

Ferramentas de testes e qualidade de software para a plataforma .NET

Resumo

O Objetivo deste artigo é mostrar ferramentas para a plataforma .NET, que permitem a realização de testes e a melhoria da qualidade do produto de software. Vamos falar sobre ferramentas para testes unitários, automação de builds, cobertura de código, análise estática de código e integração contínua. Vamos entender onde elas se encaixam dentro de um processo de desenvolvimento de software maduro e como auxiliam na qualidade de um projeto na plataforma .NET.

Ferramentas

1. NUnit (<http://www.nunit.org>)

NUnit é um aplicativo open source que permite a você escrever testes na linguagem de sua escolha para testar uma função específica de sua aplicação. Ela é uma excelente forma de testar a funcionalidade do seu código quando escrito pela primeira vez e também fornece um método para testes de regressão da sua aplicação. A ferramenta NUnit fornece um framework para escrever unidades de testes bem como interface gráfica para executar estes testes e visualizar os resultados.

1.1. Aplicação sob teste

Para que a ferramenta seja testada será codificada uma classe que representa a tabela **Pessoa** no banco de dados. Uma pessoa tem apenas uma propriedade no sistema em questão, seu **Nome**. A classe fornecerá apenas um serviço, que é o de **Pesquisa** de uma pessoa dado o seu nome:

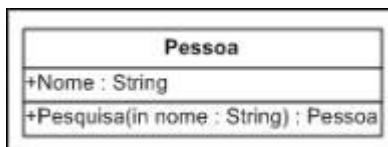


Fig.1: O Diagrama de Classes do nosso sistema

O Banco de dados contém uma única tabela, chamada **Pessoa**, com uma só coluna, chamada **Nome**, que conterá um texto de 50 caracteres.

1.2.Utilizando o NUnit

De acordo com o princípio básico do TDD, deve-se codificar os testes primeiro, e as rotinas para satisfazer os testes depois, então a primeira coisa a fazer é criar a aplicação de teste.

1.2.1. Projeto de testes em .net

O NUnit é projetado para carregar uma DLL .NET, instanciar uma classe que contém um conjunto de testes, e executar estes testes. Portanto, deve-se iniciar criando uma aplicação C# do tipo Class Library no .NET.

1.2.2. Classes de teste

Em seguida será acrescentada a classe que conterá o conjunto de testes. Remova a classe default Class1.cs, e acrescente uma classe chamada Testes.cs. Para usar as facilidades providas pelo NUnit, deve-se colocar uma referência a um dos componentes fornecidos por ele. Acrescente também um `using NUnit.Framework` à classe Testes. Para que ele reconheça a classe Testes como uma classe que contém testes, deve-se que marcar esta classe com o atributo `[TestFixture]`. Este atributo está no namespace `NUnit.Framework`.

```
using System;
using NUnit.Framework;
using System.Data.OleDb;

namespace ExemploNUnit
{
    [TestFixture] public class Testes
    {
    }
}
```

Fig. 2: A classe de teste, vazia.

Para testar a classe **Pessoa**, serão criados 3 testes:

- Um teste chamado `BancoVazio` que tentará pesquisar uma pessoa quando a tabela **Pessoa** está vazia.
- Um teste chamado `UmRegistro` que tentará pesquisar um registro, quando este registro é o único cadastrado na tabela **Pessoa**.
- Um teste chamado `RegistroInexistente` que tentará pesquisar um registro que não é nenhum dos registros cadastrados na tabela **Pessoa**.

Fig. 3: Os testes que criaremos para a classe **Pessoa**.

1.2.3. Rotina de inicialização de testes

Para que os testes fiquem independentes é interessante que se apague todos os registros da tabela **Pessoa** antes de cada teste, e criar as condições necessárias aos testes. O NUnit já fornece uma funcionalidade para isto podendo-se marcar um método da classe de testes com o atributo [SetUp]. Assim aquele método será automaticamente executado pelo NUnit para todos os testes, imediatamente antes de se começar a execução do teste. Abaixo será criado o método de setup de testes, ApagaRegistros:

```
[TestFixture] public class Testes
{
    [SetUp]
    public void ApagaRegistros()
    {
        OleDbCommand comando = new OleDbCommand();
        comando.CommandText = "DELETE FROM Pessoa";
        ExecutaComando(comando);
    }

    private void ExecutaComando(OleDbCommand comando)
    {
        OleDbConnection conexao = new OleDbConnection(
            "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Teste.mdb");
        comando.Connection = conexao;
        try
        {
            conexao.Open();
            comando.ExecuteNonQuery();
        }
        finally
        {
            conexao.Close();
        }
    }
}
```

Fig. 4: A classe de testes já com o método de setup dos testes.

1.2.4. Os Testes

O primeiro teste, que será chamado BancoVazio que tentará pesquisar uma pessoa quando a tabela **Pessoa** está vazia.

```
[Test]
public void BancoVazio()
{
    Pessoa p = Pessoa.Pesquisa("nome xyz");
    Assert.IsNull(p, "Pesquisa com banco vazio deve retornar null");
}
```

Fig. 5: primeiro teste.

O método está marcado com o atributo [Test]. Isto faz com que o NUnit o reconheça como um teste. Além disto, todos os métodos que são testes devem seguir a assinatura `public void NomeDoMetodo()`, ou seja, público, sem retorno, e sem parâmetros. Outro aspecto é o uso de uma classe chamada Assert, a qual está no namespace NUnit.Framework, e permite que sejam feitos vários tipos de verificações, o que realmente é bastante útil em um programa de testes. Nesse caso, a classe Assert verifica se o objeto p do programa é nulo, e em caso negativo, emite a mensagem "Pesquisa com banco vazio deve retornar null". Há também o uso de uma classe "que não existe": a classe Pessoa. Mas é assim que o TDD funciona. De uma forma geral, os seus testes serão a modelagem do seu sistema, pois é deles que sairão as classes da sua aplicação. Obviamente, deve-se colocar as pessoas mais experientes da equipe de desenvolvimento para bolar os testes, e, uma vez estes construídos - e consequentemente já definidas as classes da aplicação - você distribuirá grupos de testes para os outros desenvolvedores, que serão encarregados de fazer os testes funcionarem. E ao fazerem os testes funcionarem, os desenvolvedores estarão criando o código necessário para que a aplicação se comporte de acordo com o esperado nas várias situações testadas. Os testes serão automatizados permitindo que realize-se testes de regressão. Ou seja, haverá um processo de desenvolvimento mais integrado, de maior qualidade, e que se adapta muito bem a mudanças de requisitos. Vemos aqui mais um princípio do TDD:

1.3.Executando os testes no NUnit

A primeira coisa a fazer é compilar a aplicação de teste. Para fazermos isto, temos que criar a nossa classe , e colocar um pouco de código lá dentro.

```
using System;

namespace ExemploNUnit
{
    public class Pessoa
    {
        public static Pessoa Pesquisa(string nome)
        {
            return null;
        }
    }
}
```

Fig. 6: O método Pessoa.Pesquisa, versão 1.

Este código deve ser colocado em uma nova classe que foi acrescentado a aplicação de teste, chamada **Pessoa.cs**.

O NUnit necessita d eum projeto. Para isso abra o NUnit e crie um novo projeto, através do menu **File → New Project**. Depois, carregue a DLL da nossa aplicação de testes, através do menu **Project → Add Assembly**, selecionando a DLL **bin\debug\ExemploNUnit.dll**. Veja que o NUnit automaticamente reconhece a classe Testes como um conjunto de testes, e o método BancoVazio como um teste.

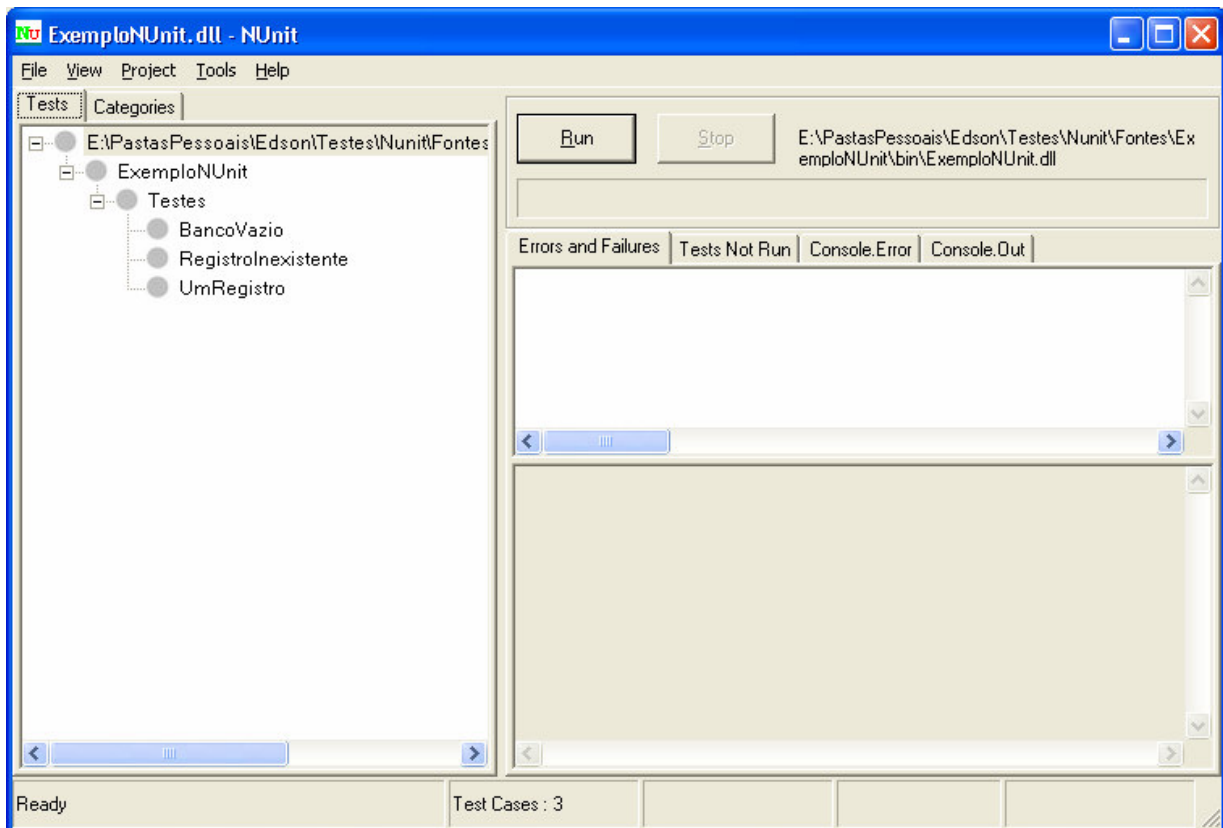


Fig. 7: O projeto de teste carregado no NUnit.

