

# Implementation of Best-Practice Software-Engineering Setup in .net Application Development

Developing a testing environment for the CSS GmbH in the scope of a  
project lab for the Technical University of Vienna

18th August 2004

## Author

**KALS**, Stefan, 0027476, 534,  
E-Mail: [stefan@kals.net](mailto:stefan@kals.net)

## Supervisors

**SCHATTEN**, Dr. DI. Alexander, Institute of Software Technology and Interactive Systems  
E-Mail: [alexander@schatten.info](mailto:alexander@schatten.info)

**SCHWEIGER**, DI. (FH) Sven, CSS computer-systems-support GmbH  
E-Mail: [sven.schweiger@css-web.net](mailto:sven.schweiger@css-web.net)

# Contents

<b>1</b>	<b>Introduction</b>	<b>4</b>
<b>2</b>	<b>Definition of Concepts</b>	<b>4</b>
2.1	Extreme Programming (XP)	5
2.1.1	Planning Rules	5
2.1.2	Designing Rules	6
2.1.3	Coding Rules	6
2.1.4	Testing Rules	7
2.2	Test Driven Development	7
<b>3</b>	<b>The chosen concept features</b>	<b>8</b>
<b>4</b>	<b>The Coyote Project</b>	<b>8</b>
4.1	Project Description	8
4.2	Coyote Architecture	9
<b>5</b>	<b>Daily Build</b>	<b>10</b>
5.1	The Daily Build concept	10
5.2	Goals of Daily Build	10
5.3	Implementing Daily Build and Smoke Test	11
5.4	Daily Build tools	12
5.4.1	Overview of the available Daily Build tools	12
5.5	Daily Build in practice	13
<b>6</b>	<b>Black box and White box testing</b>	<b>15</b>
6.1	Black box testing	16
6.1.1	The concept of Black box testing	16
6.1.2	Goals of Black box testing	16
6.1.3	Bugs to be found by Black box testing	17
6.1.4	Implementing Black box testing	17
6.1.5	Black box testing tools	18
6.1.5.1	Overview of the available Black box testing tools	18
6.1.5.2	JUnit	18
6.1.5.3	NUnit	19
6.1.6	Black box testing in practice	20
6.2	White box testing	24
6.2.1	The concept of White box testing	24
6.2.1.1	Static Analysis	25
6.2.1.2	Dynamic Analysis	25
6.2.1.3	Code Inspection	25
6.2.2	Goals of White box testing	26
6.2.3	Bugs to be found by White box testing	26
6.2.4	White box testing tools	27
6.2.4.1	Overview of the available White box testing tools	27
6.2.4.2	FxCop	27

6.2.5	White box testing in practice . . . . .	27
<b>7</b>	<b>Performance testing</b>	<b>28</b>
7.1	The concept of Performance testing . . . . .	28
7.2	Performance testing tools . . . . .	28
7.3	Performance testing in practice . . . . .	29
<b>8</b>	<b>Conclusion</b>	<b>30</b>

## Abstract

The ever increasing complexity of contemporary software products advise the application of state-of-the-art software engineering techniques. However, the setup, configuration and implementation in the project team is a demanding task. This paper acts as a guideline through the forest of several important aspects of software development as there are: daily builds using automated testing, black box testing, white box testing and finally performance testing. While discussing features, field of application and possible goals at an abstract level, theory is supplemented by descriptions of exemplary use in a real-world .NET software project.

## 1 Introduction

This guideline evolved through the work on an extensive testing concept for a new software project at the CSS computer-systems-support GmbH company [5] in the scope of a project lab for the Technical University of Vienna [37] assisted by Alexander Schatten [39] from the Institute of Software Technology and Interactive Systems [36].

While being targeted to future use within the recently started “B-Web” (business web) called project, the testing environment was developed using a smaller subproject of B-Web for try-out purposes called “Coyote Trace Solution” (shortcut: “Coyote”). The Coyote project was finished in February 2004 and targeted to be used within the B-Web project wherefor perfectly suited to be used as a pilot project for the testing environment. Wherever a chapter “... in practice” appears in the following, one can look forward to examples of the practical use of the just before discussed testing concepts on the Coyote project. Because both the B-Web and the Coyote project are developed using Microsoft’s C#.NET programming language [16], the practical examples fit into the scope of .NET. Nevertheless this paper is held as general as possible to not limit its usability just for .NET developers.

This paper should help the interested developer team improving (or firstly implementing) their testing environment in practice covering the concepts of Daily Build, Black box and White box testing as well as a small overview of Performance testing.

## 2 Definition of Concepts

In the field of testing architecture in software engineering, many different concepts exist. While unequal in name these concepts share many equal features as there are “test-before-code” or daily build for example. For being sure about the surrounding environment, these architectural testing concepts in the field of software engineering will shortly be defined and described.

As many other companies, the CSS GmbH mainly used the relatively old software engineering concepts MSF (Microsoft Solutions Framework [15]) and some waterfall approaches. To introduce new concepts to the company while not overstraining the capacity for new techniques in the heads of the decision-makers, the planned software engineering strategy update mainly comprised automatically executable tasks. So the two widely accepted software engineering concepts Extreme Programming (see Chapter 2.1 Extreme Programming) and Test Driven Development (see Chapter 2.2 Test Driven Development) were taken into focus.

## 2.1 Extreme Programming (XP)

The Extreme Programming strategy for developing software in a different way mainly consists of a set of rules which should be obeyed to obtain the full value of more efficient development and higher quality including better customer satisfaction.

Don Wells states on his XP website “Extreme Programming: A gentle introduction.” [42] that:

“XP is different. It is a lot like a jig saw puzzle. There are many small pieces. Individually the pieces make no sense, but when combined together a complete picture can be seen.”

The whole Extreme Programming approach bases on the following rules divided into four areas:

### 2.1.1 Planning Rules

**User Stories:** Apart from analyzing the customers needs and writing long requirement documents, the customer just should write stories about what he wants the system to do for him.

**Release Planning:** The release plan is the master plan for the software project and is used for creating iteration plans for each individual iteration.

**Small Releases:** The software development team has to release iterative builds of the project to the customer often to quickly get feedback on the newly integrated features. This concept ensures that the project later not suffers from misunderstandings in early phases.

**Project Velocity:** The project velocity is measured all the time and gets considered in the iteration planning.

**Iterations:** The whole project should be divided into a dozen iterations of about one to three weeks length ensuring easy velocity measurement and quick project problem discovery.

**Iteration Planning:** On the beginning of each iteration, in the iteration meeting the work for this iteration gets defined with the help of the user stories.

**Move People:** Let the developers change their area of project they are working on to avoid coding bottlenecks and knowledge concentration/loss.

**Stand-up Meetings:** Introducing a daily stand-up meeting offers the amount of communication mostly needed but seldom practised.

**Fix XP:** When the process breaks, fix it by changing the responsible concepts to meet the project specific requirements.

## 2.1.2 Designing Rules

**Simplicity:** The simplest designs mostly are the best (and surely take the least time).

**System Metaphor:** Choose a system metaphor for ensuring the team using the same naming conventions and having all the same picture of the whole system.

**CRC Cards:** Use CRC (class, responsibilities and collaboration) cards to design the system as a team.

**Spike Solutions:** Use small prototypes (spike solutions) for discovering and solving hard design or technical problems.

**No Early Adding:** Don't introduce features into the project you think will be used later — they probably won't.

**Refactoring:** Advise the software team to refactor their code whenever and wherever possible.

## 2.1.3 Coding Rules

**Customer Availability:** The customer should be available to the development team all the time. This ensures that upcoming questions and problems are handled as quickly as possible.

