

Evaluations done in this thesis revealed that there is a lack of build tools which provide a high-level integration of testing and code quality tools for C/C++ software development projects. Other platforms, like Java and .Net provide more unified, end-to-end development environment having most of these tools available. The C/C++ development field is somewhat scattered and there are lots of tools available [17], but the integration to build and server tools requires more manual work. The result is observable in the CI system feature matrix at [25]; a major part of the systems are targeted for Java and .NET. While we are able to integrate practically any tool into Buildbot, a higher level of integration is still desired. That would make providing detailed feedback from testing and code quality analysis significantly easier.

**CI System Infrastructure and Management**

The CI system infrastructure consists of the following parts:

- deploying the selected build tools,
- server environment installation,
- and declaring a process for the build management.

Three central requirements can be recognised determining the design choices:

- managing multiple projects in the build system,
- delegating the build responsibilities to the project teams,
- and separating build environments.

The first two requirements were found to be conflicting with each other when technical implementation was considered. In order for a project personnel to setup and maintain the build, they need certain level of access and administrative privileges on the server side. On the other hand, those privileges cannot be granted for the sake of server environment integrity. Also, it is required to maintain a possibility for restricting access to different projects as we are working on a corporate environment.

Our technical approach proved to be highly efficient. By using a guest operating system it was possible to create arbitrary number of separated build environments. Project's Build Manager can devise the software packages and tools without granting any additional user privileges inside the sandbox. The system is fairly robust and does not suffer performance losses which are unavoidable when virtual machines are used. Creating the system required writing a few small utility scripts, but mostly it was done by using tools provided by the distribution. The main disadvantage encountered was software bugs in these tools. In particular, the tool for granting fake user privileges (fakechroot) suffered compatibility problems when 32-bit guest OS was used in 64-bit host OS. The effects were often very subtle and hard to detect, but were eventually resolved by upgrading to the latest possible package version.

The overall functionality was evaluated by introducing one internal project and a couple of open source projects into the build system. Build Managers in those projects had less Linux experience, so it was also a test for the documentation and usability of the build system. Experimenting with live software projects with real-life dependencies and build issues provided insight what need to be improved within the process. Especially, the concept of chroot was experienced a bit confusing as there is only a single command prompt but two entirely different environments. The following steps were gathered to improve the awareness and usability of chroot:

- Command prompt changes to reflect the current environment: `user@hostname$` changes to: `user@chroot$`.

- Simplifying creation and usage of chroot environment: a utility script called `create_chroot.sh` can conceal the numerous default parameters for calling debootstrap.
- `create_chroot.sh` can generate another helper script called `chroot_login.sh` to achieve a single-command login to the newly created chroot environment.

Apart from the individual improvements to the system, the major observation was that the creating the build process itself has a continuous nature. This cannot be achieved in one go, but requires feedback from users and constant improvements via small changes much like Kitiyakara et al. describes their work on the article *Growing a Build Management System from Seed* [9]. It is also agreed with the authors that the changes should be as simply as possible and respond directly to a specific or potential problem. This helps to eliminate the need to introduce tools or methods that are not adding any real value but making the system heavier and bloated.

## 6.2 Implications for the Organisation

**Preparing for Continuous Integration**

Implementing the CI process may require massive investments for the organisation since it affects all stages of development [46]. The key factor is not just crossing the barriers of entry into CI, but to be able to automate the end-to-end building to produce a reliable release. Prior to introducing the process to an actual development project, majority of the CI components must be in place. A piecemeal introduction of the process may turn out to bee too prohibitive for the overall success when problems are solved in an ad-hoc manner [15]. While this thesis work has provided means and implementation for many of the components, it does not declare the complementary organisational guidelines how to develop. By following these guidelines developers understand fundamental issues,

such as the definition of done for a development task, verifying the acceptable quality, integrating new code to the code base, reporting defects, and so on. The importance of established guidelines and disciplines simply cannot be overemphasised.

