Automated Integration Testing – an Evaluation of CruiseControl.NET

CHRISTIAN JÖNGREN

Master of Science Thesis
Stockholm, Sweden 2008
Automated Integration Testing
– an Evaluation of CruiseControl.NET

CHRISTIAN JÖNGREN

Master’s Thesis in Computer Science (30 ECTS credits)
at the School of Computer Science and Engineering
Royal Institute of Technology year 2008
Supervisor at CSC was Karl Meinke
Examiner was Stefan Arnborg

TRITA-CSC-E 2008:123
ISRN-KTH/CSC/E--08/123--SE
ISSN-1653-5715

Royal Institute of Technology
School of Computer Science and Communication
KTH CSC
SE-100 44 Stockholm, Sweden
URL: www.csc.kth.se
Automated Integration Testing – an Evaluation of CruiseControl.NET

Abstract

Testing is an important task of the software development process. It gives assurance of many things, for example that the software executes correctly or that its performance is within acceptable limits – it gives assurance of the quality for the software. Despite this fact, testing is the task that often gets down prioritized because of budget constraints or time shortage. Often this depends on the fact that testing must be performed manually and thus becomes a very time consuming task. Automation is something seen as a potential and partial solution to this problem.

This report presents the results of using a continuous integration server – CruiseControl.NET – and the NUnit framework, as a means for automating the integration testing in a specially selected project at a Swedish IT-company. It also presents evaluation results regarding the use of CruiseControl.NET as a pure software testing tool. The evaluation was performed in parallel with the work of automating the integration testing.

With the approach presented above, the integration testing could be partially automated. However, it was also confirmed that an initial investment is needed to achieve even, only a partial automation of the software testing. Further it was established that CruiseControl.NET has some potential of increasing cost-effectiveness for the integration testing as well as contributing to a higher, final quality of developed software.

Automatisk integrationstestning – en utvärdering av CruiseControl.NET

Sammanfattning


Preface

The master’s project presented in this master’s thesis has been performed under the School of Computer Science and Communication at the Royal Institute of Technology. It was further conducted within the Computer Science and Engineering program and was supervised by Professor Karl Meinke.

The subject for this master’s project was assigned by Brainpool AB in Solna, Stockholm. The work in this master’s project was applied to a selected project called MEK, at Brainpool AB. This work was supervised by Torbjörn Dahlin and Stefan Skoog.

I want to thank all my supervisors for their aiding in the performance of this master’s project. Special acknowledgement is given to Torbjörn Dahlin, for his ability to help me get past especially tough issues.

I also want to give my appreciation to Ingela Linered for her overall support during the performance of this work.
Table of Contents

1. Introduction ......................................................................................................................................... 1
   1.1 Background ................................................................................................................................. 1
   1.2 Problem Definition ....................................................................................................................... 1
   1.3 Goals/Purpose ............................................................................................................................. 2
   1.4 Delimitations ............................................................................................................................... 3
   1.5 Organization of this Report ........................................................................................................... 3
2. Theoretical Background ...................................................................................................................... 4
   2.1 Agile Development ....................................................................................................................... 4
   2.2 Software Integration ...................................................................................................................... 4
   2.3 Continuous Integration .................................................................................................................. 4
   2.4 Continuous Integration Practices ................................................................................................... 5
      2.4.1 Maintain a single source repository ...................................................................................... 6
      2.4.2 Automate the build ............................................................................................................... 6
      2.4.3 Make your build self-testing .............................................................................................. 6
      2.4.4 Everyone commits every day ............................................................................................. 6
      2.4.5 Every commit should build the mainline on an integration machine ................................. 6
      2.4.6 Keep the build fast ............................................................................................................... 7
      2.4.7 Everyone can see what is happening .................................................................................... 7
   2.5 Build Automation ......................................................................................................................... 7
   2.6 Software Testing ............................................................................................................................ 7
      2.6.1 Software testing in general .................................................................................................. 8
      2.6.2 Testing on different levels .................................................................................................. 8
   2.7 Automated Testing ......................................................................................................................... 9
      2.7.1 Automated integration testing ............................................................................................ 9
      2.7.2 What to automate? .............................................................................................................. 11
   2.8 CruiseControl .............................................................................................................................. 11
3. Methodology ..................................................................................................................................... 13
   3.1 About the Evaluation Model ....................................................................................................... 13
   3.2 Performance of the Evaluation .................................................................................................... 13
      3.2.1 Theoretical studies ............................................................................................................. 13
      3.2.2 Practical work ..................................................................................................................... 13
4. Evaluation Results ............................................................................................................................. 14
   4.1 Performance of CruiseControl .................................................................................................... 14
      4.1.1 Time per test iteration ......................................................................................................... 14
4.1.2 Costs of testing ..................................................................................................................... 15
4.1.3 Test coverage ........................................................................................................................ 16
4.1.4 Testing frequency .................................................................................................................. 16
4.1.5 Summary .............................................................................................................................. 17
4.2 Process Definition Ability ....................................................................................................... 17
4.3 Non-Intrusiveness ..................................................................................................................... 18
4.4 Ease-of-Use .............................................................................................................................. 19
  4.4.1 Sub criteria for ease-of-use ............................................................................................... 19
  4.4.2 Summary .............................................................................................................................. 26
4.5 Project Preconditions ............................................................................................................... 26
5. Automated Integration Testing Capability of CruiseControl ...................................................... 28
6. Other Conclusions ....................................................................................................................... 30
  6.1 Product Quality Using CruiseControl? .................................................................................. 31
  6.2 Cost Effectiveness Using CruiseControl? ............................................................................... 32
7. Overall Conclusions ................................................................................................................... 35
8. Recommendations ...................................................................................................................... 36
9. Work Not Performed ................................................................................................................... 37
List of References ........................................................................................................................... 38
  Books............................................................................................................................................. 38
  Articles .......................................................................................................................................... 38
  WWW............................................................................................................................................ 39
Appendix 1 – The Evaluation Model .............................................................................................. 41
Appendix 2 – the NUnit Graphical Test Runner ............................................................................... 45
Appendix 3 – Example of a Build Report, displayed in the Web Dashboard .................................... 46
Appendix 4 – Example of a NUnit Test Report, displayed in the Web Dashboard ............................ 47
Appendix 5 – The XML Configuration File .................................................................................. 48
Appendix 6 – The Web Dashboard, Start Page .............................................................................. 52
1. Introduction

This thesis presents the results of an evaluation regarding the continuous integration server CruiseControl.NET, version 1.2.1 (from here only referred to as CruiseControl) [CC]. This evaluation was further performed with the more specific intention of finding out if, and how CruiseControl could be used to automate the integration testing at Brainpool AB.

1.1 Background

Software integration work (with belonging testing) is an activity preferably performed according to an incremental model (continuously) and not as an individual, independent phase at the end of the software development process [Sommerville 2001: p452]. One of the reasons for this is it becomes easier to find potential faults in the system under development [Sommerville 2001: p452]. If you integrate the system in a stepwise manner the area for error localization is delimited [Sommerville 2001: p452].

Brainpool is an example of a company that conducts its integration work, with accompanying testing, continuously. However this is done according to manual techniques in an ad hoc manner. This is not considered to achieve good enough results regarding, among others the final quality of the system and the cost effectiveness for the testing of the system. The company believes a tool supported automatization of the integration work, including testing, can give better results in mentioned areas. CruiseControl was selected for this purpose but the question remained if it fulfills the special requirements of the business.

1.2 Problem Definition

The main problem at Brainpool was that the integration testing was not cost effective enough. This often entailed executing the same tests repeatedly. Foremost it was the repeated execution of the tests that took unnecessarily long time to perform. In addition, this work was of highly routine nature, as well as very monotonous. Consequently the wish was it would be made more effective. If this work could be automated developers could spend more time performing tasks demanding higher creativity and thus provide more value for the business.

The problem at hand became especially manifest when the company integration tested a SOA-based system (called MEK). This system consisted of four layers, as illustrated in figure 1 below.

---

1 SOA – Service Oriented Architecture is a design approach for integrating across diverse systems [SOA]. It enables IT resources, like applications or whole systems to be accessed as services [SOA].
The integration testing was done through direct interaction with the user’s interface of the application (manual input of data) and afterwards the database was manually controlled to be in a valid state (corresponding to system inputs). The integration testing consisted of running 51 different test cases and the duration of one test run (input of data according to all test cases and checking of values in database) was approximately three days. This manual running of the tests implied there was no easy way to rerun the test cases, which was necessary after a change to the system. Consequently another three days of integration testing was necessary every time a minor or major change was made to the implementation of the system.

The hope was that a tool supported automation of the integration testing to some part would be able to solve this problem. The tool Brainpool had special interest of investigating closer for this purpose was CruiseControl [CC]. Consequently the problem specific for this master’s project came to be evaluating CruiseControl, in order to establish if and how the tool could be used to implement automated integration testing.

1.3 Goals/Purpose

The main goals of this master’s project were:

1. To conduct an evaluation of CruiseControl as a test tool, on the basis of specially selected criteria
2. Examine if and how CruiseControl could help implement automated integration testing.

The aim was also to draw theoretical conclusions regarding two specific questions at issue that were of special interest to Brainpool. These were as follows below:

1. Could it be justifiable from a cost effectiveness perspective to incorporate CruiseControl into a product/service development process similar to the one at Brainpool?
2. Could an incorporation of CruiseControl into a product/service development process similar to the one at Brainpool give rise to a higher final quality of the products/services developed?

1.4 Delimitations

In the work of evaluating CruiseControl, partly through theoretical studies of the tool and partly through practical usage of the tool in a selected project, it was not included to perform a fully functional incorporation of CruiseControl into the development process at Brainpool. Nor did it include performing a complete and fully functional incorporation of CruiseControl in a delimited part of the development process – for example a certain project. Only a practical try-out of CruiseControl (in a selected project) was as earlier described, conducted. These limitations depended on the limited time frame of a master’s project (20 weeks).

Further it was not included in the project to find out what possible long term effects an eventual automatization of the integration testing (via CruiseControl) could have for a software development organization similar to Brainpool. The reason for this is also that it would require a more extended evaluation, extending beyond the end of this master’s project.

1.5 Organization of this Report

Chapter 2 gives a summary of the literature studied as a pre phase to the practical work conducted in this master’s project. Chapter 3 explains the methodology used and to some detail how this master’s project was conducted.

Chapters 4, 5 and 6 concern the results of this work. 4 discusses the results directly related to the criteria in the evaluation model used, while 5 and 6 presents more general results of this evaluation. Chapter 5 especially, discusses CruiseControl and its ability to implement automated integration testing.
2. Theoretical Background

This section summarizes the theory studied as a pre phase to this master’s project. It will discuss the core concepts and techniques essential for the performance of this master’s project.

2.1 Agile Development

A brief and general description of agile development is given here. This is because of the close relationship between CruiseControl and the agile practice Continuous Integration.

Agile development is a reaction to the traditional heavy processes for software development [Schuh, 2004: p2]. A well-known example of the latter is the Waterfall Model [Sommerville, 2001: p45]. An example of the former is the Extreme Programming methodology [Beck and Andres, 2004].

Very simply agile software development can be divided in a theoretical, almost philosophical part and one more practical part. The theoretical part consists of factors agile developers consider to be of high value. Among these are Individuals and interactions and the ability to respond to change [Agile Alliance]. Further it is emphasized in the theoretical part to follow certain principles when developing software. These are meant to guide software development, because the values are too abstract to directly guide your behaviour [Beck and Andres, 2004: p23]. Examples of these are to welcome changing requirements, to keep it simple or to deliver working software frequently (iteratively) [Agile Alliance].

The practical part consists of the practices performed on a daily basis to ensure the fulfilment of the principles and to bring accountability to the values [Beck and Andres 2004]. Examples of these practices are Continuous Integration, Build Automation and Automated Unit Testing.

2.2 Software Integration

What does software integration mean? A very simple explanation is given by Siegel. He explains the integrating of a component as the process of adding the component to the system [Siegel, 1996: p236]. Or more practically – the process of putting the component under version control. He further says integration can happen with different time intervals – everything from once a day to once per implementation iteration. He recommends doing it as often as once a day.

Further, Sommerville generally speaks of integration as an activity that should be performed according to an incremental model (continuously) [Sommerville, 2001: p452]. It should not be performed as its own phase at the end of the development process. One of the reasons for this is that it becomes easier to find possible errors in the system [Sommerville 2001: p452]. As you integrate the system in a stepwise manner the area for error localization is delimited [Sommerville 2001: p452].

2.3 Continuous Integration

CruiseControl implements Continuous Integration and therefore it is good to have a basic understanding of this agile practice. Consequently this section is devoted to a thorough discussion of the topic.

Continuous Integration has the goal of changing old habits of integration in which separate branches of code reside on the workstations of programmers and then on a daily, weekly or (even worse) monthly basis is integrated into a single, working system [Schuh, 2004: p76]. This
means the practice encourages integrating small bits of functionality into an up-to-date and clean code base, as soon as they are completed [Schuh, 2004: p76].

