

Combining Testing Power of Dependent Software Engineering Projects

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering / Internet Computing

eingereicht von

Stefan Dösinger

Matrikelnummer 0526822

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dr. Stefan Biffl
Mitwirkung: Dr. Richard Mordinyi

Wien, 29.1.2013

(Unterschrift Verfasser)

(Unterschrift Betreuung)

Combining Testing Power of Dependent Software Engineering Projects

MASTER'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering / Internet Computing

by

Stefan Dösinger

Registration Number 0526822

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dr. Stefan Biffl
Assistance: Dr. Richard Mordinyi

Vienna, 29.1.2013

(Signature of Author)

(Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Stefan Dösinger
Anschützgasse 17/10, 1150 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum)

(Unterschrift Verfasser)

Acknowledgements

This work has been supported by the Christian Doppler Forschungsgesellschaft and the BMWFJ, Austria.

Abstract

This thesis introduces and evaluates the *Continuous Change Impact Analysis Process*, short *CCIP*, an extension to continuous integration.

Continuous integration (CI) is a well-established concept to automate building and testing of software projects, with the goal of improving quality and time to delivery. Automated tests, which are an integral part of CI, test whether a piece of software behaves according to its specification. However, the behavior of software is not defined by its code alone, but also by external dependencies that provide part of the functionality and may be developed by outside organizations or outside the engineering domain. Currently CI falls short of its potential because it does not take into account changes to dependencies and therefore executes tests in an isolated environment.

CCIP attempts to break up the isolation by introducing communication between CI servers. If a dependency is modified, a notification is sent to its dependents, which import the changes, run their own tests and send feedback. This provides quick feedback for dependency developers and helps to discover regressions earlier.

This thesis uses a prototype CCIP implementation to answer research questions concerning the costs and benefits of the extension, feedback quality and expected issues in large-scale deployments. This prototype implementation has been evaluated with two sets of interdependent open source projects consisting of a total of 10 projects. One set of projects originates from the Linux 3D driver and gaming stack, the other centers on the Apache Foundation's OSGi implementation.

The empirical evaluation showed that CCIP can discover additional regressions that slip through the dependencies' own tests, but API changes, false-positives and random test failures severely reduce the usefulness of CCIP.

Keywords: software testing, software dependencies

Kurzfassung

Diese Diplomarbeit beschreibt den *Continuous Change Impact Analysis Process*, kurz *CCIP*, und untersucht und bewertet dessen Eigenschaften.

Der CCIP ist eine Erweiterung für den Prozess der kontinuierlichen Integration (engl. *continuous integration* oder *CI*). CI ist eine verbreitete Methode, um das Kompilieren und Testen von Softwareprojekten zu automatisieren, mit dem Ziel die Softwarequalität zu verbessern und Auslieferungszeiten zu verkürzen. Eine der tragenden Komponenten von CI sind automatisierte Tests, die überprüfen, ob sich die Software entsprechend ihrer Spezifikation verhält. Das Verhalten der Software wird jedoch nicht nur vom Quellcode alleine bestimmt, sondern auch von externen Komponenten, die importiert werden und wichtige Funktionen bereitstellen. Diese Abhängigkeiten werden oft außerhalb des Projektteams entwickelt und entspringen manchmal von einem gänzlich anderen Fachgebiet. Da CI Änderungen an Abhängigkeiten nicht betrachtet, werden Tests in einer zu isolierten Umgebung ausgeführt, wodurch CI nicht sein ganzes Potential entfalten kann.

CCIP versucht diese Isolation aufzubrechen, indem es CI-Server untereinander kommunizieren lässt. Wenn eine Abhängigkeit aktualisiert wird, wird eine Benachrichtigung an die CI-Server von abhängigen Projekten geschickt. Diese importieren die geänderten Artefakte, führen ihre eigenen Tests aus und liefern das Ergebnis zurück. Dadurch erhalten Entwickler von Abhängigkeiten frühzeitig automatisiertes Feedback darüber, wie sich ihre Änderungen in den von ihnen abhängigen Projekten auswirken.

Diese Diplomarbeit verwendet eine prototypische Implementierung von CCIP um das Verfahren zu testen und Forschungsfragen zu beantworten. Diese Forschungsfragen betreffen die Kosten und Nutzen von CCIP, die Qualität des Feedbacks und den Umgang damit, sowie Fragestellungen und Herausforderungen bei großflächigem Einsatz von CCIP. Um diese Fragen zu beantworten, wird CCIP mit zwei Gruppen von Projekten aus unterschiedlichen Entwicklungsumgebungen und Ökosystemen getestet. Eine Gruppe stammt aus dem Umfeld des 3D-Renderings unter Linux, die andere aus der OSGi-Implementierung der Apache Foundation.

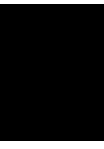
Die empirische Evaluation zeigt, dass CCIP zusätzliche Fehler finden kann, die den Tests eines Projekts entgehen. Jedoch bereiten Änderungen an den Programmierschnittstellen, falsche Fehlermeldungen und zufällige Testfehler Probleme und reduzieren die Nützlichkeit von CCIP.

Schlüsselwörter: Softwaretest, Softwareabhängigkeiten

Contents

1	Introduction	1
1.1	Overview	1
1.2	Motivating Scenarios	2
1.3	Intended Benefits	3
1.4	Thesis Structure	3
2	Testing and Continuous Integration	5
2.1	Goals of Continuous Integration	5
2.2	Automated Building	6
2.3	Testing	8
2.4	Source Code Repository	10
2.5	Continuous Integration Servers	11
2.6	Best Practices	13
2.7	Experiences	14
3	Software Dependencies	15
3.1	What are Software Dependencies	15
3.2	Linking Types	16
3.3	Dependency Management	17
3.4	Dependency Anomaly: The Bootstrapping Problem	19
4	Impact Management	21
4.1	Terms	21
4.2	Tool Support	22
4.3	Collaboration Strategies	24
4.4	Managing Open Source Software	24
4.5	Building GNU/Linux Distributions	25
5	Research Issues	27
5.1	Research Issues	29
5.2	Use Cases	30
5.3	Evaluation	30
6	Proposed Solution Approach	35

6.1	The Continuous Change Impact Analysis Process Workflow	35
6.2	Communication Protocol	38
6.3	Artifact Transfer and Merging	39
6.4	Prototype Implementation	39
7	Evaluation	45
7.1	Evaluated Projects	45
7.2	Evaluation Setup	53
7.3	Evaluation Results	54
8	Discussion	63
8.1	Findings	63
8.2	Answers to Research Issues	65
9	Conclusion and Future Work	69
9.1	Conclusion	69
9.2	Future Work	69
A	Tools Used and Copyright Attribution	73
	List of Figures	74
	Bibliography	75



Introduction

This chapter provides an overview of the contribution of the thesis, the existing problem scenarios that motivated the research and intended goals.

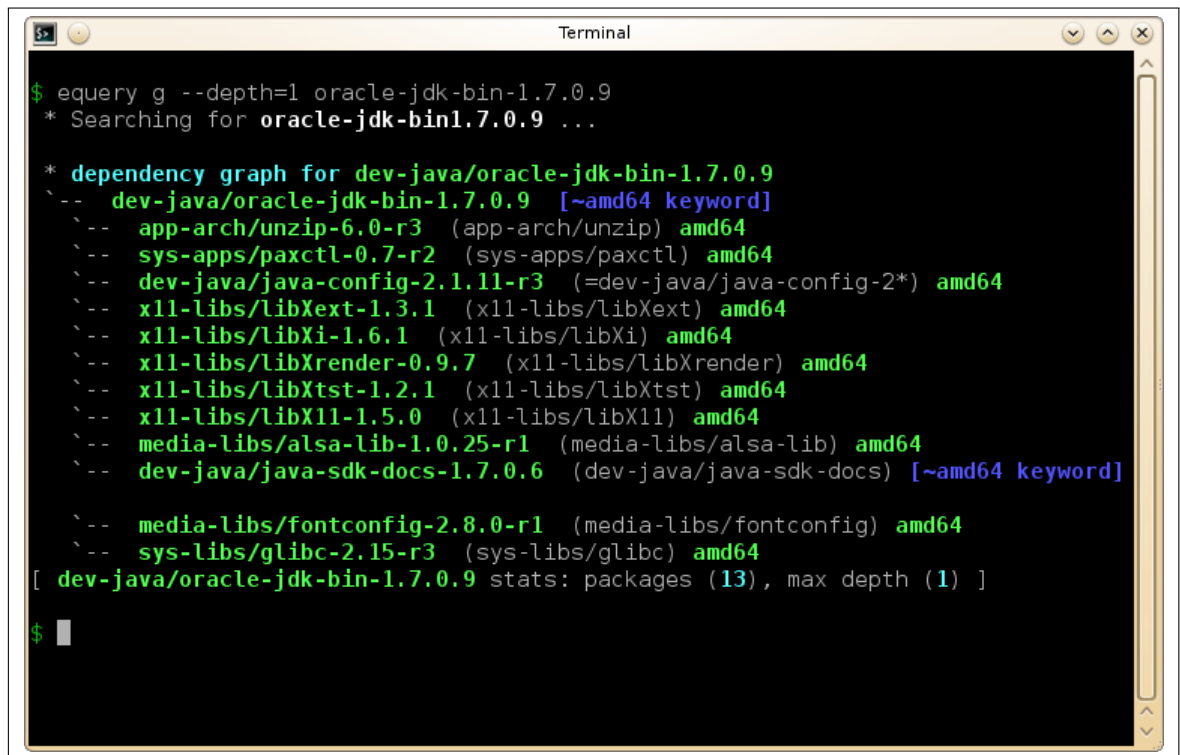
1.1 Overview

In today's software engineering landscape, *continuous integration (CI)* is a well-established concept to improve software quality and reduce time to delivery. Continuous integration automates certain tasks, in particular code compilation and executing tests.

Testing is an important part of continuous integration, but testing happens in a kind of isolated environment. Tests evaluate a specific code artifact and are written based on and according to this code artifact's specifications. However, the code of a software artifact is not isolated from the rest of the world. Most software imports other software to provide a part of its functionality. These *dependencies* are often developed outside the project's organization or engineering domain. Dependencies themselves may depend on other software, causing transitive dependencies. Some software is imported by other projects (the software *dependents*). Figure 1.1 shows the dependencies of the Java Development Kit package in Gentoo Linux as an illustrative example.

The first relationship is handled to a varying degree by build systems and package managing software, while the latter has almost no tool support. Upgrading dependencies and finding problems introduced by those updates is a tedious and slow process that has to be done manually [13]. It is common for dependents to wait for final releases of dependencies before they update. As a result, feedback from dependents arrives late and problems are not detected until after a release. At this point, fixing a regression and delivering the fix to dependent projects requires more effort than earlier in the development cycle.

To address these shortcomings, this thesis proposes, implements and evaluates the *Continuous Change Impact Analysis Process*, or *CCIP*, an extension to continuous integration which enables CI servers to enter a two-way communication. This communication allows them to no-



```
$ equery g --depth=1 oracle-jdk-bin-1.7.0.9
* Searching for oracle-jdk-bin-1.7.0.9 ...

* dependency graph for dev-java/oracle-jdk-bin-1.7.0.9
  -- dev-java/oracle-jdk-bin-1.7.0.9 [~amd64 keyword]
    -- app-arch/unzip-6.0-r3 (app-arch/unzip) amd64
    -- sys-apps/paxctl-0.7-r2 (sys-apps/paxctl) amd64
    -- dev-java/java-config-2.1.11-r3 (=dev-java/java-config-2*) amd64
    -- x11-libs/libXext-1.3.1 (x11-libs/libXext) amd64
    -- x11-libs/libXi-1.6.1 (x11-libs/libXi) amd64
    -- x11-libs/libXrender-0.9.7 (x11-libs/libXrender) amd64
    -- x11-libs/libXtst-1.2.1 (x11-libs/libXtst) amd64
    -- x11-libs/libX11-1.5.0 (x11-libs/libX11) amd64
    -- media-libs/alsa-lib-1.0.25-r1 (media-libs/alsa-lib) amd64
    -- dev-java/java-sdk-docs-1.7.0.6 (dev-java/java-sdk-docs) [~amd64 keyword]

    -- media-libs/fontconfig-2.8.0-r1 (media-libs/fontconfig) amd64
    -- sys-libs/glibc-2.15-r3 (sys-libs/glibc) amd64
[ dev-java/oracle-jdk-bin-1.7.0.9 stats: packages (13), max depth (1) ]

$
```

Figure 1.1: Dependencies of the Java Development Kit package in Gentoo Linux

tify each other about dependency upgrades, transfer code artifacts and report back test results for automated testing of changes beyond the project boundary.

1.2 Motivating Scenarios

This section describes real-world development scenarios that provided motivation for the research topic of this work.

As virtually all software depends to some extent on other software, changes to the dependencies can introduce problems for the software that depends on them, for example [18] or [40]. If this problem is detected after a new version of the dependency is released, removing the problem not only requires a fix to the dependency, but also necessitates waiting for a new release of that dependency. This is highly inconvenient for the developers of the dependent software, because they cannot upgrade to the new version to profit from improvements and new features.

To make matters worse, in some situations the developers of the dependent do not have control over which version of the dependency is used, e.g. because it is controlled by a Linux distribution (See chapter 3). In this case, the developers of the dependent software may have to employ complicated workarounds for the problem, explain to disappointed users why their software doesn't work and why they can't fix it or guide users through the complicated steps of

installing a different version of the dependency. Neither of these options is particularly appealing.

To add insult to injury, the dependent's developers might realize that their own conformance tests recognize the bug, and could have provided an automated way to detect the problem much sooner.

The developers of the dependency might realize that the regression occurs in a complicated, but legitimate scenario, and that their own tests did not catch it because they either did not think of this scenario, or because not all the components necessary to reconstruct the problem are available to them. At the time when the problem is reported, it is often difficult to isolate the change that caused it because a lot of time has passed since it was introduced.

1.3 Intended Benefits

The intended benefits of the contribution are threefold:

- Help find cross-project regressions earlier and with less manual work.

This benefits dependencies and dependents alike. Dependencies profit by receiving bug reports from their dependents earlier, ideally minutes after a change introduced a regression. Dependents profit by being able to detect upcoming problems quickly and with little or no manual effort.

- Give dependencies an easy way to extend their test coverage.

Ideally, a software project has automated tests that cover its entire codebase, and a continuous integration infrastructure that executes those tests on all supported platforms. In practice, complete coverage requires a lot of effort to achieve, and some configurations or conflicts with other software may slip through the net. Including the tests of dependent projects helps close the gap in the test coverage and highlights problems the developers of the dependency have not considered.

- Create a network of communicating CI servers.

With support for cascading updates, a large-scale deployment of communicating CI servers is possible. Such a network of communicating servers could do testing on a huge set of interdependent open source software for example.

1.4 Thesis Structure

The remainder of the thesis is structured as the following: Chapters 2-4 report on background and related work regarding testing, software dependencies and impact management. Chapter 5 describes use cases to clarify the problem statement and presents research issues concerning the effective and efficient handling on change impacts. Chapter 6 explains the main contribution of the paper, while chapter 7 and 8 discuss the evaluation and its results. Finally, Chapter 9 concludes the paper and presents future work.

Testing and Continuous Integration

This chapter provides background information about the goals, methods and limitations of software testing and continuous integration. State-of-the-art continuous integration concepts form a basis for the ideas presented in this thesis.

2.1 Goals of Continuous Integration

The goal of continuous integration is to automate key parts of the software engineering process to improve software quality, accelerate development and free developers from manually executing repetitive tasks. Advantages of continuous integration are [24]:

- Reduced risk: Early testing helps detect problems earlier.
- Find bugs early: Constant testing and small changes pin down new bugs to the recently changed areas of code.
- Fewer entangled bugs: Since bugs are detected and fixed early, complicated interactions between multiple bugs are less likely.
- Faster deployment cycles: Changes can be shipped faster.
- It supports communication between developers (see chapter 4).

To achieve these goals, a few changes to the technical project setup and the development process are required. The key parts of continuous integration are an automated build system, automated tests, the use of a revision control system and the requirement that developers commit changes to the central code repository early and often and regularly pull in changes from their coworkers. Finally a continuous integration server is responsible for building and testing the project whenever changes are committed [16].

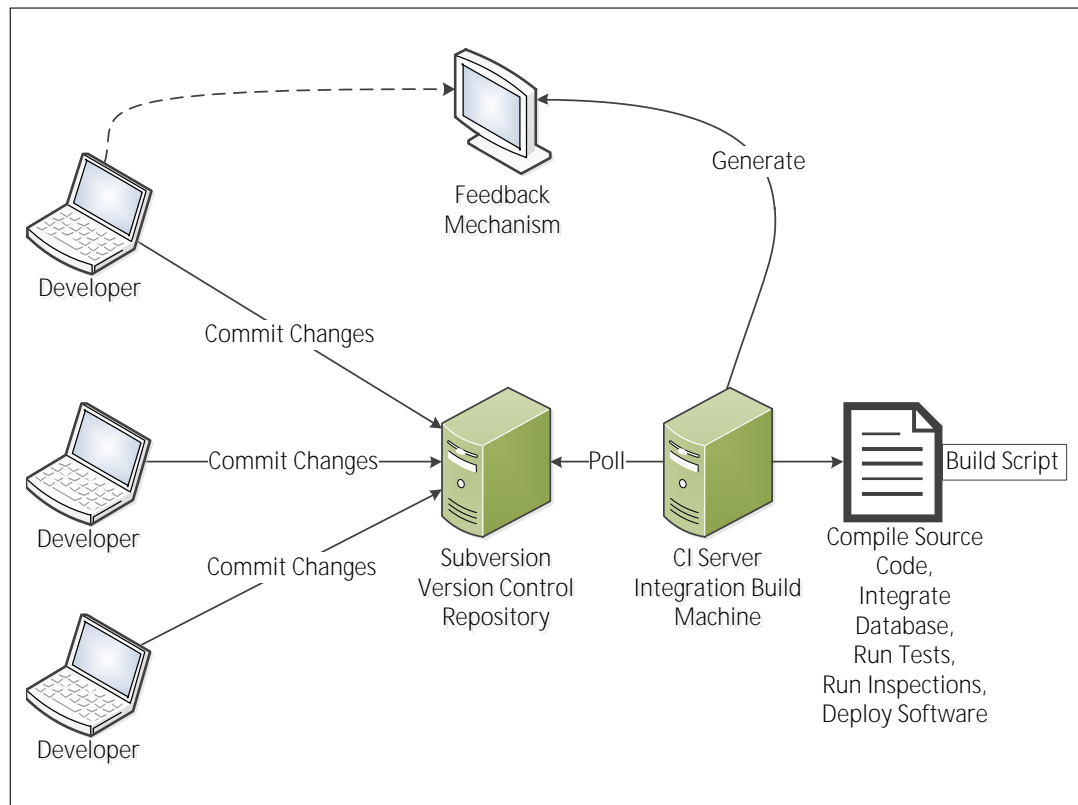


Figure 2.1: CI process overview [16, Fig. 1.1]

Figure 2.1 gives a high-level overview of the continuous integration process, described briefly in the following: Developers commit their changes to a source code repository. The continuous integration server polls the source code repository. Whenever it detects changes it uses the build script to build and test the code. It generates a report which developers can receive via different feedback mechanisms (e.g. e-mail, webpage).

2.2 Automated Building

The first requirement of continuous integration is an automated build system that is able to build the entire project by executing one command without any manual intervention. This chapter gives an overview over the topic and describes two major build systems used by software used in the evaluation part of this thesis.

All major development platforms (e.g. Java, GNU/Linux, Microsoft Windows, Microsoft .NET) provide such build systems with easy-to-use templates and strongly encourage their use. It is up to the developers of the project to make use of them and integrate their custom build tasks into the existing templates.

While build systems can be as simple as a platform-specific script performing a few compiler

invocations, modern build systems provide a number of services like dependency management, operating system abstraction, release management, etc. The build systems used by most of the projects tested in this thesis are Apache Maven¹ or the GNU Build System², which are described in more detail in the following subsections. Other popular build systems include Ant³, CMake⁴ and MSBuild⁵.