Achieving extensive testing coverage by starting from scratch can be difficult. It may require rethinking the organisation's assets, as it was done in [46]. Instead of focusing on the mere application code, they took a more holistic view of the process and considered testing and validation as part of the application. Currently, our organisation is in a similar situation. There were no well-established, organisation-wide testing guidelines available. The level and quality of testing is up the project team and customer requirements. Due to this, testing was left out of the thesis work scope. It would be feasible to start a project to study and define the guidelines. The results of that projects should then be used to complement the work done in this thesis.

**Continuous Integration and Agile Software Development Methods**

Continuous integration seems to be fitting well with agile development methods, as experienced in [19, 22]. Both agile methods and CI are incremental in nature and share many common practices, thus adopting agile can help introducing CI. On the other hand, CI can also help adopting agile development methods. CI was experienced as one of the key factors for success in [46] where a large organisation, called Gap Inc. Direct, decided to adopt an agile development method. They selected an ambitious, large-scale pilot project for the transformation to prove the value of agile approach. Massive investment were made in continuous integration to be able to handle such a large project. Significant investments were also put to automation and testing. The system was robust enough to handle 600-1200 builds a day and moving terabytes of compiled code and data between environments. In essence, by following the CI practices they were still able to have code that is deployable every day. The results also show that the importance of frequent integrations becomes even more critical when managing very large projects.

Sesca Mobile Software Oy has already introduced agile practices and is fairly accustomed working with Scrum. We have established training material and whole teams, not just the developers, are trained to Scrum practices as needed. We are not expecting any major issues combining CI into our Scrum process. We also conclude that it is recommended to complement CI with an agile development method.

## 6.3 Conclusions

Continuous integration can significantly improve the overall results of a software development project. Common risks involved can be reduced by integrating changes frequently. This way defects can be detected and fixed earlier, at the time when the change is checked into the version control system rather than during late-cycle testing. Because the integration includes automated tests and code inspection, the health of software is measurable. Continuously measuring the state of the code helps to take corrective actions before the defect accumulates. Continuously integrating in a clean environment also reduces assumptions towards individual developer's local environment.

CI reduces repetitive processes by extensive automation throughout all project activities. The integration process will be run automatically the same way every time a change occurs in VCS. This frees people to do more higher-value and productive work. Another important point is being able to generate deployable software at any point in time. From the outside perspective this is the most tangible asset. The clients or users do not want to concern so much about improving the process quality or reducing the risks, but desire to see the working product. The benefit compared to a non-CI process is being able to apply fixes to the software immediately, integrate and release without any additional cost.

Project visibility can be improved by providing continuous feedback. Having the latest information about build status and quality metrics is important for making effective decisions. Often, the situation is that project members collect this information manually which quickly becomes a tedious effort and may be neglected. The team can also observe current trends in the build results and overall quality when the frequent integrations provide the continuous statistics.

This thesis work has studied in detail how the described benefits can be achieved and gathered practical results from earlier attempts to introduce the process. The CI process is built on top of the fundamental software engineering practices described in chapter two. They are considered as best practices by the industry and should be seen as disciplines to strive after. We have also learned through the experiences in chapter three, that it is advisable to establish a major part of the CI components and developing guidelines prior to introducing the process. Another issue discovered among the related work was keeping the build times down. It can have significant effect on the team behaviour.

The main objective was implementing the technical infrastructure for CI. During evaluation the various build tools and CI servers it was discovered that more time and first-hand experience would have been needed. This is simply due to the vast amount of tools. They all have trade-offs, but recognising and evaluating them according to our criteria cannot be done without first learning the tool and using it. In the organisation's case, we tend to use the building tools already familiar to us. The CI server was selected so that it integrates nicely with the build tools and made sure that all requirements can be fulfilled. We believe that the chosen tools fit the best to our work, which is the most important point in the end.