Pressione o botão **Run** para que o NUnit execute os testes.

O segundo teste a ser definido que será chamado **UmRegistro** que tentará pesquisar um registro, quando este registro é o único cadastrado na tabela **Pessoa** (veja a figura 3 com a descrição dos testes).

```
[Test]
public void UmRegistro()
{
    IncluiRegistroTeste("nome xyz");
    Pessoa p = Pessoa.Pesquisa("nome xyz");
    Assert.IsNotNull(p, "Pesquisa deveria retornar objeto");
    Assert.AreEqual("nome xyz", p.Nome, "Nome errado");
}

private void IncluiRegistroTeste(string nome)
{
    OleDbCommand comando = new OleDbCommand();
    comando.CommandText = "INSERT INTO Pessoa VALUES (?)";
    comando.Parameters.Add("@nome", nome);
    ExecutaComando(comando);
}
```

Fig. 8: O segundo teste.

Há agora um outro teste do NUnit: um método marcado com o atributo [Test], que é `public`, `void`, e sem parâmetros. O que ele faz é incluir um registro de teste cujo nome é "nome xyz", e chama o método `Pessoa.Pesquisa` para recuperar este registro. Em seguida, o teste verifica se um objeto foi retornado (`Assert.IsNotNull`), e se a propriedade `Nome` desse objeto contém o valor "gilberto" (`Assert.AreEqual`), ou seja, o teste verifica se a pessoa correta foi recuperada pelo método `Pessoa.Pesquisa`.

Ao recompilar a aplicação, uma mensagem é exibida como o erro "'ExemploNUnit.Pessoa' does not contain a definition for 'Nome'". Para isso não ocorrer deve-se alterar a classe para que ela tenha agora a propriedade `Nome`. Acrescente a propriedade `public string Nome` à classe `Pessoa`, e recompile a aplicação de teste. Isto mostra uma característica do TDD: a classe é construída a medida em que construímos os testes. Ao voltarmos para o NUnit, note que ele já recarregou a nova versão da DLL (a que acabamos de compilar), e já mostra o teste `UmRegistro`. Mas, ao rodar os testes, percebe-se que `BancoVazio` continua ok, enquanto que `UmRegistro` falhou.

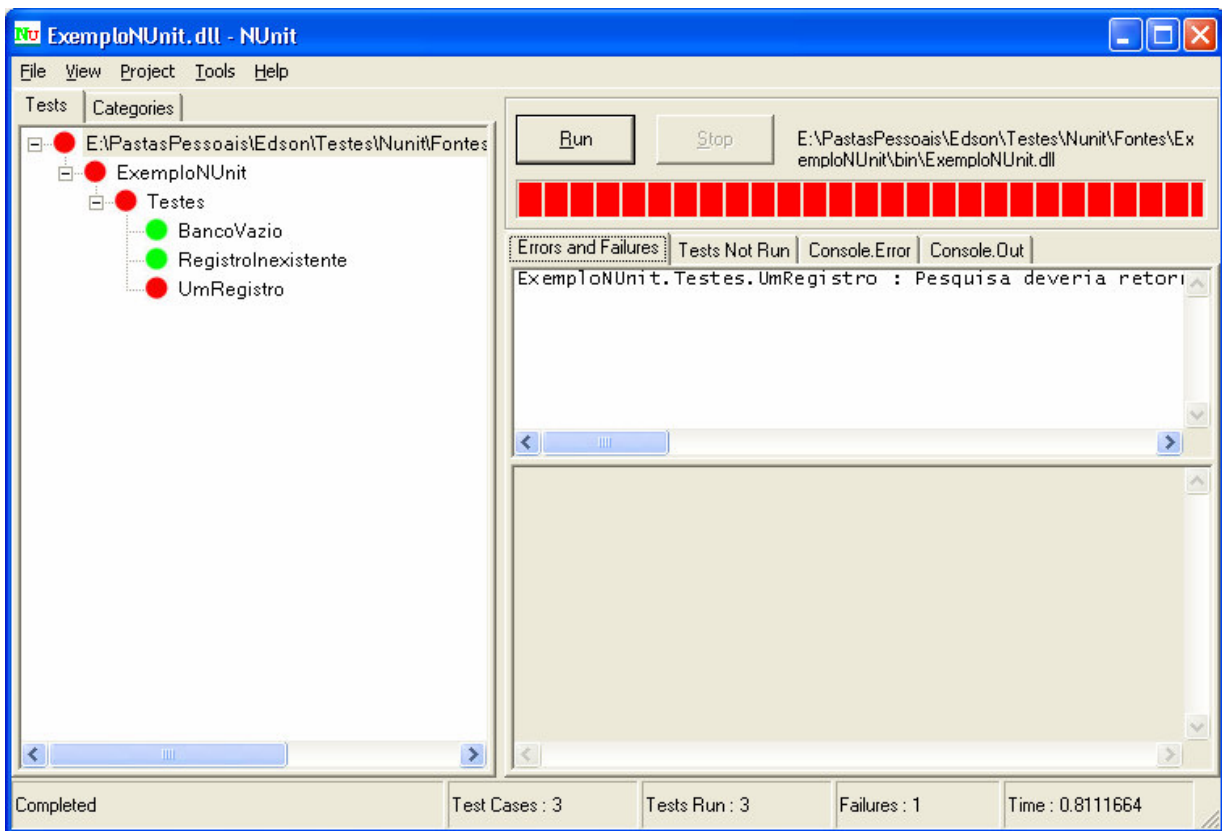


Fig. 9: Red bar não é bom sinal.

Óbvio, pois a única coisa que o método `Pessoa.Pesquisa` faz é retornar null. O teste nesse caso falhou para essa funcionalidade, então, seguindo o princípio básico do TDD (*Você só deve codificar uma funcionalidade se você tem um teste que falha para ela*), agora a funcionalidade pode ser codificada.

```
public static Pessoa Pesquisa(string nome)
```

```

{
    OleDbConnection conexao = new OleDbConnection(
        "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Teste.mdb");
    OleDbCommand comando = new OleDbCommand("SELECT * FROM Pessoa",
        conexao);
    OleDbDataReader dados = null;
    try
    {
        conexao.Open();
        dados = comando.ExecuteReader();
        dados.Read();
        Pessoa p = new Pessoa();
        p.Nome = (string)dados["Nome"];
        return p;
    }
    finally
    {
        dados.Close();
        conexao.Close();
    }
}

```

Fig. 10: O método Pessoa.Pesquisa, versão 2.

Recompile a aplicação de teste, e no NUnit, clique com o botão direito no teste UmRegistro e selecione **Run**. Você verá uma barra verde. Mas se re-executarmos todos os testes (clique com o botão direito em **Testes** e selecione **Run**), vemos que agora o teste BancoVazio está falhando. Pela leitura da mensagem de erro, sabemos que é a linha que tenta ler o valor da coluna **Nome** que está dando problemas. Óbvio, pois no teste BancoVazio não há linhas para serem lidas, então temos que lidar com esta situação. Vamos trocar a linha

```
dados.Read();
```

para

```
if(!dados.Read()) return null;
```

e a *green bar* voltará a aparecer.

O exemplo será fechado codificando o terceiro teste chamado RegistroInexistente que tentará pesquisar um registro que não é nenhum dos registros cadastrados na tabela **Pessoa**." (veja a figura 3). O nosso teste será:

```

[Test]
public void RegistroInexistente()
{
    IncluiRegistroTeste("nome aaa");
    Pessoa p = Pessoa.Pesquisa("nome xyz");
    if(p != null) Assert.Fail("Pesquisa achou o registro " + p.Nome);
}

```

Fig. 11: O terceiro teste.

