UPTEC IT08 014
Examensarbete 30 hp
November 2008

Introducing Continuous Integration in an Enterprise Scale Environment

Staffan Persson

This page intentionally left blank.

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Introducing Continuous Integration in an Enterprise Scale Environment

Staffan Persson

This thesis investigates how continuous integration best should be incorporated into a current development environment in order to acquire a shorter time for the regression testing feedback loop. The product in target is a high availability enterprise scale software product within mobile telecom. It utilizes a large commercial relational database for customer associated data and employs approximately one hundred designers.

A continuous integration system needs automatic database integration, compilation, testing, inspection, deployment, and feedback. Measures that are valuable to the build process regardless of build time should be given the highest priorities of implementation. Feedback has been developed in the form of Continuous Integration Web. The first priority at this stage is to introduce unit testing of the database which also has to be automatically built whenever updated or changed.

The amount of time used to integrate should be as close to ten minutes as possible. This goal is reachable if the build process is split into at least two integrations by differentiating unit tests from component tests and, if the software's functionally divided parts are built separately on local integration servers.

Starting integrations automatically could potentially allow broken source code to reside in the source code repository. This problem can however be mitigated with the aid of good routines and sufficient communication between developers and the continuous integration server. If the latter is possible it is recommended to make use of a continuous integration server. Otherwise it is recommended to perform integrations manually.

Handledare: Euler Jiang
Ämnesgranskare: Roland Bol
Examinator: Anders Jansson
ISSN: 1401-5749, UPTEC IT08 014
Tryckt av: Reprocentralen ITC

This page intentionally left blank.

Sammanfattning

Kontinuerlig integrering (Continuous Integration, CI) används för att på ett tidigt stadium hitta problem som uppkommer när källkod från flera utvecklare integreras. Det är en del av en systemutvecklingsmetodik (Extreme Programming, XP) som är kvick och bygger på återkoppling, antagande om enkelhet och god attityd till förändring. Regression Testing hittar problem vilka uppkommer som en konsekvens av att tidigare fel åtgärdats eller ny funktionalitet implementerats. Detta är möjligt genom att även tidigare testfall körs när källkod har ändrats. Produkten som studeras är ett High Availability system inom mobil telekommunikation i Enterprise-skala. Regression Testing används för produkten och det finns en målsättning att introducera kontinuerlig integrering för att erhålla en kortare återkopplingsloop.

För att realisera kontinuerlig integrering måste en rad kriterier uppfyllas och den komplikation produktens storskalighet innebär beaktas. Målet för denna studie är att analysera hur kontinuerlig integrering bäst bör integreras i den nuvarande utvecklingsmiljön.

En litteraturstudie som utförts inom ämnet har bland annat visat att ett kontinuerligt integreringssystem behöver automatisk databasintegrering, kompilering, testning, inspektion, installation och återkoppling för att kunna utnyttjas fullt ut. Återkoppling har visat sig vara en av de viktigaste delarna eftersom den binder samman de övriga genom att kommunicera det totala resultatet till utvecklaren. Behovet av återkoppling har prioriterats och också tillgodosetts genom utveckling av verktyget Continuous Integration Web. Continuous Integration Web visar produktens byggprocess från kompilering till funktionell testning i realtid.

Statistik framtagen för den nuvarande byggprocessen visar att kompilering samt enhets- och komponenttestning tillsammans behöver 171 minuter i medeltal för att slutföras. Den tid som en byggprocess bör ta i anspråk rekommenderas inom XP till *10 minuter*. För att nå målet med den kortare byggtiden föreslås att byggprocessen, med de delar ett kontinuerligt integreringssystem består av, delas upp i minst två integreringscykler. Den första integreringscykeln tidsmässiga mål är att vara så nära 10 minuter som möjligt och ska alltid först fullbordas utan fel på utvecklarens privata arbetsstation innan källkoden kan vidarebefordras till en central första integreringscykel.

Den andra integreringscykeln ska startas när den centrala första integreringscykeln fullbordats utan fel. Om den andra integreringscykeln tar för lång tid att fullbordas kan en tredje integreringscykeln skapas. Den tredje integreringscykeln kan, om den utformas korrekt, utföras parallellt med den andra. Alla cykler bör utarbetas för att så snabbt som möjligt hitta eventuella fel.

Eftersom den enda testning som utförs i den första integreringscykeln består av enhetstester är det av högsta vikt att dessa tester tillsammans med övriga delar av den första integreringscykeln inger tillräckligt förtroende för att källkoden ska kunna vidarebefordras till en centrala första integreringscykeln. Den första integreringscykeln måste även kunna garantera att andra utvecklare kan basera sitt arbete på den byggda källkoden samt att den kan användas för den andra integreringscykeln.

Statistik framtagen för *enskilda komponenter* visar att kompilering samt enhets- och komponenttestning behöver i medeltal 33 minuter för att slutföras. Detta talar för att en uppdelning mellan enhets- och komponenttestning skulle göra det möjligt att komma nära rekommenderade 10 minuter om endast en av produktens funktionellt uppdelade komponenter byggs samtidigt på *en* integreringsserver. När enhets- och komponenttestning delas upp bör även imiterade objekt användas till enhetstester för att göra dessa snabbare. Samtidigt bör också stumpar användas för att tranformera komponenttester till enhetstester om det behövs för att första integreringscykeln ska inge tillräckligt förtroende.

Innan en övergång till kontinuerlig integrering kan implementera måste alla nödvändiga åtgärder prioriteras. Att realisera kontinuerlig integrering utan att först åtgärda undermåliga eller saknade delar i den nuvarande konfigurationen skulle göra värdefull tid avsatt för integrering omotiverat ineffektiv.

Under hela transformeringen måste feedbackverktyget hållas uppdaterat för att alltid kunna kommunicera output från introducerade verktyg. Detta eftersom en integrering inte är till gagn om dess resultat inte kan förmedlas till berörda utvecklare på ett bra sätt. Vidare bör åtgärder som är värdefulla för byggprocessen oavsett byggtid prioriteras högst.

I nuläget har databasintegrering högst prioritet. Eftersom databasen är en del av produkten bör den följaktligen behandlas på samma sätt som källkod genom att skapas automatiskt och enhetstestas vid varje berörd integrering. Differentieringen av enhets- och komponenttester bör utföras sist, eftersom den inte är till nytta innan en uppdelning av de olika integreringscyklerna samt konfigurering av lokala integreringsservrar har skett.

Kontinuerlig integrering startas vanligen automatiskt genom en kontinuerlig integreringsserver men kan även startas manuellt. En kontinuerlig integreringsservers huvudsakliga uppgift jämfört med nuvarande funktionalitet uppbyggt av skript är att kontinuerligt starta integreringar baserat på ändringar i ett centralt system för versionshantering. Komplexiteten i att utveckla en kontinuerlig integreringsserver tillsammans med flexibiliteten hos redan tillgängliga enheter gör det emellertid alltid fördelaktigt att konfigurera och använda en existerande server.

När en integrering startas manuellt görs det av en utvecklare som har intentionen att integrera och därmed är intresserad av resultatet. Detta kan däremot inte garanteras om integreringen startas automatiskt. Emellertid garanterar en kontinuerlig integreringsserver att en integrering alltid startas när den behövs. Det huvudsakliga problem som kan uppstå om resultatet av en integrering inte beaktas är att trasig källkod kan residera i systemet för versionshantering. Trasig källkod i ett system för versionshantering kan dels resultera i att utvecklare inte kan utföra den första integreringscykeln på sin arbetsstation samt i värsta fall att den utförs med trasig källkod.

Ingen kontinuerlig integreringsserver har idag ett bra stöd för att hålla trasig källkod borta från ett system för versionshantering. Införs bra rutiner för att omhänderta automatiska integreringars resultat rekommenderas användandet av en kontinuerlig integreringsserver. I annat fall rekommenderas att integreringar utförs manuellt.

This page intentionally left blank.

Contents

Figures

This page intentionally left blank.

1 Introduction

1.1 Background

The product in target of this thesis is a High Availability enterprise scale software product within mobile telecom. It utilizes a large commercial relational database for customer associated data and employs approximately one hundred designers. Regression testing [1] uncovers problems in software that occur when other errors are fixed or when new functionality is implemented. The development process of the product incorporates regression testing. A goal has been set up to start utilizing continuous integration in order to acquire a shorter time for the feedback loop.

Continuous Integration is one of Extreme Programming's primary practices [2]. In Martin Fowler's highly references article [3] on the subject it is described as:
"…a software development practice where members of a team integrate their work frequently, usually each person integrates as least daily – leading to multiple integrations per day. Each integration is verified by an automated build (including test) to detect integration errors as quickly as possible."
The amount of time used to integrate should according to [2] be as close to ten minutes as possible.

Practices stated by [4] include the importance of all members using a central source code repository as well as the need of the build being an automated process. Additionally, [5] states the significance of developers running private builds and do not commit source code which failed local compilation tests to the source code repository. Getting broken code from the source code repository should be avoided and to fix broken builds should have the highest priority for the whole development team.

The product's build process currently consists of many steps from check in to system testing where some, to a certain degree, are automated and some are not. The relation between the different steps of the build process is to the most part not automated and to always make the next step in line to trigger automatically is being worked on. The current 3.3 Development

Environment is also in need of a global visualization for the complete build process from check in to system testing. This thesis is a part of the current project Automation Roadmap which introduces various agile methods as well as automates the build process.

1.2 Problem Statement

The thesis will investigate how Continuous Integration best should be incorporated into the current development environment. This includes specifying parts of the development environment that needs to be changed or created as well as to prioritize between them. The thesis also aims to develop a solution visualizing the product's entire build process.

1.3 Limitations

Only parts of Continuous Integration that are particularly interesting to Continuous Integration on an enterprise scale and the product will be assessed. The reader is expected to be familiar with Extreme Programming and to have basic knowledge about its practices.

Focus will mainly be on the technical side of Continuous Integration as opposed to its more behavioral aspects. The actual deployment of Continuous Integration for the product will not be carried out.

1.4 Report Structure

The remainder of this report is structured as follows:

- Chapter 2 describes the research methodology of the thesis;
- Chapter 3 analyses the basic elements of Continuous Integration, how it should be applied into an Enterprise scale environment, the product's development environment, continuous integration servers, and the difference between manual and automatic integration;
- Chapter 4 explains the tools developed during the thesis;
- Chapter 5 discusses the most important findings from the study and presents conclusions along with recommendations.

2 Method

2.1 Literature Study

Concluding the best way to introduce Continuous Integration in the product's development cycle requires great amounts of knowledge about the current build process and the subject of Continuous Integration. Learning about Continuous Integration and how it usually is introduced was done by performing a literature study on the subject, 3.1 Continuous Integration. The aim of the literature study was to explain and clarify how all different part of the development cycle in a Continuous Integration environment should be constructed. It was also in conjunction with 3.2 Enterprise Scale to tell if it would be beneficial for software projects on an enterprise scale to make use of Continuous Integration or what was needed for it to become valuable asset.

Concluding the best way to introduce Continuous Integration in the product's development cycle was done by evaluating all parts of the development cycle based on the literature study along with the product's build process. Absent parts or parts lacking features were identified and given recommendations for improvements. Furthermore, priorities of the different recommendations and improvements have been discussed in order to reach a conclusion on which actions that are most beneficial to carry out first.

Another concern that became apparent for the introduction of Continuous Integration was if a Continuous Integration server should be used and if the current build scripts could be altered to support Continuous Integration with a Continuous Integration server. The option to make use of a Continuous Integration server was evaluated by mapping out the current build processes, noting its strengths and weaknesses, and by analyzing how complex the alteration of it would be compared to the possible benefits of incorporating a Continuous Integration server. These two options were also compared to the complexity, benefits, and disadvantages of replacing the current build scripts with another build tool.

2.2 Prototyping

A feedback tool is one of the key components in a Continuous Integration system. It visualizes the build process and thereby also requires gaining knowledge about all its parts. Learning what was in need of changes or had to be created was achieved by prototyping a feedback tool.

It was evaluated whether to use an evolutionary or extreme prototyping method. Since extreme prototyping is commonly used within development of web applications it proved to be a good candidate. Extreme prototyping consists of three steps where the first two steps consist of a static web pages followed by a working user interface but simulated service layer. When this was weigh against evolutionary prototyping it proved to be less beneficial since the software easily could be divided into evolutionary steps and that continuously new usable functionalities were prioritized higher than a system that only would provide usable functionality in the end of the development process. Another factor was that the system could evolve in its future environment and that a good communication had been established with the indented users of the system.