**Coding Standards:** The development team should write code according to the common coding standards defined by the team to keep the code consistent and easy to understand.

**Unit Test First:** Similar to TDD (see Chapter 2.2 Test Driven Development), XP wants the developer to first write a unit test before starting to code it.

**Pair Programming:** All code included in a production release is programmed by two developers at a single computer. XP claims to produce as much functionality by using this concept as a “one programmer per one computer” strategy, but with higher quality.

**Sequential Integration:** Because of complete integration of all different software modules often cause integration problems which are difficult to detect, XP recommends sequential (step-by-step) integration.

**Integrate Often:** The programmers should integrate and release their code modules into the code repository as often as possible to ensure no bigger integration problems.

**Collective Code Ownership:** This concept allows every developer to change every code file, add functionality, fix bugs and refactor.

**Simplicity:** Keep the optimization part on till the end.

**No Overtime:** Encouraging the developers not to work overtime keeps the motivation in the project.

## 2.1.4 Testing Rules

**Have Unit Tests:** Just have unit tests for all developed classes is very important to make the concept work.

**Unit Tests Pass:** All unit tests have to pass before a module is allowed to be released.

**Arising Bugs:** When bugs are arising, ensure they are not coming back by writing unit tests for exactly that piece of code.

**Acceptance Tests:** Using the user stories, acceptance tests are developed to ensure that the system meets the customer scenarios.

## 2.2 Test Driven Development

In the last years, Test driven development (TDD) is evolving as one of the most successful software developing concept using unit testing. It utilizes a three-step concept: write test, write code, refactor, which apparently improves the process of software development.

Jeff Miller, Eric Vautier and David Vydra define Test Driven Development on their well known TDD community page [33] as follows:

”TDD is a lightweight programming methodology that emphasizes fast, incremental development and especially writing tests before writing code. Ideally these follow one another in cycles measured in minutes.”

The TDD concept recommends adding functionality (using test cases) in very small chunks, typically starting the first cycle with some simple cases like: what happens with a stock running out of cash, a list is empty or the user enters wrong data. Once these simple cases are handled, one can add more functionality, step-by-step.

While XP sets its focus on both, human code testing (e.g. “Pair Programming”) and automated code testing, TDD mainly concentrates on automated testing, especially unit testing. Both concepts agree that unit tests are essential to fast, incremental coding.

Test Driven Development (TDD) has a variety of different names [21] which all apply to the same concept as there are:

- Test First Design (TFD)
- Test First Programming (TFP)
- Test Driven Development (TDD)

Dave Chaplin [4] states that practicing TDD evolves the design of a system starting at the low level of writing unit test cases for a class: Writing those test cases and implementing the correspondent class object and methods then leads to a need for other object classes and methods. One important

rule of TDD so is: “If you can’t write test for what you are about to code, then you shouldn’t even be thinking about coding.”

According to Kent Beck, one of *the* experts in the area of XP, “test-first code tends to be more cohesive and less coupled than code in which testing isn’t a part of the intimate coding cycle.” [2] which can be seen as one of the results of Chaplin’s thesis. Another reason for this reduced coupling may be because of developing classes that are actually needed to fit the architecture requirements and testcases rather than building objects that are *thought to be needed* [21].

### 3 The chosen concept features

While some XP rules like Stand-Up Meetings, Design Simplicity, Refactoring, Customer Availability and Pair Programming already get practiced in the CSS GmbH company partly, others were chosen to be introduced to the software development team as part of the project lab using the Coyote Trace Solution project as a proof of concept:

- **Daily Build:** The software project is being built on a daily base for ensuring a consistent package at every point of time suiting to the “Integrate Often” XP concept.
- **Black box testing:** Implementing the “write test, write code” concept by writing unit test cases checking the conformity of classes to their specification. Both, TDD and XP recommend using unit testing as a main step to success.
- **White box testing:** Testing software classes using the knowledge about their innards especially by inspecting code lines for well-known issues or patterns on syntactical and semantical kind. While this item is similar to the “Coding Standards” concept of XP, it additionally fits very well into the automated daily build process.
- **Performance testing:** Inspecting the different ways of testing an application for its performance on several important tasks. While looking at functional tests mainly during the project lab, performance testing was chosen to round up the testing palette.

## 4 The Coyote Project

### 4.1 Project Description

Coyote Trace System is a framework that provides easy to use monitoring and logging capabilities for distributed and non distributed .NET applications.

To know what’s going on within an application is essential for operating software even if there are no upcoming error reports. If errors happen in applications it’s necessary to have some means of monitoring the activity in the application — historically and/or in the live system. Therefore every application should have some monitoring and logging capabilities. For distributed applications this is even more important because the erroneous component may be on each tier of the application and it may cost a lot of time to identify which component was the originator of the problem. With

distributed systems there is one more problem: Distributed applications are commonly installed on machines located in one or more DMZ's (Demilitarized Zones) separated by firewalls. The administrators of firewalls want to keep the necessarily open ports for applications in the firewalls as few as possible. Though, if a system operator wants to monitor the activity of a distributed system, he must minimize the amount of ports being used. Also impact on the application caused by the trace system must be minimized: A trace system should not influence application design by imposing upon the application designer to derive every class from a tracing class. Application performance ought not to be reduced significantly and last but not least the trace system must not harm the runtime behavior by waiting for returning method calls.

This is where the Coyote Trace System comes in. It has the following main features:

1. Easy to use tracing components that automatically gather context information.
2. Though the only use of static methods, the application developer doesn't need to instantiate a tracing class for writing trace messages.
3. Ensures good performance and does not block the application by using a multithreaded trace subsystem.
4. Configurable trace message publishing and filtering.
5. Publishing of trace messages to a remote or local trace server.
6. Integrated live monitoring application that offers configurable filters and connections to multiple trace servers.

## 4.2 Coyote Architecture

The application sends trace output with the use of the trace agent. The trace agent collects context information of the calling type, constructs a trace message and forwards it to the trace manager. The trace manager verifies that the message passes the configured filters and asynchronously sends the message to all configured publishers. Publishers are responsible to make some meaningful operation with the trace message, usually store the message in some kind of storage system. The Coyote Trace Solution includes three publishers: The event log publisher writes all trace messages to the application event log. The text file publisher writes the trace publishers to an XML file with rollover options. The trace server publisher enables the trace system to write the messages to a remote or local trace server.

The trace server service may operate as trace message relay, as trace message aggregator and as event source for the coyote trace message monitor application as needed.

The Coyote trace monitor is one of the main parts of the Coyote Trace Solution that enables the user to view the current trace message activity on one or more trace servers. This makes the trace monitor highly useful during the development and productive phase of a software project (see Figure 1).