Apache Maven

Apache Maven is a Java-based build tool and is commonly used to build Java-based projects, although it can be used with other programming languages as well. Maven is a highly modularized tool - the main program is a simple container for numerous plugins which do the actual work. As such, it is highly customizable. Since a big selection of tools is readily available, this customization takes little effort [38].

Maven aims to limit the configuration burden by providing a reasonable default configuration for every plugin and allowing the developer to adjust the defaults where needed. This process is called *Convention over Configuration* [38].

One of the outstanding features of Maven is its repository concept and dependency management. Library developers upload their releases to a public repository server. Developers who wish to use these libraries only have to declare the dependency's name and version in the maven configuration files (by convention called "pom.xml"), and Maven will automatically download the dependencies from the remote repository and store them in a local cache. Locally compiled artifacts are stored in the same local cache as well. Different versions of the same artifact can be stored in the same repositories without conflicts.

Criticism [51] of Maven mainly claims that Maven fails to live up to its promises. Despite the convention over configuration guidelines a substantial amount of configuration is still required and the XML syntax of the configuration files makes this process more awkward than necessary. The remote and local software artifact repository can introduce unnecessary side effects and hide bugs (e.g. by accidentally linking against an outdated component rather than reporting an error) and error messages are cryptic.

GNU Build System

The GNU Build System originates from the GNU/Linux operating system and its use is widespread among open source software. It is usually used to compile C and C++ software on Unix-like operating systems, although it can be used for other languages and non-Unix systems as well.

The GNU Build System provides operating system abstraction by checking for required components and choosing a suitable compiler and compiler options. If properly configured, it supports incremental builds where only changed source files are recompiled, which dramatically improves compilation times for developers [19].

¹<http://maven.apache.org/>

²<http://www.gnu.org/software/autoconf/>

³<http://ant.apache.org/>

⁴<http://www.cmake.org/>

⁵<http://msdn.microsoft.com/en-us/library/0k6kkbsd.aspx/>

While the build system can check the availability of dependencies, it cannot install missing ones automatically and usually reacts by aborting with an error or disabling features in the compiled program. It is then up to the user to install the required dependencies in the correct version.

The main criticism [37] [33] of the GNU Build System focuses on the complicated configuration of the build system, the slow execution performance of the build scripts and the difficulty to use it to build programs on Microsoft Windows.

Build Reproducibility

One often overlooked aspect of build systems is the ability to reliably reproduce builds, preferably even on different computers. To achieve this, all aspects of the build process have to be controlled by the build system, including the versions of dependencies, the compiler, system include files and build settings. Reproducibility is important to allow developers to reproduce each other's build or runtime issues and to retest past code versions. [17]

Unfortunately neither Maven nor the GNU Build System can reliably reproduce builds. Neither build system controls the compiler. The GNU Build System has limited control over dependencies. While Maven has more control over dependencies, locally overwritten dependencies and remotely downloaded snapshot versions can influence the build process, as can accidentally used old versions of code artifacts that are available on one system but not the other.

A build system that addresses these issues has been proposed [17], but since it did not find widespread adoption it is of little value to this thesis. As a result, it may be harder to perfectly reproduce the evaluation results in chapter 7. Readers who attempt to do so may get different results depending on their software versions. It is not known to the author why the build system from [17] and [44] was not adopted broadly. A speculative reason is that it is considered too complex because it demands fairly tight control over the host operating system. It is in effect a GNU/Linux distribution⁶) and not just a build system.

2.3 Testing

This section describes the basics of testing that are relevant to continuous integration and this thesis and provides further literature on the subject.

What is Testing?

According to Ian Sommerville, the intention of testing “*is to show that a program does what it is intended to do and to discover program defects before it is put to use*” [50]. To do this, the software is executed on artificial input data and the output is compared to expected results.

Testing can be done on different layers of the software using different test techniques. In a typical software engineering project testing occurs at the following levels:

- Unit testing [7]:

⁶<http://nixos.org/>

It tests the smallest possible components of a software system, e.g. on a function or class level. If a class utilizes other objects these objects are usually replaced with mock objects that have no program logic and return data controlled by the test. Unit tests are simple to write and execute, but they cannot find bugs that result from incorrect interaction between modules.

- Integration testing [32]:

Integration testing assembles all the parts of a software system to form the complete system and tests it as a whole. No mock objects are used, but the testing is still confined to the development environment. Integration tests are more complex than unit tests, but they are able to detect problems that spread across multiple software modules.

- Release testing [50, Ch. 8.3]:

This is the final stage of testing before a software artifact is released for use outside of the development team. External testers can participate in the testing and the system might be installed on the production environment of a customer, but still operate on artificial data.

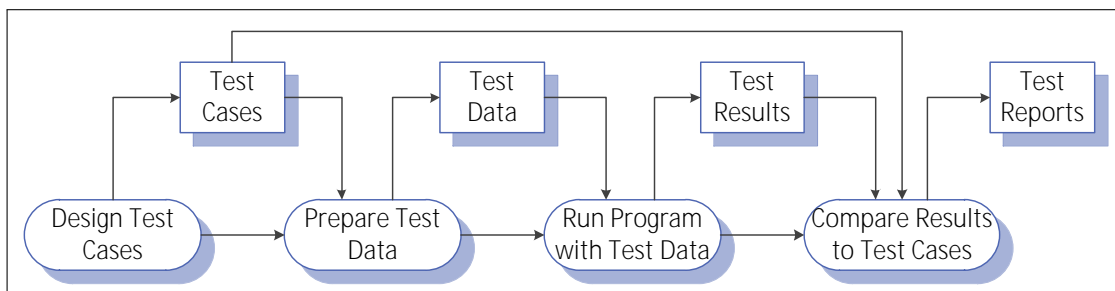


Figure 2.2: A model of the software testing process [50, Fig. 8.3].

Figure 2.2 is an abstract model of the traditional test process. A test case is a specification of the input to the test, the expected output and a statement of what is being tested. Test data are inputs to the system being tested. Sometimes test data can be generated automatically, but test cases cannot be generated automatically because people who understand the system have to specify the expected test results. Test execution and comparing the test results to the expected results can be done manually or automatically.

Test Automation

Continuous integration requires that the CI server is able to execute tests without human interaction. This means that the tests have to be automated in the form of an executable program which interacts with the tested program. The test program can interact with the tested code in a variety of ways:

- Programming interfaces:

The usually preferred way is to call the exported functions or methods of a module in the same way the other modules in the assembled system would do. This requires a clean API abstraction of the various modules, e.g. a separation of business logic and user interface (UI).

- User interface:

If some testing goals cannot be achieved with API tests, e.g. because the business logic is not separated from the UI or the UI itself is tested, then the test program has to interact with the UI. This is more complicated than interacting with an API, but there are several tools that aid in this process, e.g. Selenium⁷ (for websites), Abbot⁸ (Java) and AutoHotkey⁹ (Windows).

An automated test can be as simple as a stand-alone program that returns success or error when exiting to indicate the test result. To structure tests test frameworks like JUnit¹⁰ can be used. Test frameworks usually provide additional tools to organize test results.

Limitations

One important limitation of testing is that it is not a formal proof of correctness. As Edsger W. Dijkstra eloquently stated, “*Program testing can be used to show the presence of bugs, but never to show their absence!*” [11].

The reason for this is simple: Testing can only test a finite set of input parameters, and no matter how extensive the testing is, there is still the theoretical possibility that some untested inputs trigger a bug in the program, the compiler, the operating system, the processor hardware or another dependency component.

Further reading

As far as continuous integration and the rest of this thesis are concerned, all that is needed is a test program for each tested code artifact that returns true if the program behaves as intended and false if it does not. Thus this section skips details on how to write tests, how to manage testing and the role of testing in the software engineering process. The interested reader can find more information on testing in [50, Ch. 8] and [57].

2.4 Source Code Repository

Continuous integration requires a central code repository to allow the CI server to access the source code to make test builds and detect changes to the code.

While in theory this code repository could be as simple as a shared directory on a file server, revision control systems [55] are more suitable for this task. In addition to giving the CI server

⁷<http://seleniumhq.org/>

⁸<http://abbot.sourceforge.net/doc/overview.shtml/>

⁹<http://www.autohotkey.com/>

¹⁰<http://www.junit.org/>

access to the code, revision control systems also provide features that make collaboration between developers much easier, like access to previous versions of the code, finding out which code lines were changed by which developer, branching and merging and many others. These features are outlined in chapter 4.2.

Commonly used revision controls systems are Apache Subversion¹¹, Git¹², Mercurial¹³ and CVS¹⁴. Which revision control system(s) are used and their precise inner workings are not important to CI or this thesis. The only aspect that matters is that the CI server can interact with the revision control system.

2.5 Continuous Integration Servers

This section describes the role of the CI server and provides information about popular CI server implementations and the OpenCIT CI implementation used in this thesis.

The CI server is the component in the CI process that brings the individual parts together. It decides when to execute the build and test processes, it manages build results and built artifacts and reports the results to the developers.

The CI workflow

The CI workflow as described by Fowler [24] is essentially a sequential automation of the manual build process. Whenever a developer commits a change to the code artifact, the CI server checks out the code and builds the artifact. After the build process the artifact's tests are run. Optionally, the produced artifacts can be deployed, for example to provide nightly builds on an FTP server. Regardless of the outcome, the CI server sends a notification with the results to the developer. Figure 2.3 visualizes the process.

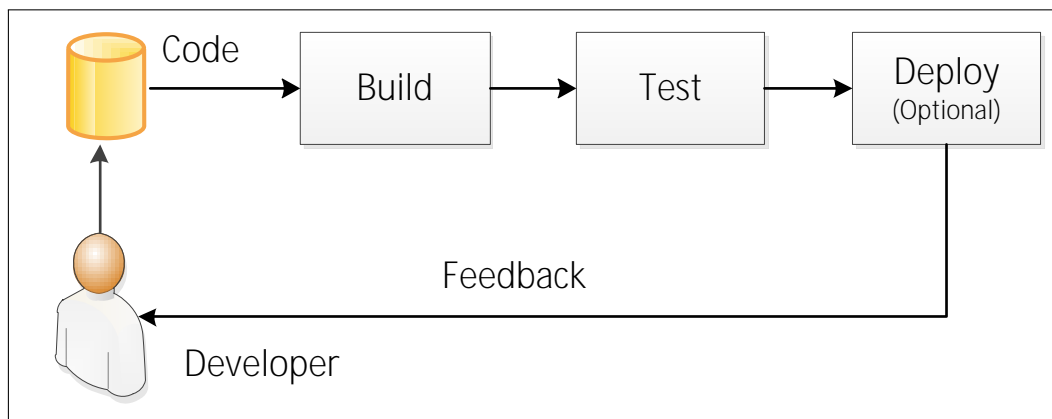


Figure 2.3: An outline of the CI workflow

¹¹<http://subversion.apache.org/>

¹²<http://git-scm.com/>

¹³<http://mercurial.selenic.com/>

¹⁴<http://www.nongnu.org/cvs/>

In the event of a build failure, the test and deploy steps are skipped and an error notification is sent. The development team may have different notification rules depending on the outcome. It may also make sense to reject or revert the change if it introduces build or test failures. State of the art CI servers offer a wide range of customizations.

Popular CI servers

Among the most widely used CI servers are the CI servers Hudson¹⁵ and its fork Jenkins¹⁶. The main features leading to their popularity are an easy to use browser-based user interface and a vast amount of available plugins that allow using the build servers with virtually every available build system and revision control system and add custom features to the build servers.

Hudson and Jenkins have basic support for dependency tracking¹⁷. However, this support is limited to projects building on the same server, does not include a way to communicate with other servers and works only with projects using Maven as their build system. Some remoting support also exists, but it is aimed at load balancing and covering multiple build environments only, similarly to BuildBot's master/slave system¹⁸.

Another CI server is BuildBot¹⁹. Its strength is the master-slave system that separates the user interface and build control processes (the master) from the actual build and test processes (the slaves). This not only provides load balancing support, but also allows BuildBot to test the tested code artifacts on multiple hardware platforms and operating systems but still manage the results in a central location.

The shortcoming of BuildBot's remoting support in distributed development environments is that the interface is not suitable to communicate with unknown remote parties. The setup process to add a slave is complicated and the amount of control the master has over the slave leads to security concerns if the master is not trusted [56, Sec. 2.6].

OpenCIT

The OpenCIT CI server²⁰ is a very simple CI server that mainly serves as a demonstration of the Open Engineering Service Bus (OpenEngSB) middleware²¹.

While OpenCIT is a quite limited build server for everyday uses compared to Hudson, Jenkins or BuildBot, it has certain advantages for the purpose of this thesis. Its small codebase (about 5000 lines of Java code and XML configuration) makes it easy to modify.

Furthermore, the underlying OpenEngSB framework [39] provides all the infrastructure required for setting up message-based communication, workflow management and tool integration.

¹⁵<http://www.hudson-ci.org/>

¹⁶<http://www.jenkins-ci.org/>

¹⁷<https://wiki.jenkins-ci.org/display/JENKINS/Dependency+Analyzer+Plugin>

¹⁸<https://wiki.jenkins-ci.org/display/JENKINS/Distributed+builds>

¹⁹<http://www.buildbot.org/>

²⁰<http://opencit.openengsb.org/>

²¹<http://www.openengsb.org/>

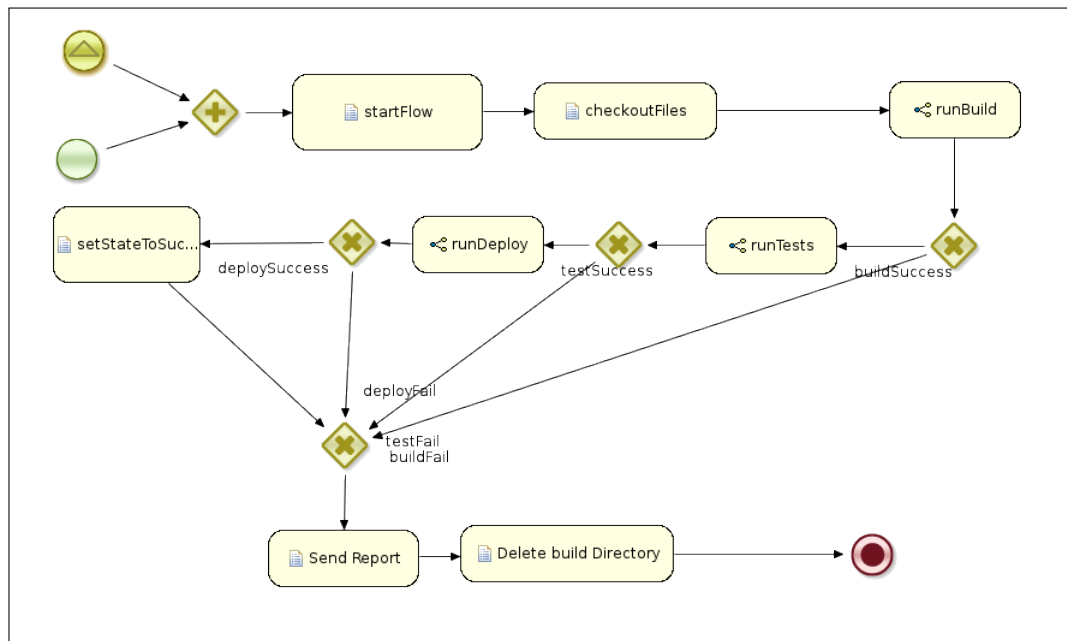


Figure 2.4: Drools rendering of the OpenCIT workflow.

OpenCIT supports the Git revision control system and Maven build systems. The core CI workflow is implemented as a JBoss Drools²² workflow (Figure 2.4). OpenEngSB provides the necessary infrastructure to add plugins for other build systems and revision control systems.

2.6 Best Practices

In addition to the infrastructure and project changes, continuous integration requires the developers to follow a set of practices to achieve its full potential. Duval lists the following rules that developers should follow [16, Ch. 2]:

- Commit code frequently!

Committing code frequently not only gives coworkers access to one's work early, but it also keeps changes small and easy to understand.

- Do not commit broken code!

If developers commit broken code, they break the build process for everyone else and causing disruption in coworkers' work. Therefore developers should build and test their changes on their local workstations before committing. Recent CI developments also allow CI servers to test incoming commits before they are persisted in the code repository and reject them if they cause failures²³.

²²<http://www.jboss.org/drools/>

²³<http://wiki.hudson-ci.org/display/HUDSON/Gerrit+Plugin>

- Fix broken builds immediately!

If the CI server reports that the project is in a broken state, developers should address the issue immediately. Otherwise they run the risk of always ignoring test failures, which makes continuous integration useless. A possible policy to enforce this is to reject any changes that do not fix a build or test failure, once such a failure is detected [29].

- Avoid getting broken code!

Developers should avoid checking out code from the code repository if the CI server reports failures, unless they plan to find and fix the breakage. Otherwise it is better to wait for the developer who caused the breakage to fix it.

2.7 Experiences

This section outlines scientific research that investigates if the practical experiences with CI live up to the expectations and promises.

The available literature on the actual results of automated testing is thin, but Dudekula Mohammad Rafi et. al. have written a comprehensive overview with a systematic literature review and a practitioner survey [42]. Although their work focuses on test automation in general, the results apply to continuous integration as well. Their key findings are:

- Automated testing reduces the effort required for testing. However, this effort reduction should not be used to reduce the testing budget of a project, but to improve testing with the available resources.
- Automated testing is most useful when multiple regression testing rounds are needed.
- Test automation increases test coverage, which means that it has benefits even when repeated testing is not required.
- A key limitation is the high initial cost in designing test cases, obtaining a test automation tool and training the staff.
- Maintenance of automated test cases is perceived as problematic by practitioners.
- Automated testing cannot fully replace manual testing, only augment it. The skills and intuition of testers are still required to detect new bugs.

The authors also bemoan that the available literature focuses on the strength of test automation and limitations are only reported by practitioners. They attribute this to a publication bias towards papers describing benefits.

Software Dependencies

This chapter provides an outline of software dependencies: What dependencies are and why they are necessary, ways of importing dependencies, distributed software engineering, common dependency management systems and dependency anomalies.

3.1 What are Software Dependencies

Software dependencies are a result of software modularization and code reuse [10]. Using external libraries for performing common tasks instead of reimplementing everything from scratch is an old [35] technique and considered good software engineering practice [50, Ch. 16]. Furthermore software is split up into independent modules to make the code easier to manage and make team coordination easier. As a consequence, the different modules and external libraries have to be provided to the program in the correct version.

Formally, a code artifact CArt0 depends on another code artifact CArt1 when it relies on it for some of its functionality [41]. If CArt1 is not available or not functional, CArt0 does not work, or works with reduced functionality only. A code artifact can depend on multiple other code artifacts (its *dependencies*) and a code artifact can be used by multiple other code artifacts (its *dependents*). Dependencies themselves can have dependencies (*transitive dependencies*). Taken together, the individual code artifacts and their dependency relations form a *dependency graph*, which Podgurski and Clarke [41] define as an acyclic directed graph. While the big majority of dependency relations is indeed cycle-free, this is not true for all of them (section 3.4).

Dependencies are of particular interest in software engineering because the task of managing external dependencies and dealing with library upgrades can be a major cost factor in software evolution [47].

3.2 Linking Types

This section describes common ways of interacting with dependencies, their advantages and disadvantages, common uses and implications for software development.

Code Copy

The simplest way to include a dependency is to simply copy the dependency's code into the project that needs the dependency and instruct the build system to build it. To update the dependency the source files have to be updated manually and the project has to be recompiled.

The advantages of this method are full control over the dependency version, the ability to make modifications to the dependency and the guarantee that the dependency is available at runtime.

The disadvantages are the difficult update process, the need to build the dependency, which increases compile time and requires understanding of how to build it. The source code has to be available and there may be licensing issues.