As a major part in evolutionary prototyping is to make full use of requirements engineering, Appendix A Continuous Integration Web Software Requirement Specification was created. The software requirement specification was also updated and maintained according to new requirements throughout the development process.

3 Analysis

3.1 Continuous Integration

According to [5], the major and most common components of a Continuous Integration system are automatic database integration, compilation, testing, inspection, deployment, and feedback. The following sections will describe these components in more detail.

3.1.1 Database Integration

The key concept of continuous database integration is that the database used by the software solution is recreated with all its test data at each integration occasion [5]. This requires that it is possible to create the database as well as to manipulate it, by adding test data automatically. Testing and inspection of the created database is also a part of integrating the database.

The database should, in a Continuous Integration perspective, be treated exactly like source code. Changes to the database should thereby result in a new integration. Furthermore, [5] [6] also both agree that each developer should have their own local database, not affecting other developers with untested database changes which is likely to happen if the database is used centrally.

3.1.2 Compilation

Automatically compiling source code is, as stated in [5], a very basic part of a Continuous Integration system and is usually carried out with the aid of a build tool such as Ant™ or Maven™.

3.1.3 Testing

Automatic testing is a very important part of Continuous Integration. Moreover, [5] even argues that Continuous Integration without any automated tests is not to be considered Continuous Integration. Human testing with a focus on usability is however still considered necessary and cannot be neglected since the software will be used by humans and thereby still needs to be tested by humans.

The tests can be divided into four subgroups [5] [1]. A unit test challenges the smallest testable part of a software system and has no dependencies. A Component test[1] analyzes the software system on a higher level than a unit test and utilizes internal or external dependencies. The component test is however different from a system test and a functional test[2] since the latter two will test the entire software system and thereby requires it to be fully deployed and installed. The functional test differs from the system test by testing it from the perspective of a client instead of testing the software system it self.

3.1.4 Inspection

Automatic inspection evaluates low-level details such as coding standards, design metrics, complexity, and code duplication. It is cheap to run and can thereby also be run continuously in order to faster discover problem areas. In addition to this, [5] also states that continuous inspection will shorten the time it takes from the moment a problem has been identified to its resolution. This is because more and more knowledge of the source code and its design will be lost during the private build before a problem is discovered and can be dealt with.

Even if automatic inspection is a great asset in the daily work, it is obvious that it cannot be a substitute for manual inspection [7]. Instead, [5] [7] both agree that using continuous automatic inspection allows manual inspection to focus on higher level parts such as if maintaining the source code ultimately is going to be unproblematic, design issues, and other requirements. This will make the manual inspection more efficient.

3.1.5 Deployment

Deployment of a software system is the last piece in the Continuous Integration chain. This is where packaging of all software components takes place and the release ready for higher level testing is created. According to [3] [8], this should also make it easy to do automatic deployment not only for higher level testing but also to the end user since many of the existing procedures can be used. Moreover, [8] even argues that not automating the deployment is a waste of time and that it can have a huge impact on the ability to provide new features to the end user more frequently.

[1] Component tests can also be referred to as subsystem tests and integration tests
[2] Functional tests can also be referred to as acceptance tests

If a Continuous Integration system should be able to release good working software to the *end user* at any place and at any time. In order to accomplish this, [5] declares the following necessary steps:

1. *Labeling repository assets*
2. *Clean environment*
3. *Labeling builds*
4. *Successfully run tests at all levels in the clean environment*
5. *Build feedback reports*
6. *Possibility to roll back by using labels*

These steps will assure that the product is fully functional when deployed in the end user's environment. Labeling repository assets means both that many files can be grouped together as well as that these labeled groups can be tracked via history; just like files in a repository normally can be. It is then possible to backtrack if there is a problem with a build. Labeling the actual binary files that were produced during the build of the labeled group of source code files is to label a build. Producing a clean environment is to make sure that nothing in the environment is assumed to be present upon deployment. Everything needed for deployment should be removed and put back into the environment automatically. This includes both operating system as well as other software required by the product.

All previously discussed quality measures should also be run in this clean environment even though they already have been run by the build machine in the production environment. This is to eliminate any possible environmental issues.

Feedback reports should contain information about defects addressed as well as features implemented and changes, supplied by the version control system, to source files used in the build. Finally, the roll back capability comes in handy when a problem occurs and a roll back has to be made in order to produce a working release after a malfunctioning change or addition to the software's source code.

Rollback using two different forms of backtracking, simple backtracking and true backtracking, is proposed by [9]. When one or more components in a build fail due to its dependencies failing, simple backtracking will search backwards in history to find the latest successful build of that very component. The set of built components used by the found component will be used if and only if the two following preconditions are satisfied [9]:

1. *The requirements of the component with dependencies failing and the found component are the same*
2. *The component with dependencies failing does not have successfully built child components of another version than the found component*

The benefit of using simple backtracking is according to [9] that it almost eliminates failed builds as an obstacle to delivery of releases. True backtracking is still considered to be experimental and in need of further study [9].

3.1.6 Feedback

Both [5] [10] agree that continuous feedback is the key corner stone and most important part of a Continuous Integration system. This since the essence of Continuous Integration is to shorten the time from that a problem occurs until it has been solved. Continuous feedback in this scenario is obviously of highest importance as none of the other parts can be utilized to their full extent if feedback from them cannot be acquired fast enough.

All feedback is not necessary or even helpful for every one. The importance of providing the right information to the right people at the right time and in the right way is highly emphasized in [5]. The type of information that is useful to project managers, architects, developers, and testers differs.

Ways to send feedback discussed in [10] include text based tools such as RSS, e-mail and SMS as well as sound and different visual devices. X10 is a protocol that with certain hardware allows usage of almost any electrical device such as lava lamps or sirens. Another popular example shown in Figure 1: The Ambient Orb has functions such as flashing or glowing in different colors which is used to show different states of a build [11].

Figure 1: The Ambient Orb

3.2 Enterprise Scale

As discussed in the previous section, time is of essence when waiting for integration feedback. The key issue of utilizing Continuous Integration for software projects at enterprise scale such as the product is the rapidly increasing build time. A longer build time results in slower feedback from integrations. According to [12], this leads to longer time between integrations performed by the developers and thereby a loss to the benefits of practicing Continuous Integration.

It is of highest importance to fail builds as fast as possible [5]. The faster a build fails, the faster developers can attend to the cause of the problem. The practice of Continuous Integration can according to [12] be beneficial for enterprise scale software projects if build time is short enough.

Data has been collected using 4.2 Status Monitor during a period of 44 days from 15 different branches and 91 builds where more than zero components were built. The measured time for builds, compilation and the current setup of unit tests included, ranges form 14 to 769 minutes with a mean value of 171 minutes and a median value of 119 minutes. For further information, see Appendix B Build Data.

3.2.1 Splitting the Integration Cycle

The integration can be divided into multiple parts in order to shorten the time before necessary feedback integration can be made available to the developers. Splitting the integration can be done in many ways depending on different priorities. The integration cycles below are based on priorities from [12] [5]. In order to accomplish a split of the integration cycle it is however of highest importance to achieve distinction between unit tests and component tests [12].

A private build is done by the developer every time before committing new or changed source code to the source code repository [5]. This is done in order to limit the possibility that changes made will break the first integration cycle's build. The private build consists of getting the latest source code from the source code repository followed by running all steps of the first integration cycle. If the first integration cycle's build breaks it is likely that other developers become

affected. It could in the worst case lead to developers getting broken source code for use in their private builds or otherwise, to developers not being able carry out their private builds.

The first integration cycle is triggered when the changes from a successful private build are committed to the source code repository. It is carried out to guarantee that the source code in the repository not only builds without errors on one single workstation. A successful build in the first integration cycle must assure that the source code can be used by other developers as well as the second integration cycle. All parts of the first integration cycle should thereby be selected so that a balance between a fast enough and stable enough build is achieved [5]. The order of items in the cycle should reflect upon what usually fails first since failing builds fast always is of essence.

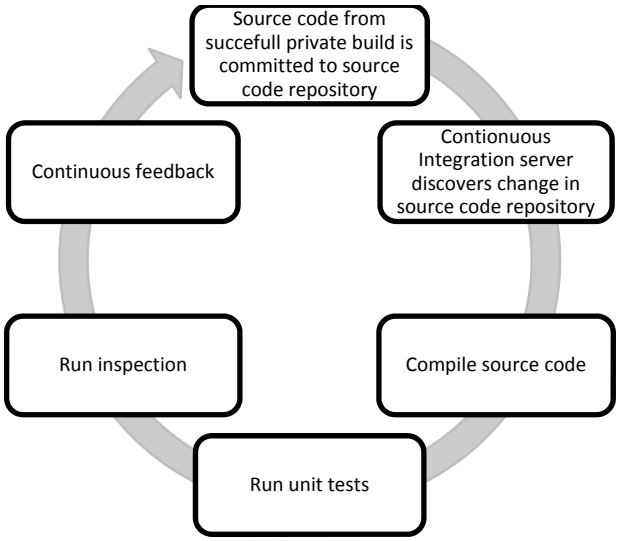

Figure 2: First Integration Cycle

The second integration cycle is usually run periodically and will generally require a much longer completion time than the first. It should also consist of more time consuming tasks with a lower feedback priority than those of the first integration cycle. If the second integration cycle includes too many tasks or is too slow, it should be considered to allow a third integration cycle to execute the most time consuming tasks. The third integration cycle could be constructed in such a way that it can run in parallel with the second [3].

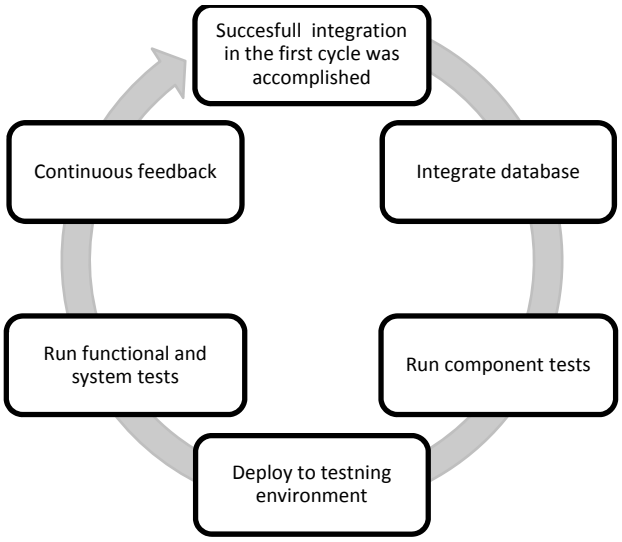

Figure 3: Second Integration Cycle

3.2.2 Compilation

Two solutions to shorten the time needed for compilation are proposed by [13]. The first solution is to make use of incremental compilation instead of always performing a full compilation of all components in the software system. The importance of still performing a full compile is however stressed. As later explained about testing, the compilation can also be split into the different integration cycles. The faster incremental compiling can be used in the first integration cycle while the slower full compile can be run less often by the second integration cycle. Furthermore, only components of the software system that are new or contain changes have to be compiled if the software system is component based [9].

Compilation is however according to [12] seldom a major reason for long build time in Extreme Programming projects. Instead it is stated that the normal cause of long build time is the increased number of tests performed during integration. Reorganization of the compilation might thereby not be necessary to meet targeted build time. Instead, it should be the first priority to achieve the targeted build time by focusing on the testing stage. A second solution proposed by [13] is to remove as many tasks as possible from the first integration cycle that are not linked to feedback.

3.2.3 Testing

Unit tests, being the fastest tests, should not only always be run at each integration occasion but also by the developer before source code is committed to the source code repository [3] [5] [14]. Sadly, unit tests cannot alone catch all problems within a software system. There is also a need for component tests, system tests, and functional tests. Component tests, system tests, and functional tests are all not only great assets when it comes to finding problems within the software on a higher level. They are also a great source of information to find new areas in need of unit testing [3].

The Extreme Programming guideline of a ten minutes build time might not be achieved if all tests have to be run for each integration occasion. A solution to the problem that the fast unit tests might fall short when it comes to finding bugs on a larger scale in the software system and that component test, system tests, and functional tests can be too slow to run at every integration is presented in [3] [13] [15]. The stated solution is to divide the tests into slower and faster categories. The fastest tests will be executed at each integration occasion on the original integration server while slower tests can be run by an additional machine. This will allow for fast feedback for the developers without losing the necessary component tests, system tests, and functional tests. The machines dedicated to performing slow tests can run with a cycle of a few hours while even slower test can run on nightly bases.