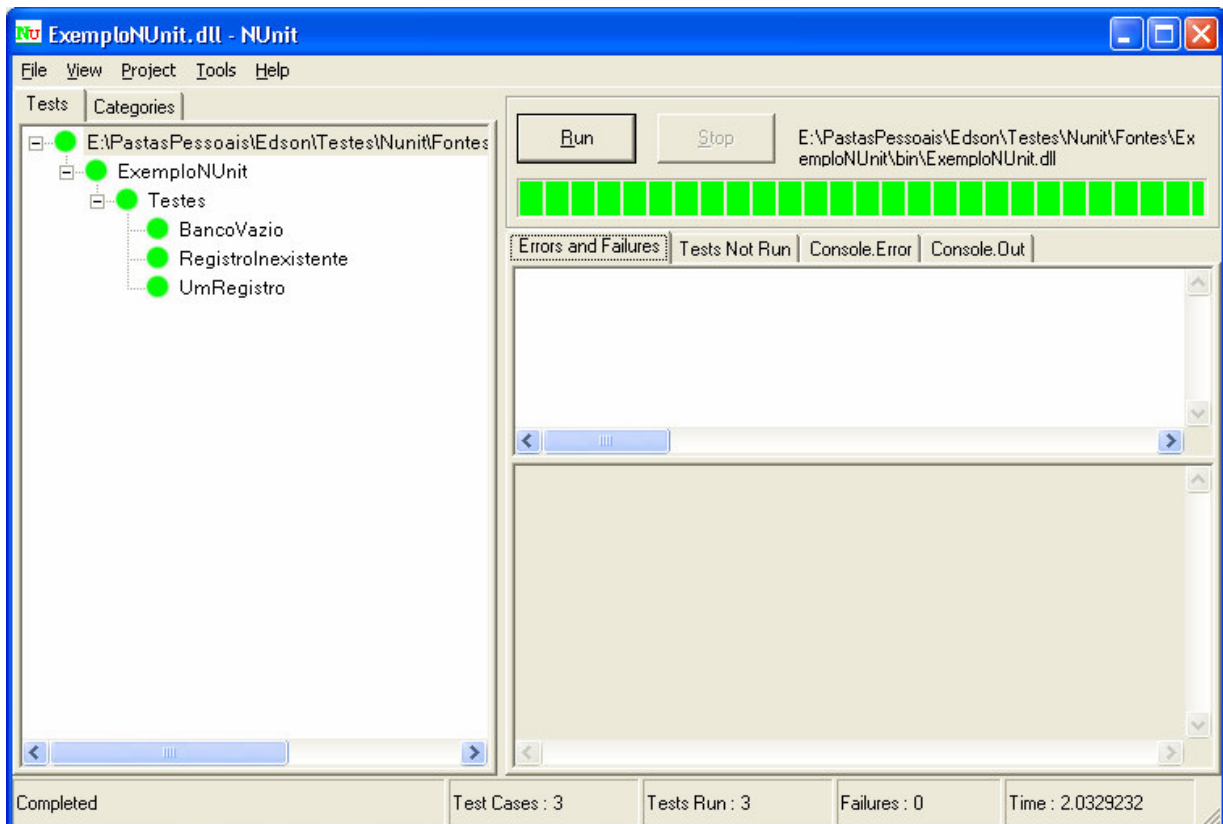
Recompile a aplicação de teste, e no NUnit, re-execute todos os testes. Os dois primeiros executam ok, mas o nosso novo teste falha, emitindo a mensagem "ExemploNUnit.Testes. RegistroInexistente : Pesquisa achou o registro nome aaa.

Isto porque o nosso método Pessoa.Pesquisa atualmente está abrindo a tabela **Pessoa**, e retornando o primeiro registro que ele encontra lá. Temos que filtrar pelo nome recebido como parâmetro! Então vamos corrigir o nosso método:

```
public static Pessoa Pesquisa(string nome)
{
    OleDbConnection conexao = new OleDbConnection(
        "Provider=Microsoft.Jet.OLEDB.4.0;Data Source=Teste.mdb");
    OleDbCommand comando = new OleDbCommand(
        "SELECT * FROM Pessoa WHERE Nome = ?",
        conexao);
    comando.Parameters.Add("@nome", nome);
    OleDbDataReader dados = null;
    try
    {
        conexao.Open();
        dados = comando.ExecuteReader();
        if(!dados.Read()) return null;
        Pessoa p = new Pessoa();
        p.Nome = (string)dados["Nome"];
        return p;
    }
    finally
    {
        dados.Close();
        conexao.Close();
    }
}
```

Fig. 12: O método Pessoa.Pesquisa, versão final.

Agora os testes devem estar ok, conseqüentemente a aplicação também.



2. DotNetMock (sourceforge.net/projects/dotnetmock)

O objetivo do teste unitário é exercitar somente um método por vez. Mas o que fazer quando esses métodos dependem de outras coisas como rede, banco de dados ou cotações em tempo real ? Ou até mesmo se parte do código depende de outras partes do sistema.

Uma alternativa é a criação de stubs, substituindo partes do sistema que por ventura ainda nem estejam prontas, por classes que simulam um determinado comportamento.

Suponha que você precise em algum momento retornar a data e hora correntes. A seguinte função poderia ser criada:

```
public DateTime Now {  
    get {  
        return DateTime.Now;  
    }  
}
```

Para determinarmos uma configuração em tempo de depuração poderíamos fazer a seguinte melhoria:

```
public DateTime Now {  
    get {  
        if (DEBUG)  
            return currentTime;  
        else  
            return DateTime.Now;  
    }  
}
```

Para situações mais complexas isso ficaria inviável, pois teríamos que alterar o código fonte em vários locais. Para resolver esse problema

2.1.Definição

Para resolver o problema acima há um padrão chamado mock que é um substituto do objeto real. Há diversas situações onde esse padrão pode nos ajudar:

- Objetos com comportamento não determinísticos
- Objetos difíceis de serem configurados
- Eventos e exceções difíceis de serem simuladas
- Objetos extremamente lentos
- User Interface
- Objetos que ainda não existem e precisam ser integrados

Utilizando mock objects podemos minimizar ou até mesmo solucionar esses problemas seguindo os seguintes passos:

1. Utilizar uma interface para descrever um objeto
2. Implementar a interface em código de produção
3. Implementar a interface

O código sob teste referencia seus objetos através de suas interfaces, desacoplando-se do objeto “mock” ou real. O exemplo anterior ficaria, então da seguinte maneira:

- Definição da interface

```
public interface Environmental {  
    DateTime Now {  
        get;  
    }  
    // Outros métodos omitidos  
}
```

- Implementação da interface pela classe real

```
public class SystemEnvironment : Environmental {  
  
    public DateTime Now {  
        get {  
            return DateTime.Now;  
        }  
    }  
  
    public void PlayWavFile(string name) {  
        // bop into core.dll and invoke PlaySound...  
    }  
  
}
```

- Implementação da classe Mock

```
using System;  
  
public class MockSystemEnvironment : Environmental {  
  
    private DateTime currentTime;  
  
    public MockSystemEnvironment(DateTime when) {  
        currentTime = when;  
    }  
  
    public DateTime Now {  
        get {  
            return currentTime;  
        }  
    }  
  
    public void IncrementMinutes(int minutes) {  
        currentTime = currentTime.AddMinutes(minutes);  
    }  
  
}
```

Observá-se que pode-se passar a data e hora desejada como valor inicial no construtor. Há também nesse exemplo um método que possibilita o incremento do número de minutos. Isso permite que você controle a data e hora retornada por um objeto mock.

Suponha que para efeito de testes crie-se na aplicação um novo método denominado `Reminder()` que emite um alerta sonoro após as 17:00hs.

```
public class Checker {  
    Environmental env;  
  
    public Checker(Environmental env) {  
        this.env = env;  
    }  
  
    public void Reminder() {  
        DateTime now = env.Now;  
  
        if (now.Hour >= 17) {  
            env.PlayWavFile("va_pra_casa.wav");  
        }  
    }  
}
```

O código que irá ser utilizado em produção utilizará a classe `SystemEnvironment`. O código de testes por outro lado utilizará a classe `MockSystemEnvironment`. A aplicação sob teste não saberá a diferença entre o ambiente de teste e o ambiente real, pois a mesma interface é implementada. Nesse caso pode-se definir testes que utilizam a classe mock o tempo para valores conhecidos e verificando seu resultado. Com um código extra podemos adicionar alguns testes para verificar se o método `PlayWavFile` foi chamado sem termos que utilizar o dispositivo de som do computador.

```
public class MockSystemEnvironment : Environmental {

    private DateTime currentTime;

    public MockSystemEnvironment(DateTime when) {
        currentTime = when;
    }

    public DateTime Now {
        get {
            return currentTime;
        }
    }

    public void IncrementMinutes(int minutes) {
        currentTime = currentTime.AddMinutes(minutes);
    }

    // Implement the PlayWavFile part of the interface,
    // and provide methods to check to see if it was
    // played

    private bool soundWasPlayed = false;

    public void PlayWavFile(string fileName) {
        soundWasPlayed = true;
    }

    // For convenience, check the sound played
    // flag and reset it in one method call
    public bool CheckAndResetSound() {
        bool value = soundWasPlayed;
        soundWasPlayed = false;
        return value;
    }
}
```

A configuração final para esse testes seria a seguinte:

```
using System;
using NUnit.Framework;

[TestFixture]
public class TestChecker {

    [Test]
    public void QuittingTime () {

        DateTime when = new DateTime(2004,10,1,16,55,0);
        MockSystemEnvironment env;
        env = new MockSystemEnvironment (when);
        Checker checker = new Checker (env);

        // No alarm sounds at 16:55
        checker.Reminder();
        Assert.IsFalse (env.CheckAndResetSound(), "16:55");

        // Now try 17:00
        env.IncrementMinutes (5);
        checker.Reminder();
        Assert.IsTrue (env.CheckAndResetSound(), "17:00");

        // And finally 19:00
        env.IncrementMinutes (120);
        checker.Reminder();
        Assert.IsTrue (env.CheckAndResetSound(), "19:00");
    }
}
```

O código acima cria um objeto do tipo Date and Time contendo a hora a ser utilizada para o teste, o qual é então passado para o ambiente de testes “mock”.