Situations where this technique are used are: If custom changes to the dependency are required, for code modules developed in the same development organization, for static data or when only a minor part of an otherwise bigger library is used.

Static Linking

With static linking [28] the linker includes a compiled copy of the dependency in the generated executable. Compiling the dependency is not necessary. Updating the dependency generally requires a recompilation of the source code, although relinking is technically enough if precompiled object files of all parts of the code artifact are available.

The core advantage is that the dependency is guaranteed to be available in the correct version at runtime without the requirement to build the dependency itself and making the dependency easier to update compared to a source code copy. This method can be used even if the source code of the dependency is not available. Furthermore multiple programs that require conflicting versions of the same dependency can coexist without issues.

The disadvantages are the increased size of the executable, which matters if multiple executables using the same dependency are used, as every one of them contains a copy of the dependency. Replacing the dependency is not practical without recompilation. Some libraries like the GNU Lesser General Public License v2.1 place certain restrictions on static linking [25].

This way is used for internally developed dependencies or dependencies that are hard to obtain by users. It is also used for system binaries that run early in the boot process of operating systems, where external libraries may not be available and sometimes on embedded operating systems without a dynamic loader.

Dynamic Linking

Dynamic linking [52, Appendix 7A] means that a dependency library is only referenced by its name in the dependent program. The dependency artifacts are stored in a different file and loaded

by the host system as needed.

This allows different programs to share common components, thus reducing the disk space and runtime memory requirements. The dependencies can be upgraded easily by replacing the library files, without the requirement to recompile or relink programs using them.

The main problems with dynamic linking are missing libraries and version conflicts. The required libraries have to be provided somehow. Usually the libraries are a part of the host operating system or runtime environment. If a required library is missing for some reason or available in the wrong version, dependent programs fail to run or function properly. Especially older versions of Microsoft Windows used to have major library management issues known to users and system administrators as *DLL Hell* [3].

Dynamic linking is a widely used mechanism. It is used for linking against components of the host operating system or runtime environment and often used for linking internal components of a software system.

Remote Invocation

A more recent way to interact with dependencies is communicating with them over a network, e.g. the Internet. Multiple communication patterns are used [53, Ch 4], the most commonly used being remote procedure calls (RPC), message-oriented communication (MoM) or shared resources. A contractual interface agreement defines the communication format. Ideally this interface is independent of the service implementation, allowing clients to switch between different service providers.

Remote invocation is used by services that are distributed in nature, e.g. an online music store or an online multiplayer game.

3.3 Dependency Management

This section describes various way of managing dependency artifact on different operating systems and software ecosystems.

GNU/Linux

On GNU/Linux systems managing dependencies is the job of package managers, which manage programs and libraries in software packages. To install a program, the user requests the installation of the package containing the program. The package manager knows the dependency requirements of all software packages known to it and installs the dependency together with the packages that require it. When the user requests the uninstallation of a package, all dependencies that were not explicitly installed by the user and are not required by other packages can be removed again. System updates are also handled by the package managers.

There are numerous package managers, used by different GNU/Linux distributions: yum¹ (Fedora), dpkg² (Debian, Ubuntu and derivatives), Portage³ (Gentoo) and others.

¹<http://fedoraproject.org/wiki/Yum>

²<http://www.debian.org/doc/manuals/debian-faq/ch-pkgtools.en.html>

³<http://www.gentoo.org/doc/en/handbook/handbook-x86.xml?part=2&chap=1>

When software is installed or built without using the package manager, the user is responsible for providing the required dependencies, for example by instructing the package manager to install the dependency. The common build systems on Linux like the GNU build system (see section 2.2) can only detect the absence of required dependencies and abort the build process with a descriptive error.

The available software for GNU/Linux systems forms a big software ecosystem. This is reflected by a huge amount of packages, forming a complex dependency tree. Figure 3.1 shows an artistic rendering of the dependencies between packages on the Gentoo Linux system that was used in the experimental evaluation in chapter 7. The picture was generated by the tool Pacgraph⁴ written by Kyle Keen. The underlying data is the Portage tree from July 31st 2012. This picture illustrates the number of packages a GNU/Linux distribution is built from and the complexity of dependencies between them.

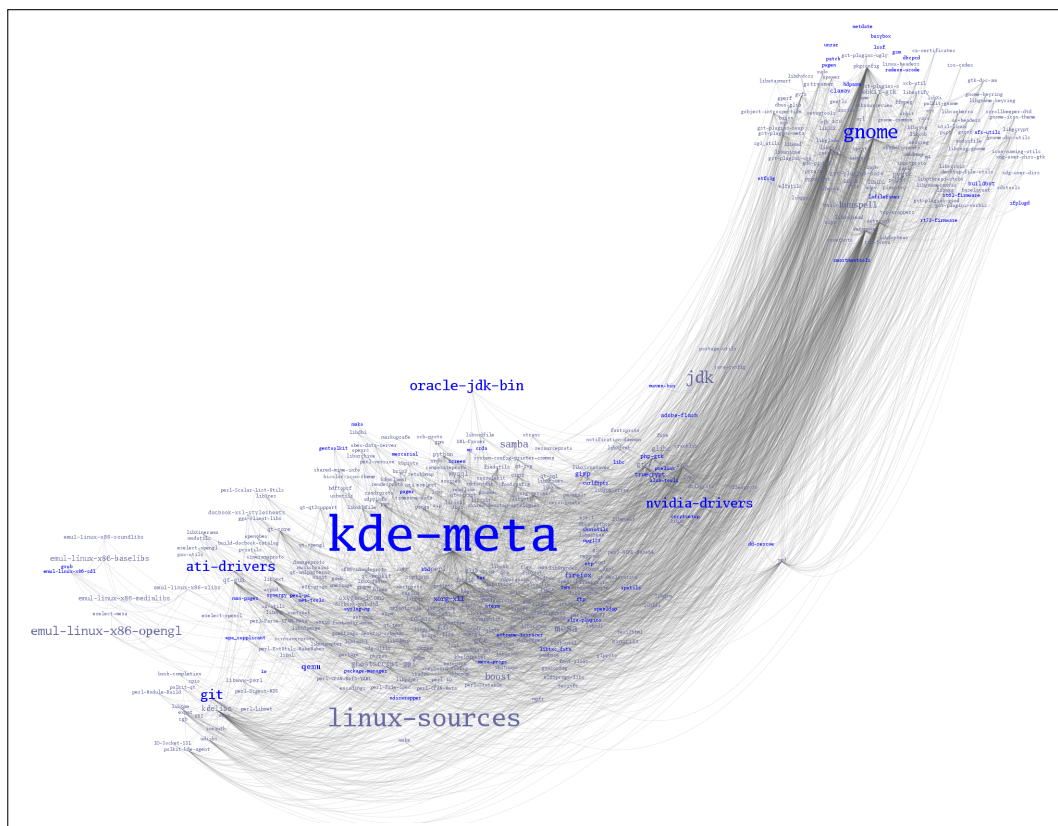


Figure 3.1: A visualization of the package dependencies on the test system.

The Gentoo installation is a desktop installation with both the KDE and Gnome desktops installed. It also contains various C and Java development tools. The graph contains a total of 435 nodes. The installed packages occupy 4717.2 megabytes of disk space.

⁴<https://github.com/keenerd/pacgraph>

Microsoft Windows

Microsoft Windows is a fairly monolithic system. Windows ships a lot of libraries that provide common services needed by Windows applications, starting from basic things like file handling and user interfaces to things like multimedia functionality and network communication. As such, few external dependencies are needed.

There are however some libraries that are not always available. These libraries can be redistributed by software vendors, so the installation software of the dependent program can make sure they are available on the target system. The most commonly redistributed packages are the Microsoft Visual C++ runtime ⁵, the .NET framework⁶, the DirectX runtime⁷, the Visual Basic⁸ runtime and others.

Windows does not have a sophisticated system to manage conflicting versions of the same library. Ideally libraries are backwards compatible and installers do not replace newer versions with older libraries they may have bundled. Unfortunately this is not always the case, leading to the aforementioned DLL Hell. Microsoft has tried multiple schemes to combat DLL Hell, ranging from file protections to disallow overwriting of system libraries [9] and so-called *Side-by-Side Assemblies*, which essentially allow library versioning [8].

Java Ecosystem

Java itself does not provide any dependency management support. However, dependency management is provided by build tools like Apache Maven.

Maven provides public online repositories where software developers can upload their code artifacts. Code artifacts are uniquely identified by their publishing organization, name and version. If a dependency is not available on the build system, it is downloaded and stored in a local cache. Locally built artifacts are also stored in that cache.

When a program is packaged for distribution outside the Maven repository system, Maven includes a copy of all dependencies in the packaged archive.

3.4 Dependency Anomaly: The Bootstrapping Problem

This section discusses an anomaly in the dependency graph that has to be considered.

The dependency graph is usually described as a directional, acyclic graph [41]. This is true for almost all software packages, but it does not always hold up in low level components of an operating system, especially concerning compilers.

For example, the GNU Compiler Collection (gcc, ⁹), a popular C compiler, is written in C itself. As a consequence, a working C compiler binary is required to compile the compiler. Furthermore, this host compiler requires a C runtime binary (e.g. GNU libc, ¹⁰). Thus the compiler

⁵<http://www.microsoft.com/en-us/download/details.aspx?id=5555>

⁶<http://www.microsoft.com/en-us/download/details.aspx?id=17851>

⁷<http://www.microsoft.com/en-us/download/details.aspx?id=8109>

⁸<http://www.microsoft.com/en-us/download/details.aspx?id=20429>

⁹<http://gcc.gnu.org/>

¹⁰<http://www.gnu.org/software/libc/>

depends on the C runtime and itself. Because the C runtime is written in C and compiled by a C compiler, it also depends on the compiler, forming a dependency cycle. A compiler that is able to compile itself is called *self-hosted*.

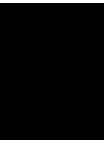
The problem of compiling compilers that are implemented in the language they compile is known as the *bootstrapping problem* [4]. The first challenge it presents is creating the first executable binary of a compiler for a new language. There are various techniques to achieve this, but they are not of concern for this thesis. The more relevant problems are how complex systems like GNU/Linux handle compiler updates and the implications on dependency tracking.

Linux distributions that are compiled from source like Gentoo address this dependency cycle by compiling the compiler and C runtime twice [26]. Likewise, when the GNU Compiler Collection is compiled with its standard build scripts, it compiles itself three times - once with the host compiler, then with the just-generated compiler and then with the 2nd compiler build. Obviously a working compiler binary is needed to perform bootstrapping. This compiler is shipped with a small precompiled host system. During the bootstrapping process, this host compiler is replaced with the newly compiled compiler. The cyclic dependency between glibc and gcc is represented in Gentoo's packages: ¹¹ and ¹².

The main implication on dependency tracking is that dependency relationships are not guaranteed to be cycle-free.

¹¹<http://sources.gentoo.org/cgi-bin/viewvc.cgi/gentoo-x86/sys-libs/glibc/glibc-2.15-r2.ebuild>

¹²<http://sources.gentoo.org/cgi-bin/viewvc.cgi/gentoo-x86/sys-devel/gcc/gcc-4.6.3.ebuild>



Impact Management

This chapter explores problems that occur when coordinating development work inside a team or between separate teams as well as common solutions to these problems. This background information provides understanding how the contributions of this thesis can be used to support developers in their work.

4.1 Terms

This section gives a background of research literature on software development coordination, with a focus on papers that define the problem and terms for it.

One of these papers, written by Cleidson R. de Souza et al [14], called the activities to handle wanted and unwanted influence between developers *impact management*. Impact management consists of three aspects: First, knowing your *impact network*, the people that are affecting one's work or are affected by one's work. Second, *forward impact management*, making sure others are aware of one's changes and third, *backward impact management*, making sure one is aware of others' work. Their work concludes with an empirical study of two different software development teams describing their impact management techniques.

Impact management becomes more difficult when the team is spread across the globe, like in many open source projects. Halloran and Scherlis analyzed a number of open source projects and documented their development approaches [29]. Their main goal was to find out how open source projects coordinate, enforce quality standards, and what is necessary to have new development practices or tools accepted by open source projects.

The contribution of this thesis can be used for all three aspects of impact management: Notifications from dependencies to a project aid the project's backward impact management. Notifications from the project to its dependents help automate forward impact management. And finally, the ability to subscribe to these updates helps clarify the impact network.

4.2 Tool Support

This section outlines common tools that are used for development coordination.

Generic Communication Tools

The most important set of tools are those that allow basic communication: Meeting coworkers in person, telephones, email, chat software, video telephony, etc. They provide unstructured communication between humans.

Other software tools can send information and updates to developers by generating emails using predefined templates. An example for this is a continuous integration server that sends an email to the developers when a build fails. The reverse direction is also possible, but usually more limited - e.g. a chat bot [27] that accepts commands via chat messages.

Revision Control

Revision control systems, already mentioned in section 2.4, not only provide a central storage space to store one's work and synchronize it with coworkers, they also provide essential functionality to analyze other people's work and parallelize development of different features.

A revision control software maintains a complete history of the development of the project. For each modification to the project's files it stores information about who changed the file, when the change was made, which files were modified, as well as a message from the developer describing the purpose of the change. Old versions can be retrieved from the repository at any time to analyze potential regressions and changes can be reverted if necessary [34, Ch. 3].

Another central concept of modern revision control systems is that of branching and merging. A *branch* is a split of the development line [34, Ch. 7]. It can be used to develop a new feature without affecting other developers, or to maintain an older version of the software, e.g. for post-release maintenance.

To combine the development of two separate branches, a *merge* is used [34, Ch. 9]. A merge unites two branches and the changes from both branches appear in the combined branch. If conflicting changes were made during the separate development, *conflict resolution* is necessary. To a certain extent conflict resolution can be automated, but in general manual intervention is required to reconcile two conflicting changes.

Figure 4.1 shows an example of a development tree with extensive use of branches and merges. The tree shows the development history of the Git tool, which is self-hosted and uses Git as its revision control system. The graphical rendering is a screenshot of its built-in GUI *gitk*¹. Git is a distributed revision control tool, which means that users can make full copies of the entire project history. This encourages heavy use of branches since users can make modifications in their private trees and then send a *pull request* to the maintainers of the upstream tree, which contains all changes, allows the maintainers to review the changes and merge them into the upstream tree if they agree with the modifications.

¹<http://www.kernel.org/pub/software/scm/git/docs/gitk.html>

4.3 Collaboration Strategies

This section describes problems that occur in big and possibly globally distributed software development teams and tactics used to overcome them.

Software companies can grow to sizes of hundreds or thousands of employees who may even be distributed across the entire globe, but still attempt to work together on the same product. Careful use of the previously described communication tools and a good team and code structure are required to maintain a high level of productivity.

Challenges introduced by big team sizes include finding a proper separation of work, keeping separately developed components compatible and spreading project knowledge inside the team [46]. If, in addition, the team is geographically distributed, further issues like timezone offsets, cultural differences, as well as slow and/or expensive communication are added [27]. Most of the challenges in distributed development environments can be traced back to the lack of in-person informal communication [46].

To address these challenges, organizational measures as well as tools to support distributed teams have been proposed, implemented and analyzed.

Organizational measures aim at reducing the need for communication and improving informal communication, mutual understanding and strengthening the team. The need for communication can be reduced by splitting the development work into well-defined, isolated pieces of work [6]. Possible examples are fixing well-understood bugs, writing documentation or porting the application to a new platform.

Established strategies to improve informal communication include [6] regular conference meetings and phone calls, offering employees to visit other branches of the company and weekend team-building activities.

The tools described in the previous sections provide a formal communication framework to uphold a certain quality of information exchange to keep the development on track.

4.4 Managing Open Source Software

This section describes how open source projects are managed on a small, per-project scale. The specific focus is on team coordination and efforts to keep code quality high.

The central way of control over an open source project is limiting write access to the project's code [29]. Usually this means that only trusted developers have permissions to push their changes to the projects revision control server. While open source licenses allow everyone to obtain, study and modify the source code, as well as publish those changes as they see fit, the official code remains under control of the project maintainers.

The same paper also found that open source projects rely heavily on computer-mediated communication and tools [29]. Almost all communication takes place via mailing list, public chatrooms and bug trackers rather than person-to-person communication. Contrary to proprietary software, too much person-to-person communication can become a problem for open source projects. The danger it poses is the creation of an exclusive circle of core developers that make it difficult or impossible for new developers to join the project. These findings are

mirrored by development guidelines, e.g. the Apache Foundation's guide for new project proposals [22, Sec. Known Risks].

The right to redistribute the open source software freely poses a danger to highly popular projects that target the consumer market [5]. On occasion, modified versions of such projects are made available on the Internet, with the malicious intent that a user downloads the modified version instead of the original. Restricting commit access is useless against this because the perpetrators never attempt to have their changes included. Those modifications may include adware or charging a fee for the download. Projects like Mozilla [20], Debian [48] and LibreOffice [49] protect themselves against such tactics by registering the project's name as a trademark and disallowing the use of the name for modified versions.

4.5 Building GNU/Linux Distributions

GNU/Linux differs from commercially developed operating systems in the way that there is no body exercising central control over the development of the software system. Furthermore, GNU/Linux distributions are too large to manage them simply with the techniques from section 4.4. This section explores how Linux distributions are built, work on system component is shared, and how compatibility between distributions is preserved.

Eric S. Raymond has written a famous essay [43], in which he compared the development of the Linux kernel and some other free software projects to a bazaar, and the development of commercial software (and some free software, most notably then-current versions of gcc) to building a cathedral. He elaborates that cathedral-style development is marked by central control and micromanagement over the direction of the project, whereas on the bazaar no central control exists and multiple solutions for one problem compete with each other. Yet, through mechanisms similar to that of a free market, a certain order establishes itself and high-quality software is written. Creators of Linux distributions select what they think are the best components to solve given problems, and users judge their selection by picking the distribution that works best for them. Eventually this market mechanism eliminates inferior software.

The bazaar model has not been without criticism [33] [12]. The criticism focuses on the inhomogeneity of the GNU/Linux ecosystem and burdening the user with unnecessary decisions and confusing differences between Linux systems. To fight incompatibility in a faster way than just letting the market decide, various standardization organizations were formed, most notably the Linux Standard Base² and freedesktop.org³. Furthermore, platform independent industry standards are used, like the C [30] and C++ [31] language standards or the OpenGL 3D API [45].

²<http://refspecs.linuxbase.org/lsb.shtml>

³<http://www.freedesktop.org/>

Research Issues

In distributed development environments with multiple independently developed projects, the different projects are not sufficiently included into the testing process because the project management and testing techniques described in chapters 2-4 do not take into account the dependency relations between projects.

This thesis describes a testing process that extends classic continuous integration. This process improves testing effectiveness by extending testing to project dependencies and dependents. This chapter describes the research issues that arise and the approaches used to answer them.