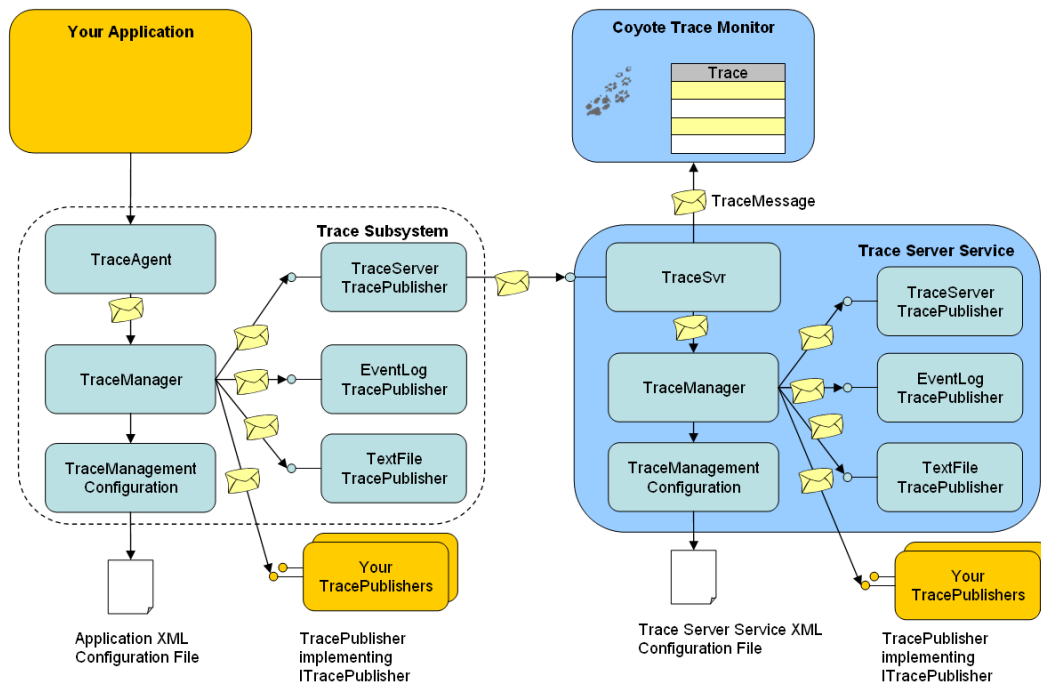


Figure 1: Coyote Architecture

## 5 Daily Build

### 5.1 The Daily Build concept

The *Daily Build* concept is nowadays integrated in nearly every new Software Engineering approach. Kent Beck [1] for example suggests the use of a separate build server applying the *Continuous Integration* strategy of Extreme Programming. This explicit Daily Build server is responsible for automatically requesting the current project sourcecode on a periodical base (every day at 00:00 for example) and then completely building the project.

While compiling and linking the project together successfully is a quite basic job, the build process can be extended with automated unit testing (using tools like JUnit [24] or NUnit [35]) also referenced as *Black box testing* and code analysis (using tools like FxCop [13] or Teamstudio Analyzer [27]) also known as *White box testing* to raise the value of a Daily Build server.

### 5.2 Goals of Daily Build

The main goal of a Daily Build process integrated in the software development architecture is to periodically put together all pieces (i.e. source files, class files, resources) and build them to see if the whole software project compiles without errors. Perhaps some class interfaces changed or one programmer removed a assumed to be useless class, which another programmer referenced to just that moment.

These issues apply on a rather basic level: they just let the build process fail, which is really a bad problem for the project manager. If that happens, he or she has to locate the problem and solve it - or, even better, let the responsible programmers search for the problem in their code. If using XP including the “Collective Ownership” concept (every programmer must be familiar with every code component), such problems theoretically should not appear.

On the other hand, the so-called *Smoke Test* including different testing techniques checks, if the new build meets the test specifications. A Smoke Test is a test or test suite that represents the first line of defense for changing a software component. So, a developer has to write a few test cases which ensure the correct behavior of the main features or methods of his components.

Steve McConnell [32] mentions four big benefits of using Daily Build with Smoke Test:

1. Firstly and already addressed is the **minimized integration risk**. McConnell agrees, that one of the greatest risks a team project faces is different team members combining or integrating code they have been working on separately. The resulting composite code might not work well. Depending on how late in the project the incompatibility is discovered, debugging might take longer than it would have if integration had occurred earlier, program interfaces might have to be changed, or major parts of the system might have to be redesigned and re-implemented. In extreme cases, integration errors have caused projects to be cancelled. The daily build and smoke test process keeps integration errors small and manageable, and it prevents runaway integration problems.
2. Secondly it **reduces the risk of low quality**. Without using a good Daily Build and Smoke Test strategy the state of the project is unknown. The risk of integration and quality problems through hidden faults is much higher - a valid Smoke Test would have discovered such issues hours after their entrance into the project.
3. Thirdly it **supports easier defect diagnosis** by giving exact indications of what happened to the test manager. If the Daily Build of day 5 succeeded and that one of day 6 failed, the issue is to be searched between those two days.
4. Last but not least it **improves the team morale**. Seeing their project building successfully is a morale boost for developers - even if it doesn't do anything exciting. Daily Build enables the programmer team to watch their software growing a little bit every day - and that makes them happy.

### 5.3 Implementing Daily Build and Smoke Test

One important aspect of Daily Build is the word *Daily*. Steve McConnell mentions two similar metaphors for a *Daily* Build in his article about Daily Build and Smoke Test [32]:

- On the one hand, Jim McCarthy [31] compares the Daily Build with the **beat of a heart**. Once the heart does not beat any more, the project has died. From another point of view, the Daily Build is an integral part of a functional software project in this context: A project without Daily Build is nonviable.

- On the other hand, Michael Cusumano and Richard W. Selby [17] think of Daily Build as a **synchronization pulse**. Code fragments of different programmers are allowed to come a little bit out of sync *between* two pulses but after the next sync pulse, all parts of the project must fit together again.

The next point to think about is a useful broken build management. It is very important to find a good balance between a too weak and a too strict contract. Such a contract should contain all the show stoppers that really break our project but exclude uncritical defects that only interrupt the software building process while being no threat to our projects healthiness.

Next issue to address is the need for a matching *Smoke Test*. The best Daily Build is worthless without a good designed and balanced Smoke Test. Such a Smoke Test must not include *all* features of the whole system (which would be very difficult to test) but should comprise the major parts of it. So if this test fails, we will know that we are in trouble.

Another important point is to let the development team know about the importance of not breaking the Daily Build. Most teams do so by introducing a penalty for that developer who is responsible for the clash, like a small fee. That programmer should then work on the problem until it is solved. Goal of this strategy is to let a breaking build be the exception, not the rule.

Last but not least it is important to continue using Daily Build in stressful times, too. Especially in such situations, the effort for the build process can seem useless but it isn't. Stressed developers make more mistakes, quality is decreasing - Daily Build is then more necessary than ever.

## 5.4 Daily Build tools

### 5.4.1 Overview of the available Daily Build tools

The following overview over the available .NET daily build tools can be found in full length in Gunderloy's article from 2003 [22]:

CruiseControl.NET is a continuous integration utility for .NET. Continuous integration works by monitoring your source code control system and automatically starting a build and running unit tests whenever a new file is checked in. CruiseControl.NET is open source, and integrates with VSS, CVS, Perforce, StarTeam, PVCS, NAnt, and NUnit, among other software.

Daily Build lets you set up a treeview of tasks nested into folders. Each task can run whatever you like - provided that it can be run from the command line, which most anything can. It's meant to be driven by Windows Scheduler.

Draco.NET is another free continuous integration utility for .NET projects. It supports NAnt and Visual Studio for builds, and CVS and VSS for source code control. It includes source code if you'd like to tweak it yourself.

FinalBuilder is a complete IDE for setting up complex build processes. It lets you stitch together dozens of tasks from running a VS .NET build to executing a SQL Server DTS package to interacting with various source code control systems. You can also write additional command-line tasks, though you may not need to. You can debug and log the build process as well.

Hippo.NET is an open source build tool specifically for .NET projects. It uses a client/server model so that the entire team can share a single build machine. The developers are planning to add continuous integration and scheduled build features.