## 6.4 Future Work

The work done in this thesis does not yet provide a complete CI process and quite a few improvements to the build system can be still made. The most notable issue in the process side is the lack of guidelines. It may not be possible to develop a single guideline for all project types conducted in the organisation, but they could be done according to the software platform or framework: Symbian/S60, Maemo, Qt, GTK+, generic C/C++, and so on. The second issue is related to testing the code. Achieving automatic testing requires that the proper tools are found for each of the main project types and the guidelines for implementing the tests are established.

The build system usability can be improved by implementing the changes proposed in chapter 6.1. The Buildbot software itself is also being developed steadily, meaning that newer versions are released. Furthermore, plans for the Buildbot version 1.0 have been specified [41]. The new major release sets more emphasis towards software testing, thus the 1.0 version is currently described as *Python-based continuous integration testing framework*. From our point of view, this looks a promising development trend and could be further evaluated when the appropriate release is out.

# REFERENCES


1 V. Stavridou. Integration in software intensive systems. Journal of Systems and Software archiveVolume 48, Issue 2, pages 91 - 104. 1999. ISSN: 0164-1212.

2 W. Royce. Software project management: a unified framework. Addison-Wesley Longman Publishing Co., Inc., pages 12 - 16. 1998. ISBN: 0-201-30958-0.

3 P. Duvall, S. Matyas, and A. Glover. Continuous Integration: Improving Software Quality and Reducing Risk. Addison-Wesley Professional, 336 pages. 2007. ISBN: 9780321336385.

4 B. Boehm and P. Papaccio. Understanding and Controlling Software Costs. IEEE Transactions on Software Engineering, pages 1462-1477. 1998. ISSN: 0098-5589.

5 M. Fowler. Continuous Integration. [http://www.martinfowler.com/articles/continuousIntegration.html]. Cited 21.2.2009. Updated May 2006.

6 S. Mcconnell. Daily Build and Smoke Test. IEEE Software Volume 13, Issue 4, pages 142-143. 1996. ISSN: 0740-7459.

7 M. Zawadzki. Drawing the Line: Continuous Integration and Build Management. [http://downloads.urbancode.com/Drawing%20the%20Line_Continuous %20Integration%20and%20Build%20Management.pdf]. Cited 2.3.2009. Updated 2007.

8 D. Spinellis. Software Builders. IEEE Software, Volume 25, Issue 3, pages 22-23. 2008. ISSN: 0740-7459.

9 N. Kitiyakara and J. Graves. Growing a Build Management System from Seed. Proceedings of the AGILE 2007, pages 401-407. 2007. ISBN: 0-7695-2872-4.

10 Subversion open source version control system. [http://subversion.tigris.org/]. Cited 6.3.2009. Updated 2008.

11 Git - Fast Version Control System. [http://git-scm.com/]. Cited 6.3.2009. Updated May 2009.

12 CVS - Concurrent Versions System. [http://www.nongnu.org/cvs/]. Cited 6.3.2009. Updated December 2006.

13 Stephen P. Berczuk and B. Appleton. Software Configuration Management Patterns: Effective Teamwork, Practical Integration. Addison-Wesley Longman Publishing Co., Inc., pages 82-84. 2002. ISBN: 0201741172.

14 K. Schwaber and M. Beedle. Agile Software Development with Scrum. Prentice Hall PTR, 158 pages. 2001. ISBN: 0130676349.

15 S. Koroorian and M. Kajko-Mattsson. A Tale of Two Daily Build Projects. Proceedings of the 2008 The Third International Conference on Software Engineering Advances, pages 245-251. 2008. ISBN: 978-0-7695-3372-8.

16 QTestLib Manual. [http://doc.trolltech.com/4.5/qtestlib-manual.html]. Cited 4.3.2009. Updated 2009.

17 Open source testing tools. [http://www.opensourcetesting.org]. Cited 11.4.2009. Updated April 2009.

18 K. Beck and C. Andres. Extreme Programming Explained: Embrace Change (2nd Edition). Addison-Wesley Professional, 224 pages. 2004. ISBN: 0321278658.