Stubs and mocks can shorten the build time even further [16]. Objects that are expensive to create and used in test scenarios where they are not the aim of the test should be substituted by mock objects. The mock object mimics the behavior of the real object but is less expensive to create. Stub objects should be created if tests relying on external dependencies are run. Stub objects can mimic the behavior of services by providing preconfigured answers to specific calls and will allow tests that rely on services to run much faster.

3.2.4 Local Integration Servers

Only using one integration server in a large software project with many developers will make the developers commit their code less frequently in order to break the build as seldom as possible [12]. Consequently, this will lead to more complicated integrations.

The suggested solution to this problem is to divide development into smaller teams where every team has their own integration server, hence allowing for more frequent committing of source code. Dividing development into smaller teams is however only possibly if the source code can be modularized in such a way that it does not lead to a complicated final integration later. The by [12] recommended way to modularized is by functionality and not by application layers, hence minimizing dependencies between teams.

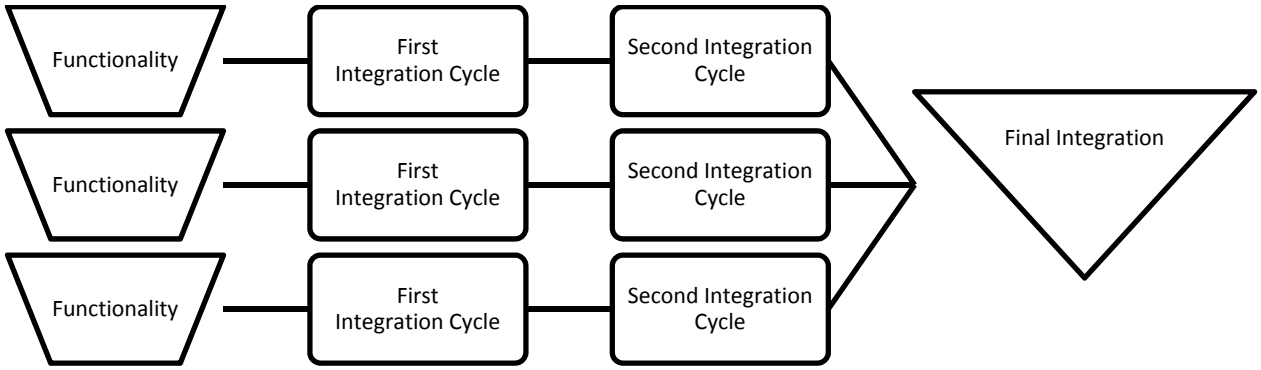

Figure 4: Local Integration Servers

The product's system is functionally divided into multiple components. Data from different parts of the build process has been collected for analysis during a period of 44 days; the compile and unit testing part of the build process with the most sustainable amount of data. The number of components in the builds ranges, during this period of time, from 1 to 392 and the build time per component ranges from 1 to 234 minutes. The mean value of compiling and unit testing a component is based on this data 33 minutes. The median value is 19 minutes. The mean value of ISO creation and distribution is 51 minutes. For further information, see Appendix B Build Data.

3.3 Development Environment

3.3.1 Revision Control

As explained in 1.1 Background, revision control is a fundamental prerequisite of Continuous Integration. The software tool for revision control used in the product's development environment is Rational ClearCase™.

Data in Rational ClearCase™ is stored in versioned object bases and the data in a versioned object base is accessed using ClearCase™ views [17]. A mounted ClearCase™ view will appear as a normal file system to the end users; displaying selected versions of files and directories based on a configuration specification. Configuration specifications in the product's development environment are used by multiple views and are referred to as branches. A developer working on a branch of the product shares the same versions of files and directories with other developers on the same branch. Altered source code that is checked into the source control repository thereby needs to be merged with existing source code.

3.3.2 Build Environment

The build environment consists of a considerable amount developed bash, tcsh, and Perl™ scripts. Scripts that are in use differ between the branches as a different extent of automation of the build process has been achieved among them. All active branches are also, to different extent, built on nightly bases.

The hierarchy of scripts is presented according to branches that are most developed in the respect of automation. Included scripts are only those of higher importance to the build process. No scripts used as libraries or for single functionalities are thereby included. The goal of this overview is not to present the complete script hierarchy but to increase the understanding of the build environment and its process. This is needed in order to understand the complexity of switching from the current scripts, make, and Ant to other build automation tools where applicable.

Key actions of scripts with single functionality not included in the listing are however explained in the description of scripts calling them. The developed tool statusMonitor.pl, even though not of importance to the build process, is included for its importance to the feedback tool Continuous Integration Web and for clarity of its whereabouts.

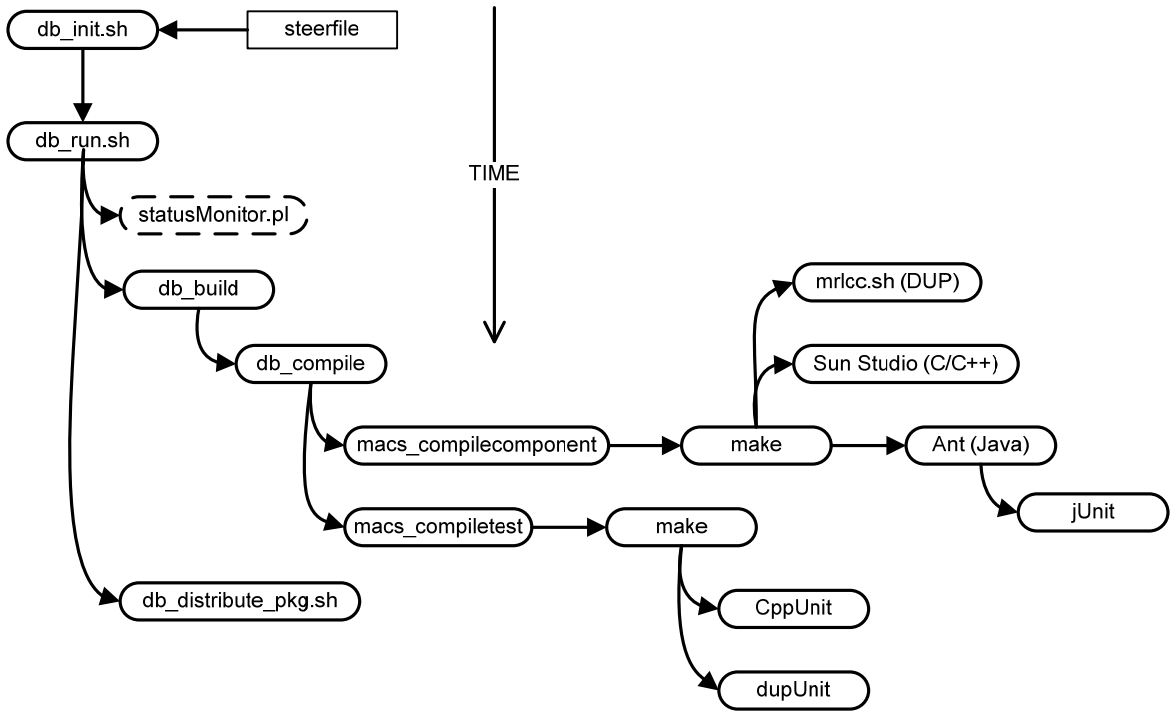

Figure 5: The product's Build Process

steerfile

Path	/vobs/sog/…/tools/dailybuild/
Description	• The steer file is highly important in the build environment. Records set up in the steer file guides the build process by deciding what to compile and what to test.

db_init.sh

Path	/vobs/sog/.../tools/dailybuild/bin/db/
Description	• Initialize the daily build and make sure that the chosen view is available and then set it up.

	• Make sure that the steer file is present. • Create a sorted list of components based upon component.nfo files in the selected view. • Filter out components from the created list that are not included in the provided original steer file. • Assemble the ordered and filtered steer file.

db_run.sh

Path	/vobs/sog/.../tools/dailybuild/bin/db/
Description	• Provide the possibility to start the build in three different ways: Debug, fast, or debug and fast. The build of both fast and debug versions can either be built on one machine or separately on two machines. • Collect all components needed to increase build number. • Loop through all components in order to find all changed components to compile. • Give the option to terminate the current build. • Create build logs, javadoc, ccdoc, and keep track of successfully built or failed components. • Send out e-mails about the success or failure of built components. • Create a websites of collected the result from the build, including dupUnit and CppUnit test.

db_build

Path	/vobs/sog/.../tools/dailybuild/bin/db/
Description	• Start parallel compile processes on multiple machines of components without dependencies.

db_compile

Path	/vobs/sog/.../tools/dailybuild/bin/db/
Description	• Check if all dependency components of components to build

	have been built. • Only build components if that has changed since last build or are set to force build.

macs_compilecomponent

Path	/project/.../ccenv/admin/tools/ccmacs/bin/
Description	• Use different sets of make files to compile components with mrlcc.sh for DUP code, Sun Studio for C/C++ code, and Ant to compile Java code and to start jUnit.

macs_compiletest

Path	/project/.../ccenv/admin/tools/ccmacs/bin/
Description	• Use a make file to compile and run dupUnit and CppUnit tests.

db_distribute_pkg.sh

Path	/vobs/sog/…/tools/dailybuild/bin/db/
Description	• Create an ISO or a package. • Distribute the ISO to testing environment.

3.3.3 Database Integration

The database can at the present time be built automatically via scripts and is available to developers as a central shared database. Local versions can be easily be set up by the developers themselves. Higher level testing is due to performance recommended to be executed on the shared database. The local databases are suggested to be run on dedicated machines. No automatic testing or inspection of the database is currently carried out.

3.3.4 Compilation

The compilation part of the build process is carried out automatically by making use of shell and Perl™ scripts as well as make and is more thoroughly explained in 3.3.2 Build Environment. Source code is compiled using Sun Studio™, Ant™, and the product specific mrlcc.

3.3.5 Testing

The automated testing consists of unit tests, component tests, and TTCN. TTCN is however not set up to start automatically in the build process but is on occasion started manually. Unit tests and component tests are not separated and thereby not run in any specific order.

3.3.6 Inspection

Inspections using Coverity Prevent™ for C and C++ as well as Checkstyle, FindBugs™, and PMD for Sun Java™ are currently run on multiple branches.

3.3.7 Deployment

Functionally to package and to distribute for functional testing is currently available and set up for a few of the product's branches. Automatic installation is being worked on and is run occasionally.

3.3.8 Feedback

The feedback available before the development of Continuous Integration Web mainly consisted of separate tools showing different parts of the build process with varying level of detail. Only raw logs were available for compilation, packaging, distribution, and installation. Tools and logs were found in different places and it was not possible see the collected result of the whole build process.

3.4 Continuous Integration Server

3.4.1 Manual and Automatic Integration

The Continuous Integration server's main responsibility, when compared to the scripted configuration currently in use, is to continuously check the repository for updated source code, and when this occurs, to execute the continuous integration cycle. Using a Continuous Integration server is optional in a Continuous Integration system. Additionally, [18] [19] even argues that it is better to not use a Continuous Integration server and suggest the following steps to practice Continuous Integration:

1. *Run build locally*
2. *Get the latest source code from the source code repository and build locally to see if the source code committed by other developers has integration problems*
3. *Make sure that other developers know that an integration is ongoing and that they do not update from the source code repository*
4. *Commit source code*
5. *Run build on an integration machine to assure that the success of the build is not dependent on a local environment*
6. *Announce that the integration has been completed*

Argued advantages using the above approach are that source code in the source code repository always can be built successfully and that possible problems can be easily isolated by looking at the step where the problem occurred. The difference in a scenario with a Continuous Integration server is that step five would be automated. The main reason to perform integration manually is according to [5] the lack of support in current Continuous Integration servers of preventing source code that breaks the build to reside in the source code repository. Making use of a Continuous Integration server in a Continuous Integration scenario is however a common practice as can be seen by examples and descriptions in [13] [14].

3.4.2 Continuous Integration Servers

This section will only list and compare Continuous Integration servers that are applicable for the product. The listing is based upon criteria in [20]. The criteria are put in relation to the product's

development environment in such a way that none of the most important key components in the current setup have to be changed. Hence, only Continuous Integration servers with support for Rational ClearCase™ version control and shell script will be included. Furthermore, only Continuous Integration servers that are freely available are included. All of the following Continuous Integration Servers can be used for the product.

Continuum [21]

Features	Build tools supported are Ant, shell script, Maven, Maven2. ClearCase version control is supported.
Reliability	Released in 2005
Longevity	Open source community
Target environment	As a minimum JDK 1.5 is needed. Cross-platform.

CruiseControl [22]

Features	Build tools supported are Ant, shell script, Maven, Maven2, Nant, make, windows batch. ClearCase version control is supported.
Reliability	Released in 2001
Longevity	Open source community
Target environment	As a minimum JDK 1.4 is needed. Alternatively a servlet container. Servlet and JSP versions are not stated. Cross-platform.