NAnt is an open-source build utility based on the Ant project that does the same for Java solutions. It uses XML files to hold information on tasks that should be performed, and supports Mono as well as Microsoft's .NET Framework.

.NET Solution Build & Deployment Process & Tools is a set of documentation and utilities from Microsoft to help creating a structured build process. There are also some preview bits that work with the "Whidbey" version of .NET.

OpenMake is a high-end build management tool that handles scenarios such as multiple developers building the same project with full reproducibility and support for both Java and .NET projects (as well as much else). The latest release includes integration with the Visual Studio .NET and Eclipse IDEs.

Visual Build Pro is a full IDE for managing complicated build processes. It supports dozens of tools and has a full macro language for custom requirements. It will keep log files for a history and integrates directly with various versions of Visual Studio.

Finally, though it can't actually be used yet, one should know about MSBuild - a NAnt-like build utility that Microsoft is integrating into Visual Studio .NET "Whidbey". This is likely to provide stiff competition for some of the existing free and commercial programs, just by getting delivered with Visual Studio .NET by default.

## 5.5 Daily Build in practice

Because of one former project in the CSS GmbH company using NAnt and the so already existing knowledge about that tool, it was chosen for the implementation of the daily build in this scope, too.

Daily build using NAnt utilizes a main XML-Document, which contains all needed configuration items for defining the daily build process. This is the slightly shortened build-file for the Coyote Trace Solution:

```
1 <?xml version="1.0"?>
2 <project name="TraceSolution" default="build" basedir=".">
3   <target name="initsolution">
4     <!-- Solution Properties -->
5     <property name="solution.name" value="TraceSolution"/>
6     <property name="solution.sourcesafe.db"
7     value="\\Devw2kvss011\VSS\MoreVSSDataBases\CSSRepository\srcsafe.ini"/>
8     <property name="solution.build.debug" value="false"/>
9     <property name="solution.build.binsubdir" value="release"/>
10    <property name="solution.build.srcdir" value="src"/>
11    <property name="solution.build.bindir" value="bin"/>
12    <property name="solution.build.xmldir" value="xml"/>
13    <property name="solution.build.deploydir" value="deploy"/>
14    <!-- Included Projects -->
15    <property name="solution.coyote" value="Coyote"/>
16    <property name="solution.licenseproviderlib"
```

```

17         value="Css.Licensing.LicenseProviderLib"/>
18     <property name="solution.licensegenerator"
19         value="Css.Licensing.LicenseGenerator"/>
20     <property name="solution.tracecomlib"
21         value="TraceComLib"/>
22     <property name="solution.traceeventsinklib"
23         value="TraceEventSinkLib"/>
24     <property name="solution.traceservice"
25         value="TraceService"/>
26     <property name="solution.tracesvrlib"
27         value="TraceSvrLib"/>
28 </target>
29 <target name="initfolderstructure">
30     <mkdir dir="${nant.project.basedir}\${solution.build.srcdir}"
31         failonerror="true" />
32     <mkdir dir="${nant.project.basedir}\${solution.build.xmlmdir}"
33         failonerror="true" />
34 </target>
35 <target name="build"
36     depends="initsolution,initfolderstructure,buildprojects">
37     <echo message="Trace Solution build complete."/>
38 </target>
39 <target name="buildlatest"
40     depends="initsolution,cleansources,initfolderstructure,
41         getsources,buildprojects">
42     <echo message="Trace Solution build (latest) complete."/>
43 </target>
44 <target name="cleansources">
45     <delete dir="${nant.project.basedir}\${solution.build.srcdir}"
46         failonerror="false" />
47 </target>
48 <target name="getsources">
49     <echo message="Getting Trace Solution sources from VSS."/>
50     <vssget user="stefan" password="stefan"
51         localpath="${nant.project.basedir}\${solution.build.srcdir}"
52         recursive="true" replace="true" writable="true"
53         dbpath="${solution.sourcesafe.db}"
54         path="$/0 Development/TraceSystem/TraceSolution"/>
55 </target>
56 <target name="buildprojects" depends="tracecomlib">
57 </target>
58 <target name="tracecomlib">
59     <echo message="Building TraceComLib"/>
60     <property name="output.dir"
61         value="${nant.project.basedir}\${solution.build.srcdir}
62             \${solution.tracecomlib}\${solution.build.bindir}
63             \${solution.build.binsubdir}"/>
64     <mkdir dir="${output.dir}" failonerror="true" />
65     <csc target="library" warnaserror="false"
66         debug="${solution.build.debug}"
67         output="${output.dir}\${solution.tracecomlib}.dll"
68         doc="${nant.project.basedir}\${solution.build.xmlmdir}

```

```

69         \${solution.tracecomlib}.xml">
70         <sources basedir="\${nant.project.basedir}
71             \${solution.build.srcdir}\${solution.tracecomlib}\">
72             <includes name="**/*.cs"/>
73         </sources>
74         <arg value="/unsafe-"/>
75         <arg value="/nologo"/>
76         <arg value="/define:TRACE"/>
77     </csc>
78 </target>
79 </project>

```

The first target in the build called `initsolution` initializes all the project properties holding all informations later used. This target gets called by `build` and `buildlatest`.

The `initfolderstructure` target creates the folder structure afterwards accessed: one source directory and one documentation directory. This target gets called by `build` and `buildlatest`.

The `build` target builds the solution using the actual source code files existing in the source directory. This target is designed to called directly from the command line.

The `buildlatest` target cleans up the sources, initializes the folder structure, gets the latest version of the source code files from MS Visual Source Safe and builds them. This target is designed to called directly from the command line.

The `cleansources` target cleans up the sources in the source directory. This target gets called by `buildlatest`.

The `getsources` target gets the latest version of the source code files from MS Visual Source Safe into the source directory. This target gets called by `buildlatest`.

The `buildprojects` target builds all included projects of the trace solution. To save space, only one sample project (the `TraceComLib`) is included in this shortened version of the build file. By just adding slightly different targets for the other project, one can achieve a build of the whole solution. This target gets called by `build` and `buildlatest`.

The `tracecomlib` target builds the `TraceComLib` project of the Coyote Trace Solution with the help of the `csc` which calls the .NET compiler. This target gets called by `buildprojects`.

With the help of the Windows Scheduler the build with NAnt was configured to run every day at 02:00 in the morning.

## 6 Black box and White box testing

While the area of software testing covers a bunch of different techniques, this paper focuses on two main concepts: On one hand white box testing requires the intimate knowledge of program internals, while on the other hand black box testing is based solely on the knowledge of the system requirements.

It is obvious that software engineering experts mainly concentrated on developing testing methodologies based on white box testing because of being primarily concerned with program internals.

In the last years, the importance of black box testing has gained general acknowledgement and triggered the development of a certain number of useful black box testing techniques.

As Thaller [41] says, a software developer never should do testing on his or her self code files - it is most likely to oversee defects in your code lines.

## 6.1 Black box testing

### 6.1.1 The concept of Black box testing

The “Black box testing” concept describes the area of software testing techniques without using knowledge about the code inside of a class or method. Following this principle, one looks at the software classes to be tested as black boxes - that's where the name comes from. So the testing engineer and his unit test cases (“Unit testing” is used as another synonym for Black box testing) only see the interface(s) of a class. The process of testing such a class could be seen three-stepped:

1. The unit test case enters some input into the class/method to be tested by passing parameters to a method or setting class attributes.
2. The desired method of the class is executed without knowledge about the workflow within.
3. The unit test case gets some output of the class/method to be tested by a return value or some class attributes and decides whether the class matches the specification (Test: passed) or not (Test: failed).