Figure 5.1 illustrates the problem space. It shows a setup with 5 engineering projects - for simplicity reasons represented by code artifacts (CArt) - whereas CArt0 depends of CArt1 and CArt2, and CArtM and CArtN depend on CArt0, and, implicitly, on CArt1 and CArt2. However, despite the dependency relationship between these projects, automated testing and most of the development happens in isolation, i.e. without considering change effects on dependents and ongoing changes to dependencies. The circled numbers describe problems that occur with classic testing and management techniques in this setup:

1. Developers of dependencies do not know the impact of their changes on their dependents, and might not even know that those dependents even exist. If they want to know the impact, they have to test manually. This means high manual effort for the developers of the dependencies, and usually the readiness to deal with dependent projects is low.
2. Communication between interdependent projects is done manually, or semi-automatic at best. Aside of the manual effort, this further delays feedback for developers of dependencies. Usually this communication consists of filing and commenting on bug reports, writing on mailing lists or directly to developers, and sometimes automatically generated human-readable mails when new releases are published.
3. Software is regularly updated, and developers of dependent projects want to consider new dependency releases in the expectation that they add features or improve stability. However, they have to test new versions of their dependencies manually. As a result, they do

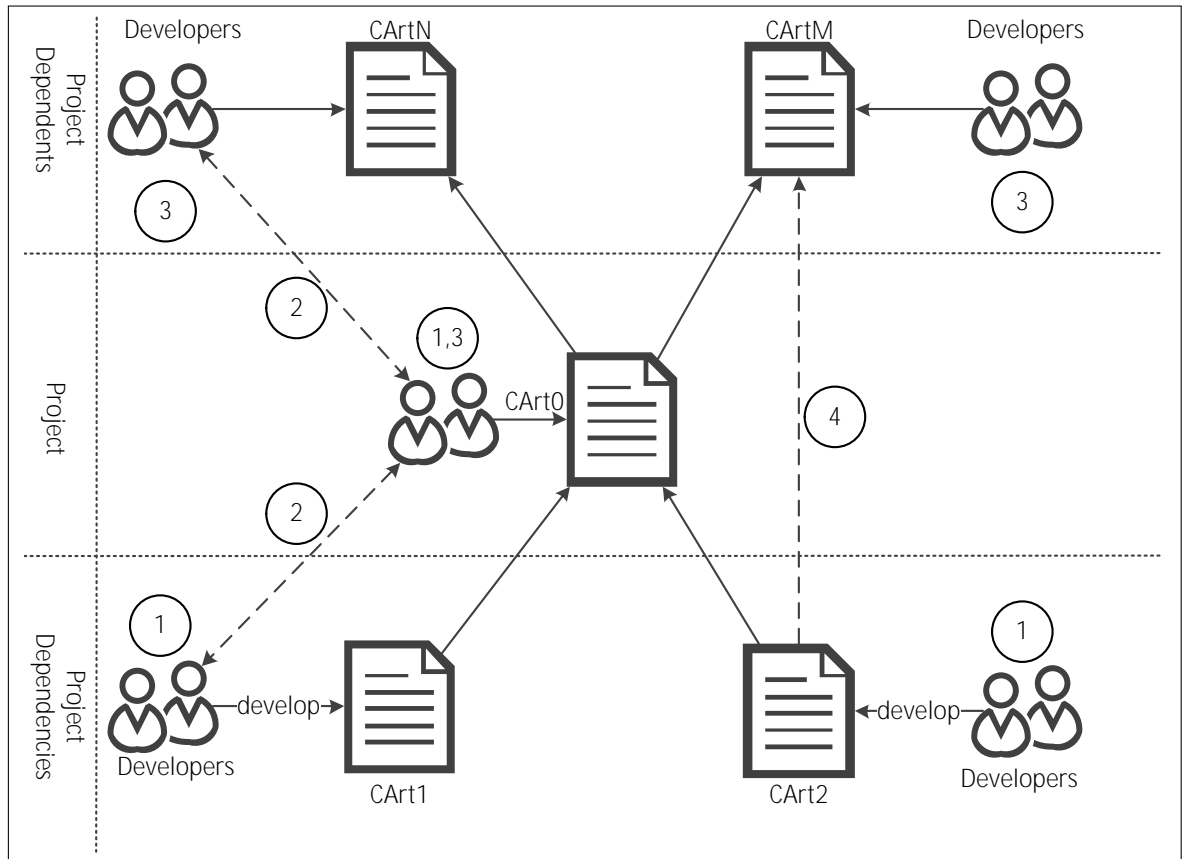


Figure 5.1: Limitations of classic testing and management techniques in an environment with multiple independently developed projects

not find regressions in the dependencies that affect their code until they decide to update, usually after a new version of the dependency has been released.

4. Dependencies may depend on other dependencies. Some software faults are caused by defects in transitive dependencies (for example [18], [40]). Since an intermediate project is involved, debugging those kinds of dependencies and working with their developers to get them fixed are difficult and time consuming. Due to the aforementioned manual update and communication process those defects are discovered late.

The CCIP described in this thesis aims to break up the isolation by actively taking into account the complexity of dependencies between code artifacts and automating update propagation, testing and feedback reporting across project boundaries. The main stakeholders are the developers, who benefit from higher testing effectiveness, as the concept provides automated tool support and reduces manual work (e.g. communication with project dependencies and dependents).

5.1 Research Issues

The key research issue is to evaluate whether automated update notifications, testing and feedback processing provide an advantage over the manual process (RI-1). Another issue is filtering feedback to eliminate feedback with poor quality (RI-2). Finally, further issues need to be addressed that arise in large scale deployments (RI-3).

Costs and Benefits of the Automation

The first research issue is about evaluating the advantages and disadvantages of the new process in comparison to the currently established ones:

RI-1.1 How much effort is required to implement the CCIP in existing continuous integration implementations?

The CCIP is intended to be integrated in CI servers. While the communication protocol is implementation independent, the CCIP workflow has to be integrated in each CI implementation separately.

RI-1.2 How much initial effort is required to deploy the CCIP initially in existing projects and in case of dependency reconfiguration?

If a project's CI server supports the CCIP, the project setup may have to be changed to make use of it. When the project makes use of a different dependency, this setup may need adjustments. What is the setup effort, if any, is required depending on metrics like project size, number of dependencies, etc.

RI-1.3 How are costs and benefits distributed between dependencies and dependents?

Both projects in a dependency relation have to adopt the CCIP in their project setups to make use of it. The setup costs, testing resource cost and benefits may be distributed unevenly between the dependency and the dependent.

RI-1.4 How can the benefits of the CCIP be measured to justify the increased costs?

Objective metrics are needed to measure the advantages, or lack thereof, the CCIP provides to a projects development process.

Feedback Quality

Feedback from dependent projects is a cornerstone of the Continuous Change Impact Analysis Process. The second research issue deals with requirements the dependent projects and their feedbacks have to meet to provide useful, and how low quality feedback can be dealt with.

RI-2.1 How can developers deal with test results from other projects?

Developers of dependencies most likely do not know the code of their dependent projects well enough to fully understand test failures reported to them. The test failure may be a bug in the dependent project, and analysis of the failure report may turn out to be a waste of time. How can developers use incoming feedback efficiently?

RI-2.2 Are there guidelines that dependent projects can follow to ensure maximum usefulness of their feedback for dependencies?

Scalability Concerns

The examples in this section consist of a small number of projects. When using the CCIP on a larger scale additional issues have to be dealt with:

RI-3.1 How can large scale deployments be handled?

Projects at the top of the dependency tree will receive many update notifications, which might trigger more test runs than the continuous integration server can handle. Which strategies can be used to handle or reduce heavy load?

RI-3.2 How relevant are cascading notifications?

Cascading updates (changes to a dependency of a dependency) likely cause a lot of CI server load, but are they likely to find regressions?

5.2 Use Cases

Figure 5.2 outlines the use cases for developers using CCIP:

1. Developers of dependent projects receive automatic notifications of dependency updates together with testing results pointing out potential problems.
These passive notifications improve dependency management not only by pointing out available dependency updates, but also by providing automated integration results.
2. Developers are automatically informed about the impact of their changes in dependent projects.
Whenever a developer commits changes to his project's source code repository, the CCIP provides automatic test results of his changes in dependent projects. This increases the developer's confidence in his changes or points out potential problems early.
3. The system provides a way to expand test coverage in dependencies with little or no effort.
Developers can use the tests of their dependent projects to expand test coverage. This is especially helpful in complex projects where complete test coverage is very hard to achieve with conventional means.

5.3 Evaluation

The CCIP concept and research issues will be evaluated by writing a prototype implementation¹ and using this implementation to monitor the development of selected open source projects. Literature research and investigating the currently established manual processes will provide the basic design of the prototype. The results of the evaluation can be found in chapter 7.

¹<https://github.com/stefand/opencit/tree/da-sdo>

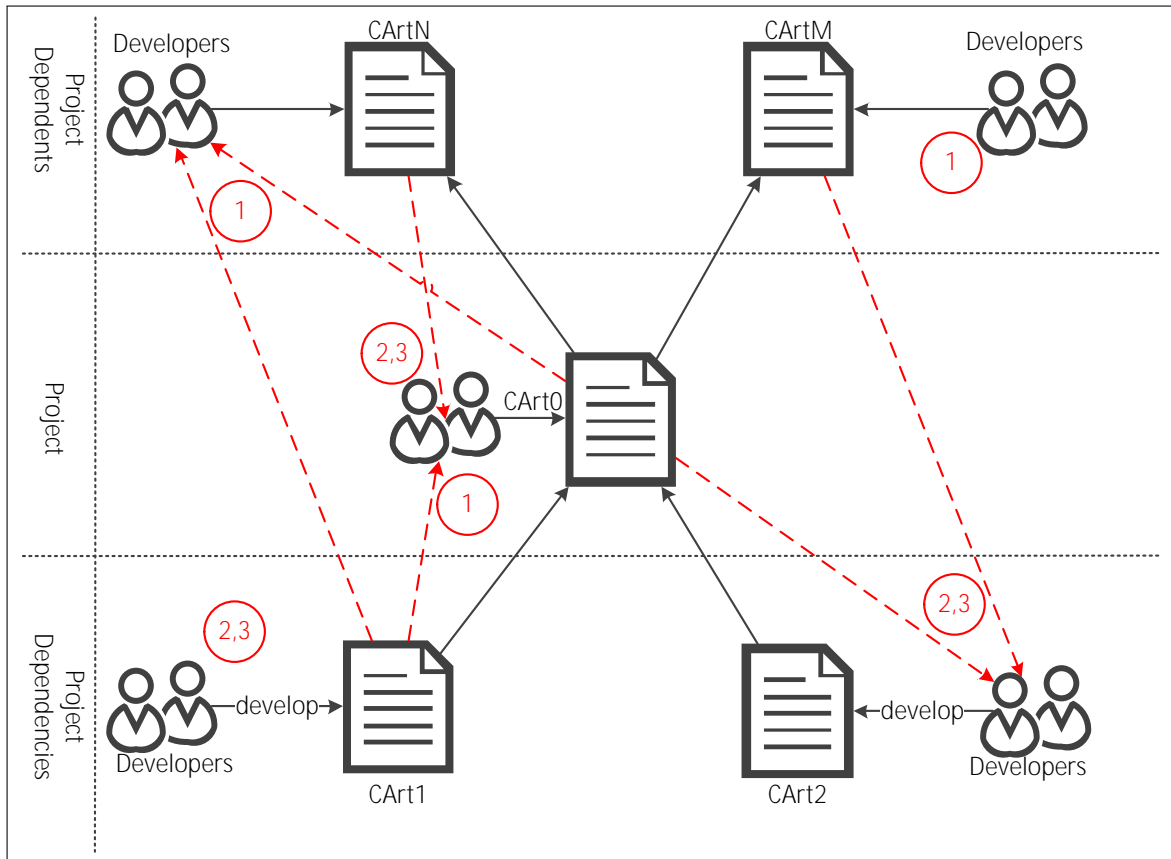


Figure 5.2: Use cases of CCIP for developers

Figure 5.3 is a high-level overview of the Continuous Change Integration Process and the additions it introduces:

1. CARt1 notifies its dependent project CARt0 about a new build.
2. CARt0 imports the new CARt1 build into its build system and runs its tests over the merged system.
3. CARt0 reports the test results to its dependency CARt1.
4. If CARt0's tests executed successfully, it announces the newly available build to its dependents CARtN and CARtM.
5. CARtN and CARtM import the new CARt0 build and run their tests.
6. CARtN and CARtM report the test results to CARt0.
7. CARt0 forwards the feedback to the originator of the change, in this case CARt1. This way a cascading update and feedback mechanism is realized.

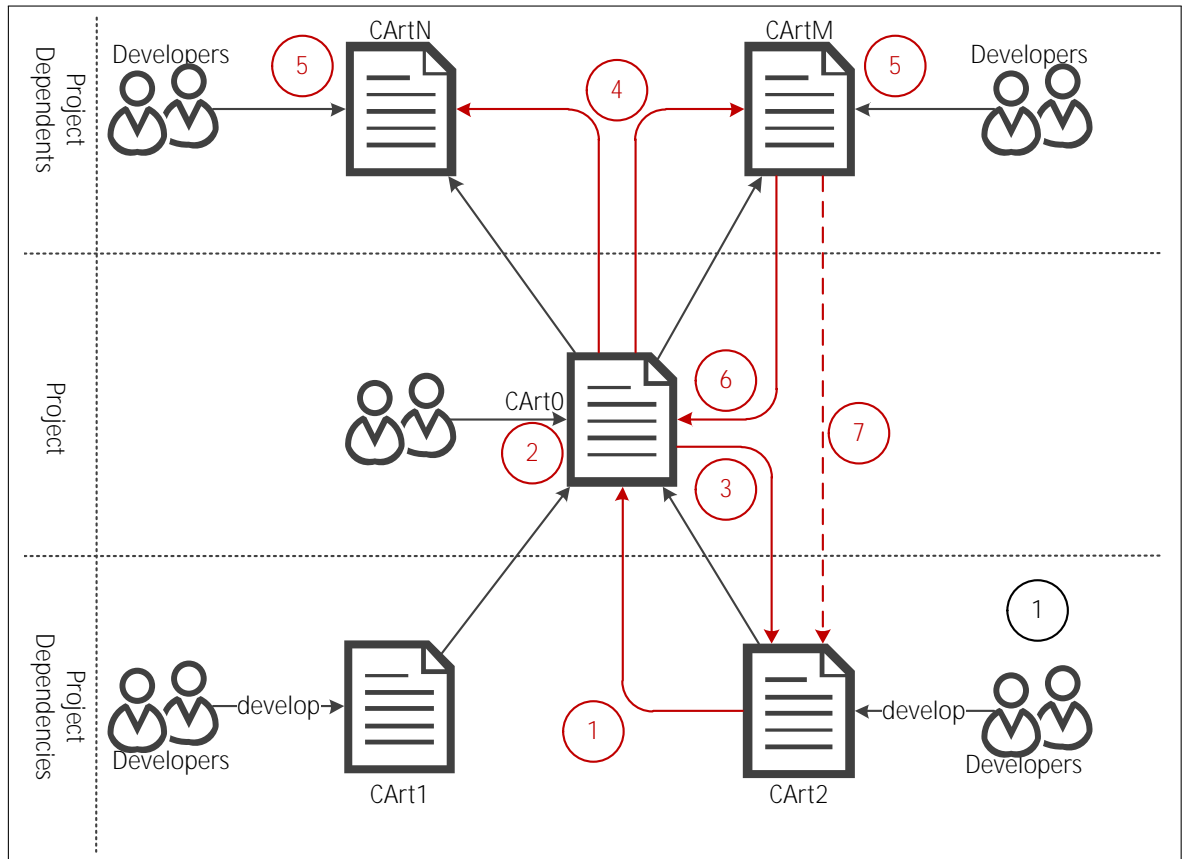


Figure 5.3: Solution overview

Prototype Implementation

To evaluate the concept, a prototype implementation of the CCIP will be written based on the OpenCIT² CI server. OpenCIT was chosen because of its small size, flexibility and adaptability. It is free of features that are not relevant to the CCIP and would complicate a prototype implementation. This prototype will be used to monitor selected open source projects and their dependencies over a prolonged period of time.

Testing the implementation with open source projects

Open Source software provides a good testbed for the CCIP concept and prototype implementation because the revision control systems are publicly available and changes are generally published as soon as they are committed by the developers. It is easily possible to set up a continuous integration infrastructure for a private evaluation without interfering with the development of the project.

²<http://opencit.openengsb.org>

Thus the prototype implementation of the CCIP will be set up to monitor a set of projects from different ecosystems. One of these projects is OpenCIT itself together with the OpenEngSB framework³ it depends on and major Java-based technologies like Apache Karaf⁴, JBoss Drools⁵, Apache Wicket⁶ and Google Guava⁷. Aside from the obvious goal of self-hosting the OpenCIT build server these projects actively developed and widely accepted in the software engineering industry.

The other set of projects is Wine⁸ together with the GNU/Linux graphics stack (Mesa⁹, the X.org X server¹⁰ and GPU drivers). These projects were chosen because Mesa implements OpenGL¹¹, a complex yet stable realtime 3D graphics API, and runs on a wide range of platforms, which makes testing with traditional approaches challenging. Wine has a big number of 3D rendering tests which have been used to find and isolate many OpenGL implementation bugs in the past. To extend testing even more, free 3D game engines like Ogre¹² and Irrlicht¹³ will be run on top of Wine.

Literature Research

A thorough literature research will be conducted to identify related work and other attempts at achieving similar goals. A study of currently established continuous integration and dependency management processes will provide a starting point for the prototype implementation and its communication protocols.

³<http://www.openengsb.org>

⁴<http://karaf.apache.org>

⁵<http://www.jboss.org/drools>

⁶<http://wicket.apache.org>

⁷<http://code.google.com/p/guava-libraries>

⁸<http://www.winehq.org>

⁹<http://www.mesa3d.org>

¹⁰<http://www.x.org>

¹¹<http://www.opengl.org>

¹²<http://www.ogre3d.org>

¹³<http://irrlicht.sourceforge.net>

Proposed Solution Approach

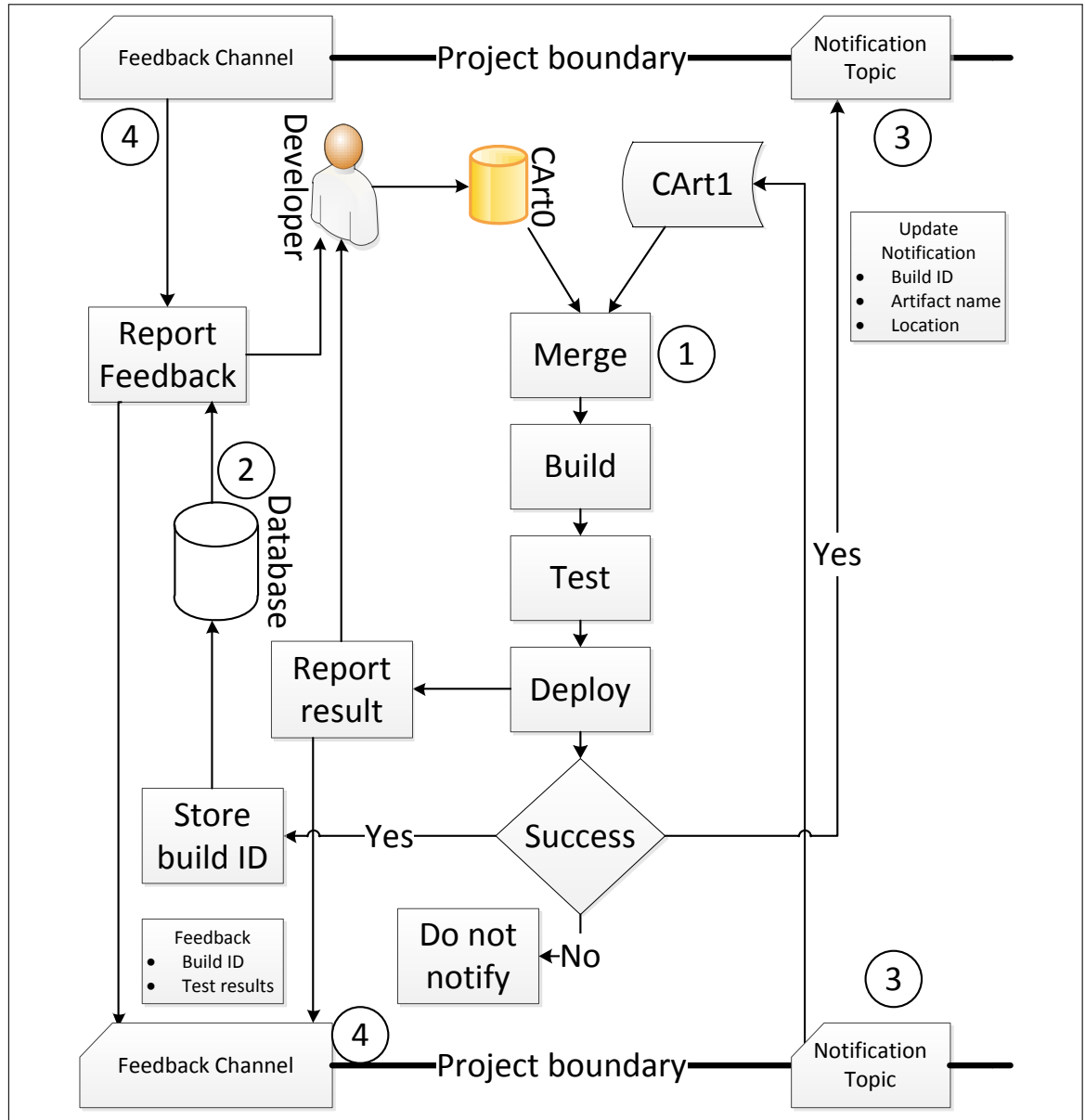
This section describes the changes CCIP makes to the continuous integration workflow, the artifact transfer and merging steps and the prototype implementation used for the evaluation.