3. NAnt (<http://nant.sourceforge.net/>)

NAnt é uma ferramenta para automação de builds e releases, baseada na ferramenta open source do projeto Apache Java Ant, criada para a plataforma Java. O arquivo que define como é um build está em formato XML(diferente de uma ferramenta como o make, do Unix).

Essa ferramenta possui uma importância fundamental, pois “uma parte importante de qualquer processo de desenvolvimento de software é conseguir builds confiáveis do software.”(Fowler e Foemmel, 2004). A Microsoft, em sua ferramenta IDE(Integrated Development Environment) Visual Studio .NET, não disponibilizou nenhuma ferramenta nesse sentido. A comunidade viu a necessidade e portou a idéia já consolidada do Ant para a plataforma .NET. A comunidade ainda criou um repositório para contribuições de novas tasks que podem ser interessantes dentro do NAnt em <http://nantcontrib.sourceforge.net/>.

O NAnt possibilita a realização de uma série de tarefas, através de suas tasks. O mais interessante é que ele não necessita do Visual Studio .NET ou outra ferramenta de desenvolvimento instalada, apenas o .NET Framework(Gehtland, 2004c). Dessa forma, o NAnt permite a criação de um processo automatizado de geração de builds, permitindo a construção do software inteiro a partir de apenas um único comando.

De acordo com Clark(2004), um processo de builds deve ser:

- Completo: não necessita de outros passos além daqueles descritos na ferramenta de build.
- Repetível: Os arquivos de configuração que geram o build também se modificam conforme o tempo e, dessa forma, devem ser integrados à ferramenta de controle de versões, como o restante do software.
- Informativo: Permite saber com facilidade se um build falhou ou teve sucesso.
- Esquedulável: Pode ser colocado para ser rodado de tempos em tempos. Um exemplo clássico e uma prática fundamental é o daily build and smoke tests(McConnell, 1996).
- Portável: Significa que as máquinas onde o build pode rodar só contêm o estritamente necessário. Ele não precisa ser dependente de uma IDE particular, de um endereço IP ou de um diretório específico.

Para termos um exemplo do poder e facilidade de construir uma configuração de build automatizado, segue abaixo o arquivo XML e uma explicação do que ele está realizando.

```
<?xml version="1.0"?>
<project name="Trab_Testes" default="release">
  <property name="build.dir" value="C:\Builds\Trab_Checkout\bin\" />
  <property name="src.dir" value="C:\Builds\Trab_Checkout\src\" />
  <property name="nunit.dir" value="C:\Arquivos de programas\NuNit 2.2\bin\" />
  <property name="ncover.dir" value="C:\Arquivos de programas\NCover\" />

  <target name="release" depends="clean,checkout,build">
    <timestamp>
      <formatter property="DSTAMP" pattern="yyyy-MM-dd"/>
      <formatter property="TSTAMP" pattern="HH:mm"/>
    </timestamp>
  </target>
</project>
```



```

    </tstamp>
    <echo message="Build iniciou em: ${DSTAMP} - ${TSTAMP}"/>
</target>

<target name="checkout">

    <exec program="cvs" commandline="-d :local:C:\Repositorio co Trab_Testes\src"/>

</target>

<target name="clean">
    <delete dir="${build.dir}" failonerror="false"/>
    <delete dir="${src.dir}" failonerror="false"/>
</target>

<target name="build">
    <mkdir dir="${build.dir}" />
    <mkdir dir="${src.dir}" />

    <csc target="library" output="${build.dir}Trab_Testes.dll" debug="true">
        <sources>
            <includes name="${src.dir}/*.*" />
        </sources>
        <references>
            <includes name="${nunit.dir}nunit.framework.dll" />
        </references>
    </csc>

    <exec program="nunit-console" commandline="${build.dir}/Trab_Testes.dll"/>

    <exec program="${ncover.dir}ncover.console" commandline="/q /o ${build.dir}coverage.xml
    /c "${nunit.dir}nunit-console" "${build.dir}Trab_Testes.dll" />

    <exec program="C:\Arquivos de programas\Microsoft FxCop 1.312\fxcopcmd.exe"
        commandline="/f:${build.dir}Trab_Testes.dll /out:${build.dir}fxcop.xml"
    failonerror="false"/>

</target>
</project>

```

Basicamente, o arquivo XML acima faz as seguintes tarefas, de forma automatizada, ao chamarmos o Nant com o parâmetro release:

1. Limpa os diretórios com os fontes e binários que existam anteriormente(para garantir que ele não deixe de compilar nada por existir já um arquivo no local).
2. Faz um checkout do código-fonte mais recente que está dentro da ferramenta de controle de versões(utilizei no exemplo o CVS).
3. Compila o código-fonte, gera uma class library(dll), chama os testes de unidade dos métodos da biblioteca, gera um relatório de cobertura de código e faz uma análise estática do código-fonte.

A saída do nant é a seguinte:

```
C:\Builds\Trab_Checkout>nant release
NAnt 0.84 (Build 0.84.1455.0; net-1.0.win32; release; 26/12/2003)
Copyright (C) 2001-2003 Gerry Shaw
http://nant.sourceforge.net
```

```
Buildfile: file:///C:/Builds/Trab_Checkout/Trab_Checkout.build
Target(s) specified: release
```

clean:

```
[delete] Deleting directory C:\Builds\Trab_Checkout\bin\
[delete] Deleting directory C:\Builds\Trab_Checkout\src\
```

checkout:

```
[exec] cvs -d :local:C:\Repositorio co Trab_Testes\src
      U Trab_Testes\src\Largest.cs
      U Trab_Testes\src\TestLargest.cs
cvs checkout: Updating Trab_Testes\src
```

build:

```
[mkdir] Creating directory C:\Builds\Trab_Checkout\bin\
[csc] Compiling 2 files to C:\Builds\Trab_Checkout\bin\Trab_Testes.dll
[exec] nunit-console C:\Builds\Trab_Checkout\bin\Trab_Testes.dll
      NUnit version 2.2.0
      Copyright (C) 2002-2003 James W. Newkirk, Michael C. Two, Alexei A.
      Vorontsov, Charlie Poole.
      Copyright (C) 2000-2003 Philip Craig.
      All Rights Reserved.
```

```
      OS Version: Microsoft Windows NT 5.1.2600.0 .NET Version: 1.1.432
      2.2032
```

```
.....
Tests run: 8, Failures: 0, Not run: 0, Time: 0,09375 seconds
```

```
[exec] C:\Arquivos de programas\NCover\ncover.console /q /o C:\Builds\Trab_
Checkout\bin\coverage.xml
      /c "C:\Arquivos de programas\NuNit 2.2\bin\nunit-console" "C:\Builds\Trab_
Checkout\bin\Trab_Testes.dll"
      NCover.Console v1.3.3 - Code Coverage Analysis for .NET - http://nco
ver.org
```

```
Command: C:\Arquivos de programas\NuNit 2.2\bin\nunit-console
Command Args: C:\Builds\Trab_Checkout\bin\Trab_Testes.dll
Working Directory:
Assemblies:
Coverage File: C:\Builds\Trab_Checkout\bin\coverage.xml
Coverage Log:
```

```
***** Program Output *****
```

NUnit version 2.2.0
Copyright (C) 2002-2003 James W. Newkirk, Michael C. Two, Alexei A.
Vorontsov, Charlie Poole.
Copyright (C) 2000-2003 Philip Craig.
All Rights Reserved.

OS Version: Microsoft Windows NT 5.1.2600.0 .NET Version: 1.1.432
2.2032

.....
Tests run: 8, Failures: 0, Not run: 0, Time: 0,078125 seconds

***** End Program Output *****
Copied 'C:\Arquivos de programas\NCover\Coverage.xml' to 'C:\Builds\
Trab_Checkout\bin\Coverage.xml'
[exec] C:\Arquivos de programas\Microsoft FxCop 1.312\fxcopcmd.exe /f:C:\Bu
ilds\Trab_Checkout\bin\Trab_Testes.dll /out:C:\Builds\Trab_Checkout\bin\fxcop.xml
1

Microsoft FxCopCmd v1,312
Copyright (C) 1999-2004 Microsoft Corp. All rights reserved.

Loaded DesignRules.dll...
Loaded GlobalizationRules.dll...
Loaded InteroperabilityRules.dll...
Loaded MobilityRules.dll...
Loaded NamingRules.dll...
Loaded PerformanceRules.dll...
Loaded PortabilityRules.dll...
Loaded SecurityRules.dll...
Loaded UsageRules.dll...
Could not resolve reference to nunit.framework.
Loaded Trab_Testes.dll...
Initializing Introspection engine...
Analyzing...
Analysis Complete.
Writing 11 messages...
Writing report to C:\Builds\Trab_Checkout\bin\fxcop.xml...
Done.

release:

[tstamp] domingo, 28 de novembro de 2004 18:17:28.
[echo] Build iniciou em : 2004-11-28 - 18:17

BUILD SUCCEEDED

Total time: 12.9 seconds.