Hudson [23]

Features	Engage Build tools supported are Ant, shell script, Maven, Maven2, windows batch. Additional build tools are supported via plug-ins. ClearCase version control is supported via plug-in.
Reliability	Released in 2005
Longevity	Open source community
Target environment	As a minimum J2SE 1.5 is needed. Alternatively a servlet container that supports Servlet 2.4 and JSP 2.0 support. Cross-platform.

Luntbuild [24]

Features	Build tools supported are Ant, shell script, Maven, Maven2, Rake. ClearCase version control is supported.
Reliability	Released in 2004
Longevity	Open source community
Target environment	As a minimum JDK 1.4 is needed. Alternatively a servlet container with Servlet 2.3 and JSP 1.2 support. Cross-platform.

An advantage of using open source Continuous Integration servers mentioned in [5] is that they allow for debugging into their source code; this is a helpful feature when developing plug-ins for, or in other ways extend the functionality of a Continuous Integration server. The following Continuous Integration servers meet the requirements of implementing Rational ClearCase™ and shell script support but are not freely available and thereby not included in the listing.

- *anthillpro*
- *Bamboo*
- *Build Forge*
- *OpenMake Meister*
- *Parabuild*
- *QuickBuild*
- *TeamCity*

4 Developed Tools

4.1 Continuous Integration Web

As previously acknowledged it was decided to learn the current environment by prototyping a web based feedback tool. Continuous Integration Web is a feedback tool for the product's build process and will allow designers to view the state of all branches of the product concurrently as they are being built. This will help designers to know more easily the current total state of the software developed. The following sections will describe the different key parts of Continuous Integration Web in detail as well as issues during development and decisions made.

4.1.1 Functionality

The software's main components are: A full overview for the current state of all branches, displaying specific build process task's details, and preferences. Figure 6: Continuous Integration Web Overview Screen Shot shows Continuous Integration Web displaying continuous feedback from build processes of several active branches. The real branch names in the screenshot have been alias with branch1-9. Red areas communicate that a problem was encountered during the related step in the cycle and a checkmarks assures that a step has been completed. As seen in the live screen shot, the branches branch2 and branch6 are in the compile state while branch7 is in an ISO state. Other branches have completed their most recent cycle and their total result is display.

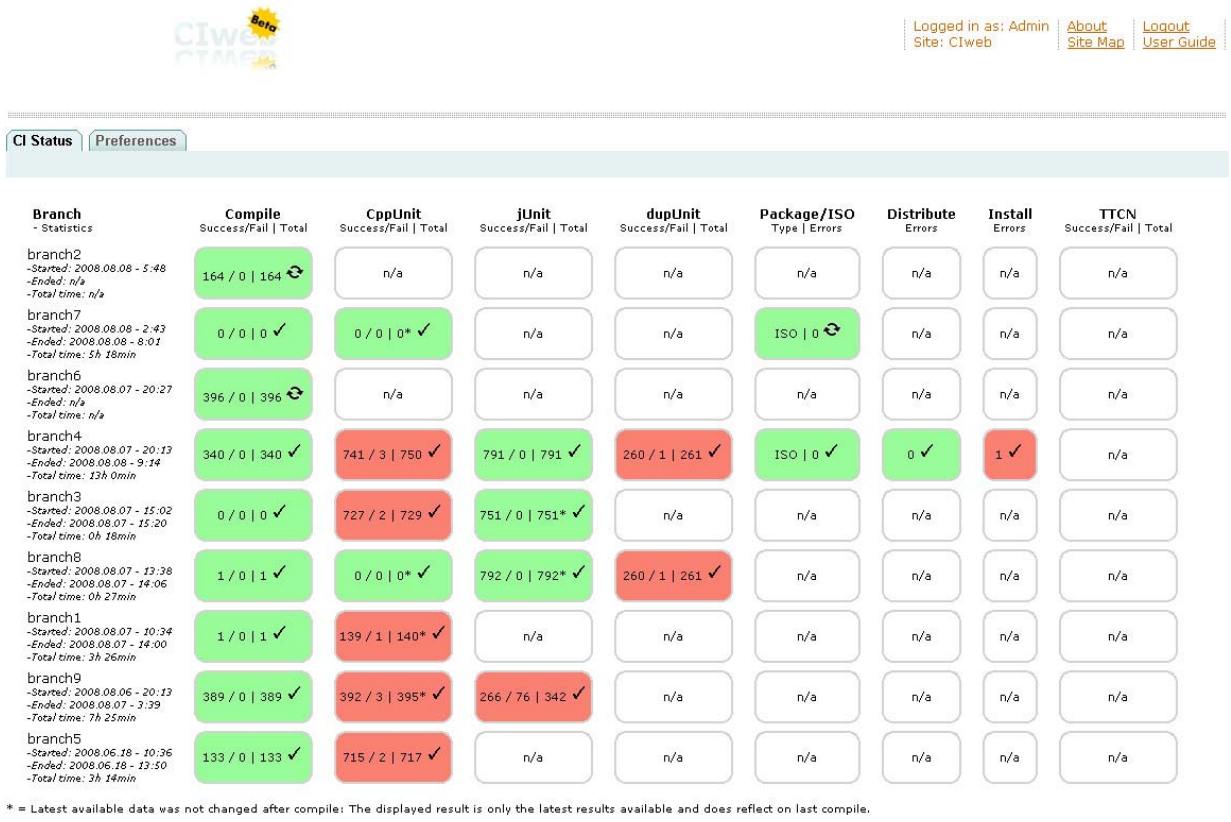

Figure 6: Continuous Integration Web Overview Screen Shot

Users can choose to view parts of the build process in higher detail by clicking on a desired element. The elements' details are displayed differently depending on if a structured report of its results is available outside of Continuous Integration Web or not. Existing structured reports are displayed when available and other elements' results are displayed in a unified way within Continuous Integration Web. Figure 7: Status Overview Use Case Diagram shows how the results of elements without existing reports can be acquired in relation to the branches overview displayed in Figure 6: Continuous Integration Web Overview Screen Shot. Refer to Appendix A Continuous Integration Web Software Requirement Specification for higher details about the functionality and requirements of the software.

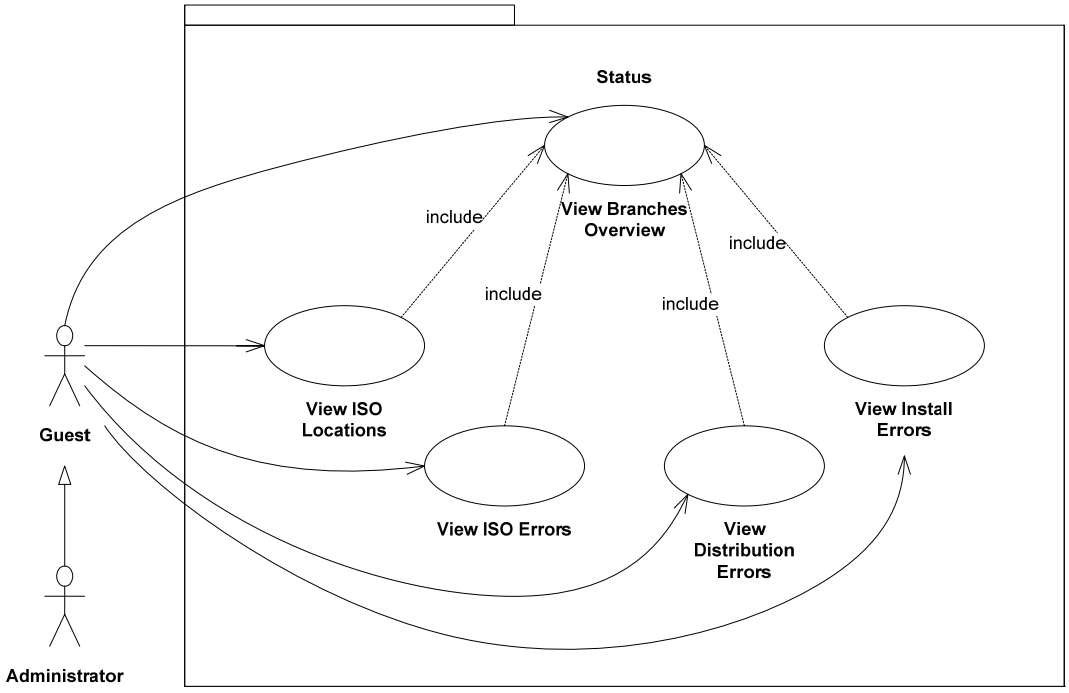

Figure 7: Status Overview Use Case Diagram

Preferences consist of the possibilities to configure project names of branches and parse locations of logs. All branches found by Continuous Integration Web are always displayed in the project name configuration. The list allows users to set project names for branches and only branches with a name set up will be displayed in the overview. Given project names are also used to parse logs. Since all parts of the build process usually make use of different project names for the same branches it has been made possible to enter multiple project names. These project names are then all used when searching for logs. The configuration of parse locations is implemented in order to allow for Continuous Integration Web easily to run in both the Solaris and the Windows environment. Parse locations can also be changed if any logs are stored at other locations than the current.

Continuous Integration Web only shows the current results from every part of the build process. Unit tests are however shown with an accumulated result as it is needed to understand the total status of the branch. The availability of up-to-date TTCN logs is at present not guaranteed. It could thereby be argued that the latest known results of TTCN also always should be shown.

4.1.2 Architecture and Structure

It was proposed that the system should be developed by using a series of scripts. Even though straight forward, this would however lead to difficulties in maintaining the software in the long term for people with no previous knowledge of it. It was instead decided to use a technology that would be more suitable for the Model-View-Controller design pattern [25] in order to assure the ease of which developers would be able to manage the system's source code. The Model-View-Controller design pattern isolates the user interface from business logic by making use of a controller to handle flow control, input data, and interpretation of data returned from the model.

Means to display continuously updated information using web, in accordance to the software requirements of instant updates without user interaction, had to be researched. Even though a Java applet would make this possible, the company's development road map states that applets should be phased out. Another solution that also would allow continuously updated information was Ajax. PHP and J2EE both provide numerous libraries for Ajax but previous experience in J2EE led to the decision of developing Continuous Integration Web using the Ajax together with J2EE.

The Model-View-Controller design pattern for actions that did not require any measures without receiving user interaction was implemented using Struts [26] and Tiles [27]. Struts is a Model-View-Controller framework implementation for J2EE while Tiles is a framework for web application user interfaces. Figure 8: Continuous Integration Web Flow shows how a request of a web page without Ajax is received and handled.

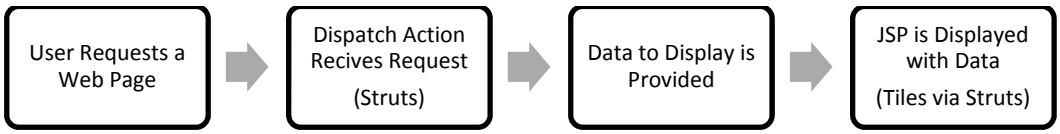

Figure 8: Continuous Integration Web Flow

The Model-View-Controller design pattern for actions that only receive user interaction to request the web page and then need to show continuously updated data was implemented using reverse Ajax. Reverse Ajax was accomplished by making use of the technology Comet with the

aid of the library DWR [28]. Comet utilizes infinitely long lasting HTTP connections to send updated data to user sessions accessing the overview. Figure 9: Continuous Integration Web Continuously Updated Flow shows how a request of a continuously updated web page is handled using Struts and Tiles together with Comet.

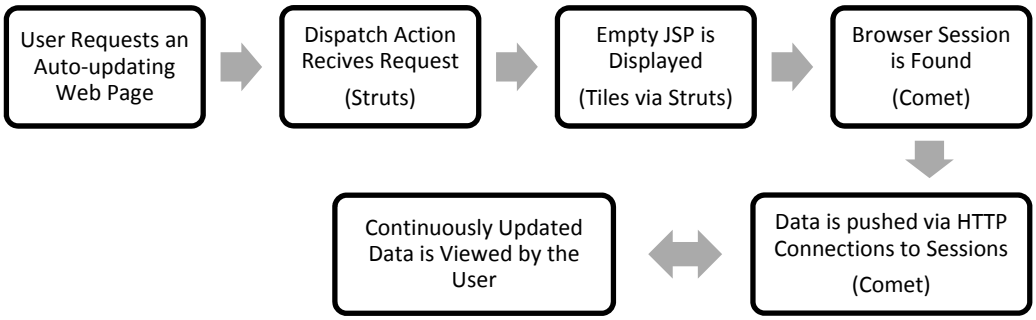

Figure 9: Continuous Integration Web Continuously Updated Flow

In order for Continuous Integration Web to provide up to date data to users as soon as they request the status overview, it was decided to utilize the javax.servlet.ServletContextListener. The context listener listens for the context to be initialized and is thereafter able to start the singleton LogReader. LogReader in hand collects data and keeps it up to date in one Branch object for every branch. Whenever new data is available, LogReader notifies StatusPusher to update available sessions with the new data so that it is displayed to all users. Figure 10: Data Package Class Diagram shows associations, inheritances, and dependencies between the above discussed classes.