### 6.1.2 Goals of Black box testing

The goals of Black box testing differ according to the desired testing category / strategy as there are:

**Functionality:** At a high percentage, black box testing could mainly be assigned to the category of functionality testing which primary objective is to verify whether the software program does what it is supposed to do, i.e. what it is specified to do in the requirements.

**Usability:** Test cases in this category are intended to assess an application against its usability requirements to determine if it contains any usability defects.

**Volume:** Volume testing is targeted to find limitations of software classes/methods by processing huge amounts of data [41]. Test cases in this category can detect bugs that cover wrong buffer sizes, memory leaks and other efficiency problems.

**Recovery:** Recovery tests aim for validating the recovery functionality of software applications including tests like: is it possible to recover all data or part of it, is the recovered data correct and consistent?

**Stability:** Stability testing could be considered as a mix of volume and stress testing: By using different types of testing concepts, the tester tries to find stability problems in the application.

**Security:** Another important concern of software is security. By trying to gather resources that the user is not allowed to access or testing the authentication framework, security holes could be found.

From another point of view there exist only two main goals of black box testing:

1. Find cases where the program does not do what it is supposed to do.
2. Find cases where the program does things it is not supposed to do.

### 6.1.3 Bugs to be found by Black box testing

Using the different kinds of testing concepts, different types of errors/faults can be found. Because black box testing doesn't use any knowledge of the source code of an application, it is better suited to find bugs in the implementation of the specification in the following categories:

- Incorrect or missing functions
- Interface errors
- External database / file access errors
- Performance errors
- Initialization / termination errors

### 6.1.4 Implementing Black box testing

Implementing black box testing means writing the appropriate test cases for the classes / methods to be tested and the desired types of errors to be found. While just writing a few test cases testing some functionality is quite easy, crafting a complete set of test cases (test suites) completely covering any chosen type of error class is a much more difficult job.

For doing so, one firstly has to identify the equivalence classes of the application's input domain: Try to isolate all the conditions within the code associated with the programm's input. Next determine the valid and the invalid states of each condition. Each state corresponds to one of these equivalence classes.

Depending on the type of input data, different test cases have to be defined to cover an equivalence class:

**Range of values:** One valid and two invalid states. Example: Valid range for variable a is 10 to 20. Valid test state: a = 15, invalid test states: a = 9, a = 21.

**Number of input values:** One valid and two invalid states. Example: Valid number of input values: between two and three objects. Valid test state: two objects, invalid test state: one object, four objects.

**Set of values:** *Each member processed differently:* one valid and one invalid state per member.  
*All members processed similarly:* one valid and one invalid sufficient. Example: connection could be closed, connecting and open. Valid test state: connection=open, invalid test state: connection=closing.

**A must be condition:** One valid and one invalid states. Example: a greater than 5. Valid test state: a = 6, invalid test state: a = 5.

Also considered in the examples, the use of boundary values makes sense. These values (9 and 21 in the first 10-20 range example) are statistically (and logically) the most likely states for finding faults in the implementation of the specified behavior.

## 6.1.5 Black box testing tools

### 6.1.5.1 Overview of the available Black box testing tools

Most Black box testing tools utilize the concept of testing within other test scripts to ensure that the product is performing within acceptable parameters. All major components of the program being tested are provided with a work load, after which the output of them are matched against a range of valid results. A “pass” for each aspect of the series of tests indicates that the piece of software functions as expected. A “fail” indicates that the program returned invalid results or performed inadequate.

There are a bunch of tools available on the market of black box testing tools, the major ones are the following:

- Mercury WinRunner [12]
- Segue SilkTest [26]
- IBM Rational Robot [11]
- Compuware QACenter Enterprise Edition (QARun and TestPartner) [7]
- JUnit [24] [20]
- NUnit [35]

### 6.1.5.2 JUnit

JUnit [24], a major - if not *the* major open source tool is a regression testing framework written by Erich Gamma and Kent Beck. Its main functionality is to provide the developer who implements unit tests in Java with a collection of testing classes simplifying the process of writing test code and so-called “TestRunners” which execute the test cases and summarize the results. JUnit is Open Source Software which was released under the Common Public License Version 1.0 and is hosted on SourceForge [20].

A developer going to write test cases using JUnit has to know about the way JUnit divides up the work: Test Case - Fixture - Suite.

The smallest unit of JUnit is a “Test Case”. A Test Case is implemented by just writing a Java class inheriting von `junit.framework.TestCase` and overriding the base method `runTest`. Within this method the developer can write his/her own test code involving creating objects, setting parameters, executing methods and asserting the results. Test cases include a common set of objects to be used and a common initializing method.

If the developer wants to (and this is recommended!) write more than one test for a similar set of objects, the concept of a “Fixture” is the one to use. Straightly forward as JUnit is, a Fixture is nothing more than a set of test cases operating on a common collection of objects.

Once the programmer has developed a set of test cases or fixtures, he probably wants to run them together - thats what a “Test Suite” can be used for. JUnit provides this class to act as a testing container which calls all desired test cases.

Additionally to these business classes, JUnit gets delivered with several different “Test Runner” classes, which provide the tester with a textual or graphical interface for executing the written test suites and viewing the results.

### 6.1.5.3 NUnit

Kent Beck says about NUnit: “NUnit 2.0 is an excellent example of idiomatic design. Most folks who port a new xUnit tool derivative just transliterate the Smalltalk or Java version. That’s what we did with NUnit at first, too. This new version is NUnit as it would have been done in C# to begin with.” [35].

NUnit firstly was developed by porting JUnit to Microsoft’s .NET language families. Though, NUnit and JUnit are very similar except the programming language they are designed for. The current version of NUnit, 2.1 is the third major release and was completely redesigned utilizing the advantages of the .NET framework and written in C#. Similar to Java, .NET incorporates the concept of a common language runtime for all supported languages, so NUnit can be used with C#.NET, C++.NET, VB.NET and J#.NET.

The concept structure including test cases, fixtures and test suites was completely taken over from JUnit. As NUnit was used to implement black box testing in practice, the testing possibilities are shortly addressed in the following:

#### Assertions

As in any of the xUnit frameworks, assertions belong to the most important components for implementing test cases, and NUnit is no exception. NUnit provides the developer with a big collection of different static assertion methods within the `NUnit.Framework.Assert` class. If one assertion within a test case fails, the method call does not return and NUnit reports an error. Because of this concept, assertions after this first assertion will not be executed in this case. Knowing about this design implementation, it would be best to include only one assertion per test case to avoid ambiguous situations.