Podemos notar como ter um script que automatiza o build a partir do código-fonte que se encontra em um repositório é útil. Não necessitamos de uma IDE ou outra ferramenta sofisticada para realizá-lo(lembrando que é apenas o .NET Framework e o próprio NAnt). Garantimos que os desenvolvedores estão alinhados com o processo de desenvolvimento do software e com o código-fonte. Conforme Clark (2004), é comum, em locais sem um

processo automatizado de build, ouvimos a seguinte frase: “mas na minha máquina estava compilando e funcionando!”.

Para finalizar, algumas boas práticas no uso do NAnt (Nantz, 2004):

- Tudo depende de sua estrutura de diretórios. Como esta ficará dentro de um sistema de controle de versões, mudanças futuras podem ser complexas. Vale a pena gastar um tempo inicial do projeto para pensar em como montá-la (Em Clark (2004) há muitas dicas interessantes para a estrutura de diretórios).
- Crie targets diferenciados para a geração de builds diários e para a criação de releases para os clientes (já que os clientes não precisam receber o código-fonte e o resultado dos testes).
- Mantenha o arquivo XML atualizado sempre que necessário e guarde-o junto com a versão em um branch do sistema de controle de versões.

4. FxCop (<http://www.gotdotnet.com/team/fxcop/>)

O FxCop, conforme descrito em seu site, é uma ferramenta gratuita de análise de código que o verifica e checka sua consistência de acordo com os Guias de Design do .Net Framework definidos pela Microsoft em: <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpgenref/html/cpconNetFrameworkDesignGuidelines.asp>.

Ele possui um dicionário de regras que valida o código nas seguintes áreas(seguido de um exemplo de cada uma delas. Estas regras e seu detalhamento podem ser encontrados na documentação que é enviada com a ferramenta):

1. Design – Evitar interfaces vazias
2. Globalização – Especifique MessageBoxOptions
3. Interoperabilidade – Chame GetLastError imediatamente depois de pinvoke
4. Mobilidade – Não use prioridade de processos idle
5. Nomes – Evite tipos de nomes em parâmetros
6. Performance – Evite criação desnecessária de strings
7. Portabilidade – Campos de tipo Valor devem ser portáveis
8. Segurança – Ponteiros não devem ser visíveis
9. Uso – Implemente ISerializable corretamente

É interessante notar que certos códigos gerados pela própria IDE do Visual Studio(no Forms Designer, por exemplo) podem gerar certos erros no FxCop, pois este código auto-gerado não segue os guidelines (Robbins, 2004). A Microsoft nas próximas versões do ASP .NET e do Visual Studio irá realizar alterações em seus wizards e geradores de código para seguir todos os guidelines do FxCop.

Para termos uma idéia, no pequeno exemplo criado, o seguinte pedaço do arquivo XML foi gerado:

```
<?xml version="1.0" encoding="utf-8"?>
<?xml-stylesheet type="text/xsl" href="c:\arquivos de programas\microsoft fxcop
1.312\Xml\FxCopReport.Xsl"?>
<FxCopReport Version="1.312">
```

```

<Targets>
<Target Name="C:\Builds\Trab_Checkout\bin\Trab_Testes.dll">
<Modules>
<Module Name="trab_testes.dll">
<Messages>
<Message TypeName="AssembliesShouldDeclareMinimumSecurity" Category="Microsoft.Usage"
CheckId="CA2209" Status="Active" Created="2004-11-28 20:17:25Z" FixCategory="Breaking">
<Issue Certainty="99" Level="CriticalError">No valid permission requests were found for assembly
'Trab_Testes'. You should always specify the minimum security permissions using
SecurityAction.RequestMinimum. If assembly permission requests have been specified, they are not
enforceable; use the PermView.exe tool to view the assembly's permissions. Whidbey customers can use
PermCalc.exe which gives even more detailed information.</Issue>
</Message>

.....

<Url>http://www.gotdotnet.com/team/fxcop/docs/rules.aspx?version=1.312&&url=/Design/MarkA
ssembliesWithComVisible.html</Url>
<Email>askfxcop@microsoft.com</Email>
<MessageLevel Certainty="99">CriticalError, Error</MessageLevel>
<File Name="DesignRules.dll" Version="1.312.0.0" />
</Rule>
<Rule TypeName="StaticHolderTypesShouldNotHaveConstructors" Category="Microsoft.Design"
CheckId="CA1053">
<Name>Static holder types should not have constructors</Name>
<Description>Instances of types that define only static members do not need to be created. Many compilers
will automatically add a public default constructor if no constructor is specified. To prevent this, adding an
empty private constructor may be required.</Description>
<Resolution Name="Default">Remove the public constructors from '{0}'.</Resolution>
<Owner />

<Url>http://www.gotdotnet.com/team/fxcop/docs/rules.aspx?version=1.312&&url=/Design/StaticH
olderTypesShouldNotHaveConstructors.html</Url>
<Email>askfxcop@microsoft.com</Email>
<MessageLevel Certainty="90">Error</MessageLevel>
<File Name="DesignRules.dll" Version="1.312.0.0" />
</Rule>
</Rules>
</FxCopReport>

```

Como podemos notar, é uma ferramenta interessante para projetos que necessitam de um ótimo nível de qualidade de código. Permite que revisões ou inspeções de código(em processos mais formalizados) sejam feitas com o apoio dos resultados gerados por essas ferramentas.

5. NCover (<http://www.ncover.org/>)

O NCover é uma ferramenta open source de cobertura de código para .NET. Seu diferencial mais interessante é que não precisa de uma compilação especial, pois utiliza a API Profiler do próprio framework .NET. NCover não altera, portanto o código pré-compilado IL.

Ele foi criado especialmente para trabalhar junto com o NUnit. Como no processo de criação e execução de testes de unidade é complexo verificar quais linhas de código

realmente foram executadas, o NCover veio para garantir uma maior qualidade dos testes gerados pelo desenvolvedor .NET.

O NCover cria basicamente dois arquivos importantes:

- Coverage.xml – é a saída com os dados da análise do NCover.
- Coverage.xsl – é um arquivo que permite a transformação do XML em HTML, para tornar a saída XML mais legível.

Abaixo um exemplo do relatório lido por um browser(utilizando o arquivo XSL). É importante notar que as linhas que não aparecem são as linhas que não executam como linhas em branco, parênteses ou as chaves(as que abrem um if ou for, por exemplo):

Code Coverage Report

Trab_Testes

Cmp.Largest

Visit Count	Line	Column	End Line	End Column	Document
12	27	14	27	33	c:\Builds\Trab_Checkout\src\Largest.cs
12	29	3	29	24	c:\Builds\Trab_Checkout\src\Largest.cs
1	31	4	31	46	c:\Builds\Trab_Checkout\src\Largest.cs
11	42	8	42	17	c:\Builds\Trab_Checkout\src\Largest.cs
32	44	4	44	26	c:\Builds\Trab_Checkout\src\Largest.cs
22	46	5	46	23	c:\Builds\Trab_Checkout\src\Largest.cs
32	42	43	42	50	c:\Builds\Trab_Checkout\src\Largest.cs
43	42	19	42	41	c:\Builds\Trab_Checkout\src\Largest.cs
11	49	3	49	14	c:\Builds\Trab_Checkout\src\Largest.cs
11	50	2	50	3	c:\Builds\Trab_Checkout\src\Largest.cs

TestLargest.TestEmpty

Visit Count	Line	Column	End Line	End Column	Document
1	41	3	41	29	c:\Builds\Trab_Checkout\src\TestLargest.cs
0	42	2	42	3	c:\Builds\Trab_Checkout\src\TestLargest.cs

TestLargest.LargestOf3Alt

Visit Count	Line	Column	End Line	End Column	Document
1	49	3	49	26	c:\Builds\Trab_Checkout\src\TestLargest.cs
1	50	3	50	14	c:\Builds\Trab_Checkout\src\TestLargest.cs
1	51	3	51	14	c:\Builds\Trab_Checkout\src\TestLargest.cs
1	52	3	52	14	c:\Builds\Trab_Checkout\src\TestLargest.cs
1	53	3	53	40	c:\Builds\Trab_Checkout\src\TestLargest.cs
1	54	2	54	3	c:\Builds\Trab_Checkout\src\TestLargest.cs

6. CruiseControl .NET (<http://ccnet.thoughtworks.com/>)

O CruiseControl .NET é uma ferramenta que traz os benefícios da Integração Contínua automatizada para a plataforma .NET. A prática da integração contínua representa uma mudança importante no processo de construção de software. Ela torna a integração uma atividade diária e trivial dos desenvolvedores, ao invés de uma tarefa pouco freqüente e dolorosa. A integração contínua acredita na filosofia incremental e iterativa de progresso contínuo e em passos pequenos.

É interessante notar que Cockburn (2004) diz que os elementos de gerência de configuração, integrações freqüentes e testes automatizados deveriam fazer parte do núcleo básico de qualquer processo de desenvolvimento de software e que ele fica até constrangido de ter que mencioná-los.

A freqüência da integração pode variar(e a ferramenta permite essa variação) de minutos e horas e pelo menos uma vez por dia, seguindo a filosofia de "daily builds and smoke tests" conforme descrito por McConnel (1996). O único fator limitante para essa freqüência é o tempo que demora para a realização de todo o ciclo de build e testes. Além disso, é interessante lembrar que ferramentas como o CruiseControl .NET só disparam um build no caso de alguma alteração ter ocorrido no código-fonte ou alguém forçar um build (Clark, 2004).