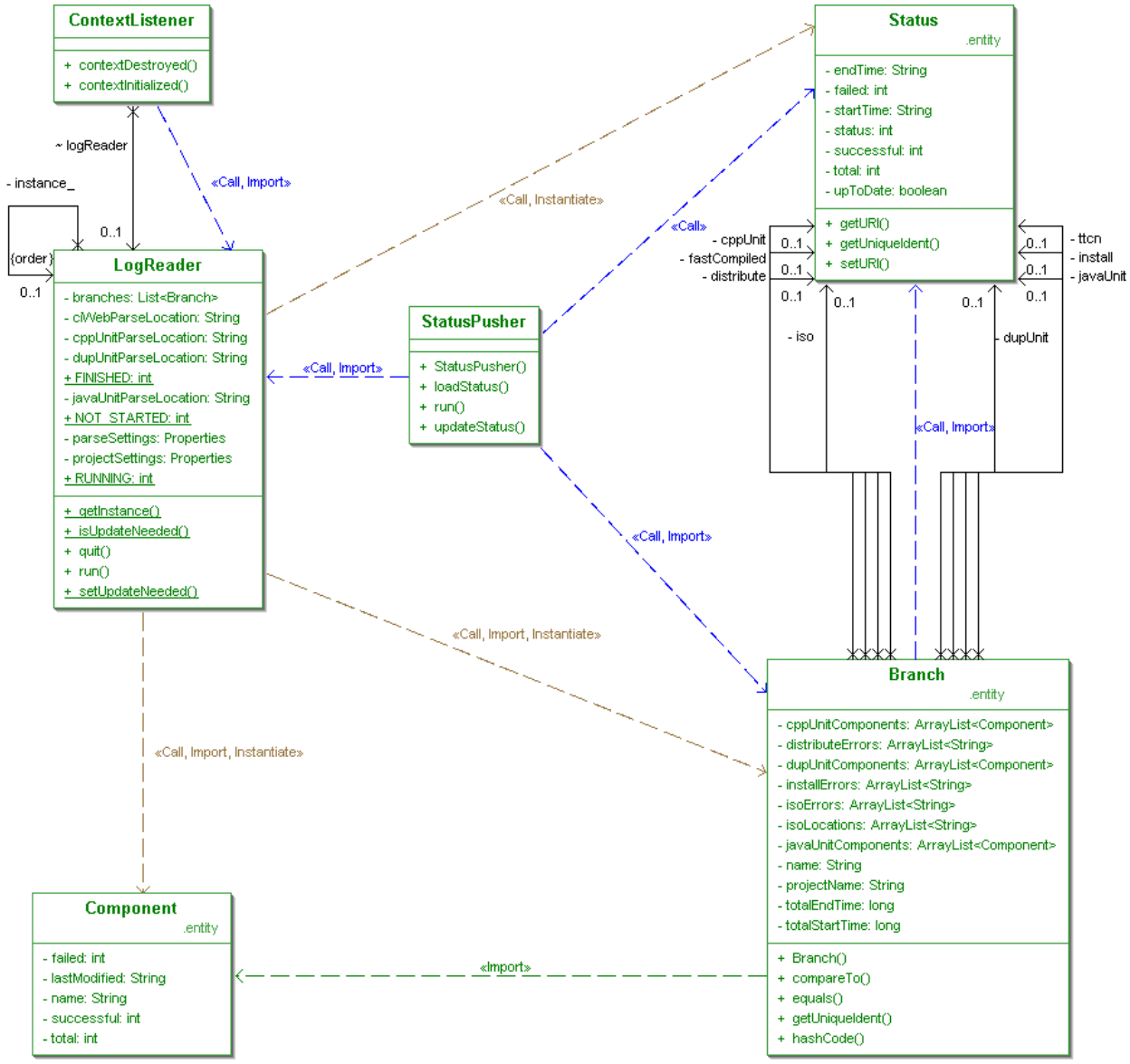

Figure 10: Data Package Class Diagram

4.1.3 Data Collecting

Data is currently collected from all automated tasks of the build process. Since all build process tasks from compilation to distribution are executed within different views for all branches, their log data are not possible to access without using the same view. ClearCase has support for a view referred to as a super view which provides access to all currently available views. Using a super view would however, due to the nature of the network, require the machine which is running Continuous Integration Web to constantly be logged into the zoned ClearCase environment.

A better solution was to develop a script to run along the build script in the build view of every branch. Compilation, package and ISO creation, distribution, as well as installation were thereby in need of a secondary tool to keep track of log files and data. The tool 4.2 Status Monitor was created to collect this data constantly from running builds and reports it in real-time, outside of the used view, where it could be easily accessed by Continuous Integration Web.

The result from unit tests was however already presented through web pages produced by scripts that were run after the complete result was available. This meant that displaying unit test results in real-time would require refactoring of already available functionality. Instead of refactoring already available software it was decided to prioritize output that was not presented to the designers in an acceptable way. The accumulated results of unit tests would be shown when compilation and unit testing had been completed. This would still provide the unit test results to the designer before the start of the next task in the build process: ISO or package creation.

Neither installation nor TTCN test is carried out within a view. Their logs can thereby be parsed in real-time directly by Continuous Integration Web. Figure 11: Continuous Integration Web and Status Monitor Data Collecting shows from where and how Continuous Integration Web acquire all its data about the build process and explains Status Monitor’s and Continuous Integration Web’s whereabouts in the build environment.

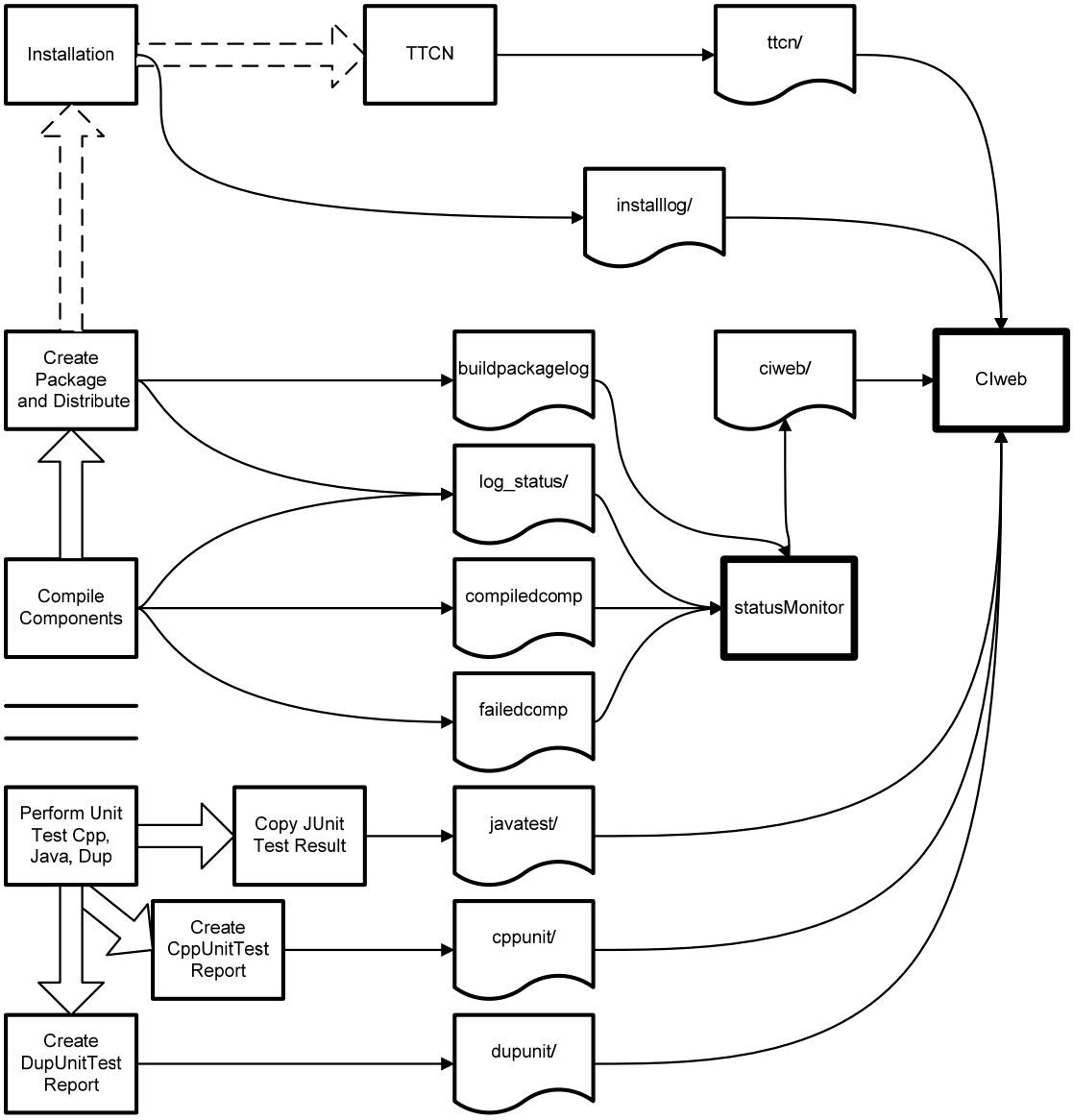

Figure 11: Continuous Integration Web and Status Monitor Data Collecting

ttcn/

Path	/project/.../www/ttcn/[project name]/base/details/
Description	Web report is available but not always as updated as the raw log files. Raw log files are not in a view and are parsed by Continuous Integration Web in real-time.

installlog/

Path	/project/.../www/installlog/[project name]/
Description	Raw log files are not in a view and are parsed by Continuous Integration Web in real-time.

ciweb/

Path	/project/.../www/ciweb/[branch name]/
Description	Produced by status monitor. Contain data about all parts of the build process performed in a view. Log files are loaded as Properties object by Continuous Integration Web in real-time.

log_status/

Path	/vobs/sog/…/tools/dailybuild/buildlogs/[date]/
Description	In a view and is parsed by status monitor in real-time. Contains status files created by scripts whose logs are not parsed.

buildpackagelog

Path	/vobs/sog/…/tools/dailybuild/buildlogs/[date]/
Description	In a view and is parsed by status monitor in real-time. Contains data about ISO and package creation and distribution.

compiledcomp

Path	/vobs/sog/…/tools/dailybuild/buildlogs/[date]/
Description	In a view and is parsed by status monitor in real-time. Contains data about compiled components.

failedcomp

Path	/vobs/sog/…/tools/dailybuild/buildlogs/[date]/
Description	In a view and is parsed by status monitor in real-time. Contains data about failed compiled components.

javatest/

Path	/project/.../www/javatest/[branch name]/[component]/junit/
Description	Web report is available and updated after completion of unit tests and compilation. Web report is parsed by Continuous Integration Web.

cppunit/

Path	/project/.../www/cppunit/[branch name]/logs/[date]/cppunit/
Description	Web report is available and updated after completion of unit tests and compilation. Web report is parsed by Continuous Integration Web.

dupunit/

Path	/project/.../www/dupunit/[branch name]/
Description	Web report is available and updated after completion of unit tests and compilation. Web report is parsed by Continuous Integration Web.

4.2 Status Monitor

The status monitor reports all acquired data to Continuous Integration Web in real-time by taking advantage of the fact that it automatically always is in the correct view. The structure of the report created by the status monitor is on the form of a java properties file for easy handling within Continuous Integration Web.

The following data from the build process is obtained by the status monitor:

- *Total compiled components*
- *Failed compiled components*
- *Build start time*
- *Build end time*
- *ISO creation start time*
- *ISO creation end time*
- *ISO errors*
- *ISO locations*
- *Package creation start time*
- *Package creation end time*
- *Distribution start time*
- *Distribution end time*
- *Distribution errors*

All data is acquired by parsing of files as they are created or log files as they are written. Different approaches were tested in search of the most efficient way. Since the logs had to be parsed as they were written, searching through them from line one to the last line on every update, similar to the UNIX command grep, was not an option. This solution, even though functional, would when tackling log files of approximately 5000 lines result in an unnecessarily high processor load with a total of $\sum_{i=0}^{5000} i \approx 1.25 \times 10^7$ lines to parse.