6.1 The Continuous Change Impact Analysis Process Workflow

CCIP makes a number of modifications to the regular CI workflow (see section 2.5). The core modifications are the addition of a dependency merge step and generation of update notifications. Furthermore, CCIP introduces feedback messages, but they are handled outside the core workflow.

Figure 6.1 outlines the CCIP workflow. At the core of the CCIP is a publish-subscribe message system to send update notifications and feedbacks. It introduces the following changes to the standard CI workflow in circled numbers:

1. Merge: In addition to changes to the project's own source code, the CCIP workflow can be triggered by dependency update notifications. The dependencies have to be integrated into the build environment. For more details please refer to section 6.3. The CI server has to keep track of the origin of the change to be able to send the feedback to the correct destination.
2. Build Database: In order to handle feedback correctly, the CI server has to store information about old builds, most importantly which change triggered the build. Most CI servers already do this for other reasons, like allowing the user to look up the build history.
3. Extended notification: On successful builds a notification message is sent via the update notification topic. Other CI servers can subscribe to this topic to receive them. If the build fails, no public notification is sent, but the regular project-internal build report is still generated. See section 6.2 for details about these messages.



4. Feedback handling: If a build was triggered by an update notification, the builds result (positive or negative) is sent to the feedback queue of the CI server that sent the update notification. For details see section 6.2.

To initiate the process, the CI server of the dependent code artifact has to make contact to the CI servers of its dependencies. Ideally, the necessary information can be extracted from the existing build system configuration if the build system supports dependency management and automatic dependency downloading. Otherwise the administrator of the CI server has to provide the necessary information manually.

The lines at the bottom and top of figure 6.1 symbolize the project boundaries to dependencies and dependents. As the process is cascading, update notifications can be sent to dependents when a dependency changes. Feedback from dependents is forwarded to dependencies if it concerns an update originating from that dependency. One change can result in multiple update notifications if there is more than one registered dependent and one update notification can lead to more than one feedback message. Sections 5.1 and 8.2 discuss scalability concerns related to this cascading operation.

Execution Example

This subsection provides an execution example to illustrate the operation.

To begin the process, a developer commits a change to CArt1. This starts the first iteration, inside CArt1's CI server:

1. Merge: No special steps are needed because no dependency was changed. Different actions are possible, see section 6.3.
2. Build, test: The regular build process is executed and passes successfully.
3. Deploy: CArt1's CI server uploads the artifacts to a public location.
4. Store result: CArt1's CI server generates a unique identifier for this build and stores it with the build parameters.
5. Update notification: An update notification is published in the update notification topic.
6. Developer report: A build report may be sent to the developer as a regular CI server would do.

The update notification triggers the second iteration, which runs inside CArt0's CI server:

1. Merge: The CI server has to fetch the new build of CArt1 from the location provided in the update notification and instruct the build system to use it.
2. Build, test, deploy: They are executed in the same way as in the first iteration.
3. Store result: The server has to store which dependency triggered the new build and the unique build ID.

4. Update notification, Developer report: Same as in the first iteration.
5. Feedback report: A feedback report is generated and sent to CArt1's CI server.

Let us assume that a third code artifact CArtM exists, which imports CArt0. The update notification from CArt0's CI server triggers a third execution iteration, which is assumed to fail. In this case, the merge, build, test and feedback steps are executed in the same way as in the second iteration, and the deploy and update notification steps are skipped.

When CArt0's CI server receives the feedback message from CArtM's server, it proceeds to process it:

1. Database lookup: The first step is to look up the build that the message refers to.
2. Developer notification: A notification email with the build and test reports is sent to CArt0's developers.
3. Cascading feedback: Because the build the feedback refers to was triggered by an update of CArt1, a feedback message is sent to CArt1's CI server.

CArt1's server handles the feedback in the same way, except that the build of CArt1 was triggered by a change to CArt1, so no further feedback messages have to be generated. At this point, the CCIP execution finishes.

6.2 Communication Protocol

As described before the CCIP relies on communicating CI servers. Therefore, it is necessary to clarify the type and amount of information exchanged between the various servers. An update notification needs at least the following information:

- A unique build ID.

This is necessary to relate feedback to builds.

- The artifact name.

The CI servers receiving the update notifications need this information to know which dependency has been changed and which projects have to be rebuilt. An alternative way is to have one update topic per project.

- Information where to obtain the artifact.

This information depends on the build system and cannot be generalized. Considerations regarding artifact transfer and merging are described in section 6.3.

- How to report feedback.

A feedback message from project dependents contains:

- The build ID to which this feedback is a response to.

- The build result: Build failure, test failure, success or failure in a dependent project.
- Debugging information like logging output in the case of a failed build.
- Contact information of the feedback sender.

Obviously it is up to the receiver of an update notification which information he sends or if he sends feedback at all. Privacy may be a concern here as test names or test output may reveal confidential information.

6.3 Artifact Transfer and Merging

How artifacts are transferred and integrated depends on the kind of the artifact and the build system the project uses. Some build systems like Apache Maven download dependencies automatically. In this case the only necessary information is the new version. To merge the new version into the dependent project, the merge step merely has to adjust the version information in the Maven configuration files. On the other hand, a dependency like an operating system kernel or device driver may make a reboot of the test machine necessary.

As a consequence, it is not possible to define the layout of the download information and the merge step in a way that will work for every software artifact. To keep the setup effort low for average projects it may be conceivable to provide templates for popular build systems and handle the remaining cases with user-written scripts.

Handling of Multiple Dependencies

There are multiple dependency configuration options if a project has multiple dependencies. For example, consider a setup similar to figure 5.3, where CArt0 has two dependencies CArt1 and CArt2. Now the CCIP server receives an update notification from CArt1, followed by an independent update notification from CArt2. When building CArt0 in response to the update of CArt2, it can revert the version of CArt1 back to the version referenced in CArt0's source code or keep using the version from the most recent update notification. The prototype implementation uses the first option.

Ideally, a CCIP implementation would support both ways and allows the user to choose the desired behavior. If a user requests a manual rebuild, all available dependency versions should be listed, with the ability to select an arbitrary combination for the build. In either case, the versions used for the build must be recorded for reproducibility.

6.4 Prototype Implementation

The prototype implementation will be used to evaluate CCIP (Chapter 7). This section describes how the concept from section 6.1 is implemented in the prototype.

The prototype is based on the OpenCIT¹ CI server introduced in section 2.5. This build server was chosen because of its small size, which makes it easy to modify, and the strength and flexibility of the OpenEngSB² middleware it is built on.

As described in section 2.5, OpenCIT already supported the Git revision control system and Maven build system. An additional connector calling arbitrary commands was written to support other build systems via custom scripts.

Dependency domain and connectors

The first substantial modification to OpenCIT itself was the addition of the merge step. To abstract different build systems, a new *dependency* domain was introduced. The domain's Java interface is given in listing 6.1:

```
1 public interface DependencyDomain extends Domain {  
2     @Raises({ MergeSuccessEvent.class , MergeFailEvent.class })  
3     void merge(OpenEngSBFileModel path ,  
4         String dependencyLocation , long processId );  
5 }
```

Listing 6.1: Dependency Domain Interface

The domain has only one method, *merge()*, which accepts two main parameters: The path to the project's source code and a connector-specific string identifying the dependency location. The *processId* parameter is used for interacting with the workflow engine. Following the general OpenEngSB convention [54, Ch. III], the operation is asynchronous and raises a *MergeSuccessEvent* after successful completion, or a *MergeFailEvent* in case of an error.

The merge domain is invoked by the OpenCIT CI workflow (see figure 2.3) after checking out the source and before invoking the build domain. See figure 6.2 for the modified workflow.

Two dependency connectors were written: *MavenDep* and *DummyDep*. *MavenDep* is used for Maven projects. Its *dependencyLocation* parameter is the new version of the dependency. A connector instance parameter set by the user specifies the path to the pom.xml file and the attribute which defines the version of the dependency that should be used. The *merge()* method replaces the existing version string with the new version string. Maven is responsible for the actual artifact transfer.

The *DummyDep* connector is used for non-Maven projects. Its *merge()* method is a no-op method that simply raises a *MergeSuccessEvent*. The artifact is installed into a common file location by the deploy script of the dependency and the dependent's build scripts are instructed to source the artifact from this location. This works on one system only, unless the common location is on a shared file system, but it is a simple and working solution for the evaluation.

¹<http://www.openengsb.org/>

²<http://opencit.openengsb.org/>

Update Notification

The notification and feedback mechanism is implemented with the Java Message Service (JMS). OpenEngSB provides the Apache ActiveMQ³ JMS provider. OpenCIT spawns an ActiveMQ server on startup to accept subscriptions from other OpenCIT instances. For each configured project, a topic is created.

The update notifications are JMS ObjectMessages containing the object from listing 9.2:

```
1 public class UpdateNotification implements Serializable {
2     private UUID buildId;
3     private String artifactLocation;
4     private String feedbackQueue;
5     private String dependencyName;
6
7     // Getters and setters removed
8 }
```

Listing 6.2: Update Notification Contents

Getters and setters exist in the class definition, but have been removed to keep the listing short. To generate the notifications, an additional step was added to the workflow after the successful execution of the deploy step. The string *artifactLocation* is generated by the deploy connector and later passed to the dependency connector.

Feedback Handling

Feedback is accepted by a JMS queue. There is one global queue for each server instance. The queues name is communicated to other servers in the update notifications. Feedback messages contain the object defined in listing 18.3:

```
1 public class BuildFeedback implements Serializable {
2     public enum BuildResult {
3         SUCCESS,
4         MERGEFAIL,
5         BUILDFAIL,
6         TESTFAIL,
7         DEPLOYFAIL,
8         NESTEDFAIL // A dependent project reported failure
9     };
10
11     private UUID buildId;
12     private BuildResult result;
13     private BuildFeedback nestedFeedback;
14     private String info;
15     private String contactInfo;
```

³<http://activemq.apache.org/>

```

16     private String projectName;
17 }

```

Listing 6.3: Feedback Contents

BuildId contains the buildId from the update notification. *Result* contains the build result. *NestedFeedback* contains the original feedback message if the feedback is forwarded from a grandchild and is *null* otherwise. *Info* and *contactInfo* contain human-readable information about the failure and contact information. OpenCIT currently uses the server administrator's email address as contact information.

Workflow

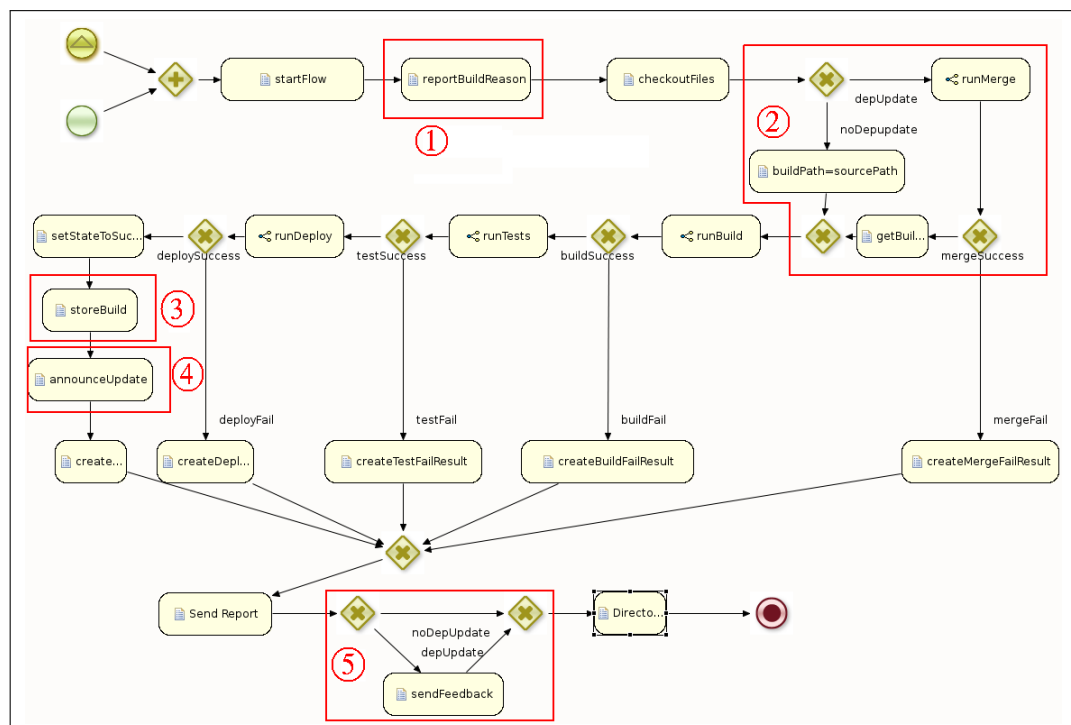


Figure 6.2: CCIP workflow implemented in JBoss Drools.

Figure 6.2 shows the implementation of the CCIP workflow in JBoss Drools. The main differences from the original CI workflow (Figure 2.4), marked in red, are:

1. Keeping track of the reason for the build:

The reason for the build is passed to the workflow. The BuildReason object is later added to notifications sent to developers and stored in the build database. Possible reasons for triggering builds are updates to the project's own code, updates to a dependency or a manual rebuild requested by the user. The commit ID or dependency build ID are stored inside the BuildReason object.

2. Merge step:

If a rebuild is triggered by a dependency update, the dependency connector is invoked to perform a merge.

3. Build database:

Successful builds are recorded in the build database.

4. Update Notification

5. Feedback:

If the rebuild was triggered by a dependency update, a feedback notification is sent regardless of the build outcome.

Like in the original CI workflow, error handling is accomplished by accepting the failure events that are defined in the tool domains used by the workflow. The gateway nodes *mergeSuccess*, *buildSuccess*, *testSuccess* and *deploySuccess* are used to react to errors and create appropriate feedback messages.

Evaluation

This chapter describes the evaluation of CCIP by presenting selected projects, their dependency relationships, the test setup and test results.

7.1 Evaluated Projects

To evaluate CCIP, 8 projects have been selected. These projects form two separate dependency trees. The prototype implementation has been used to monitor their development. Reported test failures were recorded and investigated. This section describes the tested projects, their dependency relationships and the quality assurance measures used by those projects.

The first dependency tree consists of OpenCIT itself, and the OpenEngSB middleware it is based on. OpenEngSB itself depends on Apache Karaf¹, which in turn uses Apache Aries² and JLine 2³. These projects were selected because they are direct and indirect dependencies of OpenCIT. Figure 7.1 shows the dependency tree.

The second set of projects consists of projects forming the 3D rendering stack of the GNU/Linux operating system. The bottom project is Mesa 3D⁴. Two dependents of Mesa are tested, the Irrlicht Engine⁵, and Wine⁶, a reimplementation of the Microsoft Windows API. On top of Wine, Ogre 3D⁷, another open source 3D engine, is tested. Figure 7.2 illustrates the dependency tree. These projects were selected because the author's experience with Wine development and dealing with regressions introduced by graphics driver updates was a major motivation for this the research topic.

¹<http://karaf.apache.org/>

²<http://aries.apache.org/>

³<https://github.com/jline/jline2>

⁴<http://www.mesa3d.org/>

⁵<http://irrlicht.sourceforge.net/>

⁶<http://www.winehq.org/>

⁷<http://www.ogre3d.org/>

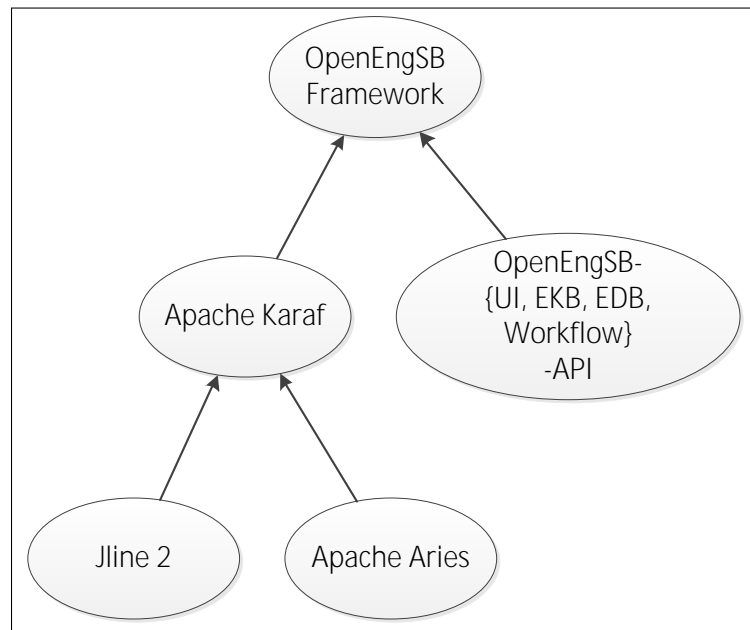


Figure 7.1: OpenCIT, OpenEngSB and their dependency tree

Mesa 3D

Mesa 3D is an open source implementation of the OpenGL [45] real-time 3D rendering API. Mesa provides software rendering⁸ as well as hardware acceleration for popular GPUs from NVidia⁹, AMD¹⁰, Intel¹¹ and other vendors.

Mesa is part of the larger Linux graphics stack. Other components of this graphics stack are:

- Direct Rendering Manager:

The direct rendering manager is part of the Linux kernel and provides subroutines and drivers necessary to initialize the graphics hardware, detect attached display devices and configure them. It also provides video memory management.

- The X.Org X11 server:

The X.Org server is a display server that implements the X11 display protocol [23]. It communicates with applications that wish to show a window on the screen and manages input devices like mice, keyboards and joysticks.

- The DDX drivers:

The DDX drivers are hardware-specific modules that allow the X.Org server to talk to the graphics hardware and the modesetting components in the kernel. The DDX drivers

⁸<http://www.mesa3d.org/llvmpipe.html>

⁹<http://nouveau.freedesktop.org/wiki/>

¹⁰<http://dri.freedesktop.org/wiki/Radeon>

¹¹<http://dri.freedesktop.org/wiki/Intel>

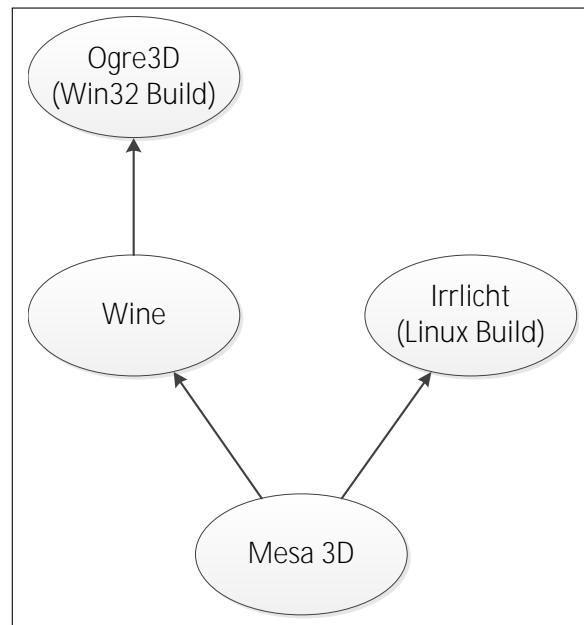


Figure 7.2: OpenCIT, OpenEngSB and their dependency tree

are fairly small since most of their functionality is provided by the kernel. Their main responsibility is providing acceleration for 2D operations.