De acordo com Fowler e Foemmel (2004), para a realização de build diários ou integração contínua é necessário:

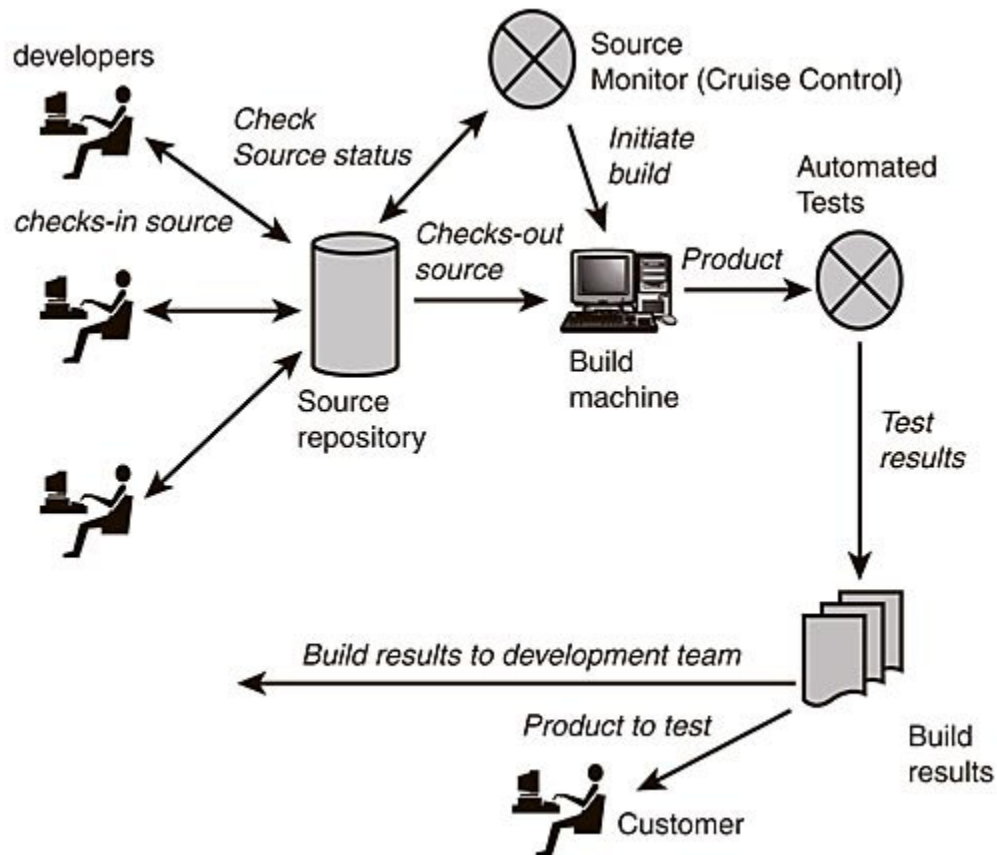
- Manter o código-fonte e outros artefatos necessários(como arquivos de configuração, deployment, etc) em um único repositório centralizado e de preferência baseado em um sistema de controle de versões.
- Automatizar o processo de build para que qualquer pessoa possa, com um único comando, construir o sistema com base nos códigos-fonte.
- Automatizar os testes de forma que se possa rodar um conjunto de casos de teste(unitários e/ou funcionais) a qualquer momento com um único comando.
- Ter certeza que qualquer um possa gerar um build com a confiança de que este é o melhor até o momento.

Os pontos acima necessitam de uma boa dose de disciplina, energia e pró-atividade. As ferramentas ajudam neste trabalho, mas o processo deve ser comprado e realizado pelas pessoas no time. Uma vez preparado o ambiente e treinado e convencidas as pessoas, o esforço de se manter a integração contínua é menor e se torna ainda mais visível como um diferencial dentro e fora da equipe.

Uma instalação e configuração padrão para o uso do CruiseControl .NET para integração contínua de um projeto consiste dos seguintes passos (Gehtland, 2004a):

1. Definir o projeto
2. Configurar as verificações de ocorrências de mudança no repositório do controle de versões.
3. Definir o intervalo de verificação do build.
4. Configurar qual o arquivo de build que será executado após um check-out bem sucedido.

A figura abaixo representa o fluxo do processo que é realizado através dessa ferramenta:



Como a figura também mostra, podemos enviar notificações por email, SMS, etc para o time sempre que um dos builds automatizados falhar. Além disso, o CruiseControl .NET permite que tenhamos um histórico de builds mantido em um site Web. Ela se torna então uma ferramenta muito útil para o time e para o gerente de projetos avaliar o progresso não só dos builds mas como do projeto como um todo.

Abaixo seguem algumas telas do site Web gerado pela ferramenta:

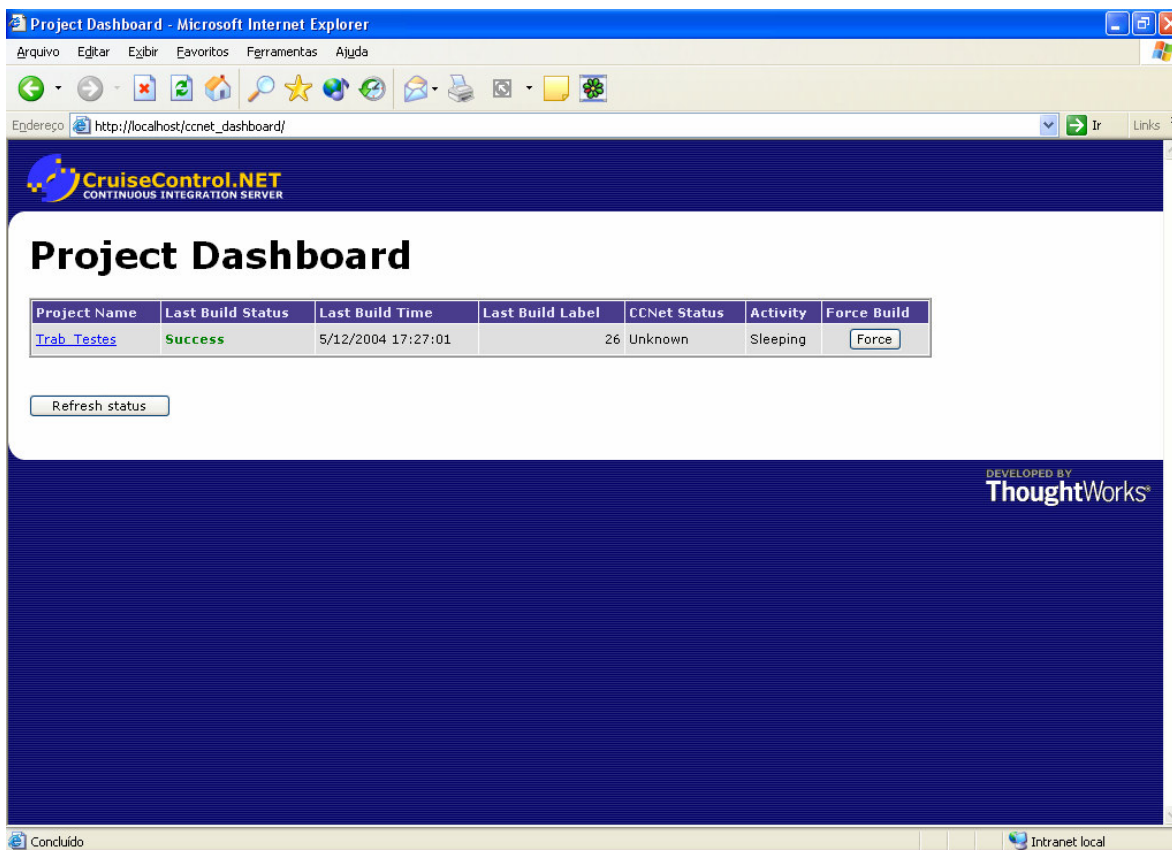

```
C:\WINDOWS\System32\cmd.exe - startCCNet

3 pasta(s) 34.275.422.208 bytes disponíveis

C:\Arquivos de programas\CruiseControl_NET\server>startCCNet

C:\Arquivos de programas\CruiseControl_NET\server>ccnet -project:Trab_Testes
[CruiseControl Server:Info]: CruiseManager: Listening on tcp://169.254.105.216:21234/CruiseManager.rem
[CruiseControl Server:Info]: Reading configuration file "C:\Arquivos de programa
s\CruiseControl_NET\server\ccnet.config"
[CruiseControl Server:Info]: Force Build for project: Trab_Testes
[CruiseControl Server:Info]: Starting integrator for project: Trab_Testes
[Trab_Testes:Info]: Starting integration for project: Trab_Testes
[Trab_Testes:Info]: Build forced
[Trab_Testes:Info]: No modifications detected.
[Trab_Testes:Info]: Building
[Trab_Testes:Info]: Build complete: Success
[Trab_Testes:Info]: Merging file: coverage.xml
[Trab_Testes:Info]: Merging file: fxcop.xml
[Trab_Testes:Info]: Merging file: TestResult.xml
[Trab_Testes:Info]: Integration complete: 5/12/2004 17:27:24
[Trab_Testes:Info]: No modifications detected.
[Trab_Testes:Info]: No modifications detected.
[Trab_Testes:Info]: No modifications detected.
[Trab_Testes:Info]: No modifications detected.
```

Inicialização do CruiseControl .NET



Página central que lista todos os projetos rodando na ferramenta CruiseControl .NET

CruiseControl.NET Build Results - Microsoft Internet Explorer

Arquivo Editar Exibir Favoritos Ferramentas Ajuda

Endereço http://localhost/CruiseControl.NET/ Ir Links »