What Continuous Integration means in more detail is that integration and testing of changes should occur often. Preferably, not more than a couple of hours of work should be completed before the next integration and test phase occurs [Beck and Andres, 2004: p49]. The cost for it will be higher and more unpredictable the longer you wait before integrating and testing [Beck and Andres, 2004: p49].

Very simplified the Continuous Integration process consists of the following practical tasks [Beck and Andres, 2004: p49]:

1. You check in your changes to a version control tool
2. The build system notices change in the source code and starts building and testing
3. When problems occur the user is notified by some sort of message (i.e. email, text message)

Before step 1 above you should add the step of building your system, locally on your own machine before checking in changes [Fowler 2006]. But the most important thing when practicing Continuous Integration (as interpreted by this author) is the build which takes place on a specific integration machine and that always bases itself on the mainline (this build is the one that occurs in step 2 above). The mainline is the current state of the system [Fowler 2006]. The state of the system should in turn be interpreted as the system source code residing in the version control system.

Furthermore you can categorize the Continuous Integration process to be either asynchronous or synchronous [Beck and Andres, 2004: p49]. The former is usually the most common one and means that integration takes place at a random time (when the developer feels like it) [Beck and Andres 2004: p49]. The latter means the integration is performed according to a predefined schedule. Beck and Andres for example, state their preferred schedule – to integrate after each pair-programming episode2 [Beck and Andres, 2004: p49].

Continuous Integration also assumes a high degree of tests are available and that these should be automated into the software [Fowler 2006]. The automation of tests into the software is something Fowler refers to as self-testing code and this is simply the practice of writing comprehensive automated tests that will accompany the functional software [Fowler, 2006]. The automated tests can be written with for example the XUnit testing frameworks. One example of these is NUnit [NUnit] for Microsoft’s .NET framework [.NET].

The tests are then used by CruiseControl as part of the continuous integration process. This means that when CruiseControl performs the continuous builds, in conjunction it also runs available tests. As is the case when using the NUnit testing framework CruiseControl will use the NUnit command line tool for running the tests. Thus one can say NUnit has, in this case an important role in the continuous integration process.

2.4 Continuous Integration Practices

Above the overall description of Continuous Integration was given. Providing more details, below follows what Fowler refers to as the key practices of effective Continuous Integration [Fowler, 2006]. It is not the full list of all of Fowler’s key practices, only a sample of what is deemed most interesting for this project.

---

2 This is an Extreme Programming primary practice [Beck and Andres, 2004: p42]. A pair-programming episode should be at a maximum a couple of hours long [Beck and Andres, 2004: p50].
2.4.1 Maintain a single source repository

When developing software lots of different files are created and these need to be managed in a controlled fashion. A good source code management system can help accomplish this. Besides the obvious production code, everything that is needed to do a build should be put in the repository [Fowler, 2006]. These artefacts include for example test scripts; properties files; database schemas; install scripts; and third party libraries [Fowler, 2006]. To cite Fowler: “The basic rule of thumb is that you should be able to walk up the project with a virgin machine, do a checkout, and be able to fully build the system” [Fowler, 2006].

2.4.2 Automate the build

It can be a complicated process getting your sources into a running system. Among other things it involves compilation, moving files around, loading schemas into databases etc. To make this work less time consuming and error prone you can automate the process by using build tools like NAnt [NAnt] or MSBuild [MSBuild]. The principle with this practice is also that you should include everything in the build. To again cite Fowler: “anyone should be able to bring in a virgin machine, check the sources out of the repository, issue a single command, and have a running system on their machine” [Fowler, 2006].

2.4.3 Make your build self-testing

The build process should include a suite of automated tests that can check a large part of the code base for bugs [Fowler, 2006]. But more importantly it means that the failure of a test should cause the build to fail [Fowler, 2006].

An example of tools ideal for this kind of testing is the XUnit testing frameworks [Fowler, 2006]. They make it very easy to set up a fully self-testing environment [Fowler, 2006]. The way in which you set up a self-testing environment lies within the implementation of the tests. These should be implemented so that the individual tests check the results (from method invocation and similar) themselves and verify their correctness [Fowler 2006]. The tests should also be automated [Fowler 2006], which quite simply means you should be able to execute by “the press of a button”.

2.4.4 Everyone commits every day

Everyone should commit each day or preferably more often than that. This is because the more often you commit the narrower is the time area in which an error may have occurred and so it is easier to find [Fowler, 2006]. Also it enhances the communication between the different developers – integration is a way for developers to tell other developers what they have changed [Fowler, 2006].

2.4.5 Every commit should build the mainline on an integration machine

With every commit a build should be made on an integration machine specially devoted for this purpose. Only if this integration build is successful the commit should be considered to be done [Fowler, 2006]. One reason for this is that developers may forget to update and build on their own machines before committing [Fowler, 2006].

One way to ensure this is done is to use a Continuous Integration Server. This acts as a monitor to the repository – when a commit against the repository finishes the Continuous Integration Server automatically checks out the sources onto the integration machine, initiates a build, and finally gives out the result of the build [Fowler, 2006]. Only when the committer is notified (usually via email) of the result is he or she done with the commit (assuming it was a successful build) [Fowler, 2006]. CruiseControl is an example of a Continuous Integration Server.

The most difficult thing to sort out for a team introducing Continuous Integration is usually the habit of fixing the mainline right away when the build of it breaks [Fowler, 2006]. This is an
important prerequisite for one of the main points of working with Continuous Integration – you should always be developing on a known and stable base [Fowler, 2006].

**2.4.6 Keep the build fast**

Practicing Continuous Integration means to provide rapid feedback [Fowler, 2006]. For this to be possible the build time must not take to long and according to Fowler the Extreme Programming guideline of a ten minute build is perfectly within reason [Fowler, 2006].

**2.4.7 Everyone can see what is happening**

A vital part of Continuous Integration is communication and one of the most important things to communicate is the state of the mainline build [Fowler, 2006]. One way of solving this is to use a Continuous Integration Server web site. This can for example display a history of changes made and who is making a build. Another advantage with this communications approach is that people not co-located can also see the status of a project [Fowler, 2006]. CruiseControl provides this sort of web site.

**2.5 Build Automation**

The reason for discussing this topic in more detail (besides as a part of Continuous Integration above) is that implementing Continuous Integration with the help of CruiseControl makes it necessary having an automated build. CruiseControl makes use of this automated build in the integration process [Pearman and Goodwill, 2005].

What you do not want is someone appointed to doing half-hour-long builds on a daily or weekly basis [Schuh, 2004: p71]. Instead the goal ought to be to reduce the build process to a simple push-of-a-button action which every programmer is allowed to use [Schuh, 2004: p71]. Within the Extreme Programming methodology, build automation is referred to as **The Ten-Minute Build**. What it means is that you should be able to “automatically build the whole system and run all of the tests in ten minutes” [Beck and Andres, 2004: p49]. In other words Extreme Programming also stresses the fact that testing should be part of the automated build.

Beck and Andres in their book on Extreme Programming argue that automated builds are of higher value than manual ones [Beck and Andres, 2004: p49]. This is because when stress levels rise the manual build is less likely to be done and if done, then not very well. Thus the build process may result in more errors and consequently more stress. If you can build automatically it becomes a stress reliever because it makes the process of checking your changes much simpler. It is easier to check if something is wrong.

There are two criteria the build automation should fulfil [Schuh, 2004: p72]. First, regardless of what component or interface a programmer is working on the build should entail all the code related to the application. By doing this, correctness of assumptions, recent checkouts and unknown dependencies are verified. Secondly, the build should fulfil the criteria of “push-of-a-button”. What this means is that the build process should be easily portable to a programmer’s workstation, very simple to learn, and not be time consuming [Schuh, 2004: p72].

**2.6 Software Testing**

This section will start with a discussion of general concepts and techniques for software testing. Following is a discussion of a topic more specific for this project – **Automated Testing**, which further will lead to the discussion of **Automated Integration Testing**, one of the main issues in this master’s project.
2.6.1 Software testing in general

What is software testing? Myers defines this in a concise way: “Testing is the process of executing a program with the intent of finding errors” [Myers, 2004: p6]. This stands in contrast to one of the more common and false definitions: “Testing is the process of demonstrating that errors are not present” [Myers, 2004: p6]. The difference between these two definitions is very important to keep in mind when working within the field of software testing, because it can make a profound difference in the success of the testing efforts [Myers, 2004: p6].

A way to reinforce the proper definition of testing is how to define a successful test case. Myers argues a test case that finds an error is successful because it has proven to be a valuable investment [Myers, 2004: p7]. Further, consider an unsuccessful test case as one that causes a program to produce a correct result without finding any errors [Myers, 2004: p7]. When trying to accept these definitions consider the fact that the assumption of a program with zero errors basically is unrealistic [Myers, 2004: p7].

2.6.2 Testing on different levels

Testing can be done on several different levels. Some of these levels are unit testing, integration testing, system testing and acceptance testing. Further discussions will consider only the ones essential for this work – unit testing and integration testing. Why unit testing is essential becomes clear in the following two subsections.

2.6.2.1 Unit testing

A unit can be defined as “the smallest possible testable software component” [Burnstein, 2003: p137]. Unit testing is about testing these small parts of a program – individual subprograms, subroutines, or procedures [Myers, 2004: p91].

There is some difference when viewing unit testing from a procedural respectively an object-oriented perspective. Traditionally a unit is viewed as a function or procedure, implemented in a procedural (imperative) programming language [Burnstein, 2003: p137]. The choice for a unit in object-oriented languages has been suggested to be both the method and the class/object [Burnstein, 2003: p137].

When unit testing, it is important to consider how the units will be combined to form the final program [Myers, 2004: p105]. This is important because it has impact on how to design the test cases. Either you can put the units together using a “big bang” approach or you can put them together incrementally. In the big bang approach, unit testing is performed with the help of driver and stub modules. These are needed to feed and receive data to and from the separately tested units. When each unit is tested they are put together in a “big bang” manner. In the incremental approach the next unit to be tested is first combined with previously integrated and tested units, before performing tests for that particular unit. This is known as incremental unit testing or integration [Myers, 2004: p105].

2.6.2.2 Integration testing

Integration testing is according to Myers not so often regarded as a separate testing step [Myers, 2004: p128]. This statement is in some way asserted by the literature in the area of testing since deeper discussions on the subject of integration testing are difficult to find. Myers for example, only refers to the area as an implicit part of incremental unit testing (see above) and does not discuss it further [Myers, 2004: p128].

Burnstein however, discusses the area more thoroughly [Burnstein, 2003]. She says among other things that in procedural-oriented systems integration testing works best as an iterative process [Burnstein, 2003: p153]. This iterative process is described by Burnstein in the same way incremental unit testing is described by Myers (see above) [Burnstein, 2003: p153]. Though with the adding of what exactly is tested – the interfaces and functionality of the new unit in combination with the previously integrated units. This implies that the two processes are
in fact the same (remember Myers also refers to integration testing as an implicit part of incremental unit testing [Myers, 2004: p128]) and that may explain to some part why integration testing is not commonly discussed in testing literature. The only difference one might see is that in addition to the individual tests for the functionality of each individual unit, tests are added to test the interfaces (the communication or interactions) between the units.

Some differences exist when speaking of integration testing for object-oriented systems. For these systems the integration process is driven by assembly of the classes into cooperating groups or clusters [Burnstein, 2003: p153, p158]. A cluster is a group of related classes which may work together (cooperate) to support a required functionality for the complete system [Burnstein, 2003: p158]. Or as Siegel describes it, a cluster is a collection of two or more software components that relate to each other through messaging relationships [Siegel, 2004: p236]. Usually you test the simpler groups of classes first and then combine these to higher-level groups until you have assembled the whole system [Burnstein, 2003: p153]. In other words, when integrating object-oriented systems it usually is not done one unit at a time as it is with procedural-oriented systems [Burnstein, 2003: p161]. More to it, Siegel speaks of an “integration test suite” as a series of tests run against the integrated cluster of software components [Siegel, 1996: p236]. Usually the tests in the integration test suite test specific components and their integration with other components comprising the cluster.

As a summary of the discussion brought about here this thesis states its own, general definition of integration testing for object-oriented systems:

Integration testing is the testing of the functionality of individual units when combined with other units. An integration test suite should exist that tests the functionality of each individual unit and the interfaces (or the messaging relationships) between the units.

2.7 Automated Testing

What is automated testing and what are its goals? A general or traditional answer to this question can be found in Chillarege’s article Software Testing Best Practices [Chillarege, 1999]. The article describes 28 best practices that supposedly contribute to improved software testing. Of these, the ones directly related to automated testing are Automated Environment Generation, Automated Test Generation and Automated Test Execution. According to the first of these three, it is important to automate the process of setting up the environment, running the test cases, recording the results and then automatically reconfigure to a new environment [Chillarege, 1999]. This is because the setting up of the test environment, is a fairly time-consuming task, especially considering when you have to test for different operating systems, versions etc. [Chillarege, 1999].

The second one of the above mentioned practices (Automated Test Generation) speaks of the need to automate the generation of test cases [Chillarege, 1999]. This is mainly motivated by the fact that writing the test cases manually can be 30% of the testing task [Chillarege, 1999].