19 A. Miller. A Hundred Days of Continuous Integration. Proceedings of the Agile 2008, EEE Computer Society Washington, DC, USA, pages: 289-293. 2008. ISBN: 978-0-7695-3321-6.

20 M. Kajko-Mattsson, M. Jonson, S. Koroorian, and F. Westin. Lesson learned from attempts to implement daily build. Proceedings of the Eighth Euromicro Working Conference on Software Maintenance and Reengineering, pages 137-146. 2004. ISBN: 0-7695-2107-X.

21 E. Karlsson, L. Andersson, and P. Leion. Daily build and feature development in large distributed projects. Proceedings of the 22nd international conference on Software engineering, ACM, pages 652-654. 2000. ISBN: 1-58113-206-9.

22 G. Brooks. Team Pace - Keeping Build Times Down. Proceedings of the Agile 2008, IEEE Computer Society Washington, DC, USA, pages 294-297. 2008. ISBN: 978-0-7695-3321-6.

23 Javadoc Tool Home Page. [http://java.sun.com/j2se/javadoc/]. Cited 19.02.2009. Updated February 2004.

24 Source code documentation generator tool. [http://www.doxygen.org]. Cited 19.02.2009. Updated February 2009.

25 Continuous Integration Feature Matrix. [http://confluence.public.thoughtworks.org/display/CC/Understanding+the+alternatives+to+CruiseControl]. Cited 21.3.2009. Updated February 2009.

26 GNU Make Manual. [http://www.gnu.org/software/make/manual/]. Cited 20.02.2009. Updated May 2006.

27 M. Doar. Practical Development Environments. O'Reilly; Associates, Inc., pages 94-99, 116-120. 2005. ISBN: 0596007965.

28 R. Mecklenburg. Managing Projects with GNU Make (Nutshell Handbooks). O'Reilly Media, Inc., pages 94-108. 2004. ISBN: 0596006101.

29 Linux Kernel Development. [http://www.linuxfoundation.org/publications/linuxkerneldevelopment.php]. Cited 20.2.2009. Updated April 2008.

30 Unix / Linux Shell Scripting Tutorial. [http://steve-parker.org/sh/sh.shtml]. Cited 20.2.2009. Updated May 2008.

31 P. Miller. Recursive Make Considered Harmful. AUUGN Journal of AUUG Inc., 19, pages 14-25. 1998.

32 SCons home page. [http://www.scons.org/]. Cited 21.3.2009. Updated February 2009.

33 Why the KDE project switched to CMake -- and how. [http://lwn.net/Articles/188693/]. Cited 21.3.2009. Updated June 2006.

34 CMake: the cross-platform, open-source build system. [http://www.cmake.org/]. Cited 21.3.2009. Updated February 2009.

35 QMake for Qt projects. [http://doc.trolltech.com/4.5/qmake-common-projects.html]. Cited 21.3.2009. Updated 2009.

36 Apache Ant. [http://ant.apache.org/]. Cited 19.02.2009. Updated January 2009.

37 CruiseControl. [http://cruisecontrol.sourceforge.net/]. Cited 21.2.2009. Updated 2009.

38 K. Sturdevant. Cruisin' and Chillin': Testing the Java-Based Distributed Ground Data System "Chill" with CruiseControl System "Chill" with CruiseControl. Aerospace Conference, IEEE, pages 1-8. 2007. SBN: 1-4244-0525-4.

39 J. Hill, D. Schmidt, A. Porter, and J. Slaby. CiCUTS: Combining System Execution Modeling Tools with Continuous Integration Environments. 15th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems, pages 66-75. 2008. ISBN: 0-7695-3141-5.