CruiseControl.NET
CONTINUOUS INTEGRATION SERVER

latest | next | previous | project stats | server log

BUILD RESULTS
Latest Build Status: Successful
Time Since Latest Build: 3 min

05 dez 2004 17:27 (26)
02 nov 2004 17:24 (25)
02 nov 2004 17:08 (24)
02 nov 2004 17:08 (23)
02 nov 2004 16:52 (22)
02 nov 2004 16:23 (21)
02 nov 2004 16:22 (20)
02 nov 2004 16:02 (19)
02 nov 2004 15:49 (18)
02 nov 2004 15:48 (Failed)
02 nov 2004 15:46 (17)
02 nov 2004 15:43 (16)
02 nov 2004 15:41 (15)
02 nov 2004 15:40 (14)
02 nov 2004 15:39 (13)
02 nov 2004 15:32 (12)
02 nov 2004 15:28 (10)
02 nov 2004 15:15 (9)
02 nov 2004 15:13 (Failed)
02 nov 2004 15:10 (8)
02 nov 2004 15:09 (7)
02 nov 2004 14:59 (6)
02 nov 2004 14:07 (5)
02 nov 2004 14:03 (4)
02 nov 2004 14:02 (3)
02 nov 2004 13:45 (2)
02 nov 2004 13:42 (1)
02 nov 2004 13:41 (Failed)
02 nov 2004 13:40 (Failed)
02 nov 2004 13:37 (Failed)

BUILD SUCCESSFUL

Project: Trab_Testes
Date of build: 5/12/2004 17:27:01
Running time: 00:00:22

Tests run: 8, Failures: 0, Not run: 0, Time: NaN seconds
All Tests Passed

Modifications since last build (0)

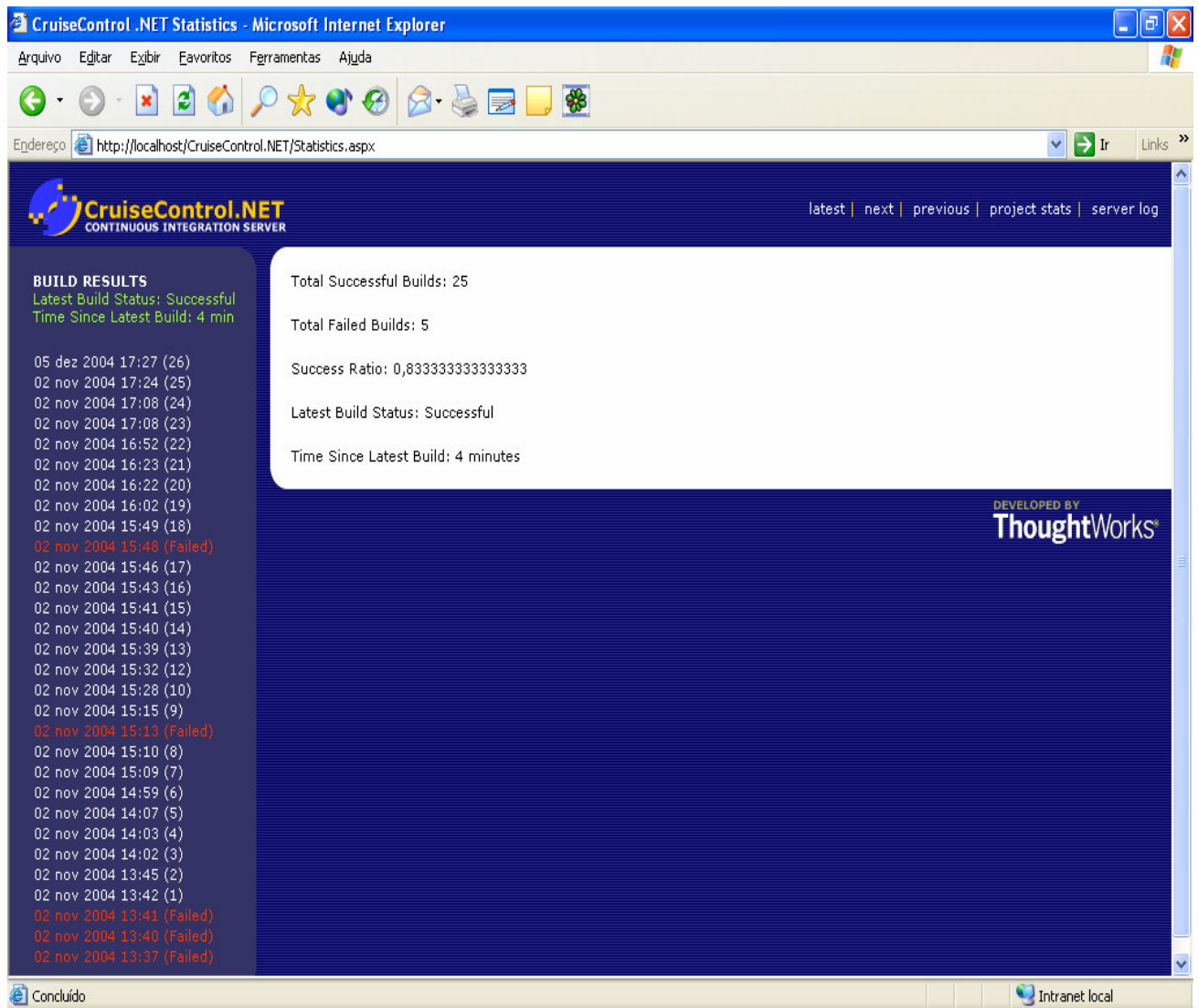
FxCop 1.312 Summary

test details | test timing | original log | FxCop | NCover

DEVELOPED BY
ThoughtWorks

Concluído Intranet local

Página inicial de um projeto, mostrando o build mais recente.



Página de estatísticas

CruiseControl.NET Build Results - Microsoft Internet Explorer

Arquivo Editar Exibir Favoritos Ferramentas Ajuda

Endereço <http://localhost/CruiseControl.NET/?log=log20041102154605Lbuild.17.xml> Links

CruiseControl.NET
CONTINUOUS INTEGRATION SERVER

latest | next | previous | project stats | server log

BUILD RESULTS
Latest Build Status: Successful
Time Since Latest Build: 6 min

05 dez 2004 17:27 (26)
02 nov 2004 17:24 (25)
02 nov 2004 17:08 (24)
02 nov 2004 17:08 (23)
02 nov 2004 16:52 (22)
02 nov 2004 16:23 (21)
02 nov 2004 16:22 (20)
02 nov 2004 16:02 (19)
02 nov 2004 15:49 (18)
02 nov 2004 15:48 (Failed)
02 nov 2004 15:46 (17)
02 nov 2004 15:43 (16)
02 nov 2004 15:41 (15)
02 nov 2004 15:40 (14)
02 nov 2004 15:39 (13)
02 nov 2004 15:32 (12)
02 nov 2004 15:28 (10)
02 nov 2004 15:15 (9)
02 nov 2004 15:13 (Failed)
02 nov 2004 15:10 (8)
02 nov 2004 15:09 (7)
02 nov 2004 14:59 (6)
02 nov 2004 14:07 (5)
02 nov 2004 14:03 (4)
02 nov 2004 14:02 (3)
02 nov 2004 13:45 (2)
02 nov 2004 13:42 (1)
02 nov 2004 13:41 (Failed)
02 nov 2004 13:40 (Failed)
02 nov 2004 13:37 (Failed)

BUILD SUCCESSFUL

test details | test timing | original log | FxCop | NCover

Project: Trab_Testes
Date of build: 2/11/2004 15:46:05
Running time: 00:00:02
Last changed: 02 nov 2004 15:45
Last log entry: Incluído o TestNegative2

Tests run: 8, Failures: 0, Not run: 0, Time: NaN seconds
All Tests Passed

Modifications since last build (1)

modified	Papo	src/TestLargest.cs	Incluído o TestNegative2

DEVELOPED BY
ThoughtWorks

Intranet local

Página de um build mostrando quem realizou modificações nele para que este rodasse.

CruiseControl.NET Build Results - Microsoft Internet Explorer

Arquivo Editar Exibir Favoritos Ferramentas Ajuda

Endereço <http://localhost/CruiseControl.NET/?log=log20041102154808.xml> Ir Links

CruiseControl.NET
CONTINUOUS INTEGRATION SERVER

latest | next | previous | project stats | server log

BUILD RESULTS
Latest Build Status: Successful
Time Since Latest Build: 6 min

05 dez 2004 17:27 (26)
02 nov 2004 17:24 (25)
02 nov 2004 17:08 (24)
02 nov 2004 17:08 (23)
02 nov 2004 16:52 (22)
02 nov 2004 16:23 (21)
02 nov 2004 16:22 (20)
02 nov 2004 16:02 (19)
02 nov 2004 15:49 (18)
02 nov 2004 15:48 (Failed)
02 nov 2004 15:46 (17)
02 nov 2004 15:43 (16)
02 nov 2004 15:41 (15)
02 nov 2004 15:40 (14)
02 nov 2004 15:39 (13)
02 nov 2004 15:32 (12)
02 nov 2004 15:28 (10)
02 nov 2004 15:15 (9)
02 nov 2004 15:13 (Failed)
02 nov 2004 15:10 (8)
02 nov 2004 15:09 (7)
02 nov 2004 14:59 (6)
02 nov 2004 14:07 (5)
02 nov 2004 14:03 (4)
02 nov 2004 14:02 (3)
02 nov 2004 13:45 (2)
02 nov 2004 13:42 (1)
02 nov 2004 13:41 (Failed)
02 nov 2004 13:40 (Failed)
02 nov 2004 13:37 (Failed)

BUILD FAILED

test details | test timing | original log | FxCop | NCover

Project: Trab_Testes
Date of build: 2/11/2004 15:48:08
Running time: 00:00:02
Last changed: 02 nov 2004 15:47
Last log entry: *** empty log message ***

Tests run: 8, Failures: 1, Not run: 0, Time: NaN seconds

Failure TestLargest.TestNegative2

Unit Test Failure and Error Details (1)

Test: TestLargest.TestNegative2
Type: Failure
Message: expected:<-11> but was:<-5>
at TestLargest.TestNegative2() in c:\Builds\Trab_Checkout\src\TestLargest.cs:line 93

Modifications since last build (1)

modified	Papo	src/TestLargest.cs	*** empty log message ***

DEVELOPED BY
ThoughtWorks

Intranet local

Página de um build que falhou, mostrando o motivo e as modificações realizadas desde o último build.