No formal documentation other than the source code exists that describes these components, but a blog post by Jasper St. Pierre¹² gives a good overview over the above and other components.

Mesa only provides 3D rendering support. It is a good candidate for testing because it can be replaced without any special system permissions or setup steps. To replace the Mesa version provided by the system, it is enough to point the application using the OpenGL API to a different libGL.so implementation by adjusting the environment variable `LD_LIBRARY_PATH`. Unlike Mesa, the X.Org server and DDX drivers requires special permissions to initialize the hardware, so they can only be modified and restarted by the root user. To replace the modesetting infrastructure the kernel has to be replaced and the system rebooted. Since a reboot terminates every application on the test system, including the CI server, those components are very difficult to test by continuous integration.

Mesa has an extensive testsuite, the Piglit OpenGL driver testing framework¹³. However, Piglit is not easy to execute. No rendering backend passes all tests¹⁴. As a result, developers have to remember which tests used to pass on their own their system and compare the results after their changes to the results before. Because of this, and the high diversity of platforms supported by Mesa, no continuous integration system is used by Mesa. Instead, Mesa relies on

¹²<http://blog.mecheye.net/2012/06/the-linux-graphics-stack/>

¹³<http://people.freedesktop.org/~nh/piglit>

¹⁴<http://people.freedesktop.org/~nh/piglit/results/all/problems.html>

a large number of developers and users who follow the day-to-day development to provide basic quality assurance.

Irrlicht

Irrlicht is an open source game engine. It provides a rich set of features¹⁵, ranging from setting up 2D and 3D rendering to character animations, particle effects, resource management and many more. Irrlicht is in use by a number of open source and proprietary games¹⁶. As a platform independent engine, it runs on Microsoft Windows, MacOS, Linux and other systems. It can use OpenGL as well as Microsoft's Direct 3D as rendering backends. Figure 7.3 shows one of Irrlicht's tech-demos.



Figure 7.3: Irrlicht tessellation tech-demo. Source: Irrlicht website¹⁷

Irrlicht has a set of regression tests that are provided to developers for pre-commit testing¹⁸. The use of a continuous integration server is not documented.

¹⁵<http://irrlicht.sourceforge.net/features/>

¹⁶<http://irrlicht.sourceforge.net/projects>

¹⁷http://sourceforge.net/apps/gallery/irrlicht/index.php?g2_itemId=1384

¹⁸<http://irrlicht.svn.sourceforge.net/viewvc/irrlicht/trunk/tests/tests-readme.txt>

For the evaluation, Irrlicht and its tests were compiled to Linux binaries and the OpenGL rendering backend was used. Irrlicht uses SVN to store the sourcecode. Because OpenCIT does not have a working SVN backend, a static revision of Irrlicht was used for the evaluation. This is an acceptable compromise because Irrlicht is a leaf of the dependency tree and regressions caused by code changes to leaf projects are not relevant to the evaluation of CCIP.

Wine

The goal of Wine is to execute applications written for Microsoft Windows on GNU/Linux, Apple's OS X and other Unix-based systems. Therefore, it provides a reimplementations of the APIs provided by Windows. Among the interfaces implemented by Wine are DirectDraw and Direct3D, which are 2D and 3D programming interfaces used by the majority of Windows-based games. Figure 7.4 shows a Linux desktop with two Windows applications and one of Wine's builtin tools running.

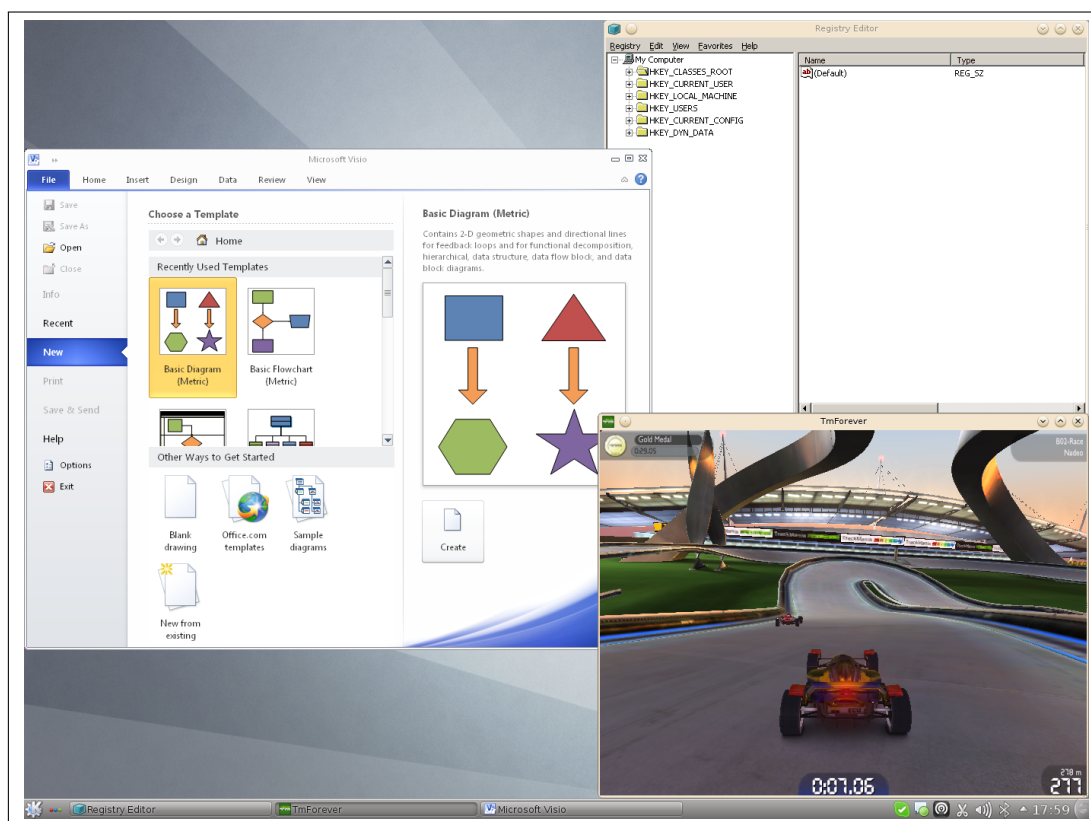


Figure 7.4: MS Visio, Trackmania Nations and Wine's registry editor on a GNU/Linux desktop

Wine contains a big amount of conformance tests¹⁹ for all components. These tests not only serve as regression tests, but also document the behavior of Windows for technical and legal

¹⁹<http://wiki.winehq.org/ConformanceTests>

reasons. All conformance tests have to pass on Windows.

Wine uses continuous integration, but with a few modifications. Instead of using a continuous integration server, Wine employs a management model where only one developer (currently Alexandre Julliard) has write access to the public repository. As part of the commit process, patches have to pass the tests on his test systems to be accepted²⁰.

Additionally, Wine uses a CI-like server infrastructure called *Wine Test Bot*²¹ to test a subset of patches, namely those that modify tests, to verify that new or modified tests execute correctly on different versions of Windows.

The Testbot does not wait for patches to be applied to the project's Git repository. Instead it tests new patches as soon as they are submitted to the patch submission mailing list. Furthermore, it allows developers to upload patches for testing manually. The main intention of the Testbot is to make it easier for developers to test on the full range of Windows versions, starting from Windows NT 4.0 up to Windows 7. It also runs the tests as 32 bit and 64 bit executables, and as 32 bit executables on 64 bit Windows.

Wine is currently working²² on extending its Testbot to pretest all patches and include testing on GNU/Linux, OS X and other target platforms. At the time of the writing of this thesis, this work was in progress and not yet used for development.

Ogre3D

Ogre3D is a game engine similar to Irrlicht. Like its competitor, it provides²³ a comprehensive framework for 3D rendering, including multiple rendering backends, material, geometry and animation management, as well as helper functions for managing game data in a platform independent way. Figure 7.5 shows a screenshot of one of Ogre's example programs.

Ogre contains a number of tests that are used in a continuous integration setup²⁴. The tests render a number of simple and complex scenes to detect regressions.

One aspect of Ogre's tests worth mentioning is that the tests do not know if the rendering output is correct or not. Instead, it is necessary to generate one reference rendering for each test. When the tests are run, their output is compared to this reference rendering. Any difference (with some wiggle room for precision issues) is interpreted as a test failure.

Because neither OpenGL [45][App. A] nor Direct3D require pixel-exact rendering, the reference images have to be generated by the system under test. To make sure this the initial rendering is correct, the reference images were manually compared to images generated on Windows.

JLine 2

JLine is a small (about 8000 lines of code) Java library for handling command line input. It allows Java programs to provide usability features like retrieving previously entered commands, editing commands and tab completion.

²⁰<http://wiki.winehq.org/SubmittingPatches>

²¹<https://testbot.winehq.org/>

²²<http://wiki.winehq.org/BuildBot>

²³<http://www.ogre3d.org/about/features>

²⁴<http://www.ogre3d.org/tikiwiki/Visual+Unit+Testing+Framework>



Figure 7.5: Ogre water rendering sample

JLine was included in the evaluation because Apache Karaf imported the current development snapshot. This required building JLine locally rather than downloading a binary version from the public Maven repository. It was also an easy opportunity to add another project to the evaluation set.

Apache Karaf

Apache Karaf is an OSGi [2] runtime implementation. Karaf bundles Apache Felix, an OSGi container, and Apache Aries, which implements core OSGi services and libraries, into a convenient to use distribution. In addition to the bundled components, it provides convenience functions like a more comfortable administration shell (compared to Felix) and branding utilities.

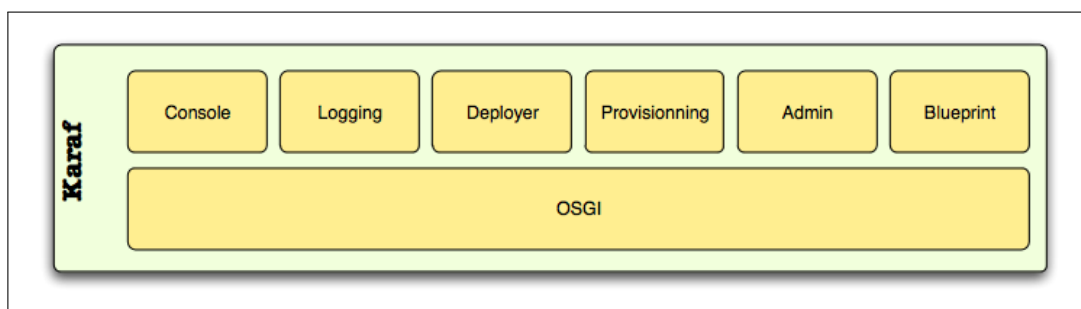


Figure 7.6: Karaf component overview [21]

Figure 7.6 shows the components forming Karaf. The OSGi box (the container) is provided by Apache Felix, although it can be replaced by other OSGi containers. The Blueprint implementation is provided by Apache Aries. The rest of the components in this figure are implemented in Karaf itself.

The Karaf source code contains unit and integration tests, which are executed by default when building Karaf with Maven. The Karaf project uses the Jenkins CI server to monitor the development. The build reports are publicly available at <https://builds.apache.org/job/Karaf/>.

Unfortunately, the Karaf tests do not pass successfully if a separate instance of Karaf is running. Since the prototype build server uses Karaf, the tests had to be skipped. Build errors introduced by changes to Karaf or its dependencies were still detected.

Apache Aries

Apache Aries implements OSGi services, libraries and the Blueprint dependency injection container. It is bundled by Apache Karaf to form a complete OSGi runtime implementation. Aries was easy to include in the evaluation because the evaluated Karaf versions import the development snapshot of Aries.

The Aries project uses continuous integration. The build results can be found at <https://builds.apache.org/job/Aries/>. Aries' tests are executed during the build process by default and worked correctly when run by the prototype CCIP implementation.

Open Engineering Service Bus

The Open Engineering Service Bus (OpenEngSB) describes itself as an easy-to-use and easy-to-adapt platform for tool integration²⁵. Its core is a middleware framework that provides services commonly used in enterprise software like workflows and persistence.

The unique feature of the OpenEngSB are tool domains and connectors. A tool domain defines an abstract interface for a certain family of tools, for example an interface for notification tools. Tool-specific connectors implement this interface, for example notification connectors for emails, IRC, Facebook²⁶ or Twitter²⁷. A business logic can be implemented as a set of workflows that react to events generated by connectors and invoke connectors to perform actions. The workflows use the interface provided by the domains and are independent of the actual connector implementation.

OpenCIT

The build server the prototype implementation is built on was included in the evaluation. See section 6.4 for more details.

OpenCIT contains a small number of unit tests that verify the functionality of its components. The main contribution of these tests is to check that the Drools workflow that imple-

²⁵<http://www.openengsb.org>

²⁶<https://www.facebook.com>

²⁷<http://www.twitter.com>

ments the CI process deploys and runs correctly and that the scheduling service starts builds. OpenCIT's tests are integrated into its Maven-based build system.

7.2 Evaluation Setup

This section describes the hardware and software environment in which the evaluation was conducted.

Hardware

The test system was a PC system with an Intel® Core i7® CPU, an AMD® Radeon® HD 5770 GPU and 8 GB of memory. All components were consumer-grade hardware. Memtest86+²⁸ was used to perform a basic reliability test of the core system components. The hardware was in use as a development and performance test machine since March 2010 and has performed reliably since then.

The Radeon HD 5770 GPU was deliberately chosen because Mesa's r600g driver, which is used for this hardware, was known to successfully pass the Wine tests. No special criteria were used to select the other hardware, other than the fact that they provided adequate performance and were available for testing.

Software

The software was a 64 bit installation of Gentoo Linux, the initial versions of software relevant to the test were: Linux 3.2.12-gentoo (kernel), IcedTea6 1.11.1 (Java SDK), Apache Maven 3.0.4, X.Org X Server 1.11.4, libdrm 2.4.34, LLVM 2.9, xf86-video-ati 6.14.3.

During the evaluation, software updates were installed as they were released by the Gentoo developers. This was necessary because some Mesa changes required updates to the kernel and libdrm. The kernel was upgraded to version 3.4 and eventually version 3.5. The X.Org server was upgraded to version 1.13.0, LLVM to 3.1 and libdrm to 2.4.40.

To simplify the evaluation, only one OpenCIT instance was used, and all tested projects were handled by it. The build server ran in an unprivileged user account. Bash scripts were used to invoke the build and tests systems of Mesa, Wine, Irrlicht and Ogre. OpenCIT generated one email report for every build and feedback messages. A dedicated email account was created to receive and archive these mails.

Regression Analysis

Once a regression was reported, a manual analysis was performed to classify the regression and find out more details. This analysis included:

- Attempt to reproduce the failure.
- Isolate the change causing the regression with the help of git-bisect.

²⁸<http://www.memtest.org/>

- A quick read of the localized change to assess the nature of the failure.
- Scan the project's mailing lists and bug trackers for reports indicating the same problem.
- Once the regression was fixed, use git-bisect again to isolate the change fixing it.

7.3 Evaluation Results

This section describes the results of the evaluation. It lists the tested range of commits and all reported regressions. Finally, it gives a closer description of imported regressions and looks at how CCIP performed for each project.

Raw Data

Table 7.1 lists the commit ranges tested for each project, the number of commits, the number of regressions caused by the project and the number of regressions found by the project.

Project	Start	End	No. Commits	Bugs in	Bugs Found	Ratio
Mesa	50b91aa3	1d0c6211	5362	16	10	0.30%
Wine	7123e441	003622c0	3054	0	8	0.00%
Irrlicht	r4233	r4233	1	0	2	-
Ogre 3D	6e912506	6e912506	1	0	0	-
JLine 2	7f0091fc	d7c9a348	9	0	0	0.00%
Aries	8186fe5b	5b9ff639	357	5	6	1.40%
Karaf	8141000e	64cbd994	283	2	1	0.71%
OpenEngSB	5eff4bf1	28d2f611	485	1	4	0.21%
Host system				3	0	-

Table 7.1: Commit ranges tested and regressions found

Start states the first tested commit, *end* the last. *No. Commits* gives the number of commits that were tested. *Bugs in* is the number of regressions caused by the project. *Bugs found* is the number of regressions found by the project. *Ratio* is *Bugs in* divided by *No. Commits*.

Note that some regressions were reported by more than one project, so the sum of the bugs found column is larger than the sum of the bugs in column. False positives are not counted in this table. Table 7.1 does not count false positives, but it does count random bugs if they could be attributed to a project and detected problems in the host system.

Figure 7.7 shows the number of regressions found for each project, and where they were located. The number in the ellipse shows the number if build or test errors discovered in the same project that caused them. An outgoing arrow symbolizes regressions found by the project where the arrow originates that were caused by the project the arrow points at. Transitive dependencies with a regression discovery count of 0 are excluded from the figure. The *Host* project is a pseudo project symbolizing the host system.

Table 7.2 lists the regressions reported by CCIP. The table contains the following information:

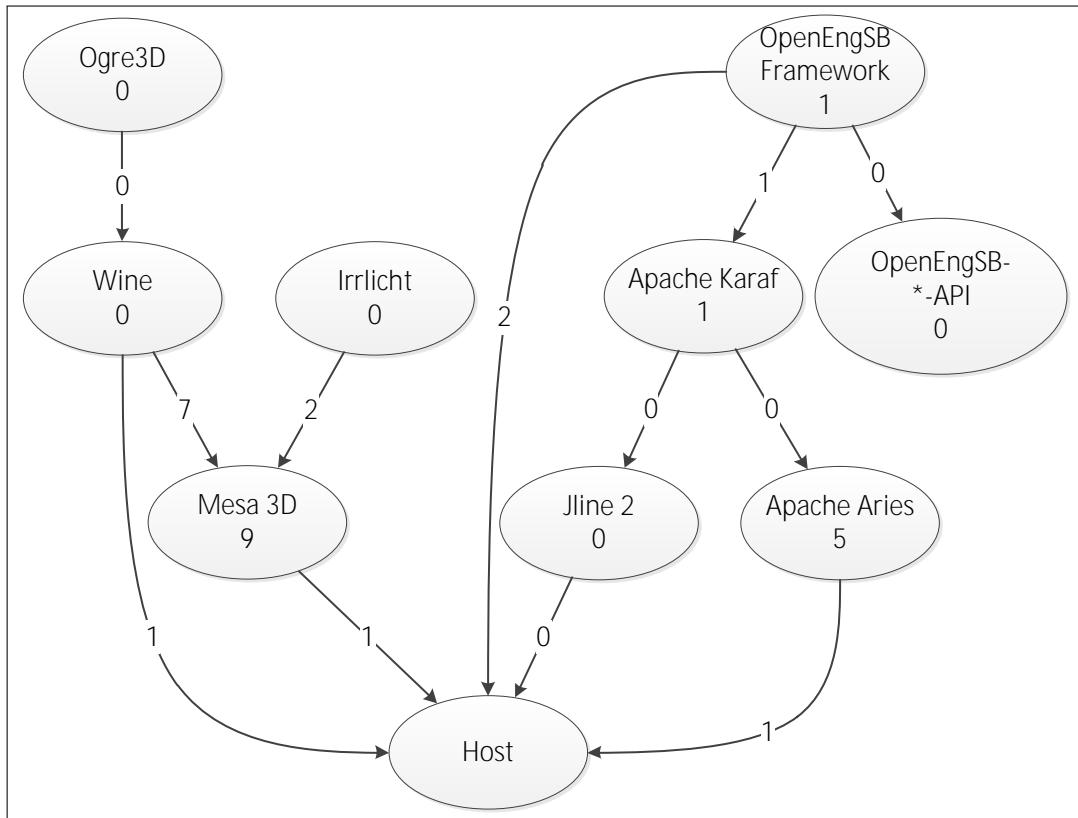


Figure 7.7: Regressions found

- **Id:** Gives each line a unique identifier.
This identifier is referenced in the later subsections describing the results in more detail.
- **Buggy Proj.:** The project which contains the regression.
- **Detecting Proj.:** The project which detected the regression. Can be the same as the buggy project.
- **Class.:** Classifies the regression as good, neutral or bad for the merits of CCIP. See below for a closer description.
- **Detect Commit:** The first 4 bytes of the SHA1 checksum of the commit where the regression was detected.
- **Detect Time:** The time when the regression report was generated.
- **Causing Commit:** The commit which introduced the regression.
- **Cause Time:** The time when the commit was pushed to the public repository.