Instead of using a shell script it was decided to make use of Perl due to its powerful regular expression features. Using Perl also allowed using the library Tie::File which ties a pointer to all

the lines of a file to the memory. The last line number of the last line scanned could thereby be saved and used as a starting point when the next update a log file had occurred. Another problem solved with the help of the Perl library Tie::File was to remove the write buffering of the output data to be read by Continuous Integration Web.

An issue with this turned out to be that one log file on multiple occasions was written to by more than one script at the same time and that this left it impossible to seek. After an amount of time had been spent trying to solve this problem it was decided to work it out in the fastest way possible. The problem was thereby solved by altering the affected scripts to create files for desired key events, hence allowing for effortless parsing.

The whereabouts of the Perl script statusMonitor in the build process is showed in Figure 11: Continuous Integration Web and Status Monitor Data Collecting and explains from where data is obtained as well as where the processed data is stored.

5 Discussion

5.1 Continuous Integration Server

Utilizing a Continuous Integration server guarantees that the integration cycle is run whenever considered necessary. Continuous Integration can however be achieved without the aid of a Continuous Integration server and an approach where the integration is started manually is perfectly functional.

The potential negative effects of getting broken source code in the source code repository when using a Continuous Integration server have to be taken into account. Integration that always is carried out with perfect timing is not beneficial if the build never is attended when it breaks. It is also not beneficial if developers acquire broken source code from the repository and thereby constantly get broken builds for the wrong reason. On the other hand it is guaranteed that every manually started integration carried out according to the procedure explained in section 3.4.1 Manual and Automatic Integration will result in problems being discovered instantly by the developer integrating.

The disadvantages of allowing an automatic start of the integration could however be limited with the aid of good communication between the Continuous Integration server and the developers. Strict standards can prioritize to revert to the latest working source code in the repository if the build breaks.

The question of whether or not to use a Continuous Integration server does thereby only make up a small part of the incorporation of Continuous Integration. The decision should be based on the complexity of setting up a Continuous Integration server in the current build environment compared to the advantages and disadvantages of manual integration. The complexity of the build process in its current state is high enough to consider manual integration. It should however be taken into account that the complexity of it might change with the change of utilizing local integration servers and splitting the integration cycle.

A third option, which has to be compared to the complexity of setting up an available Continuous Integration server, is to develop one. Developing all necessary features of a Continuous Integration server for the current development environment would however require the investment of a considerable amount of time and reengineering of already fully functional parts. Furthermore, the currently available Continuous Integration servers, listed as possible candidates in 3.4.2 Continuous Integration Servers, are very flexible and can all execute shell scripts. The time that would be required to develop a Continuous Integration server should thereby rather be put into customizing a Continuous Integration server already available.

5.2 Build Process Changes

Integrations must not only run every time new source code is checked in, but parts of the current build process need to be attended before it is possible to fully benefit from Continuous Integration. To incorporate Continuous Integration before making any changes to the current build process would cause valuable time integrating to be much less effective.

The time consumed by the build process has to be severely shortened. Possible measures to achieve a shorter build time have been explained in 3.2 Enterprise Scale. An action that has to be taken is to split the build process into at least two integrations, hence shortening the time before developers first acquire feedback. The build process should be split according to Figure 2: First Integration Cycle and Figure 3: Second Integration Cycle. The build process could also include a third integration cycle to run even slower test if necessary. The second integration cycle can alternatively be designed so that it can be run in parallel on multiple workstations.

Splitting the build process will however not shorten the time spent waiting on results enough as the build on numerous occasions still far exceeds the recommended build time of ten minutes. In order to shorten the build time further, the amount of source code to build has to be limited. This should be done by splitting it into smaller parts divided by functionality to minimize dependencies between the different parts as is explained in Figure 4: Local Integration Servers. The product's source code is already split up in this manner and its build process could thereby with a reasonable amount of effort be adapted to utilize local integration servers.

The database needs to be treated more similar to source code and should be created using the present automatic script during the first integration cycle whenever a change to it has occurred. The database should also be unit tested. It should also be automatically recreated every time the second integration cycle is run.

Developers working with database related development currently needs to make use of a central database due to performance problems with local databases. Developers utilizing a central database affect other developers with untested database changes. Enough workstations to allow running dedicated databases for at least every two developers needs thereby to be acquired. This

will also allow for the current central database to be used for integration of the database during the second integration cycle.

Splitting the integration cycle is highly possible even though still using the current build scripts. Giving each component its own integration cycle is however a more complex procedure and it should be considered to switch to a more standardized and Continuous Integration server-friendly build tool when this is performed. To replace the current build scripts with a build tool could however become a time consuming and complex refactoring and the current build scripts should thereby not be replaced unless it is decided to change the compilation process at the same time.

Unit tests need to be differentiated from component tests and TTCN should be set up to run automatically. The speed of slow unit tests should be increased up by using mocks. Component tests that are necessary to run in the first integration cycle should be transformed into unit tests by using stubs.

Only unit tests should be kept in the first integration cycle while component tests need to be moved to the second integration cycle. The time consumed to build a component of the product indicates that differentiating component tests from unit tests should lead to a build time similar to the recommended ten minutes for single component builds. However, the time consumed during unit testing of the database has not been measured and should be taken into account.

Deployment should be included for more branches and having more branches automatically installed might speed up the process of getting TTCN to be started automatically on more branches.

Feedback is the cornerstone of a Continuous Integration system. It is therefore important that it is able to supply developers with *all* necessary information from the build process in a helpful manner. Devices such ambient orbs are great Continuous Integration tools but cannot easily communicate where in the build process a problem has occurred. Developers switch branches depending on projects. Older projects sometimes need to be handled alongside ongoing projects

by developers already working on current projects due to problems found during functional or system testing.

Providing continuous feedback by using ambient orbs with multiple branches might be difficult due to the practicality of supplying the feedback from many branches to many developers in a structured way. Continuous Integration Web is designed to satisfy these needs and cover all currently automatic parts of the build process. It is self-sufficient and can thereby supply developers with up-to-date information with only minor user interaction. Continuous Integration Web could in a beneficial way be used to display the current status of projects to many developers using universal monitors. Ambient orbs on the other hand are more suitable as first level build status indicators for single developers since they can call attention to a possible problem without any user interaction. People employed at supervisory and middle management level could also benefit from Continuous Integration Web by more easily being able to keep track of the projects current state.

5.3 Priorities and Concluding Remarks

Measures to introducing Continuous Integration for the product should be prioritized in such a way that it can be implemented as soon as possible. However, it also has to be able to find problems in a helpful manner for the developers when put into practice. A Continuous Integration system that can be employed sooner but does not find problems will not be of any more help to the developers than one that is still being developed.

All parts of the build process should be prioritized by how they affect the time it takes to find problems during the private build and if they are not yet present. The first target areas should be those that are parts of the requisites of a Continuous Integration system but today are absent from the development environment. This with the purpose of gaining the most out of additions and changes to the current development environment as builds already run on a nightly bases. The key functionality to supply feedback from the build process is highly important and has also been developed in the form of Continuous Integration Web as it was given the highest. The solution has received good feedback and is used during development of the product. Continuous Integration Web needs to be kept up-to-date as the development environment evolves.

Of the remaining target areas, the first priority is to start treating the database in the same manner as source code is treated. Database unit testing has to be introduced and the database has to be automatically built whenever it has been updated or changed. Any other parts of the build process in need of attention should thereafter follow before differentiating unit tests and component tests.

Making a distinction between unit tests and component tests is only valuable and useful if it is done when the build process has been split into two integration cycles. Component tests can thereafter be moved to the second. Splitting the build process and differentiating between unit tests and component tests should thereby be carried out in connection to each other. The adaption to incorporate local integration servers should follow. The feedback tool Continuous Integration Web has to be maintained to stay up to date with changes imposed by the suggested actions.

The introduction of a Continuous Integration server should not be considered as an important question on the path towards Continuous Integration. Even though a Continuous Integration server might be useful in the product's environment it might also not be. It is however safe to say that it at the present time has a very low priority when compared to previously discussed areas which to a much greater extent are in need of valuable work.

Further research on the subject that could be interesting is to see how Continuous Integration best should be incorporated in other similar enterprise environments. A deeper study could also be carried out on how developers should act and participate in a Continuous Integration development environment and on Continuous Integration's impact on the upcoming development results of the product. Furthermore, different ways to shorten and isolate unit tests could also be investigated more thoroughly if the unit testing still proves to be too time consuming after the recommended changes have been carried out. Since the goal is to shorten the feedback loop, more research could be carried out on how to utilize regression testing in an enterprise environment to find yet other ways to speed up the process.

Bibliography

1. **Myers, Glenford J., et al.** *The Art of Software Testing.* s.l. : John Wiley and Sons, 2004. 047167835X.

2. **Beck, Kent.** *Extreme Programming Explained: Embrace Change, Second Edition.* NJ : Pearson Education, Inc., 2004. 0-321-277865-8.

3. **Fowler, Martin.** Continuous Integration. *martinFowler.com.* [Online] May 6, 2006. [Cited: April 2, 2008.] http://martinfowler.com.

4. **Lee, Kevin A.** Realizing continuous integration. *developerWorks.* [Online] September 15, 2005. [Cited: June 19, 2008.] http://www.ibm.com/developerworks/rational/library/sep05/lee.

5. **Duvall, Paul M., Matyas, Steve and Glover, Andrew.** *Continuous Integration : Improving Software Quality and Reducing Risk.* Boston : Pearson Education, Inc., 2008. 0-321-33638-0.

6. **Fowler, Martin and Sadalage, Pramod.** Evolutionary Database Design. *martinFowler.com.* [Online] January 2003. [Cited: June 30, 2008.] http://martinfowler.com/articles/evodb.html.

7. **Duvall, Paul.** Automation for the people: Continuous Inspection. *developerWorks.* [Online] August 1, 2006. [Cited: June 25, 2008.] http://www.ibm.com/developerworks/web/library/j-ap08016.

8. —. Automation for the people: Speed deployment with automation. *developerWorks.* [Online] January 8, 2008. [Cited: June 27, 2008.] http://www.ibm.com/developerworks/java/library/j-ap01088.

9. *Backtracking incremental Continuous Integration.* **Storm, Tjis van der.** Athens : 12th European Conference on Software Maintenance and Reengineering, IEEE 2008, 2008.

10. **Duvall, Paul.** Automation for the people: Continuous feedback. *devloperWorks.* [Online] November 14, 2006. [Cited: June 17, 2008.] http://www.ibm.com/developerworks/java/library/j-ap11146.

11. **Swanson, Mike.** Mike Swanson's Blog. *Automated Continuous Integration and the Ambient Orb™.* [Online] msdn, June 29, 2004. [Cited: August 14, 2008.] http://blogs.msdn.com/mswanson/articles/169058.aspx.

12. *Scaling Continuous Integration.* **Rogers, Owen R.** Garmisch-Partenkirchen : 5th International Conference on Extreme Programming and Agile Processes in Software Enineering, 2004.

13. **Fredrick, Jeffrey.** Featured Articles: Better Software Magazine Continuous Integration. *Better Software Magazine.* [Online] September 2004. [Cited: June 17, 2008.] http://www.stickyminds.com/BetterSoftware/magazine.asp?fn=cifea&id=58.
14. **Magennis, Troy.** Continuous Integration and Automated Build at Enterprise Scale. *LINQed In - Troy Magennis' View on .NET, C# and Software Development.* [Online] November 26, 2007. [Cited: June 20, 2008.] http://aspiring-technology.com/blogs/troym/archive/2007/11/26/DoesContinuousIntegrationScale.aspx.
15. **Duvall, Paul.** Automation for the people: Continous testing. *developerWorks.* [Online] March 13, 2007. [Cited: June 27, 2008.] http://www.ibm.com/developerworks/library/j-ap03137.
16. **Rasmusson, Jonathan.** ThoughtWorks. *White Papers.* [Online] October 6, 2004. [Cited: August 15, 2008.] http://www.thoughtworks.com/pdfs/long-build-troubleshooting-guide.pdf.
17. **Posner, John.** CASEVision/ClearCase User's Guide. *sgi techpubs library.* [Online] July 21, 1994. [Cited: July 10, 2008.] http://techpubs.sgi.com/library/manuals/2000/007-2369-001/pdf/007-2369-001.pdf. 007-2369-001.
18. **Shore, James.** Continuous Integration on a Dollar a Day. *James Shore: Successful Software.* [Online] February 27, 2006. [Cited: July 11, 2008.] http://jamesshore.com/Blog/Continuous-Integration-on-a-Dollar-a-Day.html.
19. —. James Shore: Succeful Software. *Continuous Integration is an Attitude.* [Online] August 18, 2005. [Cited: July 11, 2008.] http://jamesshore.com/Blog/Continuous-Integration-is-an-Attitude.html.
20. **Duvall, Paul.** Automation for the people: Choosing a Continuous Integration server. *developerWorks.* [Online] September 5, 2006. [Cited: June 17, 2008.] http://www.ibm.com/developernetworks/java/library/j-ap09056.
21. **The Apache Software Foundation.** Apache Continuum. *Continuum Features.* [Online] June 19, 2008. [Cited: July 16, 2008.] http://continuum.apache.org/features.html.
22. **ThoughtWorks.** CruiseControl. *IntegratingWithOtherTools.* [Online] May 14, 2008. [Cited: July 16, 2008.] http://confluence.public.thoughtworks.org/display/CC/IntegratingWithOtherTools.
23. **Hudson.** Plugins. *Hudson Wiki.* [Online] June 13, 2008. [Cited: July 16, 2008.] http://hudson.gotdns.com/wiki/display/HUDSON/Plugins.