Finally the last, above mentioned practice – Automated Test Execution can help to answer what automated testing is. Its goal is to minimize the amount of manual work involved in test execution and gain higher coverage through a larger number of test cases [Chillarege, 1999].

2.7.1 Automated integration testing

The discussion that follows here will speak firstly of automated unit testing. The reason for this is that no formal discussion of automated integration testing is to be found in the literature and therefore here will be discussed through the principles of automated unit testing. The formerly given definition of integration testing will also be aiding this discussion.
2.7.1.1 Automated unit testing

Automated Unit Testing is one of the agile practices and can be seen as the cornerstone of most successful agile teams [Schuh, 2004: p108]. Different authors seem to have quite different views of what automated unit testing is. For example, Vaaraniemi in his article on the subject defines the unit testing process to be automated when a suite of unit tests is run as part of the build process [Vaaraniemi, 2003]. This is according to Vaaraniemi, the way you get the full benefit from unit tests [Vaaraniemi, 2003]. Further when reading Pearman and Goodwill’s book *Pro .NET 2.0 Extreme Programming* (which is a more practical, hands-on kind of book) automated unit testing is specifically defined through the combined use of different tools [Pearman and Goodwill, 2005]. The authors specifically speak of CruiseControl as a means of wrapping other tools in automation or as a tool that “will allow you to take all the other tools you have acquired and automate their use” [Pearman and Goodwill, 2005: p96]. What this means practically is that CruiseControl runs your build tool automatically and the tests in turn, are run as part of the build process.

To derive a formal definition of what automated unit testing means, below is a list summarizing the most important criteria, according to the literature studied:

- Automated unit testing requires (at its most basic) a suite of tests that rely on restorable data and can be run at the push of a button [Schuh, 2004: p108]. One way to create and use the test suite is by the use of some unit testing framework (for example NUnit [NUnit] or JUnit [JUnit]).
- Automated unit tests should be run as part of the build process [Vaaraniemi, 2003].
- The build that runs the unit tests should be automated [Schuh, 2004: p109].
- The unit test code is part of the system in the meaning that it also, is committed to the version control tool. This makes it possible to integrate the unit testing with the build [Vaaraniemi 2003].
- The build (including the unit testing process) is run automatically according to some schema (time interval) and/or criteria (for example someone commits changes) [Beck and Andres, 2004: p50].

With help of the criteria above, the following formal definition of automated unit testing for object-oriented systems is established to apply in this master’s project:

*Automated unit testing involves having a unit test suite relying on restorable data. The unit test suite is run automatically as part of the build process. This implies the build process also, must be run automatically and preferably according to a predefined schedule. It is assumed the test suite is part of the system (the test suite is under version control together with the rest of the system).*

2.7.1.2 Automated integration testing

The general definition of integration testing reached earlier, speaks of testing the functionality of the individual units when combined with other units. This should imply that unit testing can be used for testing the specific functionality for each respective unit. But the definition also says the interfaces between the units must be tested. However, because of the characteristics of object-oriented systems (and assuming the unit tests are properly designed and implemented) the interfaces should be tested implicitly when testing the functionality of the individual units. Assuming the different units is integrated with each other.

The specific characteristic referred to here is the natural dependence between units in object-oriented systems – one unit depends upon or makes use of one or more, other units when executing its functionality. Say for example, a unit test exists for the functionality of a specific unit “A” which in turn, must make use of a unit “B” to retrieve some data. Further assume these units have been integrated. If this test does not pass it could mean the functionality of A (or B)
is implemented wrongly. But it could also mean the interface or, the messaging relationship between the two units is wrongly implemented. Consequently the interface (or messaging relationship between units A and B) is implicitly tested when running this unit test.

Having drawn the conclusions above, this author finds that the combined use of the definitions for integration testing and automated unit testing (given in previous sections) will make a proper definition for automated integration testing. Consequently this thesis’ definition of automated integration testing becomes:

**Automated Integration Testing is accomplished by adherence to the definition of integration testing in combination with the definition of automated unit testing.**

Most importantly this means that the integration test suite consists of unit tests testing the functionality of the individual units as well as the interfaces between the units and that this integration test suite should rely on restorable data and be part of the system. The automatization part for the running of this test suite comes from the definition of automated unit testing.

### 2.7.2 What to automate?

When generally thinking about automating software testing, it is important to choose what tests to automate. Meszaros et al., in the article *The Test Automation Manifesto* expresses this consideration in a clear and concise way: “Look at what you are spending a lot of time testing manually. This is what you should consider automating” [Meszaros et al., 2003].

Boehmer and Patterson, in their article *Software Test Automation – Developing an Infrastructure Designed for Success* further define in more detail what tests to automate. Among these are [Boehmer and Patterson, 2001]:

1. **Regression tests** – only tests that will be repeated are worth automating. These tests give the return on investment in automation.
2. **Tests for stable applications** – the argument for this is that if the application changes, the automated tests must also change and therefore only tests for stable applications should be automated.
3. **Non-time dependent tests** – you should not automate tests with complex timing issues. This is often easier done in a manual way.
4. **Repetitive tests** – repetitive and boring tests should be automated so as not to inhibit creativity among testers.

### 2.8 CruiseControl

As complementary information, this section contains a brief description of the target for this master’s project – CruiseControl.

In short CruiseControl is an automated continuous integration server [CC], but it really consists of a suite of applications [Pearman and Goodwill, 2005: p83]. At its core is the CruiseControl Server and this is an automated integration server [CC]. Other than this is the CCTray, a client system tray application. This lets you see the state of your project from the perspective of the integration server [Pearman Goodwill, 2005: p83]. Also, there is the Web Dashboard, a web application that allows you to see the state of the project from the integration server’s perspective [Pearman and Goodwill, 2005: p83].

Pearman and Goodwill state CruiseControl integrates several tools together and because of this is often referred to as an integration server [Pearman and Goodwill, 2005: p83]. It is true that it integrates several tools, but in this master’s project CruiseControl will be referred to as an Automated Continuous Integration Server. Mainly because the tool is based on the Extreme
Programming practice Continuous Integration and because the server’s main functionality is to automate the integration process [CC].

However, the fact that CruiseControl integrates several tools together is important. This makes it practically possible to automatically build and test your system. CruiseControl monitors the source code repository of the project and every time a developer commits a new set of modifications (for example a new unit of code) CruiseControl will check out all of the existing source code, build it using for example NAnt which in turn runs the tests with the help of a compatible testing tool (for example NUnit or MSTest) [Pearman and Goodwill, 2005: p83]. It is also possible to integrate some coverage and metrics tools with CruiseControl, like for example NCover and Vil [CC].
3. Methodology

The work performed had the form of an evaluation. Roughly, it was divided into three parts:

1. Deduction of a model consisting of specially selected criteria, against which the evaluation was to be performed
2. Practical use of CruiseControl in a, at Brainpool selected project
3. Evaluation of CruiseControl against the established model, during and after 2.

3.1 About the Evaluation Model

The evaluation model is a collection of a number of criteria against which the evaluation of CruiseControl was performed. These criteria more or less concerned practical issues regarding usage of CruiseControl as a software testing tool. Deriving the evaluation model, the deliberate assumption was made that CruiseControl was a pure software testing tool (when in fact is a continuous integration server). Brainpool wanted to know if CruiseControl can be used as a tool for performing direct test work or more exactly, if and how CruiseControl can be used to implement automated integration testing.

The full evaluation model can be viewed in appendix 1.

3.2 Performance of the Evaluation

This section shortly describes how the evaluation work was conducted.

3.2.1 Theoretical studies

The theoretical part of the evaluation work mainly concerned studying the documentation that accompanied the install package for CruiseControl. But also reading the ccnet-user group [ccnet-user] as well as initiating discussions in this group. Also the company supervising the development of CruiseControl was investigated through the aid of internet resources.

This theoretical investigation mainly aimed at evaluating CruiseControl according to some of the priority 2 criteria in the evaluation model (Documentation, Learnability and Supportability).

3.2.2 Practical work

In short the practical work performed, involved using CruiseControl on a continuous basis, in a specially selected project. In more detail the work consisted of the following parts in the order they appear below:

1. Practical studies of the system under test
2. Configuration of CruiseControl for selected project
3. Implementation of test cases, with help of the NUnit framework for unit testing
4. Continuous adding (integration) of new test cases to the system under test, thus enabling CruiseControl to make use of these test cases to build and test the system under integration
5. Evaluation of CruiseControl during the steps above, according to the criteria in the evaluation model
4. Evaluation Results

In this section the results of the evaluation with accompanying analysis and conclusions are presented. The sub headings are derived directly from the criteria in the evaluation model (see appendix 1).

4.1 Performance of CruiseControl

Performance of CruiseControl was defined through four, specific questions at issue. These questions are presented below as their own sub headings together with answers deduced as results from the evaluation work. For further explanation to this criterion and specific evaluation details, the reader is referred to the evaluation model in appendix 1.

At the end a summary is presented trying to answer the overall question of performance for CruiseControl.

4.1.1 Time per test iteration

Could CruiseControl noticeably reduce the elapsed time per test iteration?

Traditionally a test-iteration is testing that is repeated, or iterated, multiple times with the goal of achieving higher quality and/or coverage with each iteration. One test-iteration includes everything from planning and extracting tests to implementing the tests and executing them.

No evaluation work was performed according to this criterion. The reason for this was its non-applicability. A test-iteration is, as understood by this author, a very traditional way of performing your testing, in that it is seen as a separate phase of the development process. This is not the way CruiseControl is intended to be used. Rather a continuous testing process should be implemented, as to get the most benefit of using CruiseControl as part of the development process. It should be continuous in the meaning that new tests are created and added to the system throughout the development process. This way you get a more secure development process in that CruiseControl continuously runs the increasingly bigger test suites, and consequently acts as a safeguard against introducing new bugs into the system. The reason for having this criterion as part of the evaluation model from the beginning was mainly due to this author’s initial misunderstanding of how CruiseControl should be used.

However, when speaking of time for testing, this section should be the proper one to discuss what was noticed when running this work’s implemented tests via CruiseControl. From the beginning it was assumed that CruiseControl would not have any greater impact on the time for running the implemented tests. This was assumed because it runs them via the NUnit console test runner [NUnit] which in other words should have most significance for the running time. It was revealed that the running of the test cases via CruiseControl took approximately double the time as when running them locally in the NUnit graphical test runner (see appendix 2 for view of the NUnit graphical test runner).

On average, running the 51 test cases locally took 8.17 seconds and via CruiseControl the average was 16.64 seconds. There is no sure conclusion to why this was the case other than the most natural one that instead of having one process active when running the tests, two processes are active concurrently (CruiseControl and NUnit).

In this case the increase in time was no problem – 16 seconds instead of 8 did not make much difference. However it could be a problem if the number of test cases and their complexity increase and you want to adhere to the practices of continuous integration. One of these says that every commit should build the mainline on an integration machine (CruiseControl), which further means that the commit should not be considered done until the integration build
succeeds on the integration machine (CruiseControl). Conclusively, if execution of the tests via CruiseControl would take too long it may become frustrating just making a commit (this could be seen as a good argument for the ten-minute-build – one of the primary practices of Extreme Programming [Beck and Andres, 2004: p49]).

However, when new to using CruiseControl (as was the case for Brainpool) it may be enough to schedule only a nightly build (including the tests) via CruiseControl. Then the tests may take their time to run and the results may be dealt with in the morning, when people come to work.

Conclusively, to evaluate CruiseControl according to the current criterion was not doable. However, it was noted that running tests via CruiseControl, in this case took longer time than running them locally. This could mainly be of concern if wanting to implement continuous integration “by the book”. Otherwise this should not be a problem.

4.1.2 Costs of testing

Could CruiseControl noticeably reduce costs of testing by reducing manual labour?

The manual labour referring to practical testing in this work consisted of two major parts:

1. Analyzing and planning – to get a basic understanding of the system to be tested and establish how to best implement the integration test suite (the 51 test cases)
2. Implementation of the integration test suite – with help of the NUnit testing framework (the 51 test cases)

Of the total time for practical test work in this master’s project (1 and 2) the implementation part stood for approximately 62% (65 of totally 105 hours). Chillarege in his article Software Testing Best Practices mentions that writing the test cases manually can be 30% of the testing task [Chillarege, 1999]. The percentage in this case was double that. This could confirm that the test implementation consumes a great amount of resources.

What is mentioned above made one wish CruiseControl foremost could aid the implementation part of the test work. But of course this was not possible as CruiseControl is a continuous integration server and does not provide any functionality to ease the test implementation. Neither could CruiseControl reduce any other manual labour involved when performing the testing.

You could however picture a scenario where you did not have time waiting for the tests to finish their execution, when run locally (in NUnit). They would simply take too long to wait for. In this case you could commit the tests and CruiseControl would run them for you and the results of the test execution could be inspected any time thereafter, through the web dashboard, email or other.

The only real manual labour reduced was the running of your tests. However, this became a bit to farfetched as this was not a big issue. Running the tests (when not run via CruiseControl) simply meant loading an assembly-file into the NUnit graphical test runner and pushing a button to start the tests. In other words the running of the tests was already fully automated.