40 A. Holmes and M. Kellogg. Automating functional tests using Selenium. Agile Conference, pages 275-281. 2006. ISBN: 0-7695-2562-8.

41 Buildbot project. [http://buildbot.net/]. Cited 22.2.2009. Updated 2009.

42 Using Chroot Securely. [http://www.linuxsecurity.com/content/view/117632/171/]. Cited 24.2.2009. Updated 2007.

43 VirtualBox virtualisation tool. [http://www.virtualbox.org]. Cited 3.3.2009. Updated March 2009.

44 OpenSSH Public Key Authentication. [http://sial.org/howto/openssh/publickey-auth/]. Cited 25.2.2009. Updated October 2008.

45 Using CruiseControl for C++. [http://confluence.public.thoughtworks.org/display/CC/UsingCruiseControlWithCplusPlus]. Cited 11.4.2009. Updated November 2008.

46 D. Goodman and M. Elbaz. "It's Not the Pants, it's the People in the Pants" Learnings from the Gap Agile Transformation -- What Worked, How We Did it, and What Still Puzzles Us. Proceedings of the Agile 2008, IEEE Computer Society Washington, DC, USA, pages 112-115. 2008. ISBN: 978-0-7695-3321-6.

# Buildbot Example Configuration

```
# ex: set syntax=python:

# This is the dictionary that the buildmaster pays attention to.
c = BuildmasterConfig = {}

# repository URL with the latest trunk
svn_url = "https://check.svn.sourceforge.net/svnroot/check/trunk"

####### BUILDSLAVES

# the 'slaves' list defines the set of allowable buildslaves.
from buildbot.buildslave import BuildSlave
c['slaves'] = [BuildSlave("slave1", "passwd")]

# 'slavePortnum' defines the TCP port to listen on.
c['slavePortnum'] = 9991

####### CHANGESOURCES

# Use SVNPoller to check changes from the repository
from buildbot.changes.svnpoller import SVNPoller
c['change_source'] = SVNPoller(svn_url);

####### SCHEDULERS

# Add scheduler for default branch
from buildbot.scheduler import Scheduler
c['schedulers'] = []
c['schedulers'].append(Scheduler(name="all", branch=None,
                                 treeStableTimer=60,
                                 builderNames=["buildbot-full"]))


####### BUILDERS

# The factory.BuildFactory is the base class, and is configured with
# a series of BuildSteps. When the build is run, the appropriate
# buildslave is told to execute each Step in turn.

from buildbot.steps import source, shell
from buildbot.process import factory

f1 = factory.BuildFactory()
f1.addStep(source.SVN(svnurl = svn_url))
f1.addStep(shell.ShellCommand(command=["autoreconf", "--install"]))
f1.addStep(shell.ShellCommand(command=["./configure"]))
f1.addStep(shell.ShellCommand(command=["make", "all"]))
f1.addStep(shell.ShellCommand(command=["make", "check"]))

# the 'builders' list defines the Builders. Each one is configured
# with a dictionary, using the following keys:
# name (required): the name used to describe this builder
# slavename (required):
#   which slave to use, must appear in c['slaves']
# builddir (required): which subdirectory to run the builder in
# factory (required): a BuildFactory to define how the build is run
# periodicBuildTime (optional):
#   if set, force a build every N seconds
```

```
b1 = {'name': "buildbot-full",
      'slavename': "slave1",
      'builddir': "full",
      'factory': f1}

c['builders'] = [b1]


####### STATUS TARGETS

# The results of each build will be pushed to these targets
c['status'] = []

from buildbot.status import html
c['status'].append(html.WebStatus(http_port=8030, allowForce=True))

####### PROJECT IDENTITY

c['projectName'] = "Check"
c['projectURL'] = "http://check.sourceforge.net/"

# the 'buildbotURL' string should point to the location where the buildbot's
# internal web server (usually the html.Waterfall page) is visible.

c['buildbotURL'] = "http://172.18.12.21:8030/"
```