24. **Luntbuild.** Features List. *Luntbuild.* [Online] February 29, 2008. [Cited: July 16, 2008.] http://luntbuild.javaforge.com/feature_list.html.
25. *Web-application development using the Model/View/Controller designpattern.* **Leff, Avraham and Rayfield, James T.** Seattle : 5th Enterprise Distributed Object Computing Conference, IEEE 2001, 2001.
26. **The Apache Software Foundation.** Apache Struts. *Key Technologies Primer.* [Online] July 16, 2008. [Cited: August 15, 2008.] http://struts.apache.org/primer.html.
27. —. Apache Tiles. *The Composite View Pattern.* [Online] May 28, 2008. [Cited: August 15, 2008.] http://tiles.apache.org/tutorial/pattern.html.
28. **Direct Web Remoting.** Documentation. *Direct Web Remoting.* [Online] July 7, 2007. [Cited: August 15, 2008.] http://directwebremoting.org/dwr/documentation.

Appendix A Continuous Integration Web Software Requirement Specification

Contents

Figures

1 Introduction

1.1 Purpose of Document

The purpose of this document is to specify the software requirements for the Continuous Integration web (CIweb). The Software Requirement Specifications (SRS) is a complete description of the requirements for the system.

1.2 Document Structure

The document is divided into four main sections:
1 Introduction: Introduces the purpose and structure of this document.
2 Domain Description: Provides a general description of the system with a list of all system users.
3 Use Cases: Describes all use case scenarios, i.e. how functionality of the system will be used. This section is further divided into subsections resembling the logical division of the system functions.
4 Requirements: Describes functional and non-functional requirements for the system in relation to the use cases. The priority of requirements is defined as being:

- Essential: Functionality of highest importance to the system.
- Desirable: Functionality of high importance to the system.
- Optional: Functionality of lowest importance to the system.

2 Domain Description

CIweb is a feedback tool for the product's build process. It will allow designers to view the state of all branches of the product concurrently as they are being built. This will help designers to more easily know the current state of the software developed. Another key reason to the development of CIweb is to gain knowledge about the product's build process.

CIweb has two user groups:

Administrator: The administrator user group has access to and can configure system settings as well as to view the current build state of all branches of the product.

Guest: The guest user group has access to view the current build state of all branches of the product.

3 Use Cases

3.1 Subsection Use Case Diagram

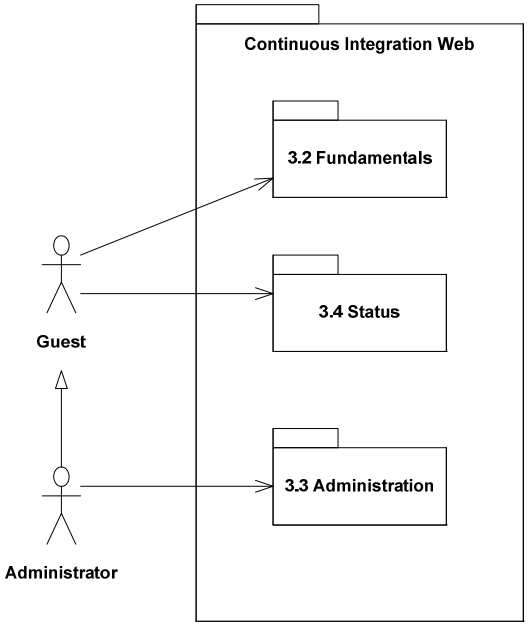

Figure 1: Subsection Use Case Diagram of Continuous Integration Web

3.2 Fundamentals

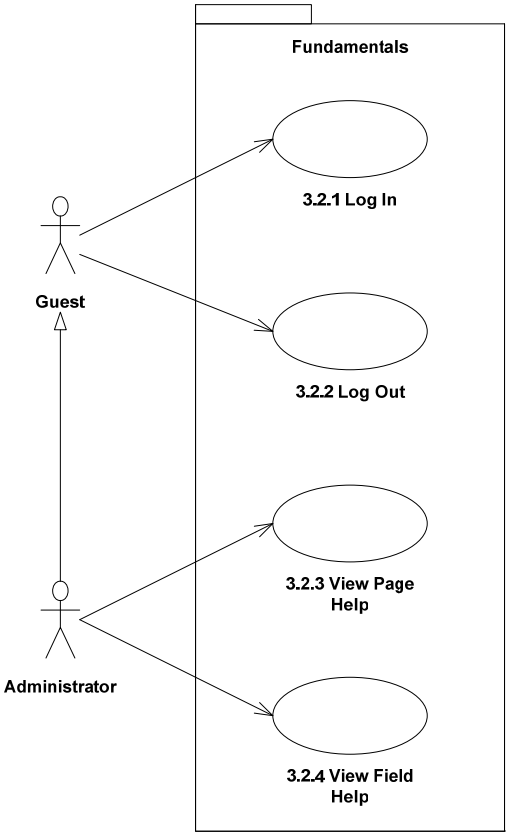

Figure 2: Fundamentals Use Case Diagram

3.2.1 Use Case: Log In

Brief Description	A user logs in to the system.
Primary Actors	Administrator, Guest
Secondary Actors	None
Preconditions	1. The user is not logged in to the system.
Main Flow	1. The user enters the system. 2. The user is asked to supply username and password in order to log in. 3. The user supplies username and password. 4. The user chooses to log in.
Postconditions	1. The user logged in to the system.
Alternative Flows	1. Invalid username or password.

3.2.1.1 Alternative Flow: Invalid username or password

Brief Description	A user logs in to the system.
Primary Actors	Administrator, Guest
Secondary Actors	None
Preconditions	1. Starts after point 4 of the main flow. 2. The user supplied invalid username or password.
Alternative Flow	1. The user is informed that the supplied username or password was invalid. 2. The user supplies username and password.
Postconditions	1. Next point is point 4 of the main flow.

3.2.2 Use Case: Log Out

Brief Description	A user logs out of the system.
Primary Actors	Administrator, Guest
Secondary Actors	None
Preconditions	1. The user is logged in to the system.
Main Flow	1. The user chooses to log out of the system.
Postconditions	1. The user logged out of the system.
Alternative Flows	None

3.2.3 Use Case: View Page Help

Brief Description	A user views page help.
Primary Actors	Administrator
Secondary Actors	None
Preconditions	1. The user is viewing a web page in the administration subsection.
Main Flow	1. The user chooses to view help intended for the current web page. 2. Help intended for the current web page is shown to the user.
Postconditions	1. The user has acquired help about what the current web page is used for.

Alternative Flows	None

3.2.4 Use Case: View Field Help

Brief Description	A user views field help.
Primary Actors	Administrator
Secondary Actors	None
Preconditions	1. The user is changing a branch project name or a parse location.
Main Flow	1. The user chooses to view help intended for the current field. 2. Help intended for the current field is shown to the user.
Postconditions	1. The user has acquired help about any restriction for the current field and what the current field is used for.
Alternative Flows	None

3.3 Administration

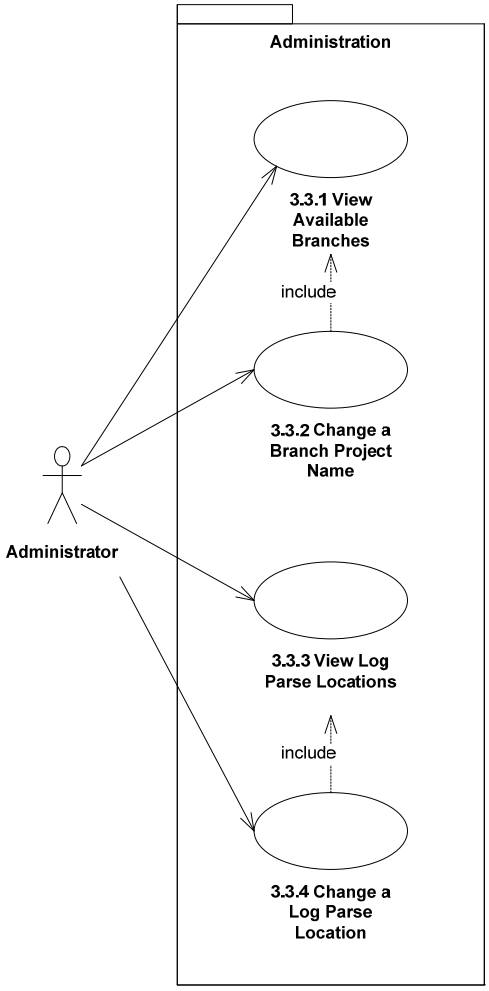

Figure 3: Administration Use Case Diagram

3.3.1 Use Case: View Available Branches

Brief Description	A user views branches available to the system.
Primary Actors	Administrator
Secondary Actors	None
Preconditions	1. The user is logged in to the system.
Main Flow	1. The user chooses to view branches available to the system.
Postconditions	1. Branches available to the system are displayed to the user with the following details: a. Branch Name b. Project Name

Alternative Flows	None

3.3.2 Use Case: Change a Branch Project Name

Brief Description	A user changes a branch project name.
Primary Actors	Administrator
Secondary Actors	Guest, Administrator
Preconditions	1. The user views branches available to the system. 2. The number of branches available to the system is greater than zero.
Main Flow	1. The user chooses to change a branch project name. 2. The user is asked to supply a project name for the corresponding branch. 3. The user supplies a project name. 4. The user is shown the result of the change.
Postconditions	1. The project name of the branch is changed. 2. If the project name was changed to a name with a length greater than zero, the branch is shown in the overview to all users. 3. If the project name was changed to a name with a length of zero, the branch is not shown in the overview to all users.
Alternative Flows	None

3.3.3 Use Case: View Log Parse Locations

Brief Description	A user views parse locations of logs to be set in the system.
Primary Actors	Administrator
Secondary Actors	None
Preconditions	1. The user is logged in to the system.
Main Flow	1. The user chooses to view log parse locations to be set in the system.
Postconditions	1. Log parse locations to be set in the system are displayed to the user with the following details: a. Log Name b. Parse Location

Alternative Flows	None

3.3.4 Use Case: Change a Log Parse Location

Brief Description	A user changes the parse location of a log.
Primary Actors	Administrator
Secondary Actors	
Preconditions	1. The user views log parse locations to be set in the system. 2. The number of log parse locations to be set in the system is greater than zero.
Main Flow	1. The user chooses to change a log parse location. 2. The user is asked to supply a parse location for the corresponding log. 3. The user supplies a parse location. 4. The user is shown the result of the change.
Postconditions	1. The parse location for the log is changed.
Alternative Flows	None