The final answer to the question above became “no” – CruiseControl could not reduce costs of testing by reducing manual labour. This was obviously because CruiseControl is not a tool designed to provide functionality for reducing manual labour when testing.

\[3\] However, bear in mind that the percentage in this case probably was higher than normal and this is likely to be due to this author’s initial inexperience with writing unit tests. Also, this author had never worked in the .NET environment or written anything in the Visual C# [Visual C#] language before and conclusively these facts have naturally put additional “not to be accounted for” time to the writing of tests.
4.1.3 Test coverage
Was CruiseControl able to noticeably improve test coverage?

As mentioned in the theory part of this thesis, according to Chillarege the goal with automated test execution is to minimize the amount of manual work involved in test execution and gain higher test coverage through a larger number of test cases. CruiseControl could fully automate the process of executing your tests. It did this by executing the NUnit tool, which in turn executed the actual tests. Thus the test execution process indeed became a very simplified process and enabled getting a higher coverage in the way that you could execute a high number of tests. However, automating the test execution was far from enough for gaining higher coverage, as was clearly shown in this work. The tests still had to be implemented. As presented in the previous section this took quite some time in this work and consequently the time you have at your disposal for testing obviously have high affect on the finally reached coverage. To some extent it might be thought a test code generation tool could aid this process. This was however out of scope for this master’s project and considering the complexity involved in the nature of the test cases implemented, this would most probably not have been a good solution anyway.

As became illustrated in this work the degree of coverage was also highly affected by the choice of design of your test cases. When planning for how to best design the test cases a conclusion was reached about testing from the business layer of the system (see figure 1 in section 1.2 Problem Definition). This became a means for gaining higher coverage because it involved executing functionality in all the layers from and with the business layer down to the database layer. As opposed to if the test cases would have been implemented to start from the web service layer.

Conclusively the answer to the question posed at the beginning of this section became “no”. As mentioned, CruiseControl to a high degree simplified the test execution process, but this was not enough for gaining a higher coverage. Using CruiseControl still required an initial investment in relatively big amounts of time (and money) for implementing tests that gave high coverage. However, if making sure the tests were developed to need minimal maintenance this initial investment in automatization hopefully could give value for the money. This came from the ability of CruiseControl to act as a safeguard against introducing new bugs into the system when making changes to it. This was done completely automatically by CruiseControl, thus allowing for developers to accidentally forget testing locally (at least to some part).

4.1.4 Testing frequency
Could CruiseControl allow testing to happen more frequently?

The answer was “yes”, CruiseControl did allow for a more frequent testing. However, this was in a more restricted form. CruiseControl allowed for more frequent testing in that it could run existing tests according to a preferred schema, for example every time someone committed a change to the system under integration or once every 24 hours. This meant the same tests were run every time (except of course, if someone committed new tests to be added to the already existing tests) thus the more restricted form of frequent testing.

More preferably frequent testing would have meant to be able to test your system often and in a different way every time. However this would probably have meant iterative planning, designing, implementing and so forth, which is nothing CruiseControl could contribute to in any way. However, the form of frequent testing CruiseControl did provide was of high value in that it acted as a safeguard against introducing new bugs into the system, when for example making changes to it (shortly mentioned in previous section). A developer could forget to update his/hers working copy before running the tests locally and thereby getting a “false” pass of the tests, because of not having the latest version of all the files. Another developer could have made changes in parallel which would collide with the changes of the first developer. CruiseControl ensured the system was tested with the latest updated files in place. However,
this assumed the developers remembered to commit their changes to the system. Otherwise CruiseControl would not reach them and consequently test with the wrong system files in place. This made for the conclusion that having good and well-known routines for updating and committing was necessary to get the full benefit of CruiseControl.

To summarize CruiseControl could be said to allow for more frequent testing. But this frequent testing only came in the form of regression testing – it could only execute existing test cases repeatedly to assure changes to the system had not introduced new errors. Further this assumed every person making any changes to the system – to the system itself, adding tests or other – also remembered updating and committing to the repository regularly.

4.1.5 Summary

Performance of CruiseControl was defined through the four above discussed questions at issue. One of these was deemed not applicable. For the remaining three, only one of these could be affected – the testing frequency. This could be increased through the use of CruiseControl (however restricted to regression testing).

If interpreting these results exactly according to the derived evaluation model, the conclusion became that CruiseControl was a low performance tool – only one yes-answer out of four. However, this was only true for its performance regarding certain aspects of testing. It should be noted that the results here did not apply, nor give justice to the tool regarding its obvious and overall, intended functionality – continuous integration. This was of course due to the fact that the evaluation model was derived with the deliberate assumption of CruiseControl being an explicit software testing tool. This was really not the case, but the reason for this deliberate assumption was explained in section 3.1 The Evaluation Model.

4.2 Process Definition Ability

Could CruiseControl help define the software development process and then especially the sub process of testing, by making it more visible and explicitly stated? If this was the case, could it reduce dependence on the few who know the development (testing) process and also allow testing to be done by staff with less skill?

CruiseControl’s main functionality is to continuously build and test your system under development. In other words the tool could not have any impact on phases where no code had yet been implemented. Consequently the phases preceding the ones where code implementation and testing occurred could not be made more visible by the use of CruiseControl and therefore also, not be more explicitly stated.

When the actual code implementation and/or test phases began CruiseControl could give some, limited insight into the ongoing process. This was enabled mainly through the web dashboard where reports of among others build and test results could be reviewed\(^4\). These reports were however quite limited in detail. They only gave a rough overview of for example, how many projects had been built and how many tests had been run. Further, for the testing aspect it was among others displayed how many tests existed for the system, how many had passed or failed etc. This was quite limited but if used with the NCover tool [NCover] a better insight into the testing process could be given. The NCover tool was never used in this work but considering its functionality one should be able to draw the conclusion of better insight into the testing process. In appendices 3 and 4 respectively, an example of a build report respectively a unit test report is given.

\(^4\) CruiseControl could also make use of, for example the code coverage tool NCover [NCover] and the code analysis tool FxCop [FxCop] which to some extent could give further insight into how well your testing covers the system code respectively how well designed your system code is.
What was presented by CruiseControl via the web dashboard (or email and other) mainly revealed the status for the build and test processes of the system. This provided a clear (however rough) picture of the overall status for your respective projects. But it was not enough to help define the development process. The belief of this author is that increased visibility into the development process may be achieved by for example, introducing well documented principles for ones process. And making sure these are easily available and well understood by the employees of the company.

Consequently it was not plausible CruiseControl would enable people, not familiar with the development and/or testing process, to better understand and contribute to these processes. This was mainly because the small extent to which the visibility into the development/testing process did increase, was not nearly enough to provide enough insight. This could instead be solved by, for example, having documented principles for ones development process.

### 4.3 Non-Intrusiveness

Was the behaviour of the software under test the same with automation as without?

It was known from the beginning of this work that the time frame did not really allow for making any real investigations regarding this criterion. This criterion could only be properly evaluated if using CruiseControl during a longer time span and preferably, from the beginning of a new project. However, because Brainpool as well as the author considered this as an important criterion some intuitive conclusion were drawn, based on theoretical and practical experience of the tool.

CruiseControl ran independently on a machine other than the local developer’s stations. It monitored the source repository to watch for changes. When changes occurred CruiseControl checked out a working copy of the system, built and tested it and finally reported the results.

The testing was (as already mentioned) done via NUnit which meant this tool had an implicit responsibility for the automation of testing and consequently was the tool that should be further investigated to answer the question posed at the beginning of this section. Even further the answer should come from how the test cases, run by NUnit were implemented.

To enable a smoother discussion on the subject at hand remember the overall structure of the tested system (discussed and illustrated in section 1.2 Problem Definition). A rough description of how the manual testing was previously performed was also given. In more detail it was done in two consecutive ways. First on a very low level in the developed system, stored procedures residing in the database layer where invoked. Secondly the functions in the user interface layer where invoked (to test the same thing) in order to get a more thorough testing. This last way became the integration testing – testing of the classes and methods working together between the different layers.

The automated test cases were implemented to invoke methods in the business layer of the system. These methods were the same that were invoked from the user interface layer, when testing manually through this lastly mentioned layer. This naturally implied that exactly the same integration testing was performed with the automated test cases as with the manual ones, with one exception – that the user interface layer was not tested. Assuming the user interface layer was not implemented correctly the software under test would behave differently when testing automatically, as opposed to when testing manually.

This delved down to one main conclusion (which was quite obvious from the beginning). Because the role of CruiseControl, when it comes to testing, really only was to run the test cases (via NUnit) it could not affect the behaviour of the software being tested. The main influence on the software had to come from the implementation specific details of the test cases.
4.4 Ease-of-Use

Because ease-of-use was a highly subjective criterion it was divided into several sub criteria in the evaluation model. In section 4.4.1 Sub Criteria for Ease-of-Use, below, each of these sub criteria are discussed individually with belonging conclusions. In the next, following section, 4.4.2 Summary, these conclusions are put together to reach a summarizing conclusion for the overall ease-of-use criterion.

4.4.1 Sub criteria for ease-of-use

Below, each of the sub criteria for ease-of-use is discussed individually, with belonging conclusions.

4.4.1.1 Ease of installation

Was CruiseControl easy to install and set up?

Depending on what type of installation you chose for CruiseControl it was more or less easy to perform the installation. The installation could be done from three formats [CC] (as summarized below):

1. **Two installers** – these were conventional Windows installation programs. One was for the CruiseControl server and the Web Dashboard. The other was for the optional CCTray utility. The installers performed some of the setup work, such as copying files to the proper locations and setting up a virtual directory for the Web Dashboard.

2. **A binary zip file** – this contained the same files as the installers. The difference was that the user could have more control over what to do with the files.

3. **A source zip** – this allowed the user to read the source code and make his or hers’ own changes to it.

Only the two Windows installers have been used in this evaluation. The reason for this was that the installation part was not considered as important as it is a one time occurrence. Not much could be said about this form of installation – it was two simple Windows installation programs. Consequently the user only had to follow some simple instructions to install the server and this only took a couple of minutes. For the one starting up with CruiseControl this form of installation seemed quite enough as it was a good starting point when being new to using this kind of tool (a continuous integration server). There was no need to for example, use the source zip when you have not even learned how the tool functions.

However, when being new to CruiseControl, you could not neglect the additional time it took to install other programs to be used with CruiseControl. CruiseControl is (as mentioned earlier) also referred to as an “integration server” in the meaning that it integrates several tools together [Pearman and Goodwill, 2005: p83]. For example, it can make use of NAnt and NUnit for building the system and testing it [Pearman and Goodwill, 2005: p83]. It was these tools and others (for a full list see the documentation of CruiseControl [CC]) that had to be additionally installed (and learned if not used before).

There was also the need of having the Microsoft .NET Framework installed. But it was assumed in this work that a potential user of CruiseControl already develops its software within this environment and therefore does not need to face the uphill of installing and learning .NET. Therefore no further discussion of this is undertaken.

To summarize, the installation of CruiseControl could be both easy and more difficult, depending on which of the installers were used. If new to CruiseControl, the two simple Windows installers should be used (as in this case). In that case it was a no timer to install the tool.
4.4.1.2 Configuration as a recurring activity

Was it easy to configure CruiseControl after having passed the initial learning phase together with your first “try-out” configuration?

This criterion was considered to be of high importance by Brainpool and this author. The reason for this was that configuration of CruiseControl becomes a recurring activity when wanting to use the tool for future, upcoming projects. It was therefore of need to establish if reconfiguration of the tool could be a time-consuming and consequently expensive activity. If so, the use of the tool would not be as attractive.

In short CruiseControl was configured via an XML-file and this file could contain configuration specifics for several different projects in parallel. Consequently the XML configuration file contained a number of main blocks corresponding to the different projects monitored by CruiseControl. Further, in these main blocks the details for each respective project were configured. This could for example concern what build tool or source control system would be used by CruiseControl and where to find the files to build for a specific project.

The main result (which to a great extent was a subjective judgment) was that once you get past the initial learning phase of configuring CruiseControl or, once you have configured CruiseControl for your first project, it was a very straightforward process to configure CruiseControl for a new project. When doing this, it mainly became a “copy-paste” process (of the old project configuration block) in which the most difficult part was to change some folder paths defining for example, where the project files were located in the source control system. This could give rise to duplication of much configuration code. However, it could be solved by creating own entities that were then invoked by different project blocks. For example the tag defining where CruiseControl could find the source control executable was the same for every configured project and could therefore be defined in an own entity. This was quite similar to the features of an object-oriented programming language. To summarize it was quite handy and simple to use.

As soon as you had understood the different elements of the configuration file, CruiseControl was easy to configure for new projects. This went somewhat in contrast with the one and only other, found evaluation of CruiseControl that was summarized in the article *Automation for the people: Choosing a Continuous Integration server* by Paul Duvall [Duvall, 2006]. This was more or less based on the opinions of different users of CruiseControl and came to the conclusion that CruiseControl was quite troublesome to configure. This opinion was in comparison with continuous integration servers that could be configured via intuitive wizards or web console applications. This author has not made a comparative study, but intuitively it seemed simpler to configure via a wizard or similar. However, as was established in this evaluation (as well as in the other, mentioned study) – configuring via an xml file gave the user greater control and insight into the functionality of the tool. This was, according to this author of greater importance, and considering it took this author less than one hour to configure CruiseControl for a new project (after having passed the uphill of understanding the configuration file) it should be quite acceptable considering the benefits you are given with bigger control over the tool.