# Utility Script for Initialising a New Project

```
#!/bin/bash

usage()
{
    echo "usage: $0 [project]"
    echo "Creates a new project called [project]. Run as root."
}

# function for checking the results after system call.
check_result()
{
    if [ $? -ne 0 ]; then
        echo "Error occurred, exiting now."
        exit 1
    fi
}

# check correct user, must be root
if [ ! $( id -u ) -eq 0 ]; then
    usage
    exit 1
fi

# check parameters
if [ $# -ne 1 ]; then
    usage
    exit 1
fi

project=$1
user=$project

# Add project account
adduser $user
check_result

# create build dir for project
mkdir /work/builds/$project
check_result
mkdir /work/builds/$project/master
check_result
mkdir /work/builds/$project/slaves
check_result

# Set owner of the project directory
chown -R $user:$user /work/builds/$project/
check_result

# Set access permissions
chmod 0770 -R /work/builds/$project/
check_result

# Do the same commands for the repos dir
mkdir /work/repos/$project
check_result
chown -R $user:$user /work/repos/$project/
check_result
chmod 0770 -R /work/repos/$project/
check_result
```

# Preparing PKI Authentication for Non-Interactive SSH-based VCS Access

The following command generates the public and private keys for PKI authentication:

```
$ ssh-keygen -t rsa
Generating public/private rsa key pair.
Enter file in which to save the key (/home/example-
project/.ssh/id_rsa):
```

The keys should be saved to the default location. The pass phrase is left empty by pressing Enter key. Using empty pass phrase is considered as an insecure method because losing the private key would let anyone to login without authentication. Hence this key pair must only be used locally on this particular server. The public key is copied into the user account's authorised keys:

```
$ cat ~/.ssh/id_rsa.pub >> ~/.ssh/authorized_keys
```

Authentication for SSH-based access method without password or pass phrase can be verified now. When accessing a server via SSH for the first time or after the IP address has changed, the authenticity of the host needs to be confirmed:

```
$ ssh localhost
The authenticity of host '(192.168.1.33)' can't be
established.
RSA key fingerprint is
a7:c6:70:3e:00:77:73:ed:90:b1:9a:bc:e7:d5:ba:32.
Are you sure you want to continue connecting (yes/no)?
```

After confirming authenticity, the following connections are usually established directly without user interaction.

# Chroot Environment Setup

Creating a new chroot environment where `$PROJECT` is the name of the project:

```
$ fakeroot fakechroot debootstrap --variant=fakechroot
--arch=i386 lenny /work/builds/$PROJECT/slaves/chroot
http://ftp.fi.debian.org/debian/
```

Login as root to the sandbox for installing software dependencies:

```
$ fakeroot fakechroot chroot /work/builds/
$PROJECT/slaves/chroot
```

Login as a normal user and verify that the code compiles by hand:

```
$ fakechroot chroot /work/builds/$PROJECT/slaves/chroot
```