3.4 Status

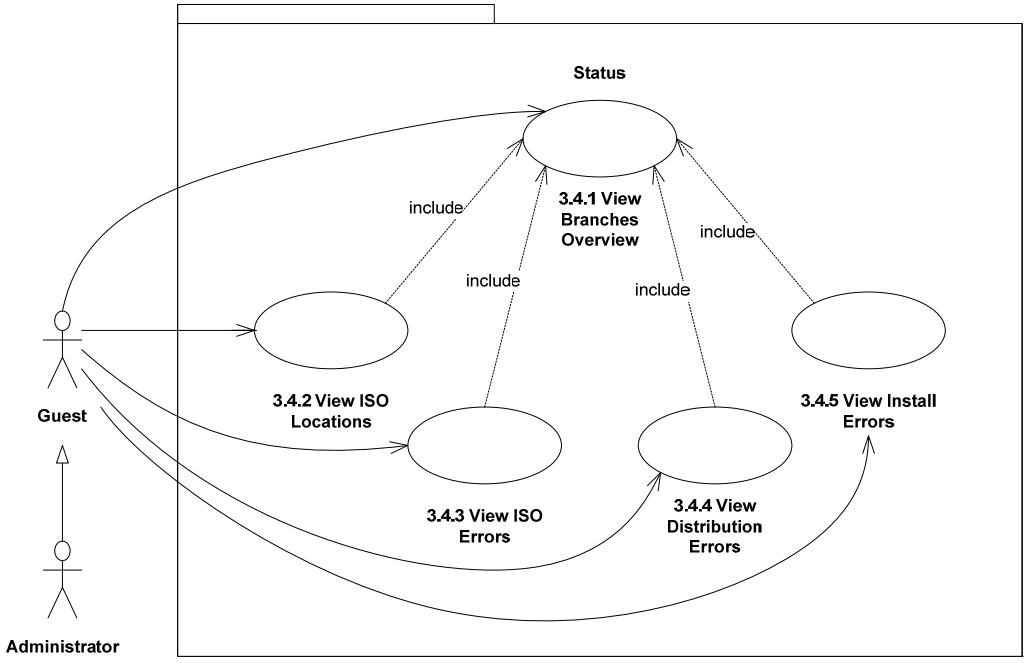

Figure 4: Status Use Case Diagram

3.4.1 Use Case: View Branches Overview

Brief Description	A user views an overview of branches.
Primary Actors	Administrator, Guest
Secondary Actors	None
Preconditions	1. The user is logged in to the system. 2. The number of branches with a project name greater than zero is greater than zero.
Main Flow	1. The user chooses to view overview of branches.
Postconditions	1. An overview of branches is displayed to the user where every branch, to the available data's highest extent, includes the following details: a. Compile, number of: Successful, failed, and total b. CppUnit, number of: Successful, failed, and total c. jUnit, number of: Successful, failed, and total d. dupUnit, number of: Successful, failed, and total e. Package: If created f. ISO: If created, number of errors

g. Distrubute: Number of errors
h. Install: Number of errors
i. TTCN:, number of: Successful, failed, and total

Alternative Flows	None

3.4.2 Use Case: View ISO Locations

Brief Description	A user views ISO locations.
Primary Actors	Administrator, Guest
Secondary Actors	None
Preconditions	1. The user views an overview of branches. 2. The number of ISO locations in the selected branch must be greater than zero.
Main Flow	1. The user chooses to view ISO locations of a branch.
Postconditions	1. ISO locations for the branch are displayed to the user.
Alternative Flows	None

3.4.3 Use Case: View ISO Errors

Brief Description	A user views ISO errors.
Primary Actors	Administrator, Guest
Secondary Actors	None
Preconditions	1. The user views an overview of branches. 2. The number of ISO errors in the selected branch must be greater than zero.
Main Flow	1. The user chooses to view ISO errors of a branch.
Postconditions	1. ISO errors for the branch are displayed to the user.
Alternative Flows	None

3.4.4 Use Case: View Distribution Errors

Brief Description	A user views distribution errors.
Primary Actors	Administrator, Guest
Secondary Actors	None
Preconditions	1. The user views an overview of branches. 2. The number of distribution errors in the selected branch must be greater than zero.

Main Flow	1. The user chooses to view distribution errors of a branch.
Postconditions	1. Distribution errors for the branch are displayed to the user.
Alternative Flows	None

3.4.5 Use Case: View Install Errors

Brief Description	A user views install errors.
Primary Actors	Administrator, Guest
Secondary Actors	None
Preconditions	1. The user views an overview of branches. 2. The number of install errors in the selected branch must be greater than zero.
Main Flow	1. The user chooses to view install errors of a branch.
Postconditions	1. Install errors for the branch are displayed to the user.
Alternative Flows	None

4 Requirements

4.1 Functional Requirements

4.1.1 Requirement: Compilation

Priority	Essential
Description	The result from compilation including the number of successful and failed as well as the total number of components compiled shall be possible to view in real-time.
Use Cases	

4.1.2 Requirement: CppUnit Test

Priority	Essential
Description	The result from CppUnit tests including the number of successful and failed as well as the total number of tests performed shall be possible to view as soon as data is made available by already existing scripts.
Use Cases	

4.1.3 Requirement: jUnit Test

Priority	Essential
Description	The result from jUnit tests including the number of successful and failed as well as the total number of tests performed shall be possible to view as soon as data is made available by already existing scripts.
Use Cases	

4.1.4 Requirement: dupUnit Test

Priority	Essential
Description	The result from dupUnit tests including the number of successful and failed as well as the total number of tests performed shall be possible to view as soon as data is made available by already existing scripts.

Use Cases

4.1.5 Requirement: Package and ISO

Priority	Essential
Description	It shall be possible see if a package or an ISO is created, ISO errors, the number of ISO errors, and locations of the ISOs in real-time.

Use Cases

4.1.6 Requirement: Distribution

Priority	Essential
Description	It shall be possible see distribution errors and the number of distribution errors in real-time.

Use Cases

4.1.7 Requirement: Installation

Priority	Essential
Description	It shall be possible see installation errors and the number of installation errors in real-time.

Use Cases

4.1.8 Requirement: TTCN

Priority	Essential
Description	The result from TTCN tests including the number of successful and failed as well as the total number of tests performed shall be possible to view as soon as data is made available by already existing scripts.

Use Cases

4.1.9 Requirement: Overview

Priority	Essential
Description	All branches given project names shall with corresponding data parsed from the build process shall in real-time and concurrently be possible to view in one web page.

Use Cases

4.1.10 Requirement: Branches Configuration

Priority	Desirable
Description	Branches shall be possible to be given project names according to user preferences and available branches should dynamically be loaded into the system for naming.
Use Cases	

4.1.11 Requirement: Parse Locations

Priority	Essential
Description	Parse locations that need to be set for log files shall be displayed and possible to change according to user preferences in order make it possible for the system to run under both Microsoft Windows and Sun Solaris environments.
Use Cases	

4.1.12 Requirement: Help

Priority	Optional
Description	Help shall be offered to the user on any page that interacts with a user.
Use Cases	

4.1.13 Requirement: Log In and Log Out

Priority	Desirable
Description	Log in and to log out shall be possible according to predefined user groups.
Use Cases	

4.2 Non-Functional Requirements

4.2.1 Requirement: Usability

Description Web page of the system shall be possible to view on one screen with a resolution not exceeding 1280 x 1024.

4.2.2 Requirement: Performance

Description Web pages shall be fully loaded within three seconds or less. The complexity and size of web pages shall thereby be held to a minimum by enforcing a maximum size of 300Kb including graphics.

4.2.3 Requirement: Reliability

Description The system shall, excluding scheduled down time, have an uptime of 95%.

4.2.4 Requirement: Security

Description The system shall only allow users logged in as a certain user group access to that user group's corresponding web pages.

4.2.5 Requirement: Scalability

Description The system shall support at least 50 concurrent users.

4.2.6 Requirement: Maintainability

Description The system shall be implemented using the Model-View-Controller design pattern to ensure the ease of which developers can manage the system's source code. The Code Conventions in the Sun Java Programming Language document shall be followed with the purpose of maintainable code. System events shall be logged into a text file.

4.2.7 Requirement: Technology

Description The system shall be compatible with web browsers Internet

Explorer 6.0 and higher as well as Firefox 2.0 and higher. It shall be deployable in Tomcat 5.0 and higher running on J2SE 1.5 or higher.

Appendix B Build Data

Compilation and Unit Test

Branch	Date	Components	Minutes	Min/Comp
	2008.06.12 - 22:47	2	142	71
	2008.06.13 - 15:20	10	492	49
	2008.06.13 - 21:38	353	553	2
	2008.06.16 - 21:16	305	351	1
	2008.06.16 - 9:43	8	18	2
	2008.06.17 - 17:25	133	156	1
	2008.06.17 - 22:51	305	376	1
	2008.06.18 - 10:36	133	194	1
	2008.06.18 - 19:23	163	284	2
	2008.06.19 - 10:30	354	385	1
	2008.06.19 - 17:52	1	22	22
	2008.06.19 - 19:23	354	436	1
	2008.06.19 - 3:11	5	112	22
	2008.06.21 - 16:39	1	109	109
	2008.06.21 - 7:31	324	392	1
	2008.06.22 - 6:38	3	124	41
	2008.06.23 - 11:17	2	26	13
	2008.06.23 - 12:29	1	21	21
	2008.06.23 - 19:22	1	97	97
	2008.06.23 - 23:09	1	140	140
	2008.06.23 - 4:48	3	100	33
	2008.06.23 - 9:21	1	30	30
	2008.06.24 - 11:06	2	29	15
	2008.06.25 - 12:48	1	20	20
	2008.06.25 - 19:23	353	435	1
	2008.06.25 - 2:10	1	112	112
	2008.06.26 - 15:36	2	21	11
	2008.06.26 - 16:40	1	17	17
	2008.06.26 - 17:22	6	23	4
	2008.06.26 - 19:23	15	117	8
	2008.06.26 - 2:57	40	124	3
	2008.06.26 - 7:36	1	98	98
	2008.06.27 - 19:23	392	757	2
	2008.06.28 - 19:23	392	769	2
	2008.06.28 - 8:35	2	187	94
	2008.06.29 - 19:23	392	670	2
	2008.07.01 - 14:08	1	27	27
	2008.07.01 - 19:23	19	120	6
	2008.07.03 - 19:16	1	234	234
	2008.07.04 - 12:32	1	67	67

2008.07.07 - 17:33	1	19	19
2008.07.07 - 19:14	1	170	170
2008.07.07 - 19:23	341	423	1
2008.07.14 - 14:41	1	34	34
2008.07.14 - 15:35	2	18	9
2008.07.15 - 19:25	343	561	2
2008.07.16 - 10:56	1	107	107
2008.07.16 - 20:18	2	30	15
2008.07.16 - 9:28	1	80	80
2008.07.17 - 19:43	364	756	2
2008.07.18 - 0:30	2	126	63
2008.07.18 - 16:53	1	22	22
2008.07.18 - 19:44	8	235	29
2008.07.18 - 2:52	4	119	30
2008.07.18 - 9:51	2	23	12
2008.07.19 - 0:07	2	36	18
2008.07.19 - 12:37	6	161	27
2008.07.19 - 19:44	8	319	40
2008.07.19 - 6:20	2	151	76
2008.07.19 - 9:12	7	183	26
2008.07.20 - 1:32	2	26	13
2008.07.20 - 19:44	8	165	21
2008.07.20 - 22:52	2	26	13
2008.07.21 - 17:24	1	27	27
2008.07.21 - 19:43	7	119	17
2008.07.21 - 19:44	8	191	24
2008.07.21 - 22:11	3	133	44
2008.07.21 - 23:19	2	56	28
2008.07.22 - 0:44	2	108	54
2008.07.22 - 10:21	12	31	3
2008.07.22 - 10:33	1	14	14
2008.07.22 - 13:39	1	18	18
2008.07.22 - 14:43	1	14	14
2008.07.22 - 19:43	354	449	1
2008.07.22 - 19:44	8	202	25
2008.07.22 - 23:29	2	30	15
2008.07.22 - 4:33	36	133	4
2008.07.23 - 12:40	143	172	1
2008.07.23 - 19:23	292	338	1
2008.07.23 - 19:44	8	216	27
2008.07.23 - 23:44	2	32	16
2008.07.23 - 3:46	327	279	1
2008.07.23 - 8:44	3	113	38

2008.07.24 - 1:33	3	113	38
2008.07.24 - 20:55	1	137	137
2008.07.24 - 20:58	2	31	16
2008.07.24 - 3:42	3	121	40
2008.07.24 - 6:02	1	95	95
2008.07.24 - 9:57	1	23	23
2008.07.25 - 10:30	5	53	11
2008.07.25 - 2:32	3	120	40
	Mean value:	**171**	**33**
	Median value:	**119**	**19**

ISO and Distribution

Branch	Date	Minutes
	2008.07.21 - 19:43	49
	2008.07.23 - 19:23	47
	2008.07.25 - 10:30	41
	2008.07.22 - 19:43	45
	2008.06.23 - 9:21	43
	2008.06.27 - 19:23	63
	2008.06.29 - 19:23	45
	2008.07.17 - 19:43	94
	2008.06.24 - 11:06	36
	2008.07.24 - 1:33	0
	2008.07.24 - 21:46	109
	2008.07.21 - 17:24	36
	Mean value:	**51**
	Median value:	**45**