What should to be kept in mind though, regarding the discussion brought about here was that configuration work in this case concerned only one “real” project and some simpler try-out projects. The try-out projects did not contain any implementation details making the configuration of CruiseControl troublesome. However, the real project (MEK) for example, contained files that could not be handled via CruiseControl which consequently made for some adjustments of the project itself. This was necessary to be able to run the project under CruiseControl. Consequently the results here should be taken with some restraint as the evaluation itself was quite limited.

An example of what the configuration file could look like can be seen in appendix 3.
4.4.1.3 Documentation

How well documented was CruiseControl?

This part of the report discusses the documentation belonging to the specific release of CruiseControl evaluated in this master’s project. This documentation was roughly the same as the one on the CruiseControl home page [CC]. The difference was that the latter one referred to the most recent development build and the former was an export of the online documentation from the main CruiseControl website [CC]. For the sake of simplicity (and availability) and because these documents only differed in small details this discussion references the online documentation. This makes it easier for the one wanting to control the discussion in this report.

The documentation accompanying the download consisted mainly of an overall summary that explained among others the principles of continuous integration and the main functionality of CruiseControl (and its accompanying tools CCTray and the Web Dashboard). The main focus was on how to set the tool up and how to use it. The usual “Getting Started” section was also available as well as more detailed descriptions on how to use and configure the tool. For the reader further interested in the contents of the documentation he/she is referred to the online site for this – http://confluence.public.thoughtworks.org/display/CCNET/Documentation.

What was most important to have in mind when reading the documentation was that it belonged to an open-source tool and as such its structure and details differed quite much from that of a commercially developed tool (as understood by this author). The overall contents of the documentation were easy to follow, but as you dug into details it got more difficult to understand the meanings of for example, some configuration details. As already mentioned, you configured CruiseControl via an XML configuration file and this contained different parameters that had to be correctly set for the tool to work properly. The descriptions of these parameters were clear at first but could after some further reading easily be misinterpreted before they were really understood. Consequently these configuration descriptions could have been more elaborative so as to make the documentation more comprehensible.

Because CruiseControl is developed as an open-source project, contributions can be made to its documentation by almost anyone. Even though this is done in a relatively controlled fashion [CC] it should open up for the possibility that the documentation becomes not so easily understood by others. This could explain why the detailed descriptions in the documentation sometimes were difficult to understand (as discussed recently).

On the other hand the overall documentation was just enough. This is something that could not be said about many of the different commercial software tools available on the market. The documentations for these are often overdone, giving the reader an overhead of information that may be troublesome to search. At least this is the opinion of this author. What is meant by this is that the documentation for CruiseControl (on an overall scale) contained only what was essential to know when configuring and using the tool. It was (as discussed earlier) only some of the detailed descriptions that could be more elaborate.

Further, the overall structure of the documentation was quite good and easy to navigate through. It was relatively easy to find what you were looking for. Most importantly, it was easy to find the “getting started” section. From there you could easily navigate further into details of configuration and use of CruiseControl. With the downloaded documentation you also got an index summary in which it was very easy to locate the exact part of the documentation you wanted to find. This was of major importance, because after having investigated the “getting started” section and wanting to go directly to some part of the documentation you wanted to do this the easy way – through an alphabetical index.

4.4.1.4 Learnability

Was it possible to learn how to use CruiseControl in a short time? Did resources exist that could aid the learning process? Examples of resources could be books, online documentation and alike.
Whether CruiseControl was easy to learn how to use or not depends heavily on whether you were a developer, (just) building and checking in code to the source control system or if you were the administrator of the tool. In the first case you did not use the tool in a direct way. You checked in your changes to your repository (with Subversion in this work) and CruiseControl automatically detected the changes in the repository, checked the whole system out and performed the tasks it was configured to do (for example building and testing the system). If the build and/or tests broke the developer was notified via, for example email (by CruiseControl) and he/she was by this urged to complement his or hers changes to the system until the build and tests passed. To summarize, when not being the administrator of the tool it was a very short learning curve (or none at all). What the developer really only needs, is to be informed that CruiseControl is to be used and to know how it works in general. In short, it should be enough for the developer to learn the principles and practices of continuous integration (which maybe is not as straightforward) and why he or she gets emails about a failing build or alike. It is also good to know about the web dashboard, where an overview of the different projects under integration is given, together with specific details for the projects (for example build and test results).

If you were to be the administrator of the tool it was a quite different thing. In this case it was a steeper learning curve, mainly because you (naturally) needed a more thorough understanding of the tool and how it worked. As the concept of understanding is quite abstract it will here be defined as when a person has learned the tool to the point that he/she can use it without needing to consult the basic documentation, other than on rare occasions. This is a quite forced definition but it enables a smoother and more explicit discussion of the subject.

Learnability of CruiseControl could be divided into two different categories of learning – the learning of CruiseControl itself and the learning of the other tools, used by CruiseControl (for example MSBuild and NUnit). Depending on to which degree a person new to CruiseControl, was familiar with these other tools, the learning curve could differ widely. In this case for example, the evaluator was not familiar with using the MSBuild tool, nor the NUnit tool. These were to be used (by CruiseControl) to build respectively test the project under integration. Consequently it took some additional time to understand and properly use these tools. (Furthermore this evaluator was not familiar with the .NET environment, which therefore also added some time to the total learning curve.) If the additional time it took to learn other tools was ignored, the learning curve for CruiseControl itself was not so very steep. It was difficult to set an exact time frame for how long it took to properly understand CruiseControl. An estimate of approximately one week (five working days) can be made with the results from this evaluation (ignoring extra time to learn other tools and considering this was a full time work). If you count in the additional time it took to install the other tools and to learn them (plus getting used to the .NET environment) it took about eight days to get a basic understanding and to be able to use CruiseControl in a relatively controlled fashion. But the important thing here was not the exact time it took to learn CruiseControl. This was a one time occurrence – once you had learned to properly use the tool (and additional tools) you had this knowledge (and hopefully it stayed).

The most important conclusion regarding this criterion was whether or not the appointed administrator of the tool was already familiar with the tools CruiseControl used as auxiliary for its work (for example MSBuild, NAnt and NUnit). If not, one had to include the additional time it would take to learn and understand these auxiliary tools.

To aid the learning process there was, of course the accompanying documentation (discussed above). Other than this there was the ccnet-user mailing list [ccnet-user]. This could be used to search information complementary to the documentation. There was also the CruiseControl.Net community site, which is a public forum for discussion of issues and features [CC]. However the latter was a rather poor complement to the main documentation. The former though, was a quite comprehensive source of information where you relatively fast could get an answer not found anywhere else.
4.4.1.5 Operability

What did the features of CruiseControl look like and feel? Were they cumbersome to use, or prone to user error?

Firstly, because CruiseControl is a server program it was difficult to speak of features in the sense of ones that could be directly accessed via some user interface. Instead features were referred to as what CruiseControl was capable of being configured to do (or what other tools CruiseControl could make use of). Secondly, because there were a vast amount of features only the most important ones are discussed here. For the one interested in a full list of the features for CruiseControl, http://confluence.public.thoughtworks.org/display/CC/CI+Feature+Matrix gives a good overview (this feature matrix is more exactly a comparison between different types of continuous integration servers).

The Source Control Feature – was maybe the most extensive feature of CruiseControl. This was in the sense of how many different types of source control systems CruiseControl could be configured to use. They were 19 to the number and among these were for example the widely known Subversion [Subversion] and CVS [CVS], but also the more specialized Alienbrain\(^5\) [Alienbrain]. In this work subversion was used, and to great satisfaction as many source control provider capabilities of Subversion are implemented for CruiseControl. Examples of these were that CruiseControl (with the help of subversion) automatically could check out source files for a new project and/or automatically update the checked out source files. For other source control systems CruiseControl did not achieve equally many source control provider capabilities as subversion (and CVS), but the mere fact that CruiseControl offered the capability to make use of as many as 19 different source control systems implied CruiseControl provides high flexibility.

The Build Feature – most importantly, CruiseControl offered the ability to use both NAnt and MSBuild for building the projects under integration. This was a big advantage because it gave you greater flexibility. If you wanted a simple build approach you could use MSBuild. When developing in the .NET environment a special solution-file is created when building in Visual Studio. This file (or other special project files – csproj-files) could then be given as argument to instruct MSBuild when this tool was used to build a project on the CruiseControl server. If you wanted a more flexible build approach NAnt could be used to define your own build scripts which NAnt makes use of when invoked by CruiseControl. In this evaluation it was considered especially important by Brainpool that CruiseControl could make use of MSBuild because they did not want the extra burden of having to write build scripts with NAnt. Instead they wanted to make use of the solution- and project-files, automatically produced by Visual Studio.

The Test Feature – or more precise, support for running unit tests via the NUnit tool was a well functioning feature. And necessary, because if you could not run your unit tests via CruiseControl the tool itself would not provide as much value. Further it would be more difficult maintaining to the principles of continuous integration. One of these says you should have an (extensive) unit test suite [Fowler, 2006] that is run (preferably) every time someone commits a change to the source code [Beck and Andres. 2004: p49].

The Reporting Feature – the main reporting capability for CruiseControl came with the web dashboard. This gave an overview of the projects currently under integration. A snapshot of the start page for the dashboard is given in appendix 6.

Via this application you could see specific details of a chosen project – like for example build results (build logs) and test results. But most importantly it presented whether builds (including testing) for your projects were successful or not. The graphic layout of the dashboard was very straightforward and simple to grasp – you got a good and simple overview of the status for the projects under integration.

---

\(^{5}\) Alienbrain is a source control system for professional media and entertainment projects [Alienbrain].
By default the dashboard was configured to show quite many details (accessed via links) but luckily this could be easily reconfigured with the help of the xml configuration file for the dashboard. Sometimes it was also necessary to make changes in special XSL-files and some CSS-files. It was not very often that you needed to make these changes. But still, some simpler way of configuring the dashboard would be desirable, for example via a wizard or web console application.

Another reporting feature was the ability to configure CruiseControl to send out mail with build results to chosen recipients. This was never tried out because only one person was evaluating the tool and consequently a try-out was not considered necessary. However theoretically, this special feature could be considered of high value because it enables notifying specially selected project members. And assuming they read there mail on a regular basis it makes for a hopefully faster reaction to failed builds. This is to be considered as of high importance because it enables having a known and stable code base on which development takes place. This is the whole point of working with continuous integration [Fowler, 2006].

Yet another reporting feature of CruiseControl was the CCTray. This was really an application in its own sense. But for simplicity it is here referred to as only an extra reporting feature, because this is all it really was – a simple reporting application, essentially providing the same information as the Web Dashboard. Although the information was very much scaled down to give the user quick feedback for the projects under integration. For example the CCTray icon was shown in different colours depending on the build status of the projects under integration and it also enabled balloon notifications for build status reporting. This application seemed a bit superfluous and has not been paid very much attention in the evaluation of CruiseControl. Although, it did provide quick and simple feedback upon the build status of your projects, which may be good if you just wanted a quick glance of the overall integration status (without having to open up the dashboard web page).

4.4.1.6 Supportability

What was the level of support for CruiseControl? Especially the long-term supportability risk of adopting CruiseControl was here considered. It is an open-source tool and as such there is the risk of the developers or maintainers abandoning it [Toth 2006]. It is clear that this could impact downstream maintenance effort and cost negatively [Toth 2006].

Because CruiseControl was an open-source project, you could not get support in the same way as for a commercial tool. It was also explicitly stated (on the CruiseControl home page) that no support was guaranteed because it was an open-source-tool [CC]

The special support you could get for CruiseControl was quite limited. It came mainly from the ccnet-user group [ccnet-user]. But you also had the CruiseControl.Net community site [CC] and a number of blogs (found via the CruiseControl.NET “People” site [CC]). However, these two, lastly mentioned information resources were very limited. The blogs were, because they did not provide much information essential for the configuration of and continuous work with CruiseControl. The community site because it was a very scarce resource, in the way that there existed little information complementary to the standard documentation. Further, not much activity was seen in this community.

The ccnet-user group was really the only resource good enough for providing additional support. It was a comparably comprehensive resource for information where you reasonably fast could find answers not found in the documentation. It also had a very good amount of activity in it and a steady increasing number of new members. Beginning of May 2007 it had 539 members. Midst of April 2008 the numbers of members was 981. During the course of almost one year nearly 442 new users of CruiseControl have joined this network. Also a steady amount of messages were posted to it every month (an average of 242 messages per month). This made the ccnet-user group a very healthy information resource in which much support could be found for your initial as well as every-day work with CruiseControl.
Of great importance was the long-term supportability risk. This could be greatly affected if the developers and/or the maintainers abandoned CruiseControl in the future. How high was this risk of the developers or maintainers abandoning it? As it was not a readily quantifiable one it was difficult to give a straightforward answer to the posed question. However some arguments could be found, speaking for support of CruiseControl also in the near future:

1. CruiseControl was developed, and is maintained, by ThoughtWorks, which is a seemingly well established company [ThoughtWorks]. For example it has offices spread out on different locations in the world. Some of these locations are Australia, Canada, USA and India [ThoughtWorks]. How this may ensure future support was not exactly clear, but it could be supposed that a big and well established company has greater possibilities to perform a non-profit project, as CruiseControl.