This time states the git *committer time*, not the *author time*. The author time states when the commit was written, which may be well before it was pushed to the repository.

- Fix Commit: The commit which fixed the regression.
- Fix Time: The committer time of the fix commit.
- Discovery Time: The time when the regression was discovered by or reported to the developers.

This can be a bug report that was filed, a mail sent to the mailing list or a message on the project's IRC channel. Sometimes a bug report is stated in the fixing commit, other reports were found by searching the mailing list archives and bug trackers. Not all regressions had such a record.

- Description: A short description of the regression.

A regression is classified as *good* if it is a valid regression that has been discovered by CCIP, but would have remained hidden with classic CI. This applies to all valid regressions where the detecting project differs from the buggy project.

A regression is classified as *neutral* if it is a valid regression that was discovered by normal CI operation. This applies to regressions where the detecting project is the project that contains the bug.

A regression is classified as *bad* if it is a false positive or a random, unreproducible issue. The consideration behind this is that false positive reports waste the dependency's developer's time, and random issues are likely to reproduce erroneous reports, even if they are valid bugs.

Lines where the buggy project is *host* are failures due to bugs in the host system, or caused by mistakes during maintenance of the host system. Such lines are classified as bad. Eliminated from the table are errors generated during the initial setup. The consideration behind this is that a sensible administrator would test his setup before taking reporting to dependencies live.

The timezone of the times in the table is UTC+0.

If a field is not relevant to the regression (e.g. a fix for a false positive) contains *x*. If some information could not be found, the table field contains a question mark. See the following subsections for a more detailed elaboration. A regression with a found date of *self-reported* was reported to the project developers by the author of this thesis. This was done if a fix for the bug was required to proceed with the evaluation. A discovery date of *history* denotes bugs that were discovered during the testing of old Mesa commits. The comparison of the CCIP discovery time and the time it was found by classic development is meaningless in this case.

Table 7.2: Test failures reported by CCIP

Id	Buggy Proj.	Detecting Proj.	Class.	Detect Commit Detect Time	Causing Commit Cause Time	Fix Commit Fix Time	Discovery Time	Description
1	mesa	wine	good	e6280c3b history	6ccda72b 2012-01-02 20:41	4becf676 2012-01-04 20:43	2012-01-03 04:03	Library link error
2	mesa	wine	good	78402613 history	a103c61d 2012-01-07 08:36	54e8dcaa 2012-01-21 12:43	never	Vertex shader clipping
3	host	mesa	bad	e81ee67b 2012-09-09 10:09	system update x	x	x	LLVM broken
4	host	openengsb	bad	87aac4bb 2012-09-10 13:29	broken setup x	x	x	Build setup change required
5	mesa	mesa	neutral	34472a0d 2012-08-18 01:23	34472a0d 2012-08-18 01:12	d0ace4e9 2012-08-18 13:53	2012-08-18 01:28	Build failure
6	unknown	wine	bad	6a85725f 2012-09-04 17:47	x x	x x	x	Unreproducible random failure
7	mesa	wine	good	428855ee history	5ec7c28f 2012-04-23 23:39	e0773da1 2012-04-23 23:39	never	Temporary refactoring bug
8	mesa	wine	good	6882381a 2012-07-13 16:40	6882381a 2012-07-12 19:29	workaround x	never	Outdated libxcb not detected
9	mesa	wine, irrlicht	good	99c65bac 2012-07-18 05:42	30257c32 2012-07-18 04:25	1ffac44e 2012-07-24 19:08	Self-reported	Fail with Linux <= 3.3.x
10	karaf	openengsb	good	initial build 2012-07-12 00:00	? ?	8141000e 2012-07-17 07:51	Self-reported	Incorrect dep. versions
11	mesa	wine, irrlicht	good	42723d8 2012-08-30 01:19	d638da23 2012-08-29 22:09	055093e3 2012-08-30 15:28	2012-08-30 03:06	Library link error
12	mesa	mesa	neutral	4a79545b history	8b902056 2012-01-26 09:07	41204252 2012-01-26 10:49	?	Compile error
13	mesa	mesa	neutral	e4340c19 history	f53e7e98 2012-01-31 17:04	8c436b4e 2012-02-01 02:33	2012-01-31 19:12	Compile error

Table 7.2: Test failures reported by CCIP

Id	Buggy Proj.	Detecting Proj.	Class.	Detect Commit Detect Time	Causing Commit Cause Time	Fix Commit Fix Time	Discovery Time	Description
14	mesa	mesa	neutral	b3ba0a7a 2012-07-13 12:05	defadf2b 2012-07-13 11:44	39d82a1b 2012-07-13 16:20	2012-07-13 14:00	Compile error
15	mesa	mesa	neutral	3469715a 2012-07-17 12:50	3469715a 2012-07-17 12:42	bf484024 2012-07-17 15:11	2012-07-17 14:08	Compile error
16	aries	aries	neutral	500be7b4 2012-07-17 16:50	e8eb2e07 2012-07-17 16:28	x x	x	Maven update delay
17	aries	aries	neutral	3e804f7c 2012-07-20 10:05	3e804f7c 2012-07-20 10:03	x x	x	Maven update delay
18	karaf	karaf	neutral	6779beb1 2012-07-23 10:26	6779beb1 2012-07-23 10:19	80981140 2012-07-24 06:16	2012-07-24 06:16	Blueprint core ver. Incorrect
19	mesa	mesa	neutral	33ef67ab 2012-08-01 16:46	33ef67ab 2012-08-01 16:37	84ead7b4 2012-08-01 17:12	2012-08-01 17:02	Compile error
20	aries	aries	neutral	e9345fa5 2012-08-01 19:26	e9345fa5 2012-08-01 19:24	a0ef3e2b 2012-08-14 07:54	?	Compile error
21	openengsb	openengsb	neutral	969a04ec 2012-08-04 20:04	969a04ec 2012-08-04 19:54	08b21335 2012-08-05 23:19	?	Test failure
22	aries	aries	neutral	89741801 2012-08-15 09:56	? ?	x x	x	Maven update delay
23	host	aries, openengsb	bad	1aa84c9d 2012-08-29 08:52	2acc2a31 2012-08-28 10:23	x x	x	pom.xml moved
24	aries	aries	bad	546f50f1 2012-09-11 08:21	546f50f1 2012-09-11 07:53	? ?	?	Random test failures
25	mesa	mesa	neutral	2bc8f03f 2012-09-15 17:05	9f37b405 2012-09-15 09:57	b6c2234c 2012-09-15 17:18	2012-09-15 11:51	Compile error
26	mesa	mesa	neutral	8d977858 2012-10-01 17:34	8d977858 2012-10-01 15:37	00d80b3a 2012-10-01 21:42	2012-10-01 20:19	Compile error with old LLVM

Table 7.2: Test failures reported by CCIP

Id	Buggy Proj.	Detecting Proj.	Class.	Detect Commit Detect Time	Causing Commit Cause Time	Fix Commit Fix Time	Discovery Time	Description
27	mesa	ogre	bad	f44bda17 2012-09-04 01:48	f44bda17 2012-09-03 20:18	x x	x	Correct precision change
28	mesa	ogre	bad	7d624799 2012-09-18 14:37	b33d7eaa 2012-09-13 21:33	x x	x	Correct precision change
29	unknown	wine	bad	? ?	? ?	? ?	?	Random test delays
30	host	wine	bad	6a85725f 2012-09-04 17:57	x x	x x	x	No cleanup after tests
31	openengsb	opencit	bad	initial build 2012-05-01 00:00	x x	x x	x	Intentional API change
32	drools	openengsb	bad	initial build 2012-05-01 00:00	x x	x x	x	Intentional API change
33	wicket	openengsb	bad	initial build 2012-05-01 00:00	x x	x x	x	Intentional API change
34	mesa	wine	good	7c8c90c4 2012-09-19 16:37	0c67fe5d 2012-09-19 15:21	f51d232e 2012-09-19 16:07	?	Accidental push of bad commit
35	mesa	wine	bad	x 2012-09-25 21:08	x x	x x	x	Out of disk space
36	mesa	mesa	neutral	59c4420f 2012-10-15 19:25	befd51f8 2012-10-15 18:53	df3721fd 2012-10-15 19:17	?	Compile error

In total 5168 reports were generated, 511 of which indicated a build or test failure. The difference in the number of failure reports and number of regressions listed in table 7.2 occurs because table 7.2 lists a regression only once, even if it persisted for more than one test run, and because one failure email was generated for each involved project. E.g. table entry 27 generated one mail for the test failure in Ogre, mail for the negative feedback sent to Wine and one mail for the negative feedback sent to Mesa.

Mesa and Wine Results

CCIP performed fairly well with Mesa and Wine. There were a total of 14 regressions in Mesa, 8 of which were found by Mesa itself, and 6 which were discovered by Wine (1, 2, 7, 8, 9, 11). The remaining regressions were compile errors that slipped into the Mesa code, were caught by classic continuous integration and fixed after a short time (5, 12-15, 19, 25, 26).

Wine triggered three reports marked as bad, 6, 29 and 30. 6 was a random failure that could not be reproduced. 29 were random test delays that could not be clearly isolated and was fixed by an unidentified update. 30 is a setup issue and in a way a follow-up problem to 29: Test runs were not properly isolated, and if a test crashed or was still running, an unexpected screen setup caused the following tests to fail.

Irrlicht

Irrlicht identified some Mesa regressions also found by Wine (9, 11), but did not report any other regressions.

Ogre 3D

Ogre 3D flagged two Mesa updates as erroneous (27, 28). These were false positives caused by correct precision changes.

Ogre compares the test rendering to a reference rendering and tolerates a certain number of pixels with different colors. However, it counts a pixel as different even on the slightest color change.

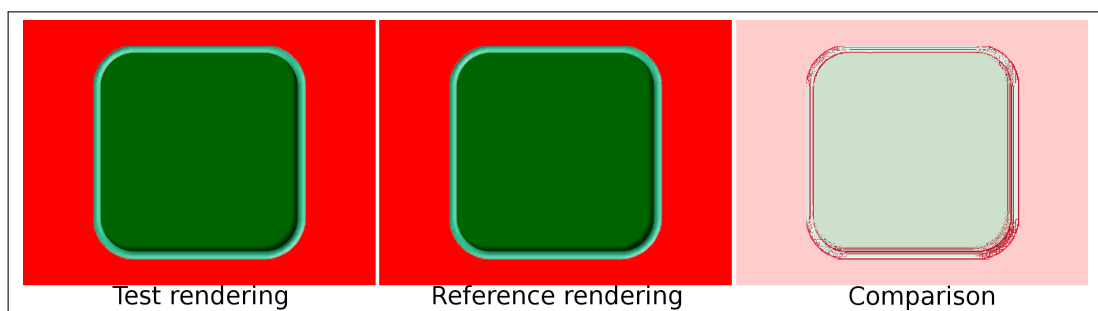


Figure 7.8: Ogre 3D false positive

Figure 7.8 shows the test rendering, the reference image and the comparison. Visually, the test and reference images are identical. A close investigation shows that some pixels in the light green border are off by one, e.g. one pixel has the color 68/157/128 in the test rendering and 67/156/128 in the reference rendering (the numbers are red/green/blue, ranging from 0 to 255). These pixels are highlighted in dark red in the image diff to the right.

It is worth noting that the second patch (28) is a partial revert of the first (27). The reason for this is outlined in the bug report it fixes²⁹. However, both reports are considered false positives because the reason for the test failure was unrelated to the cause of the bug 28 fixes.

OpenCIT and OpenEngSB

The attempt to use OpenCIT with the development version of OpenEngSB failed because of API changes (31). The build server was set up to use the stable development branch instead. This worked, but no development was done on this branch and hence no test data was produced. Using the development versions of Drools and Wicket with OpenEngSB produced similar results (32, 33).

During later testing of the OpenEngSB development branch with Karaf and Aries, one test failure in OpenEngSB was spotted (21). The initial build of OpenEngSB with the development branch of Karaf failed because of bugs in Karaf (10). These bugs were reported to Karaf developers and fixed. A later change to OpenEngSB broke the build setup, which resulted in a false positive error report being sent to the Karaf project (4).

Karaf

Aside of the initial fault in Karaf found by OpenEngSB (10), one regression in the Karaf code occurred and was found by the Karaf build system (18).

Aries

Aries had a number of build errors which were found by the Aries build and test system. One of them (20) is a reproducible test failure. 23 was a build failure caused by a JDK update. The same failure was also spotted by OpenEngSB.

A number of errors (16, 17, 22) were caused by incorrect internal linking and Maven repository update delays. Specifically, there are a number of subprojects in the source code where a subproject Foo-API declares a programming interface, and Foo-Impl implements said interface. On a number of occasions, Foo-API's version was bumped from 1.0-SNAPSHOT to 1.0, and immediately afterwards to 1.0.1-SNAPSHOT. Foo-Impl originally imported Foo-API v1.0-SNAPSHOT and was changed to import Foo-API v1.0. When building the code after all those changes, Foo-API v1.0.1-SNAPSHOT was built, but Foo-API v1.0 was not available. The Aries developers uploaded prebuilt packages to the Maven repositories, so those code revisions started building after a delay of a few days and the failures cannot be reproduced any more. Investigations whether the Aries developers intended this behavior were inconclusive.

²⁹https://bugs.freedesktop.org/show_bug.cgi?id=54877

At or shortly before commit 546f50f1, the Aries tests started failing randomly (24). Using the same code, the test executions sometimes succeeded and sometimes failed.

JLine

No error reports were generated for JLine. No major development work was done on JLine during the test period. Three releases were published and tested, JLine 2.7, JLine 2.8 and JLine 2.9.

Discussion

This chapter discusses the findings from the evaluation and provides answers to the research issues defined in chapter 5. It discusses how well CCIP met the expectations.

8.1 Findings

A number of findings can be deduced from the data gathered during the evaluation:

1. The QA measures of all tested projects are effective in preventing compile errors and test failures from entering the codebase.

The number of commits causing regressions remained below 1.5% in all tested projects. Especially outstanding in this regard is Wine, which had the 2nd highest number of changes and no regression of its own.

2. Regressions affecting dependent projects were committed to Mesa and Karaf and not detected by their tests, but their number is small.

This confirms the observations that motivated the thesis topic, but it puts into question if the effort needed for an automated testing approach is warranted.

3. Intentional API changes render CCIP ineffective.

An intentional API change will result in a build failure and negative feedback, which just reports the obvious fact that the dependent is no longer compatible with the changed API. From this point onwards, real regressions are hidden by the existing build failure. To fix this, the dependent's developers have to adjust their code to fit the new API. There are good reasons not to do this immediately, like the possibility that the API is changed a couple of times before the developers are happy with the changes, or the desire to remain compatible with the current stable release of the dependency.

4. Despite the best efforts to prevent them, there was a considerable number of false positives compared to the number of legitimate errors.
5. CCIP performed considerably better for Mesa and Wine than for the other projects.
6. Contrary to the expectations, only fairly simple errors have been found.
All regressions found by CCIP appear to be results of developer sloppiness. They were found swiftly by users, manual testers or the developers themselves. None of the regressions were in danger of being shipped within a stable release.
7. Not a single valid regression was discovered by a second-level dependent.

It is unclear if finding 1 is a result of good QA or insufficient test coverage. The low rate of testsuite breakages can occur because a low number of regressions occurred, or because a low number of regressions was detected by the tests. To separate these possibilities, a comparison between regressions reported by automated testing and regressions reported by users is needed. Unfortunately, none of the tested projects except Wine offer a way to systematically search the bug tracker for reported regressions [36].

A search for Wine bugs¹ filed between June 1st 2012 and October 17th 2012 marked with the “regression” keyword and concerning the component “directx-d3d” confirms that no bugs related to the Ogre engine or any game that uses it have been filed during the time the evaluation was active. This indicates that Wine’s QA successfully prevented regressions affecting Ogre from entering the codebase, but this is just one data point and cannot be generalized.

Finding 4 presents a big challenge. The overall rate of false positives and random errors is reasonably small, only 0.44% of all builds produced a false warning. There is room for improvement, but reducing this number becomes increasingly difficult the closer it gets to zero. However, because the number of legitimate errors found by CCIP, but not by CI, is small as well, the false positive rate has to be lowered considerably for CCIP to be a useful addition.

The evaluation offers no explanation for finding 5. One speculative reason why the Mesa-Wine duo is different may be that Mesa employs the weakest testing regime among the evaluated projects, while Wine’s testing is arguably the best. As table 7.1 shows, Wine had second highest number of changes and not a single test failure was caused by them. The constellation of a high-quality project using a comparably low quality project may seem strange. However, Wine does not depend on Mesa per se. It depends on having any OpenGL implementation available, and Mesa is just one out of many OpenGL implementations. Other common OpenGL implementations are the proprietary drivers from NVidia² and AMD³ or Apple⁴. Unfortunately, the source code of these drivers is not publicly available and no development snapshots are published, rendering them useless for this work. A hypothesis to test in future work is that CCIP works better for projects implementing a standardized, backward compatible and widely accepted programming interface than for projects introducing their own API.

¹<http://bugs.winehq.org/>

²<http://www.nvidia.com/object/unix.html>

³http://support.amd.com/us/gpudownload/linux/Pages/radeon_linux.aspx

⁴<https://developer.apple.com/devcenter/mac/resources/opengl/>

8.2 Answers to Research Issues

This section provides answers to the research issues defined in chapter 5.

RI-1: Costs and Benefits

Research issue 1 asks how costs and benefits of CCIP can be measured and how they are distributed between the dependency and dependent.

The implementation costs (RI-1.1) of CCIP are small: adding the prototype CCIP features to OpenCIT took about 160 hours of work, including some preparatory work, implementing the additional connectors and testing. This is a one-time effort for each CI server CCIP support is implemented for. To make the implementation solid enough to be used by everyday users, more work is required, but it remains a one-time effort.

Setting up a project dependency relation to migrate from CI to CCIP (RI-1.2) requires little effort in the prototype and is a matter of minutes. Handling low quality feedback is the biggest cost concern (see section 8.2).

The cost distribution question (RI-1.3) is answered as follows: The computational costs (increased CI server load) affect the dependent project because it performs additional build and test runs. The developer effort required to analyze generated reports is on whoever performs this task. Presumably this is the dependency, since they introduced the change. The immediate benefit is on the dependency's side as well, but the dependent is expected to profit as well from improvements to its dependencies.

The main benefit (RI-1.4) is discovering regression faster, and potentially discovering additional regressions. However, due to findings 1 and 4, the benefits of CCIP were limited. All tested projects are very good at avoiding regressions. In all projects, less than 1.5% of changes caused a regression. Furthermore, the majority of those regressions were caught by classic CI, not CCIP.

RI-2: Feedback Quality

Research issue 2 concerns the quality of feedback, how to handle low quality feedback and how to improve quality.

At the current state of the CCIP implementation, manual effort is required to analyze negative feedback and extract useful information from it (RI-2.1). A specific example are the steps described in section 7.2. Analyzing a regression report took an average of 30 minutes of manual work, which makes this task the most labor-intensive aspect of using CCIP. The obvious concern is that any false positive reports waste the developer's time and render CCIP useless.

The evaluation did bring up a large amount of false positive or otherwise useless reports (Figure 8.1). Which ratio of good and bad reports is acceptable cannot be answered in general, but the closer it is to zero bad reports, the better CCIP performs.

The dependent project can take a few steps to improve feedback quality (RI-2.2):

- Eliminate random test failures.