#### Comparison Assertions

NUnit offers a bunch of different comparison assertions which often are the best choice for a test case because of comparing an expected to an actual value. For these situations, NUnit provides the following comparison asserts:

```

1 Assert.AreEqual(int expected, int actual);
2 Assert.AreEqual(int expected, int actual, string message);
3 Assert.AreEqual(decimal expected, decimal actual);
4 Assert.AreEqual(decimal expected, decimal actual, string message);
5 Assert.AreEqual(float expected, float actual, float tolerance);
6 Assert.AreEqual(float expected, float actual, float tolerance,
7     string message);
8 Assert.AreEqual(double expected, double actual, double tolerance);
9 Assert.AreEqual(double expected, double actual, double tolerance,
10    string message);
11 Assert.AreEqual(object expected, object actual);
12 Assert.AreEqual(object expected, object actual, string message);
13 Assert.AreSame(object expected, object actual);
14 Assert.AreSame(object expected, object actual, string message);

```

### Condition Tests

Aside comparisons, NUnit offers simple condition tests which use a custom condition. Most condition test methods take a condition and an optional message to display. The following methods are provided:

```

1 Assert.IsTrue(bool condition);
2 Assert.IsTrue(bool condition, string message);
3 Assert.IsFalse(bool condition);
4 Assert.IsFalse(bool condition, string message);
5 Assert.IsNull(object anObject);
6 Assert.IsNull(object anObject, string message);
7 Assert.IsNotNull(object anObject);
8 Assert.IsNotNull(object anObject, string message);

```

### Fail Methods

NUnit provides you with the `Assert.Fail` method, that simply generates a failure without a test parameter. This can be useful for writing you own assertion methods:

```

1 Assert.Fail();
2 Assert.Fail(string message);

```

## 6.1.6 Black box testing in practice

As the Coyote project already existed in a final state before this testing environment was being developed, the wishful concept of “Test First Design” (see Chapter 2.2 Test Driven Development) could not truly be used. Being used as a proof-of-concept, test cases for several different classes of Coyote were developed after the project end. Nevertheless, in the future development of the B-Web project TDD should be part of the testing environment.

The following (shortened) class `FilterRuleDate` was chosen as example from the Coyote project because of its relative simplicity on one hand and its major logical/functional semantics on the other hand, which lets the class become a good candidate for black box testing unlike most other classes aimed at GUI or database functionality.

`FilterRuleDate` was developed to implement a filter rule aimed at a `DateTime` field of a trace message. Trace messages are the main messaging objects within the Coyote Trace Solution and transport the interesting information: A timestamp, the generating method, a category, a message body and so on. The Coyote Trace Monitor should receive such trace messages and display them to the user. For convenience purposes, the monitor should enable the user to filter the incoming messages on base of all message fields including the timestamp. That's what a `FilterRuleDate` object is for: It exposes a comparison date (that's the date we compare against), a rule type (should the timestamp field be equal, greater, greater or equal, smaller, smaller or equal compared to the comparison date?) and a rule detail (should the comparison consider seconds?).

In the following, only the main method `FilterRuleDate.RuleSatisfied(TraceMessage message)` is printed to keep the listing short:

```
1 namespace Css.Diagnostics.TraceMonitor.Filtering
2 {
3     [Serializable]
4     public class FilterRuleDate : FilterRuleBase, ICloneable, IFilterRule
5     {
6         ...
7
8         public bool RuleSatisfied(TraceMessage message)
9         {
10            if (this.Enabled)
11            {
12                bool satisfied = false;
13                try
14                {
15                    DateTime checkDate =
16                        (DateTime) message.GetItem(
17                            this.AssignedTraceMessageItem);
18                    DateTime compareDate = this.ComparisonDate;
19
20                    switch (this.RuleDetail)
21                    {
22                        case FilterRuleDateDetail.Second:
23                            checkDate = new DateTime(
24                                checkDate.Year,
25                                checkDate.Month,
26                                checkDate.Day,
27                                checkDate.Hour,
28                                checkDate.Minute,
29                                checkDate.Second,
30                                0);
31                            compareDate = new DateTime(
32                                compareDate.Year,
33                                compareDate.Month,
34                                compareDate.Day,
35                                compareDate.Hour,
36                                compareDate.Minute,
37                                compareDate.Second,
38                                0);
39                            break;
```

```

40         case FilterRuleDateDetail.Minute:
41             checkDate = new DateTime(
42                 checkDate.Year,
43                 checkDate.Month,
44                 checkDate.Day,
45                 checkDate.Hour,
46                 checkDate.Minute,
47                 0,
48                 0);
49             compareDate = new DateTime(
50                 compareDate.Year,
51                 compareDate.Month,
52                 compareDate.Day,
53                 compareDate.Hour,
54                 compareDate.Minute,
55                 0,
56                 0);
57             break;
58             ...
59         }
60
61     switch (this.RuleType)
62     {
63         case FilterRuleType.Equals:
64             if (checkDate == compareDate)
65                 satisfied = true;
66             break;
67         case FilterRuleType.GreaterOrEqualThan:
68             if (checkDate >= compareDate)
69                 satisfied = true;
70             break;
71         case FilterRuleType.GreaterThan:
72             if (checkDate > compareDate)
73                 satisfied = true;
74             break;
75         case FilterRuleType.SmallerOrEqualThan:
76             if (checkDate <= compareDate)
77                 satisfied = true;
78             break;
79         case FilterRuleType.SmallerThan:
80             if (checkDate < compareDate)
81                 satisfied = true;
82             break;
83         case FilterRuleType.None:
84             satisfied = true;
85             break;
86         default:
87             satisfied = false;
88             break;
89     }
90
91     if (this.Inverted)

```

```

92         satisfied = !satisfied;
93
94         return satisfied;
95     }
96     catch
97     {
98         return false;
99     }
100 }
101 else
102     return true;
103 }
104
105 ...
106 }
107 }

```

For use with this class, the following test cases were written and encapsulated within a fixture including a common set-up method:

```

1  ...
2  using NUnit.Framework;
3
4  namespace Css.Diagnostics.TraceMonitor.Filtering
5  {
6      [TestFixture]
7      public class TestFilterRuleDate
8      {
9          private TraceMessage testMessage;
10         private FilterRuleDate testFilterRule;
11
12         [SetUp]
13         public void Init()
14         {
15             testMessage = new TraceMessage(
16                 "Hello world",
17                 "Test category",
18                 TraceLevel.Information);
19
20             testMessage.TimeStamp =
21                 new DateTime(2004, 2, 20, 9, 0, 0);
22
23             testFilterRule = new FilterRuleDate(
24                 TraceMessageItemType.TimeStamp,
25                 FilterRuleType.Equals,
26                 FilterRuleDateDetail.Second,
27                 DateTime.MinValue,
28                 false,
29                 true);
30         }
31
32         [Test]

```

```

33     public void EqualsSecondsTest ()
34     {
35         testFilterRule.ComparisonDate =
36             new DateTime(2004, 2, 20, 9, 0, 0);
37         Assert.IsTrue(testFilterRule.RuleSatisfied(testMessage));
38     }
39
40     [Test]
41     public void EqualsNotSecondsTest ()
42     {
43         testFilterRule.ComparisonDate =
44             new DateTime(2004, 2, 20, 9, 0, 1);
45         Assert.IsFalse(testFilterRule.RuleSatisfied(testMessage));
46     }
47
48     [Test]
49     public void EqualsMinutesTest ()
50     {
51         testFilterRule.RuleDetail = FilterRuleDateDetail.Minute;
52         testFilterRule.ComparisonDate =
53             new DateTime(2004, 2, 20, 9, 0, 30);
54         Assert.IsTrue(testFilterRule.RuleSatisfied(testMessage));
55     }
56
57     [Test]
58     public void EqualsNotMinutesTest ()
59     {
60         testFilterRule.RuleDetail = FilterRuleDateDetail.Minute;
61         testFilterRule.ComparisonDate =
62             new DateTime(2004, 2, 20, 9, 1, 0);
63         Assert.IsFalse(testFilterRule.RuleSatisfied(testMessage));
64
65         testFilterRule.ComparisonDate =
66             new DateTime(2004, 2, 20, 10, 0, 30);
67         Assert.IsFalse(testFilterRule.RuleSatisfied(testMessage));
68     }
69 }
70 }

```

## 6.2 White box testing

### 6.2.1 The concept of White box testing

The main characteristic of white box testing is the use of knowledge about the source code inside of classes / methods.