2. ThoughtWorks is further an (allegedly) active member and supporter of the Open Source Software (OSS) community [ThoughtWorks]. This could imply, them wanting to continue the CruiseControl project. Otherwise, perhaps their statement of support for the OSS community could be debated.

3. The CruiseControl open source project development is entirely funded by ThoughtWorks [ThoughtWorks]. Assuming the software market does not decline suddenly and/or rapidly it could be concluded that they will continue to do so.

These arguments do not have a direct connection with the future supportability of CruiseControl. This is because CruiseControl is not a commercial tool and therefore support from ThoughtWorks cannot be guaranteed. But they do give some reassurance of ThoughtWorks not abandoning the project and as long as this is the case there should further, be some reassurance that users of the tool will continue to exist. In other words the current user community (primarily represented by the ccnet-user group) should stay active with a steady amount of members. And in that way also in the future, be able to provide support.

Further, the amount of activity in the ccnet-user group (discussed with statistics above) indicated this is a stable resource for support and as long it stays active it may be stated that the support for CruiseControl is a good one.

When speaking of what may negatively affect the future support for CruiseControl the main factor found was that ThoughtWorks does not make any profit of the project (because of being an open-source project). Therefore, if (or when) harsher times come it is plausible that the company will stop the funding of the project. This should not have any direct effect on the support for the tool. Because as mentioned earlier, they do not guarantee support in the common sense (because CruiseControl is not a commercial tool). However, as discussed above it may have the indirect effect that the users of the tool will also abandon it and therefore no longer contribute to, firstly the ccnet-user group (the main support source). Consequently it may die out and leave the ones deciding to continue their use of the tool with much poorer support availability.

It is difficult to draw any real conclusion from this discussion but maybe it is possible to state that as long as the software market does not plunge to deep, there may still be a high level of support for the tool. If the market goes down ThoughtWorks may abandon the tool and consequently many users as well, causing a lower level of support from the ccnet-user group. Though it is plausible, because CruiseControl is an open-source tool, some users and/or current developers will continue to develop and maintain the tool on their own, without support from the formerly responsible company. This may depend on how well established the tool has become at the time of a market plunge. Considering the current number of users (981, midst of April 2008) in the ccnet-user group and the amount of activity (~233 messages per month, midst of April 2008) in this it may be stated that the tool is reasonably well established and consequently may continue to be used even after a possible abandonment from ThoughtWorks. If so, the support should live on.
4.4.2 Summary

On an overall scale CruiseControl became a relatively easy-to-use tool. Installation was easy (of course depending on choice of method) but more importantly it was easy to reconfigure for new projects. This was important because it is a recurring activity and therefore must not consume many resources. The tool was also flexible in the way that it could make use of many tools for its integration work. Most importantly it could make use of a variety of different source control systems, different build tools (most noticeable MSBuild and NAnt but building via command scripts is also possible) depending on how you prefer your build to be done, plus you could run your unit tests via the NUnit test runner.

However some factors came up, lowering the overall ease-of-use for CruiseControl. These were as follows below:

1. The formulations in the documentation – some of these would preferably be more elaborate to ease the understanding of certain configuration details for the continuous integration server.

2. Acquaintance with other tools – this was important because CruiseControl, as a means to achieve its overall functionality (continuous integration) made use of auxiliary tools for building and testing (among others). If the administrator had at least a basic understanding of these tools it made for a faster and easier configuration/maintenance work and consequently saved time.

3. CruiseControl is an open-source tool – and as such, especially the long-term supportability risk had to be taken into account. However for the time being, this risk of lessened support should be low. Partly because of the ongoing, positive development in the software industry (as assessed by this author). But mainly because of CruiseControl’s reasonably big and active user group, which should stay active for some time even with possible abandonment of developers/maintainers.

Of 1-3 above, 3 should be the most important to consider if thinking of incorporating CruiseControl into ones development process. However, as this risk seemed low for the time being the overall high grade for ease-of-use spoke for, using CruiseControl to improve the integration process.

4.5 Project Preconditions

Were there any preconditions a project should fulfil to enable an easier use of CruiseControl for the integration work? This was something Brainpool had interest of knowing as complementary information to this evaluation.

Early in the configuration process of CruiseControl for project MEK it was noticed that you could not just take all the files in the particular project and put them under integration by CruiseControl. This depended mainly on the choice to use MSBuild as build tool (when building via CruiseControl). As the case was for project MEK it contained virtually every source file of importance to the project in one big package – application files (csproj-files), files for scripting databases (dbp-files) and setup-files (vdproj-files). When building locally in Visual Studio this was no problem. However MSBuild (which was used by CruiseControl) was not compatible with database and setup files. This meant they could not be included when building with MSBuild (via CruiseControl) and consequently they could not be part of the integration process via CruiseControl. The fact of not being able to include every system file did not pose any problem for the overall project (MEK) as the important thing was that the application files could be included. These were the ones making up the application part of the system and were consequently what was of most interest to continuously build and test.

Because of the complications presented above (and if using CruiseControl for upcoming projects) it was recommended by this author that future projects are to be more structured. This
means that instead of having all files in one package they should be clearly divided into their own individual projects. In this case it could for example have meant dividing the overall project in one project for the application itself, one for the database part and one for the setup part.

If structuring ones project the way that was recommended the configuration of CruiseControl may become an even easier task. Further it should have the positive effect that new developers (getting involved in the project some time after its initiation) get a better overview of the project and consequently faster may grasp the contents of it.
5. Automated Integration Testing Capability of CruiseControl

Was CruiseControl able to implement automated integration testing according to the definition given in the theory summary? Automated integration testing has been defined through the definitions for integration testing and automated unit testing and consequently these have been examined. The definitions are here repeated as a reminder:

Integration testing:

> Integration testing is the testing of the functionality of individual units when combined with other units. An integration test suite should exist that tests the functionality of each individual unit and the interfaces (or the messaging relationships) between the units.

Automated unit testing:

> Automated unit testing involves having a unit test suite relying on restorable data. The unit test suite is run automatically as part of the build process. This implies the build process also, must be run automatically and preferably according to a predefined schedule. It is assumed the test suite is part of the system (the test suite is under version control together with the rest of the system).

A reasonably extensive unit test suite was implemented using the NUnit framework. The implementation details for this were to some part discussed in section 4.1.4 Test Coverage. More details need not be given but it is here repeated that the test suite was directed at specifically chosen units in the business layer. Further, as these business layer units were fully integrated with the rest of the system, the test cases were implemented to make use of other units in lower layers of the system, thus enabling testing of not only the functionality for individual units, but also the integration with other units – or, the interfaces between these.

The test suite was made part of the system by simply putting it under version control together with the rest of the tested system.

The data on which the tests relied was restorable in the sense that it resided in a copy of the development database. The development database in turn was backed up every night which meant that if the tests own instance of the database where to be corrupted or wiped out by mistake, it would be an easy task to restore the test database. The task would simply consist of making a new copy of the development database.

However, the tests in this work were not run as part of the build process. The reason for this was the choosing of MSBuild as build tool which made it more difficult to integrate the running of the tests as part of the build process. As opposed to using NAnt where you make your own build scripts in which you then may define to run your tests at a certain point (as part of the build process). Though this could easily be solved by configuring CruiseControl to run the unit test suite as its own task. At the time being this author has found no downsides with this approach.

In the same way MSBuild was run as its own task to build the system and this was done just before the testing task was run. These both tasks were easily configured to be run by CruiseControl according to a preferred schedule. In this case every 24 hours was considered good to start with. Running of the build and test process on smaller intervals (for example with every new commit) was something that would be considered in the future.
The only real contribution made by CruiseControl to enable automated integration testing was to automatically run the build and test process on a predefined schedule. But of course the tool also enabled the automatic discovery of new test cases when these were finished and committed to the repository. This enabled CruiseControl to check these out and run them as part of the integration process. This was also the way the work was conducted – as each test case was completed it was committed for CruiseControl to check out and add to the automatic running with the rest of the already existing test cases.

To summarize you could say that CruiseControl added an extra layer of automation to the integration testing process. The NUnit test runner represented the first layer – this tool enabled running the tests by the push of a button, which to a high degree automated the integration testing process (this should be done by every developer locally, before checking in changes to the developed system). What CruiseControl did was running the tests purely automatically, without human intervention. Thus the “extra” layer of automation. The main benefit of this extra layer was that it worked as an extra security check if someone forgot to build and test locally. If this was the case CruiseControl made sure a build and test run would be performed and if any of it broke the developer would be notified.

Conclusively CruiseControl to a high degree enabled an automatization of the integration testing process. However, this was only in the way that CruiseControl automated the running of the test cases – by enabling no human intervention what so ever (consequently one must not forget that the design and implementation of tests for the integration testing initially required a relatively big amount of manual labour). The main contribution of CruiseControl was that the tool could run your implemented tests on a continuous basis without any manual effort. If the tests failed the developers would automatically be notified by CruiseControl. This way CruiseControl acted as a safeguard against introducing new bugs into the system (as also discussed earlier).

Further, for the conclusion drawn above to be valid, it was assumed that every potential developer and tester in a project remember to commit their changes and test cases. If they would not, CruiseControl would not be able to make use of them and consequently the functionality of CruiseControl could not fully be taken advantage of.
6. Other Conclusions

This section summarizes some theoretical conclusions drawn, regarding two specific questions at issue that were of special interest to Brainpool. The questions were as follows:

1. Can use of CruiseControl increase quality of developed products?
2. Can use of CruiseControl increase cost effectiveness of the overall development process?

Regarding question at issue number 1 above, four main points were found speaking for an increased quality of products being developed with CruiseControl:

1. CruiseControl can build on a continuous basis – this enables faster discovery of build errors, thus in turn enabling earlier fixes of these errors.
2. CruiseControl can test on a continuous basis – this enables faster discovery of implementation errors (not all, but some), thus in turn enabling earlier fixes of these errors.
3. CruiseControl can safeguard against introducing new bugs into an existing system when making changes to it – this is because of the continuous testing performed by CruiseControl.
4. CruiseControl can run tools like FxCop and NCover – this enables simple and automatic feedback of code coverage and design conformance of the developed system.

These are more closely discussed in section 6.1 below.

It was also found possible to give a positive answer to question at issue number 2. The main argument speaking for increased cost effectiveness, even though it was a purely theoretical one, was that errors may be found earlier when using CruiseControl. Using CruiseControl enabled fast notification of, for example, whether a build has succeeded or not (via the web dashboard, email or other). This in turn enabled one to quickly react to this broken build and thus repair it as soon as possible. This should lead to a higher cost effectiveness regarding both development and testing. This will be further elaborated on in section 6.2 below.

However, some practical experiences were also made, speaking against a more cost effective development process. These were as follows:

1. The uphill of learning and configuring CruiseControl for the first time
2. The extra effort needed to learn or better understand additional development tools
3. The fact that CruiseControl is an open-source tool

Of these the first two may be seen as negligible as they are one time occurrences and consequently should not affect cost effectiveness on a longer time scale. Number 3 was more uncertain whether it could affect cost effectiveness negatively or not. It was in this work concluded that it may also be considered negligible and the reason for this will be explained more thoroughly in section 6.2 below.

As a summary, it was concluded that CruiseControl should be able to increase quality of developed products as well as increase cost effectiveness of the overall development process. These conclusions are further motivated in sections 6.1 and 6.2 below.
6.1 Product Quality Using CruiseControl?

Regarding product quality, using CruiseControl, four main points were established to speak for an increased quality. The first two of these stated that CruiseControl can build and test on a continuous basis. This should enable earlier disclosure of build and implementation errors which in turn should enable one to fix these errors earlier than when not using CruiseControl. As a natural consequence it should also be easier to fix these errors because what has been done since the last build/test run is fresher in memory. When not using CruiseControl, building and testing the developed system may become a more ad hoc and less recurring activity. This could probably result in later discovery of bugs and build errors which are more difficult to find because much may have happened since the last build/test run. To summarize, using CruiseControl to continuously build and test your system should ensure a more stable code base (because of the early discovery and fixing of build/implementation errors) which in turn should contribute to a higher quality of the developed system.

In a similar way the continuous and repeated testing performed by CruiseControl every time someone commits changes should safeguard against introducing new bugs into an already existing system. These changes could for example be functionality or performance enhancements ordered by the customer of the system after it has been used for a while.

An important reservation must be stated regarding the conclusions discussed so far – mainly this concerns CruiseControl’s ability to test continuously and to safeguard against introducing new bugs. This sounds very good but also assumes tests are written and added to the system on a continuous basis. If not, it is obvious that the continuous testing performed by CruiseControl will have little effect, because of lacking enough comprehensive test suites that will thoroughly test the system – during development and after. It also assumes the tests are properly designed and implemented as to get the best coverage and most effective testing. The only solution to this problem as seen by this author is to allocate enough time for designing and implementing tests. Both during development and after when changes are made to the system in use (it could be necessary to add new tests depending on the changes made).