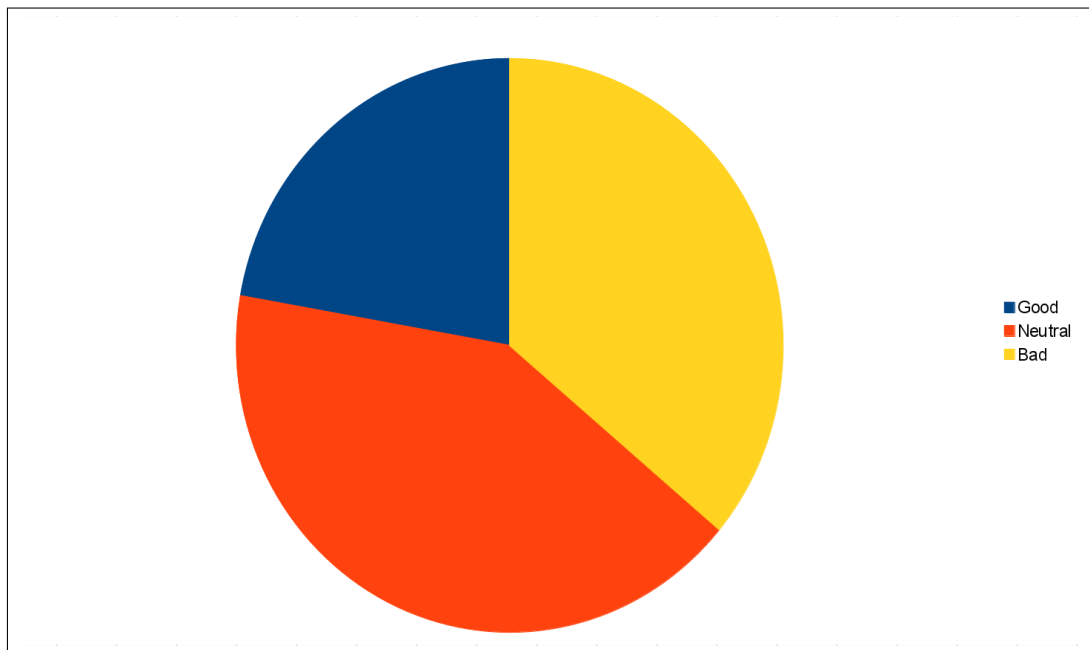


Figure 8.1: Ratio of report types

If the tests are run repeatedly without any source code or parameter changes, they should always produce the same result.

- Descriptive error messages.

Feedback is easier to understand if the error messages communicate what went wrong without requiring knowledge of the test's source code.

- Log calls to dependency libraries.

If dependency developers see which functions of their code were called prior to the test failure, they will be able to localize the problem faster. This step can be as simple as turning on the dependencies' debugging facilities.

- Document the test environment.

E.g. state which operating system was used, even if the dependency uses a platform independent development environment like Java. Which information makes sense depends on the project. For Mesa, knowing the graphics card used during the test is an important piece of information, while it barely matters for Apache Karaf.

The impact of false positives on the cost-benefit analysis depends on the cost of a false positive. The CI guidelines warn against allowing test failures to stay in the codebase for too long (see section 2.6) because developers soon start to ignore the constant complaints generated by the automated systems. However, the main CI literature like [24] and [16] does not provide

any numbers regarding the acceptable ratio of false positives, and [42] does not discuss the topic either.

The need for high quality feedback from dependent projects puts into question how willing dependencies are to listen to feedback from projects they do not know about. Thus the expectation that CCIP is a useful tool to discover one's impact network is unwarranted. For the same reason, it is unlikely that big networks of CCIP servers will be formed.

RI-3: Scalability

During the evaluation, not a single valid regression was discovered by a second-level dependent (finding 7). This provides an answer to RI-3.2: Cascading notifications are of minor importance.

By extension, this provides an answer to RI-3.1: Load can be reduced in large-scale deployments by limiting update propagation to a depth of two or maybe even one. The individual projects are likely to have an influence on this, so it makes sense to let the CI server administrators control this. A depth limit also avoids endless recursions if there is a dependency cycle in the setup, either due to actual cyclic dependencies (section 3.4) or a setup mistake.

There are certainly examples of regressions that were discovered by transitive dependents [18] [40], but the evidence so far suggests that they are too rare to justify the effort required for an automated test setup to discover them.

Conclusion and Future Work

This chapter provides a short summary of the thesis and its central results and outlines further research and engineering topics.

9.1 Conclusion

This thesis presented the Continuous Change Impact Analysis Process, or CCIP, a concept for communicating continuous integration servers for improved testing of interdependent projects. An implementation concept and prototype implementation were presented. This prototype was used to evaluate the research concept to answer research issues concerning the advantages and costs of CCIP, feedback quality as well as challenges posed by large-scale deployments.

The evaluation confirmed that dependency updates introduce regressions into dependent projects. It also demonstrated that automatic testing can be used to detect these regressions, and that CCIP can detect such regressions faster than manual testing does.

However, the evaluation uncovered a number of challenges that hinder the adoption of CCIP. Despite catching some regressions, the overall regression to commit ratio of the tested projects was low, putting the need for additional testing into question. The main issues are changes to the API of a dependency, which prevent meaningful automated testing, and false positive error reports, which waste developer time and increase the cost of using CCIP.

9.2 Future Work

This section presents research questions that arose during the implementation and evaluation of CCIP, as well as additional features for the implementation that may improve its usefulness.

Research: Percentage of Bugs Found by CCIP

In the evaluation, dependent projects found a number of regressions in Mesa and Karaf, but it is not known how the number of found regressions compares to the actual number of introduced

regressions. The main challenge to answering this question is finding a way to determine the number of regressions introduced into the code. Regressions reported by users may be a good approximation, but as discussed in section 8.1, only Wine offers an automated way of finding such reports. For other projects, a systematic manual reading of all bugreports filed during the evaluation may yield useful results, assuming such a manual task is feasible.

This research topic may provide clues as to why the overall number of regressions found during the CCIP evaluation was small - whether that the number is a result of poor regression detection by the tests, or indeed a low number of regressions that is introduced in the code.

Research: Evaluate Other Libraries That Implement Stable Standards

The evaluation indicated that CCIP performed considerably better for Mesa, Irrlicht and Wine than it did with Karaf, Aries and OpenEngSB. The possible, but unconfirmed explanation given in the discussion was that Mesa implements an open standard with a stable API (OpenGL), and Wine is a mature project with a big testsuite that does not depend on Mesa per se, but can work with any OpenGL implementation.

If other projects with similar properties can be found and the evaluation repeated, then this will confirm or reject the explanation. Potential candidates are low-level components of the GNU/Linux system that have more than one implementation, for example the C runtime (GNU libc, uclibc) and the C compiler (gcc, llvm).

Research: Investigate Cultural Impact

Some developers may dislike the idea of sending their changes immediately to outside projects, where they will be tested and in a way criticized by automated testing systems. Research if developers harbor such concerns, and which influence they have on CCIP and automated testing in general.

Research: Automatic Dependency Extraction

The Maven build system knows the download location of all its dependencies, and in some cases even the type and address of the revision control system. Using this technique, a big network of interdependent projects can be automatically imported into one or more CCIP servers for an extended evaluation of CCIP. Together with support for automatic bisections, this would enable automated testing of CCIP with entire software ecosystems.

Research the impact of false positives

For a better estimation of the costs of CCIP, further knowledge about the impact of false positives is needed. Unfortunately, there seems to be no literature on this topic concerning regular continuous integration. A future investigation should start with the effect of false positives on regular continuous integration and later on extend the research to CCIP.

Stable and Experimental Branches

Some projects, for example OpenEngSB, employ a versioning scheme called *Semantic Versioning* [1]. The basic idea is to limit API changes to major releases (v1.0, v2.0, ...), and keep the API backwards compatible in minor releases (v1.1, v1.2, v2.1, ...). By keeping track of stable and unstable branches, CCIP implementations can provide automatic testing for stable branches of projects that do not consistently maintain a backwards compatible API.

This requires additional information about the branch an update belongs to in the update notification. The CCIP implementation also has to know for every dependency which branch is used to be able to react only to compatible updates.

Implement Automatic Bisection

As part of the CCIP evaluation, a manual git bisect was performed for each test failure to isolate the change that caused the problem. This task consumed the majority of manual work time, and it may be possible to automate it.

The simplest way is to send one update notification for each change that is committed to the revision control system. Currently, if a developer commits 10 changes at once, only one update notification is sent for a build that contains all 10 changes. By separating those changes, a change that causes a regression can be isolated in a simple way, but at an increased runtime cost. An alternative approach is a binary search performed after a regression is reported. The search would be controlled by the CCIP server of the dependency, which generates intermediate builds and passes them to the dependent's CCIP server for testing.

The goal of this is to present to the developer the exact change that caused the regression by the time the regression is reported.

Tools Used and Copyright Attribution

In addition to the sources of information listed in the bibliography and programs listed in chapter 7, the following tools have been used extensively in the writing of this thesis:

- The \LaTeX typesetting system.
- Microsoft Visio 2010¹ for figures 2.1, 2.2, 2.3, 5.1, 5.2, 5.3, 6.1, 7.1, 7.2 and 7.7. The cliparts and *Calibri* font² in these figures are therefore © Microsoft Corporation.
- The Eclipse IDE³ and the JBoss Drools⁴ workflow editor. Figures 2.4 and 6.2 are screenshots of the workflow editor.
- The Git⁵ revision control system and GitHub⁶ service for storing the source code of the prototype implementation.
- The Apache Subversion⁷ revision control system and server infrastructure provided by the Institute of Software Technology and Interactive Systems of the Vienna University of Technology for storing the thesis and related material.

¹<http://office.microsoft.com/en-us/visio/>

²<http://www.microsoft.com/typography/fonts/family.aspx?FID=287>

³<http://www.eclipse.org/>

⁴<http://www.jboss.org/drools/>

⁵<http://git-scm.com/>

⁶<https://github.com/>

⁷<http://subversion.apache.org/>

List of Figures

1.1	Dependencies of the Java Development Kit package in Gentoo Linux	2
2.1	CI process overview [16, Fig. 1.1]	6
2.2	A model of the software testing process [50, Fig. 8.3].	9
2.3	An outline of the CI workflow	11
2.4	Drools rendering of the OpenCIT workflow.	13
3.1	A visualization of the package dependencies on the test system.	18
4.1	A development history with massive use of branches and merges.	23
5.1	Limitations of classic testing and management techniques in an environment with multiple independently developed projects	28
5.2	Use cases of CCIP for developers	31
5.3	Solution overview	32
6.1	CCIP implementation overview [15]	36
6.2	CCIP workflow implemented in JBoss Drools.	42
7.1	OpenCIT, OpenEngSB and their dependency tree	46
7.2	OpenCIT, OpenEngSB and their dependency tree	47
7.3	Irrlicht tessellation tech-demo. Source: Irrlicht website ¹⁷	48
7.4	MS Visio, Trackmania Nations and Wine's registry editor on a GNU/Linux desktop	49
7.5	Ogre water rendering sample	51
7.6	Karaf component overview [21]	51
7.7	Regressions found	55
7.8	Ogre 3D false positive	60
8.1	Ratio of report types	66

Bibliography

- [1] OSGi Alliance. Semantic versioning. technical whitepaper. <http://www.osgi.org/wiki/uploads/Links/SemanticVersioning.pdf>, 2010.
- [2] OSGi Alliance. OSGi Service Platform, Core Specification, Release 4, Version 4.3. Technical report, OSGi Alliance, 2011.
- [3] Rick Anderson. The end of dll hell. *MSDN Magazine*, 2000.
- [4] Andrew W. Appel. Axiomatic bootstrapping: a guide for compiler hackers. *ACM Trans. Program. Lang. Syst.*, 16(6):1699–1718, November 1994.
- [5] Holger Bleich. Angelockt und abkassiert. *c’t Magazin*, 2009.
- [6] E. Carmel and R. Agarwal. Tactical approaches for alleviating distance in global software development. *Software, IEEE*, 18(2):22–29, mar/apr 2001.
- [7] Yoonsik Cheon and Gary Leavens. A simple and practical approach to unit testing: The jml and junit way. In Boris Magnusson, editor, *ECOOP 2002 — Object-Oriented Programming*, volume 2374 of *Lecture Notes in Computer Science*, pages 1789–1901. Springer Berlin / Heidelberg, 2006.
- [8] Microsoft Corporation. About side-by-side assemblies. <http://msdn.microsoft.com/de-de/library/ff951640.aspx>, 2012.
- [9] Microsoft Corporation. Windows resource protection. [http://msdn.microsoft.com/en-us/library/windows/desktop/cc185681\(v=vs.85\).aspx](http://msdn.microsoft.com/en-us/library/windows/desktop/cc185681(v=vs.85).aspx), 2012.
- [10] Bernard Coulange. *Software Reuse*. Springer London, 1998.
- [11] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare, editors. *Structured programming*. Academic Press Ltd., London, UK, UK, 1972.
- [12] Miguel de Icaza. What killed the linux desktop. <http://tiranias.org/blog/archive/2012/Aug-29.html>, 2012.

- [13] Cleidson R. de Souza, Stephen Quirk, Erik Trainer, and David F. Redmiles. Supporting collaborative software development through the visualization of socio-technical dependencies. In *Proc. of the 2007 int. ACM conf. on Supporting group work*, GROUP '07, pages 147–156, New York, NY, USA, 2007. ACM.
- [14] Cleidson R. B. de Souza and David F. Redmiles. An empirical study of software developers' management of dependencies and changes. In *Proc. of the 30th int. conf. on Software engineering*, ICSE '08, pages 241–250, New York, NY, USA, 2008. ACM.
- [15] Stefan Dösinger, Richard Mordinyi, and Stefan Biffl. Communicating continuous integration servers for increasing effectiveness of automated testing. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ASE 2012, pages 374–377, New York, NY, USA, 2012. ACM.
- [16] P. Duvall, S. Matyas, and A. Glover. *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley, 2007.
- [17] Eelco Visser Eelco Dolstra. The nix build farm: A declarative approach to continuous integration. In *Int. Workshop on Advanced Software Development Tools and Techniques(WASDeTT)*, 2008.
- [18] Laurent Carlier et all. Assertion 'llvmoffsetofelement' when running furmark with wine. https://bugs.freedesktop.org/show_bug.cgi?id=44466, 2012.
- [19] Free Software Foundation. *automake*. http://www.gnu.org/savannah-checkouts/gnu/automake/manual/html_node, 2012.
- [20] Mozilla Foundation. Mozilla trademark policy. <http://www.mozilla.org/foundation/trademarks/policy.html>, 2009.
- [21] The Apache Software Foundation. Karaf. <http://karaf.apache.org/>, 2011.
- [22] The Apache Software Foundation. A guide to proposal creation. <http://incubator.apache.org/guides/proposal.html>, 2012.
- [23] X.Org Foundation. Documentation for the X Window System Version 11 Release 7.5 (X11R7.5). <http://www.x.org/releases/X11R7.5/doc/>, 2009.
- [24] Martin Fowler. Continuous integration. <http://martinfowler.com/articles/continuousIntegration.html>, 2006.
- [25] Inc Free Software Foundation. Gnu lesser general public license, version 2.1. <http://www.gnu.org/licenses/lgpl-2.1.html>, 1999.
- [26] Inc. Gentoo Foundation. Gentoo bootstrapping guide. <http://www.gentoo.org/doc/en/draft/bootstrapping-guide.xml>, 2012.

- [27] Steven Gianvecchio, Mengjun Xie, Zhenyu Wu, and Haining Wang. Measurement and classification of humans and bots in internet chat. In *Proceedings of the 17th conference on Security symposium*, SS'08, pages 155–169, Berkeley, CA, USA, 2008. USENIX Association.
- [28] Robert A. Gingell, Meng Lee, Xuong T. Dang, and Mary S. Weeks. Shared libraries in sunos. In *USENIX Association Conference Proceedings*, 1987.
- [29] T.J. Halloran and William L. Scherlis. High quality and open source software practices. Technical report, Carnegie Mellon University, 2002.
- [30] ISO, Geneva, Switzerland. Information technology – Programming languages – C, 2011.
- [31] ISO, Geneva, Switzerland. Information technology – Programming languages – C++, 2011.
- [32] Paul C. Jorgensen and Carl Erickson. Object-oriented integration testing. *Commun. ACM*, 37(9):30–38, September 1994.
- [33] Poul-Henning Kamp. A generation lost in the bazaar. *Queue*, 10(8):20:20–20:23, August 2012.
- [34] J. Loeliger and M. McCullough. *Version Control with Git: Powerful tools and techniques for collaborative software development*. Version Control With Git. O'Reilly Media, Incorporated, 2012.
- [35] M. Douglas McIlroy. Mass-produced software components. In *Proc. NATO Conference on Software Engineering*, 1968.
- [36] Richard Mordinyi and Richard Hirner. Relations between commits and bug reports in selected open-source software, technical report m1-tr-2012.1.16. Technical report, Vienna University of Technology, 2012.
- [37] Alexander Neundorff. Why the kde project switched to cmake – and how. *Linux Weekly News*, 2006.
- [38] Tim O'Brien, Manfred Moser, John Casey, Brian Fox, Jason Van Zyl, and Eric Redmond-Larry Shatzer. *Maven: The Complete Reference*. Sonatype, Inc, 2010.
- [39] Andreas Pieber. Flexible engineering environment integration for (software+) development teams. Master's thesis, Vienna University of Technology, 2011.
- [40] Andreas Pieber and Felix Mayerhuber. activemq web console fails. <http://issues.openengsb.org/jira/browse/OPENENGSB-1956>, 2012.
- [41] A. Podgurski and L.A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *Software Engineering, IEEE Transactions on*, 16(9):965–979, sep 1990.

- [42] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V. Mantyla. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *Automation of Software Test (AST), 2012 7th International Workshop on*, pages 36–42, june 2012.
- [43] Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12:23–49, 1999. 10.1007/s12130-999-1026-0.
- [44] Eelco Dolstra Sander van der Burg. Disnix: A toolset for distributed deployment. Technical report, Delft University of Technology, The Netherlands, 2010.
- [45] Mark Segal and Kurt Akeley. OpenGL 4.3 Core Profile Specification. <http://www.opengl.org/registry/doc/glspec43.core.20120806.pdf>, 2011.
- [46] Bikram Sengupta, Satish Chandra, and Vibha Sinha. A research agenda for distributed software development. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 731–740, New York, NY, USA, 2006. ACM.
- [47] H.M. Sneed. A cost model for software maintenance evolution. In *Software Maintenance, 2004. Proc. 20th IEEE Int. Conf. on*, pages 264 – 273, sept. 2004.
- [48] Inc Software in the Public Interest. Debian’s trademark. trademark licensing policy. <http://www.debian.org/trademark>, 2012.
- [49] Inc Software in the Public Interest. The document foundation mark policy. http://wiki.documentfoundation.org/TDF/Policies/TradeMark_Policy, 2012.
- [50] Ian Sommerville. *Software Engineering*. Addison-Wesley Longman, feb. 2010.
- [51] Kent R. Spillner. Java build tools: Ant vs. maven. <http://kent.spillner.org/blog/work/2009/11/14/java-build-tools.html>, 2009.
- [52] William Stallings. *Operating Systems*. Pearson International Edition, 2005.
- [53] Andrew S. Tanenbaum and Maarten Van Steen. *Distributed Systems*. Pearson Prentice Hall, second edition, 2007.
- [54] Open Engineering Service Bus Development Team and Contributors. *OpenEngSB Manual Version 2.0.2*. <http://openengsb.org/manual/openengsb-manual/v2.0.2/html-single/openengsb-manual.html>, 2012.
- [55] Walter F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th international conference on Software engineering*, ICSE '82, pages 58–67, Los Alamitos, CA, USA, 1982. IEEE Computer Society Press.
- [56] Brian Warner. *BuildBot Manual - 0.8.1*. <http://buildbot.net/buildbot/docs/0.8.1/>, 2010.
- [57] James A. Whittaker. *How to Break Software*. Addison Wesley, 2002.