Traditionally, white box testing (also known as “Glass box testing”) can be divided into *static* and *dynamic* analysis.

### 6.2.1.1 Static Analysis

All testing concepts belonging to static analysis have one paradigm in common: The analysis process needs no execution of the program to be tested, rather the code itself is analyzed statically - that's where the name comes from.

Hans-Ludwig Hausen [23] states that "Essential functions of static analysis are checking whether representations and descriptions of software are consistent, noncontradictory or unambiguous". Static analysis tries to find incorrect descriptions, specifications and representations of a software system and has therefore to be the first step in the chain of testing concepts. To fulfill its job, static analysis uses lexical analysis of the source code and inspects the structure and usage of the individual statements.

While static analysis can run completeness and consistency checks of the provided software on its own, a very useful extension is a pre-filled and well maintained database of "best practises" or "common failures" rules to check for.

A sample for a very common error or simply a misunderstood statement is the following:

```
if (myNumber = 5) ...
```

Implying a common high level programming language like C/C++ or Java, this statement will assign the value "5" to "myNumber" and then execute the body of the *if* clause - the condition always resolves to true. If the programmer wanted to write some meaningful code, he perhaps just forgot one "=":

```
if (myNumber == 5) ...
```

This statement does something meaningful: It compares "myNumber" to the number "5" and enters the *if* clause body if they match. Such semantically wrong code lines are very likely to be overseen by a human being but can easily be found by an automated static analysis tool using a rule database. The tool then can ask the programmer: "Did you really wanted to write *that* code?"

### 6.2.1.2 Dynamic Analysis

Compared against static analysis, the main difference of dynamic analysis is its need to **execute** the program it is instructed to test. Hausen gives us the following definition of dynamic analysis: "The analysis of the behaviour of a software system before, during and after its execution in an artificial or real applicational environment characterises dynamic analysis" [23].

The two major dynamic analysis techniques are *Path testing* and *Branch testing*. Goal of Path testing is the execution of the target program involving as much logical paths through the code tree as possible. The most important quality value measured by Path testing is the *program complexity*. A branch testing tool tries to construct a test strategy to execute each code branch at least once while testing.

### 6.2.1.3 Code Inspection

While both, static analysis and dynamic analysis completely base on the use of a computer tester, the Code Inspection concept relies on human resources. Using this concept, periodical code reviews are scheduled assigning developers to software modules they have not written themselves.

Letting a developer review code that was written by another person is very important and responsible for the good efficiency of this concept knowing that most programmers share the behavior of likely overseeing their own bugs.

### 6.2.2 Goals of White box testing

When comparing two resources on different semantic levels, i.e. a program against its specification, the *completeness* and *correctness* of the program or its implementation can be evaluated. This dynamic analysis technique called *Static verification* is used to detect problems and misconceptions in implementing the formal specifications.

Having a detailed look at branch testing, there exist several different measurement goals:

**C0 Coverage:** A C0 coverage is achieved by running a set of logical paths to execute each line of code at least once.

**C1 Coverage:** A C1 coverage is achieved by executing each code branch at least once.

**C2 Coverage:** A C2 coverage is achieved by executing each condition body at least once.

**C3 Coverage:** A C3 coverage is achieved by going through each loop more than one time.

Apart from the discussed automated testing concepts, there exist a bunch of manual testing techniques like *Walktroughs* or *Software Inspection* which will not be covered by this paper.

### 6.2.3 Bugs to be found by White box testing

Depending on the used white box testing technique there exist varying sets of failure types to be found by the testing engineer/tool. Using static analysis, an automated testing tool can find the following errors / bad style issues:

- Wrongly used API methods
- Missing method parameters (i.e. localization issues)
- Wrong naming conventions used
- Other violations to best practices
- Hard coded constants

While static analysis only can find static failures, dynamic analysis is able to detect more subtle, dynamic bugs:

- Bugs in conditions and loops (i.e. wrong break condition)
- Wrong initialization or overwriting of variables
- Memory overflow bugs
- Null pointer, bad references

## 6.2.4 White box testing tools

### 6.2.4.1 Overview of the available White box testing tools

Because of the wide area of different white box testing techniques, there exist obviously a great number of tools implementing those concepts. A few widespread tools are:

- IBM Rational PurifyPlus [10]
- McCabe QA [30]
- ParaSoft JTest [38]
- Sureshot JiveLint [40]
- FxCop [13]
- Teamstudio Analyzer for Java [27]
- NCover [6]

### 6.2.4.2 FxCop

One example of a static analysis tool is FxCop [13], which was developed by Microsoft. FxCop checks managed .NET code assemblies for conformance to the .NET Design Guidelines. To fulfill its job, FxCop uses reflection, MSIL parsing and callgraph analysis to inspect the target program against a failure database of more than 200 different defects in the following areas:

- Library design
- Localization
- Naming conventions
- Performance
- Security

FxCop provides the developer with a textual as well as with a graphical user interface.

## 6.2.5 White box testing in practice

While there exist a big number of so-called NAnt Tasks which integrate several different tools into the build tool, actually there doesn't exist one for FxCop. To integrate an automated FxCop testing process into our daily build script, an indirection over the command line was the way to go. The following code lines represent the relevant lines of the build file:

```
1 <target name="runFxCop">
2     <exec program="tools\fxcop\fxcopcmd.exe"
3         commandline="/p:${nant.project.basedir}\${nant.project.name}.fxcop
4             /o:${Build.OutputFolder}\fxcop-results.xml" failonerror="false"/>
5 </target>
```

This target called “runFxCop” just calls the FxCop executable including a project file and a result output file as parameters.

## 7 Performance testing

### 7.1 The concept of Performance testing

The concept of Performance testing can alone fill hundreds of books without completely covering all details about it. One of the major difficulties about performance testing is its “qualitative character”: In the area of black box and white box testing, one can always determine, if something is valid or not - that functionality is desired or not. Unlike tests in this area, performance testing is not targeted at finding bugs rather than at finding “bottlenecks” - devices, software components, networks or other entities which limit the performance of the whole system.

Because of this quality measurement, performance testing results depend highly on the testing environment: network performance, testing host performance, server performance (all hardware based), CPU utilization (other software running) and much more. Keeping these influences all in mind, it gets very difficult to achieve reproducible *and* significant results including hints, where to find bottlenecks.

### 7.2 Performance testing tools

Because performance testing can be applied in such a big number of different ways, each again different for testing stand-alone applications, web applications, distributed applications and so on, there exists an unmanageable number of tools for measuring performance. Some of the best-known tools are the following:

- Apache JMeter [18]
- IBM Rational Performance Tester [9]
- Segue SilkPerformer [25]
- Compuware QACenter Enterprise Edition (QALoad) [8]
- Microsoft Application Center Test [14]
- RadView WebLoad [28]

## 7.3 Performance testing in practice

The main exercise in the scope of the Coyote project was to measure the performance of and perhaps find some bottlenecks within the Coyote Trace Server, which is a central component in the architecture and though a critical point affecting the overall system performance.