CruiseControl is able to run tools like FxCop and NCover. Because of this the process of getting feedback on how well a system conforms to design guidelines and how well the unit test suite covers the system code is somewhat simplified. CruiseControl enables easy access to the reports as they may be given via for example email or the web dashboard. This could also function as a simple reminder to overview the system design and quality of unit test suites, which could be more easily neglected if not using CruiseControl. In this case it is up to the developers themselves to manually run these tools and this may result in a more ad hoc routine for doing so, or it may not be done at all.

An important remark should be made about having MSBuild as preferred choice for building your system via CruiseControl (as was the case in this work). When using MSBuild it was more troublesome to get CruiseControl to run, for example FxCop and NCover. If more easily wanting to run these tools, NAnt was a necessary build tool. This in turn was because FxCop and NCover must be run as part of the build process. CruiseControl was at the time for this evaluation work, not yet able to run these tools as their own tasks. If wanting to use NCover or FxCop you were then limited to only running these tools locally. This could be neglected when in stress and thus cause one to further neglect improving your test cases and then also in the final product quality. A solution to this could be to run NCover (or FxCop) via a smaller NAnt script in CruiseControl. However this has not been tried because the problem was discovered late in the work progress and time was not enough to fully try this.

It has been concluded in section 6 above and more thoroughly explained why in this section, that CruiseControl should be able to increase quality of developed products. These conclusions and discussions are however more or less theoretical, which is why a more well-founded answer to this question is only possible through more quantitative investigations of CruiseControl.
Furthermore CruiseControl should be used in parallel system development projects and during a longer time than this master’s project. It would also be necessary to use the tool from the start of these projects in order to get a more trustworthy answer to this question.

**6.2 Cost Effectiveness Using CruiseControl?**

Could it be motivated from a cost effectiveness point of view to integrate CruiseControl into a development environment similar to the one at Brainpool? This question concerned the general development process, including both, the system code implementation, testing and alike. It has already been stated that CruiseControl may be able to do this and it will be further motivated why in this chapter.

The main factor encountered (theoretically) that was thought could affect the cost effectiveness in a positive way was that *errors may be found earlier*. This has already been elaborated on to some extent and will here be illustrated via the following example:

Consider a scenario where CruiseControl is not used. Developer Bill checks out a copy of system Y in order to implement some new functionality. When he is done he builds and tests this new functionality. The only problem is that, at the same time developer Mary has implemented another set of new functionality and committed to the version control system (after building and testing locally). Bill forgets to update his working copy and thus he does not build his new functionality with an up to date system (including the changes of Mary). In other words it cannot be certain Bill’s functionality goes well with what Mary just implemented. After Bill has built and tested successfully he commits his changes. Now assume, no one works with the project for a week and when someone finally does, he/she discovers build errors (due to clashes between Bill’s and Mary’s changes, but this no one is aware of). In this case, because reasonably long time has passed since someone made changes to the system it may be difficult localizing the cause. In the case of using CruiseControl this error would have been discovered earlier and consequently more easily be removed (because changes made, would be fresher in mind).

Due to the limitations of this evaluation, what is claimed here cannot be backed up by quantitative arguments. A better and more realistic scenario would have been evaluating CruiseControl when configured for a system where several developers were still making changes. As was the case now, only one developer (this author) was working with the system and then not even making changes to the system itself, but “only” adding test cases to it. This could not enable the scenario described above, which in other words is a theoretical one. Though, as the scenario described above is fully realistic it should be reasonable stating that CruiseControl can increase cost effectiveness for the overall development process.

Some practical experiences were also mentioned in section 6 as possible reasons for not raising cost effectiveness in the development process. They were however considered negligible and are here further elaborated on and more thoroughly discussed as to explain why they are negligible.

First of all, *the uphill of learning and configuring CruiseControl for the first time* was noticed. Approximately 50 hours were used for the initial learning phase and trying out CruiseControl for smaller “try out” projects (configuring CruiseControl and letting it run while continuously adding changes in the form of system changes and new tests to the try-out projects). This should, according to this author be a reasonably acceptable time. But the theory is that, if already accustomed to the .NET environment and skilled in unit testing (using some XUnit framework) this work could be limited to less than 2 days of work. However, because the initial learning phase is a one time occurrence the time it takes should not affect cost effectiveness for the development process in the long term perspective and therefore be a negligible factor.

In this category (*the uphill of learning and configuring CruiseControl for the first time*) is also included an additional 50 hours needed to configure CruiseControl for the specific project.
MEK. However, if at least two of the following three criteria where to be fulfilled when configuring CruiseControl for a new project, it is believed this time could be reduced to at most one day of work:

1. **The person performing the configuration has a good basic understanding of the system that is/has or is about to be developed** – as the case was now this author had no knowledge whatsoever of the system, other than the one he acquired in parallel during the configuration work. (Of course knowledge was also acquired when introduced to the system at the beginning of the work, though this was not enough to get the basic understanding necessary. The “Learn by doing” concept was indeed applicable in this case.) Consequently the parallel learning of the system, when configuring CruiseControl added extra (unnecessary) time to this work.

2. **CruiseControl is set up from the beginning of the project (as opposed to now when the project was already completed)** – with this criterion fulfilled the criterion for knowledge of the system to develop, should not be as necessary for the person making the configuration. In this case it should be easy to set up an initial/simple configuration that may be as easily reconfigured as the system development proceeds and new needs are discovered, implicitly requiring changes to the CruiseControl configuration.

3. **The project for which CruiseControl is to be configured, is well structured** – this was discussed in section 4.5.1 Project Preconditions.

The second practical experience was the extra effort needed to learn or better understand additional development tools. In order to get the most out of CruiseControl the tool should be configured to perform more duties than just building the system. In this case CruiseControl was configured to make use of NUnit for running an extensive unit test suite for project MEK, thus adding value in the form of assurance of correctness for the system when making changes to it. But you would get even more value if using tools like NCover and/or FxCop. However, this adds time for learning and understanding these additional tools and may consequently affect cost effectiveness in the short term perspective. Especially if for example (and as in this case) you want to get the assurance an extensive test suite may provide. If you have no experience in this area it may be quite time consuming to first learn how to write effective tests to run under CruiseControl. But then again, this time should in the long term perspective give a more cost effective development. This is because, when wanting to make changes to a system for which an extensive test suite exists and is run under CruiseControl you easily get assurance of whether your changes have made a negative impact on the system. In other words, learning additional development tools is a one time occurrence and should quite easily be turned into a profit in the future and is therefore also a negligible factor.

The only real thing speaking against CruiseControl is the fact that it is an open-source tool. This fact should not be neglected, and was especially noticed when configuring the reporting functionality for CruiseControl. In order to produce styled results CruiseControl uses the XML files produced by the various tools, invoked by CruiseControl. However, MSBuild does not come with a XML logger by default and therefore needs to be complemented with an external logger. An XML-logger was provided for download from the CruiseControl homepage, but when this author was about to download it, this XML-logger was not available. A manual search for this logger had to be conducted, including reading mailing lists and posting own questions regarding where the logger (or an alternative one) could be found. This went on up until the point when it was considered to write an own logger. However luckily, this was not needed in the end as an alternative logger was finally found. But the point is, because CruiseControl is an open-source tool, additional problems may arise that are not as easily solved as when coming from the use of a commercial tool. This fact may indeed impact cost effectiveness negatively. As in this case the problem could probably have been solved easier by for example calling a support desk (if CruiseControl was a commercial tool). To summarize, the experience made, implies you should not readily dismiss the fact that CruiseControl is an open-source tool. But CruiseControl do have a reasonably big and active user community from which you mostly can get an answer to whatever problem you may have when using the tool. It was
through this user community the alternative XML logger was found, if however not as easy as one might wish.
7. Overall Conclusions

The overall concerns of Brainpool, when assigning this evaluation of CruiseControl were:

1. To know if CruiseControl could be of value in the software development process of the company, and then especially the sub process of testing?
2. To have the integration testing fully automated for a specially selected project.

As for 1 the collected conclusions of this evaluation has not found any real reason to why Brainpool should not include CruiseControl in their software development process. It does not provide any means to simplify any manual test work, except for the running of implemented tests. But it is relatively easy to use and even if not having any tests to be run by CruiseControl the tool will still provide value by continuously building the system under integration. This way build failures may be discovered earlier and thus be solved earlier, resulting in a (hopefully) more stable system. In other words provide a means for achieving higher quality of the developed systems.

When integration testing with CruiseControl the only real contribution made of the tool is to rerun the tests according to a predefined schema. However this provides value similar to the one of continuous builds of the system – bugs introduced by changes to the system are faster discovered and may then more easily be attended to. Thus this should result in a more cost effective integration test process. Recall though, that this also assumes extensive test suites of high quality.

Of course there is the initial investment required if wanting to use CruiseControl as a means for automating part of the integration testing for a new project. This initial investment was in this work represented by among others the configuration work and initial learning but foremost by the implementation part of the test cases, which stood for approximately 62% of the total time for test work (although it would probably be lower with higher experience of the tester). The configuration task should not consume much time as the initial learning phase has been passed. And lastly mentioned is also a one time occurrence.

Regarding 2 above this was successfully accomplished. An integration test suite was implemented using the NUnit framework and was then run fully automatically by CruiseControl. The main result becomes that, when making changes to this system in the future CruiseControl will act as a safeguard. CruiseControl will do this by continuously building and testing the system and thus guarding against build failures and new bugs in the system.
8. Recommendations

When performing this evaluation the major limitations were as follows below:

Only one person was involved in using the tool (undersigned) – A more realistic approach would have been to have CruiseControl integrate a project where several persons continuously performed actual development and integration work. This way a more realistic evaluation would have been made possible. Although this brings with it the risk of making the project more risk prone which may not be what a company wants as a consequence of a master’s project.

The integration work could not be performed entirely according to the principles and practices of continuous integration – this was because the project, for which CruiseControl was to be applied, was a working copy of an already finished project. This enabled only the continuous adding of new test cases to the project, no system specific code. Preferably CruiseControl would be used from the beginning of a new, real project in order to properly see how the integration work could proceed when aided by the usage of CruiseControl. However, this would probably not have been doable within the time frame of a master’s project. Further, practical issues like timing the start of the new project together with start of the master’s project and alike would have been difficult to achieve.

As a result of these limitations it is recommended that additional and more realistic evaluation work is conducted. This should be done in parallel with continuous, real-life use of the tool in some development project. In this way it should be possible to more extensively answer, especially the questions of process definition ability, cost effectiveness and product quality. As well as investigating what was not in this evaluation, due to time constraints.

Consequently a real try-out of CruiseControl, starting from the beginning of a new project, where several developers are involved in the continuous integration process, should be performed. This is something that may be tried out in upcoming projects at Brainpool.
9. Work Not Performed

Due to time constraints the following questions at issue from the evaluation model were not investigated:

1. Are there some conditions, under which CruiseControl can be run in an optimal way, as to gain the most benefit from CruiseControl?

2. Are there any preconditions that the development environment should fulfil to be able to make the project start and initial use of CruiseControl as efficient as possible?

Further, the question of whether there are any preconditions that a new project should fulfil to be able to make the project start and initial use of CruiseControl as efficient as possible was investigated only partially? The reason for this was also time constraints.

1 and 2 above as well as project preconditions should be evaluated with future use of CruiseControl at Brainpool.
List of References

Books


Articles

Bach J. (1999) Test Automation Snake Oil


Fowler M. (2006) Continuous Integration
http://martinfowler.com/articles/continuousIntegration.html
http://www.google.com/search?q=cache:3g_giQPnxKsJ:www.clrstream.com/index.php%3Foption%3Dcom_docman%26task%3Ddocclick%26Itemid%3D0%26bid%3D6%26limitstart%3D0%26limit%3D5+The+Test+Automation+Manifesto&hl=sv&ct=clnk&cd=1&gl=se&client=firefox-a

Pettichord B. (2001) Seven Steps to Test Automation Success
http://www.io.com/~wazmo/papers/seven_steps.html


http://www.codeproject.com/gen/design/onunittesting.asp

WWW

Agile Alliance. Driven by the values and principles of “The manifesto for Agile Software Development

Alienbrain. Asset management system for artists

CC. Main site for CruiseControl.NET
http://confluence.public.thoughtworks.org/display/CCNET/Welcome+to+CruiseControl.NET, last visited July 12, 2007

ccnet-user. User group for CruiseControl (Google Groups)
http://groups.google.com/ag/group/ccnet-user, last visited July 09, 2007

CVS. Version control system

FxCop. Code analysis tool

JUnit. Unit testing framework for Java

MSBuild. Build system for Microsoft and Visual Studio

NAnt. .NET build tool

NCover. Homepage for NCover – a code coverage tool for .NET
http://ncover.org/site/, last visited July 10, 2007

.NET. Software development framework

NUnit. Unit testing framework for all .NET languages

SOA. Service Oriented Architecture

Subversion. Version control system
ThoughtWorks. A global IT consultancy company
http://www.thoughtworks.com/, last visited July 12, 2007

Visual C#. Object-oriented programming language
Appendix 1 – The Evaluation Model

The purpose of the model described in this document, is to support the evaluation of the continuous integration server CruiseControl.Net (from here referred to only as CruiseControl). Further, the higher goal of the evaluation is to establish how well CruiseControl fulfils the target, specific for this master’s project – automated integration testing.