CruiseControl .NET Test Results - Microsoft Internet Explorer

Arquivo Editar Exibir Favoritos Ferramentas Ajuda


Endereço <http://localhost/CruiseControl.NET/Tests.aspx?log=log20041102154808.xml> Ir Links

CruiseControl.NET
CONTINUOUS INTEGRATION SERVER

latest | next | previous | project stats | server log

BUILD RESULTS
Latest Build Status: Successful
Time Since Latest Build: 6 min

05 dez 2004 17:27 (26)
02 nov 2004 17:24 (25)
02 nov 2004 17:08 (24)
02 nov 2004 17:08 (23)
02 nov 2004 16:52 (22)
02 nov 2004 16:23 (21)
02 nov 2004 16:22 (20)
02 nov 2004 16:02 (19)
02 nov 2004 15:49 (18)
02 nov 2004 15:48 (Failed)
02 nov 2004 15:46 (17)
02 nov 2004 15:43 (16)
02 nov 2004 15:41 (15)
02 nov 2004 15:40 (14)
02 nov 2004 15:39 (13)
02 nov 2004 15:32 (12)
02 nov 2004 15:28 (10)
02 nov 2004 15:15 (9)
02 nov 2004 15:13 (Failed)
02 nov 2004 15:10 (8)
02 nov 2004 15:09 (7)
02 nov 2004 14:59 (6)
02 nov 2004 14:07 (5)
02 nov 2004 14:03 (4)
02 nov 2004 14:02 (3)
02 nov 2004 13:45 (2)
02 nov 2004 13:42 (1)
02 nov 2004 13:41 (Failed)
02 nov 2004 13:40 (Failed)
02 nov 2004 13:37 (Failed)

Test Results:
Test Fixture Status
+ TestLargest  Progress (7/8)

DEVELOPED BY
ThoughtWorks

Concluído Intranet local

Página com detalhamento dos resultados dos testes de unidade

Conclusão

A necessidade de qualidade do software e testes está crescendo no mercado de TI. Ferramentas e boas práticas já existem no mercado, para as plataformas mais conhecidas atualmente para aplicações empresarias: J2EE e .NET. Hoje já é possível juntar e combinar as mais diversas ferramentas open source para buscar a melhoria da qualidade e maior produtividade, mas elas não são integradas de maneira uniforme.

A Microsoft e outras empresas da área de software já estão analisando esse mercado e as práticas dos desenvolvedores de software. Essa demanda está sendo notada e a Microsoft já irá atendê-la na nova versão de seu ambiente de desenvolvimento (Shirey, 2004). A IBM também não está ficando para trás, lançando sua nova suíte de ferramentas da linha de produtos da Rational.

As ferramentas mudarão, mas o processo de desenvolvimento maduro continuará similar e necessário para os nossos dias. Já é possível subirmos neste bonde rumo à maior qualidade!

Anexo

Código-fonte do exemplo usado para o trabalho:

Largest.cs:

using System;

public class Cmp
{

```
    ///
    /// <summary>
    /// Return the largest element in a list.
    /// </summary>
    /// <param name="list"> A list of integers </param>
    /// <returns>
    /// The largest number in the given list
    /// </returns>
    ///

    public static int Largest(int[] list)
    {
        int index, max=Int32.MinValue;

        if (list.Length == 0)
        {
            throw new ArgumentException("Empty list");
        }

        /*

        // ...

        */

        for (index = 0; index <= list.Length-1; index++)
        {
            if (list[index] > max)
            {
                max = list[index];
            }
        }
    }
}
```



```

        return max;
    }
}

```

TestLargest.cs:

```
using System;
```

```
using NUnit.Framework;
```

```

[TestFixture]
public class TestLargest
{
    [Test]
    public void LargestOf3()
    {
        Assert.AreEqual(9, Cmp.Largest(new int[] {9,8,7}));
        Assert.AreEqual(9, Cmp.Largest(new int[] {8,9,7}));
        Assert.AreEqual(9, Cmp.Largest(new int[] {7,8,9}));
    }

    [Test, ExpectedException(typeof(ArgumentException))]
    public void TestEmpty()
    {
        Cmp.Largest(new int[] { });
    }

    [Test]
    public void LargestOf3Alt()
    {
        int[] arr = new int[3];
        arr[0] = 8;
        arr[1] = 9;
        arr[2] = 7;
        Assert.AreEqual(9, Cmp.Largest(arr));
    }

    [Test]
    public void TestOrder()
    {
        Assert.AreEqual(9, Cmp.Largest(new int[] {9,8,7}));
    }
}

```

```

        Assert.AreEqual(9, Cmp.Largest(new int[] {7,9,8}));
        Assert.AreEqual(9, Cmp.Largest(new int[] {7,8,9}));
    }

    [Test]
    public void TestDups()
    {
        Assert.AreEqual(9, Cmp.Largest(new int[] {9,7,9,8}));
    }

    [Test]
    public void TestOne()
    {
        Assert.AreEqual(1, Cmp.Largest(new int[] {1}));
    }

    [Test]
    public void TestNegative()
    {
        int [] negList = new int[] {-9, -8, -7};
        Assert.AreEqual(-7, Cmp.Largest(negList));
    }

    [Test]
    public void TestNegative2()
    {
        int [] negList = new int[] {-18, -11, -5};
        Assert.AreEqual(-5, Cmp.Largest(negList));
    }
}

```

Referências Bibliográficas

BECK, K. – **Extreme Programming Explained**. New Jersey:Addison-Wesley Professional, 1999.

COCKBURN, A. - **Crystal Clear**, versão 5d draft. Disponível em: <http://alistair.cockburn.us/crystal/crystal.html> . Acesso em: 22 de Julho de 2004.

CLARK, M. - **Pragmatic Project Automation**. North Carolina:The Pragmatic Programmers LLC, 2004.

FIELD, J. - **CruiseControl .NET from Scratch**. Disponível em: <http://joefield.mysite4now.com/blog/articles/146.aspx> . Acesso em: 07 de nov. de 2004.

FOWLER, M.; FOEMMEL, M. - **Continuous Integration**. Disponível em: <http://www.martinfowler.com/articles/continuousIntegration.html> . Acesso em: 07 de nov. de 2004.

GEHTLAND, J. - **Continuous Integration with CruiseControl .NET and Draco .NET**. Disponível em: <http://www.theserverside.net/articles/showarticle.tss?id=ContinuousIntegration> . Acesso em: 07 de nov. de 2004a.

GEHTLAND, J - **Unit Testing in .NET**. Disponível em: <http://www.theserverside.net/articles/showarticle.tss?id=UnitTesting> . Acesso em: 07 de nov. de 2004b.

GEHTLAND, J - **Managing .NET Development with NAnt**. Disponível em: <http://www.theserverside.net/articles/showarticle.tss?id=NAnt> . Acesso em: 07 de nov. de 2004c.

HUNT, A.; THOMAS, D. - **Pragmatic Version Control using CVS**. North Carolina:The Pragmatic Programmers LLC, 2003.

HUNT, A.; THOMAS, D. - **Pragmatic Unit Testing in C# with NUnit**. North Carolina:The Pragmatic Programmers LLC, 2004.

MCCONNELL, S. – **Rapid Development**. New York:Microsoft Press, 1996.

NANTZ, B. - **Open Source .NET Development**. New Jersey:Addison-Wesley Professional, 2004.

NEWKIRK, J.; VORONTSOV, A. – **Test-Driven Development in Microsoft .NET**. New York:Microsoft Press, 2004.

ROBBINS, J. - **Bad Code? FxCop to the Rescue.** Disponível em: <http://msdn.microsoft.com/msdnmag/issues/04/06/Bugslayer/default.aspx> . Acesso em: 07 de nov. de 2004.

SHIREY, J. - **A Developer's Introduction to Visual Studio 2005 Team System.** Disponível em: <http://www.sys-con.com/story/?storyid=47003&DE=1> . Acesso em: 07 de nov. de 2004.

Web Sites das Ferramentas

NUnit - <http://www.nunit.org/>

DotNetMock - <http://sourceforge.net/projects/dotnetmock>

NCover - <http://www.ncover.org/>

FxCop - <http://www.gotdotnet.com/team/fxcop/>

NAnt - <http://nant.sourceforge.net/>

NAntContrib - <http://nantcontrib.sourceforge.net/>

CruiseControl .NET - <http://ccnet.thoughtworks.com/>

CVS - <https://www.cvshome.org/>

TortoiseCVS - <http://www.tortoisecvs.org/>