# Script for Creating Build Slave

```
#!/bin/bash

usage()
{
    echo -e "Creates a Buildbot slave \
and adds maintenance entries to user crontab\n"
    echo -e "Usage: $0 [-b basedir] [-m host:port] \
[-u username] [-p password]\n"
    echo -e "-b\tPath to builder base directory\n"
    echo -e "-m\tLocation of the build master as host:port pair\n"
    echo -e "-u\tUsername (as set in master.cfg for the slave)\n"
    echo -e "-p\tPassword (as set in master.cfg for the slave)\n"
}

basedir=
master=
username=
password=

# parse parameters
while getopts b:m:u:p: param
    do
    case $param in
        b)  basedir="$OPTARG";;

        m)  master="$OPTARG";;

        u)  username="$OPTARG";;

        p)  password="$OPTARG";;

        ?) usage
        exit 1;;
        esac
done

# Check that we got parameters
if [[ -z $basedir ]] || [[ -z $master ]] || [[ -z $username ]] || [[ -z
$password ]]
    then
    usage
    exit 1
fi

# check builder dir exists / can be created
if [ ! -e $basedir ]; then
    mkdir $basedir

    if [ $? -ne 0 ]; then
        exit 1
    fi
fi



buildbot create-slave $basedir $master $username $password
```

```
if [ $? -ne 0 ]
    then
    echo "Failed to create slave."
    exit 1
fi

# Temporary file for updating crontab task list
TMPFILE=/tmp/tmpfile

crontab -l > TMPFILE 2> /dev/null

# Clears old log files from build master directory
echo "# Slave" >> TMPFILE
echo "@weekly cd $basedir && \
    find twistd.log* -mtime +7 -exec rm {} \;" >> TMPFILE

# Start slave at boot
echo "@reboot  /usr/bin/buildbot start $basedir"  >> TMPFILE

# Set new crontab to the user and print it to console
crontab TMPFILE
rm TMPFILE

# List current crontab tasks
echo -e "\nCurrent crontable:"
crontab -l
echo
```

# Script for Creating Build Master

```
#!/bin/bash

if [ $# -ne 1 ]; then
    echo "usage: $0 [basedir]"
    echo "Creates a buildbot master and \
    adds maintenance entries to user crontab"
    exit 1
fi

basedir=$1

# check master dir exists / can be created
if [ ! -e $basedir ]; then
    mkdir $basedir

    if [ $? -ne 0 ]; then
        exit 1
    fi
fi

# Create the master
buildbot create-master $basedir

if [ $? -ne 0 ]
    then
    echo "Failed to create master, exiting now."
    exit 1
fi

# Temporary file for updating crontab task list
TMPFILE=/tmp/tmpfile

crontab -l > TMPFILE 2> /dev/null

# Clears old log files from build master directory
echo "# Master" >> TMPFILE
echo "@weekly cd $basedir && \
    find twistd.log* -mtime +7 -exec rm {} \;" >> TMPFILE
echo "@weekly cd $basedir && \
    find . -mindepth 2 -type f -mtime +14 \
    | grep -v public_html | xargs rm" >> TMPFILE

# Start master at boot
echo "@reboot /usr/bin/buildbot start $basedir"  >> TMPFILE

# Set new crontab
crontab TMPFILE
rm TMPFILE

# List current crontab tasks
echo -e "\nCurrent crontable:"
crontab -l
echo
```

# Executing Command in a Chroot Environment

```
#!/usr/bin/env python

import subprocess
import sys
import os

# helper script for running Buildbot commands inside chroot jail.
# The scripts expects to find a working chroot in the given path.

# check arguments
if len(sys.argv) < 4:
    print "usage: " + sys.argv[0] + " <chroot>" + \
    " <build_path_in_chroot>" + " <command>" + " [parameters]"
    exit (-1)

# parse parameters, take commands as a list
mylist = list(sys.argv)
mylist.pop(0) # pop script name

chroot = mylist.pop(0)
path = mylist.pop(0)
cmds = mylist

# try to enter chroot jail
try:
    os.chroot(chroot)
except OSError:
    print "Failed to access chroot: " + chroot
    exit (-1)
except:
    print "Unexpected error:", sys.exc_info()[0]
    exit (-1)


# try to change current directory in chroot
try:
    os.chdir(path)
except OSError:
    print "Failed to change directory in chroot: " + path
    exit (-1)
except:
    print "os.chdir(): unexpected error:", sys.exc_info()[0]
    exit (-1)


# Ensure that current working directory is set correctly
os.environ['PWD'] = path

# last thing, try to run the given command, save return value
try:
    ret = subprocess.check_call(cmds)
except:
    print "subprocess.check_call(): Unexpected error:", sys.exc_info()[0]
    exit (-1)

exit (ret)
```