The model consists of a number of criteria according to which the evaluation will be performed. Further these criteria are categorized into priority one, respectively priority two criteria. The priority one criteria are deemed the most important, by Brainpool AB. They are also most easily observed and quantitatively measurable. The priority two criteria demand a more subjective evaluation. They are therefore deemed less important and may not be included in the evaluation if time is running out.

The way the criteria were established was firstly by the study of a vast amount of different criteria in books and articles. These books and articles mainly discussed the topics of test automation and tools for this. The criteria were, afterwards filtered with the help of discussions between Brainpool AB and the model creator. To some extent the filtering of criteria was also made by studying opinions of relevant user groups – of CruiseControl or merely of techniques applied by CruiseControl (e.g. continuous integration).

Whenever a criterion is referenced in the document this means that the criterion was derived from or with the help of the referenced literature.

Priority One Criteria
( Observable/Quantitative )

These are the observable/quantitatively measurable criteria. They are priority one criteria which means the evaluation of CruiseControl will focus on these criteria as the most important ones. The reason for this is twofold – Brainpool AB found these criteria to be the most interesting ones plus Brainpool AB and the model creator found the observable/quantitative criteria to be the ones most realistic to evaluate, considering the time frame of the evaluation work.

The priority one criteria are threefold and are presented under their own headings, below.

Performance of CruiseControl

What this criterion means is defined by the following questions at issue (derived and interpreted from Pettichord’s discussion on reasons for why people choose to automate their tests, in his article Seven Steps to Test Automation Success [Pettichord, 2001]):

1. Can CruiseControl noticeably reduce the elapsed time per test iteration?
2. Can CruiseControl noticeably reduce costs of testing by reducing manual labour?
3. Can CruiseControl noticeably improve test coverage?
4. Can CruiseControl allow testing to happen more frequently?
These questions will be answered as simply as possible – with a “Yes” or “No” answer. A “Yes” answer to three or more of questions 1-4 will define CruiseControl as a high performance tool. A “Yes” answer to two of questions 1-4 will define CruiseControl as an average performance tool. A “Yes” answer to only one of questions 1-4 will define CruiseControl as a low performance tool.

When answering these questions with a “Yes” or a “No” the striving will be to have some quantitative data to back up the answer. For example question 1 may be measured as the difference in percent, in time, when testing with respectively without CruiseControl. Exactly what these measurements will be is not established for the time being. The reason for this is that it is considered easier (by the model creator) to establish the exact measurements to be taken, when the actual evaluation work starts.

Some comments on the four questions at issue are: 1, 3, and 4 can probably be answered through direct observations of CruiseControl in action and within the time frame of this evaluation. Though for 2, it may only be possible to draw hypothetical conclusions regarding the answer. Some direct observations may be possible, but to be able to draw any real conclusions regarding the answer to this question the performance of CruiseControl must be observed over a longer time period than what is now scheduled for this particular evaluation (20 weeks).

Important to have in mind is the subjective judgment that will also, be part of this criterion (performance of CruiseControl). For example, the present evaluator may consider a 50 percent increase in the speed of the testing process to be a “noticeable” speed increase and therefore be enough to answer this question with a “Yes”. This is a quite high increase but may yet be considered to low for someone else to justify a “Yes” answer to this particular question.

**Process definition ability**

Can CruiseControl help define the software development process and then especially the testing process [Pettichord 2001] by making it more visible and explicitly stated? If this is the case, can it reduce dependence on the few who know the development (testing [Pettichord 2001]) process and also allow testing to be done by staff with less skill [Pettichord 2001]?

**Non-intrusiveness**

What it means to evaluate CruiseControl according to this criterion, is trying to answer the question of whether the behaviour of the software under test is the same with automation as without [Bach 1999]. This is considered an important criterion by Brainpool AB and the model creator, but even so it is uncertain whether the time frame will allow for such an evaluation to take place. Therefore it is of least priority, within the “Priority One Criteria”. This further means that the “Priority Two Criteria” below, may be given a higher importance than the “Non-Intrusiveness” criterion. This is decided during the course of the actual evaluation work.

**Priority Two Criteria (Subjective)**

These are the criteria requiring a more subjective judgment in the evaluation of CruiseControl. Brainpool AB and the model creator consider these criteria to be less important because of their subjectivity. The priority one criteria (above) are considered more appropriate to use when making future decisions about a possible integration of CruiseControl in the development process. This is because these criteria are better supported by quantitative/observable data. The
subjective criteria are limited to individual judgments and are therefore not as certain as the quantitative/observable criteria. For example, one of the criteria below is *ease-of-use*. What is easy to use for one person may not be it to another.

**Ease-of-use**

Duvall, in his article *Automation for the people: Choosing a Continuous Integration Server*, says this is probably one of the most subjective criteria to consider [Duvall, 2006]. Because of this, to aid the evaluation of this particular criterion a number of sub-criteria have been listed. These are meant to be evaluated during the evaluation work and at the end be summarized and analyzed to be able to answer the overall criterion – *ease of use*.

Within the *ease-of-use* criterion the *configurability* criterion is considered the most important one (agreed upon between Brainpool AB and the model creator). Therefore the focus, when evaluating *ease-of-use* will be on this sub-criterion.

**Configurability**

Is it easy to set up new projects to work with CruiseControl? Or, formulated differently, is it easy to configure CruiseControl for different projects? The reason to the importance of this sub-criterion is that the setting up of CruiseControl for different projects is a recurring activity which therefore must work smoothly. There is the possibility that this sub-criterion is difficult to answer unless CruiseControl can be tried out for a number of different projects. Considering the time frame for a master’s project it may not be possible to do so and consequently not possible to fully evaluate CruiseControl according to this sub-criterion. So, even though it is considered as the most important sub-criterion here, chances are it may be difficult to draw any real conclusions regarding it.

**Documentation**

How well documented is CruiseControl [Kan, 2002: p384]? This will be evaluated through the study of different available documentation – developer’s documentation, user groups and others.

**Ease of installation**

Is CruiseControl easy to install and set up the first time it is to be used? As this is a one time occurrence it is not considered as important (as the configurability sub-criterion) and consequently not much focus will be on this criterion.

**Learnability**

Is it possible to learn how to use CruiseControl in a short time [Bach 1999]? Does training classes or books exist to aid the learning process [Bach 1999]? What about online documentation and alike?

**Operability**

What do the features of CruiseControl look like and feel? Are they cumbersome to use, or prone to user error [Bach 1999]?

**Supportability**

What is the level of support for CruiseControl? Is it quantifiable by some means? Especially the long-term supportability risk of adopting CruiseControl should be considered. It is an open-source tool and as such there is the risk of the developers or maintainers abandoning it [Toth 2006]. It is clear that this could impact downstream maintenance effort and cost negatively [Toth 2006].
Other Work

As a result of the discussions between Brainpool AB and the model creator, some other points were established as important for the evaluation work. These are not to be viewed as criteria, but as complementary questions to the model. It is the hope of Brainpool AB that potential answers to these questions can be derived from the overall evaluation work of CruiseControl. However it should be noticed that these questions will only be considered if the time frame allows it. The focus of this evaluation is to evaluate CruiseControl according to the above described criteria and further establish whether it at all, would be profitable to integrate CruiseControl in the development process at Brainpool AB.

The first of these questions ponders whether there are some conditions, under which CruiseControl can be run in an optimal way, as to gain the most benefit from CruiseControl? The second question concerns preconditions for a project and the development environment. In other words, are there any preconditions that a new project and the development environment should fulfil to be able to make the project start and initial use of CruiseControl as smooth and efficient as possible?
Appendix 2 – the NUnit Graphical Test Runner
Appendix 3 – Example of a Build Report, displayed in the Web Dashboard
Appendix 4 – Example of a NUnit Test Report, displayed in the Web Dashboard
Appendix 5 – The XML Configuration File

Below is an example showing what the XML configuration file for CruiseControl may look like.

```xml
<project name="BankProject">
  <!-- The Working Directory contains the checked out version of the project under integration (this is used by other blocks) -->
  <workingDirectory>
    C:\CruiseControl Projects\Bank\WorkingDirectory
  </workingDirectory>
  <!-- The Artifact Directory is the persistence location for anything you want saved from the results of the build -->
  <artifactDirectory>
    C:\CruiseControl Projects\Bank\ArtifactDirectory
  </artifactDirectory>
  <!-- Used by CCTray and the Email Publisher -->
  <webURL>
    "myWebURL"
  </webURL>
  <!-- Used by CCTray and the dashboard for providing groupings to the project. -->
  <category>Category Bank</category>
  <!-- The Interval Trigger is used to specify that an integration should be run periodically, after a certain amount of time -->
  <triggers>
    <intervalTrigger name="intervalTrigger" seconds="600" buildCondition="IfModificationExists"/>
  </triggers>
  <!-- The State Manager allows you to specify how and where data for the project is stored -->
  <state type="state" directory="C:\CruiseControl Projects\Bank"/>

  <!-- The source control block -->
  <sourcecontrol type="svn">
    <executable>
      C:\Program Files\Subversion\bin\svn.exe
    </executable>
    <trunkUrl>
      http://subversion:1030/svn/CruiseControl.NET/Bank
    </trunkUrl>
    <!-- The directory containing the locally checked out workspace -->
    <workingDirectory>
      C:\CruiseControl Projects\Bank\WorkingDirectory
    </workingDirectory>
  </sourcecontrol>
</project>
```
<workingDirectory>

<!-- Authentication for source control -->

<username>"username"</username>
<password>"password"</password>

<!-- Whether to retrieve the updates from Subversion for a particular build -->
<autoGetSource>false</autoGetSource>

<!-- Specifies root url for the WebSVN site -->

<webUrlBuilder>"rootURL"</webUrlBuilder>

<!-- Indicates that the repository should be tagged if the build succeeds -->
<tagOnSuccess>true</tagOnSuccess>

<!-- The base url for tags in your repository -->

<tagBaseUrl>http://subversion:1030/svn/CruiseControl.NET/Bank</tagBaseUrl>

<!-- Sets the timeout period for the source control operation -->
<timeout units="seconds">60</timeout>

</sourcecontrol>

<!-- used to generate the label that CruiseControl uses to identify the specific build -->
<labeller type="dateLabeller" />

</tasks>

<!-- Block defining tasks to be performed by CruiseControl -->

<msbuild>
<executable>
C:\WINDOWS\Microsoft.NET\Framework\v2.0.50727\MSBuild.exe
</executable>
</msbuild>

<!-- The directory to run MSBuild in -->
<workingDirectory>
C:\CruiseControl
Projects\Bank\WorkingDirectory
</workingDirectory>

<!-- The name of the build project to run, relative to the workingDirectory -->

<projectFile>Bank.sln</projectFile>

<!-- Any extra arguments to pass through to MSBuild -->
<buildArgs>
/v:quiet /noconlog /p:Configuration=Debug
</buildArgs>

<!-- List of the targets to run -->
<targets>Build</targets>

<!-- The full path to the assembly containing the custom logger to use -->
<logger>C:\Program Files\CruiseControl.NET\server\logger.dll</logger>

<!-- Number of seconds to wait before assuming that the process has hung and should be killed -->
<timeout>15</timeout>
</msbuild>

<!-- Instructs CruiseControl to run the unit tests contained within a collection of assemblies -->
<nunit>
<!-- Path to nunit-console.exe application. -->
<path>C:\Program Files\NUnit 2.4\bin\nunit-console.exe</path>

<!-- List of the paths to the assemblies containing the NUnit tests to be run. -->
<assemblies>
<assembly>C:\CruiseControl
Projects\Bank\WorkingDirectory\Bank.Test\bin\Debug\Bank.Test.dll</assembly>
</assemblies>
</nunit>

<!-- outputfile for NUnit test results -->
<outputfile>"nunitOutputFile"</outputfile>

<!-- The number of seconds that the nunit process will run before timing out -->
<timeout>120</timeout>
<nunit>
</nunit>

</tasks>

<!-- A set of Tasks that are run after the build is complete -->
<publishers>

<!-- Output files from different external tools (e.g. NUnit, FxCop) that need to be merged before presented -->
<merge>
  <files>
    <file>
      <!-- path to NUnit test file-->
    </file>
    <file>
      <!-- path to FxCop file-->
    </file>
  </files>
</merge>

<!-- pick up some default statistics for capturing during the build process. -->
<statistics></statistics>

<!-- Used to create the log files used by the Web Dashboard -->
<xmllogger logDir="C:\CruiseControl\Projects\Bank\ArtifactDirectory\LogDirectoryForWebDashboard" />
</publishers>

<!-- used to display project related links on the project report page of the Web Dashboard -->
<externalLinks>
  <externalLink name="My Link" url="http://somewhere" />
</externalLinks>
</project>
</cruisecontrol>
Appendix 6 – The Web Dashboard, Start Page