The to be tested server component exposes no user interface because it gets only contacted by clients using .NET remoting. Therefor instead of using a special performance tool, a custom tester class was developed, which just acts as a client and pushes trace messages with a configurable delay out to the server unless the user stops the tester:

```
1 using System;
2
3 namespace Css.Diagnostics
4 {
5     class TraceAgentTester
6     {
7         [STAThread]
8         static void Main(string[] args)
9         {
10            int delay;
11            try {delay = Int32.Parse(args[1]);}
12            catch {delay = 100;}
13
14            int count = 0;
15            DateTime before;
16            DateTime after;
17            TimeSpan diff;
18            string testMsg = "";
19            Console.WriteLine("press n and return to end");
20            while (Console.ReadLine() != "n")
21            {
22                count++;
23                testMsg = String.Format("Test Message # {0}", count);
24                Random errorLevelRand = new Random();
25                int errorLevel = errorLevelRand.Next(1,4);
26                TraceLevel level = (TraceLevel) errorLevel;
27                before = DateTime.Now;
28
29                TraceAgent.Write(level, testMsg,
30                    "this is a test message sent by tracagenttester");
31
32                after = DateTime.Now;
33                diff = after - before;
34                Console.WriteLine("Sent Msg [{0}] call duration [{1}] ms ",
35                    testMsg, diff.TotalMilliseconds);
36                System.Threading.Thread.Sleep(delay);
37            }
38        }
39    }
40 }
```

## 8 Conclusion

Today it gets more and more obvious, that a complete testing environment really gets a companies development team to a higher level. As Bobby Goerge and Laurie Williams [21] state, a company surely has to accept an extra effort for implementing and applying testing concepts, but will produce software products of a proportionally (or better) level of quality.

Especially nowadays, where a hard competition on every customer has emerged between the different software developing companies on the market, software quality and customer satisfaction have become essential for survival. Using this knowledge, every development team should reconsider their method of creating software soon.

Implementing the chosen concept features (see Chapter 3 The chosen concept features) in the CSS GmbH company turned out as no impossible challenge — in the end, the daily build server was up and running, the Coyote Trace Solution project has really proven the concept and the whole concept is waiting for getting used in the next projects.

While the whole bunch of concepts of an Extreme Programming strategy might not match every software engineering company's budget and philosophy, picking out a custom-made company testing environment really makes sense.

## References

- [1] Kent Beck. *Extreme programming explained*. Addison-Wesley Professional, Oct. 1999.
- [2] Kent Beck. Aim, fire. *IEEE Software*, 18:87–89, Sept/Oct 2001.
- [3] N. Calzolari and J. McNaught. Methods for system measurement. <http://www.issco.unige.ch/ewg95/node77.html>, 1994.
- [4] Dave Chaplin. Test first programming. *TechZone*, 2001.
- [5] CSS computer-systems-support GmbH. <http://www.css-web.net>, 2004.
- [6] Giles Cope. Ncover. <http://ncover.sourceforge.net/>, 2004.
- [7] Compuware Corporation. Compuware qacenter enterprise edition. <http://www.compuware.com/products/qacenter/entserver.htm>, 2004.
- [8] Compuware Corporation. Compuware qaload. <http://www.compuware.com/products/qacenter/qaload.htm>, 2004.
- [9] IBM Corporation. Ibm rational performance tester. <http://www-306.ibm.com/software/awdtools/tester/performance/>, 2004.
- [10] IBM Corporation. Ibm rational purifyplus. <http://www-306.ibm.com/software/awdtools/purifyplus/>, 2004.
- [11] IBM Corporation. Ibm rational robot. <http://www-306.ibm.com/software/awdtools/tester/robot/index.html>, 2004.

- [12] Mercury Interactive Corporation. Mercury winrunner. <http://www.mercury.com/us/products/quality-center/functional-testing/winrunner/>, 2004.
- [13] Microsoft Corporation. Fxcop. <http://www.getdotnet.com/team/fxcop/>, 2004.
- [14] Microsoft Corporation. Microsoft application center test. [msdn.microsoft.com/library/en-us/act/htm/actml\\_main.asp](http://msdn.microsoft.com/library/en-us/act/htm/actml_main.asp), 2004.
- [15] Microsoft Corporation. Microsoft solutions framework. <http://msdn.microsoft.com/vstudio/teamsystem/msf>, 2004.
- [16] Microsoft Corporation. Visual c# developer center. <http://msdn.microsoft.com/vcsharp>, 2004.
- [17] Michael Cusumano and Richard W. Selby. Microsofts secrets. *The Free Press*, 1995.
- [18] Apache Software Foundation. Apache jmeter. <http://jakarta.apache.org/jmeter/>, 2004.
- [19] Steven Fraser, Dave Astels, Kent Beck, Barry Boehm, John McGregor, James Newkirk, and Charlie Poole. Discipline and practices of tdd: (test driven development). In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 268–270. ACM Press, 2003.
- [20] Erich Gamma and Kent Beck. Junit on sourceforge. <http://sourceforge.net/projects/junit>, 2004.
- [21] Boby George and Laurie Williams. An initial investigation of test driven development in industry. In *Proceedings of the 2003 ACM symposium on Applied Computing*, pages 1135–1139. ACM Press, 2003.
- [22] Mike Gunderloy. Build tools for .net applications. *ADTmag.com*, Dec. 2003.
- [23] Hans-Ludwig Hausen. Comments on practical constraints of software validation techniques. In *Proceedings of symposium on software validation*, pages 323–333, 1984.
- [24] Object Mentor Inc. Junit, testing resources for extreme programming. <http://www.junit.org/>, 2004.
- [25] Segue Software Inc. Segue silkperformer. <http://www.segue.com/products/load-stress-performance-testing/silkperformer.asp>, 2004.
- [26] Segue Software Inc. Segue silktest. <http://www.segue.com/products/functional-regressional-testing/silktest.asp>, 2004.
- [27] Teamstudio Inc. Teamstudio analyzer for java. <http://www.teamstudio.com/tsv3/teamstudio.nsf/index/Teamstudio+Analyzer+for+Java?opendocument>, 2004.
- [28] RadView Software Incorporated. Radview webload. <http://www.radview.com/products/WebLOAD.asp>, 2004.

- [29] Reid Kaufmann and David Janzen. Implications of test-driven development: a pilot study. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 298–299. ACM Press, 2003.
- [30] McCabe and Associates Incorporated. Mccabe qa. [http://www.mccabe.com/iq\\_qa.htm](http://www.mccabe.com/iq_qa.htm), 2004.
- [31] Jim McCarthy. Dynamics of software development. *Microsoft Press*, 1995.
- [32] Steve McConnell. Daily build and smoke test. *IEEE Software: Best Practices*, 13(4), July 1996.
- [33] Jeff Miller, Eric Vautier, and David Vydra. Testdriven.com. <http://www.testdriven.com/>, 2004.
- [34] Miscellaneous. Wikipedia: Extreme programming. [http://en.wikipedia.org/wiki/Extreme\\_Programming](http://en.wikipedia.org/wiki/Extreme_Programming), 2004.
- [35] James W. Newkirk and Alexei A. Vorontsov. Nunit. <http://www.nunit.org/>, 2004.
- [36] Institute of Software Technology and Interactive Systems. <http://www.ifs.tuwien.ac.at/ifs>, 2004.
- [37] Technical University of Vienna. <http://www.tuwien.ac.at>, 2004.
- [38] ParaSoft. Parasoft jtest. <http://www.parasoft.com/jsp/products/home.jsp?product=Jtest&itemId=12>, 2004.
- [39] Alexander Schatten. <http://www.schatten.info>.
- [40] Sureshot Software. Jivelint. <http://www.sureshotsoftware.com/javalint/>, 2004.
- [41] G. Thaller. *Verifikation und Validation. Software Tests fr Studenten und Praktiker*. Vieweg, 1994.
- [42] Don Wells. Extreme programming: A gentle introduction. <http://www.extremeprogramming.org/>, 2003.