USO DE TESTES AUTOMATIZADOS EM PROJETOS DE *SOFTWARE*: ESTUDO DA APLICAÇÃO EM *SOFTWARE* COMERCIAL

MACEDO, Paulo Cesar
Faculdade Santa Lúcia
pcmacedo@me.com

MORAES, Marcos Roberto de
Faculdade Santa Lúcia
professormoraes@gmail.com

CATINI, Rita de Cássia
Faculdade Santa Lúcia
ritacatini@gmail.com

RESUMO

A área de engenharia de software vem experimentando, nos últimos 10 anos, uma mudança de paradigmas com relação ao advento das metodologias ágeis. Além das mudanças na forma de desenvolvimento existe uma exigência muito grande nas entregas de partes do software utilizadas como diretrizes dessas metodologias. O objetivo deste artigo é apresentar uma introdução prática a testes de software, ou seja, um forma de testar os métodos produzidos pelos desenvolvedores de software em suas aplicações através do framework da Microsoft© utilizando a linguagem C# e um ambiente de testes. Como resultado, obteve-se um modelo de exemplo de casos de testes que pode ser usado em projetos de sistemas de qualquer natureza.

PALAVRAS-CHAVE: *metodologias ágeis; teste de unidade; testes automatizados; framework.*

Recebido em 03-05-2012 • Aceito em 27-08-2012

INTRODUÇÃO

As metodologias ágeis de desenvolvimento de *software* se destacam dos processos de desenvolvimento tradicionais principalmente pelo fato de priorizarem a implementação de códigos executáveis ao invés de documentação escrita. Sendo assim, se torna relevante refletirmos sobre como podemos garantir que requisitos funcionais e não funcionais estão sendo atendidos nesta atividade, considerando a falta de amparo de artefatos documentais nessas metodologias.

Tradicionalmente, no desenvolvimento de *software*, procura-se garantir que existe fidelidade do que está sendo desenvolvido em relação aos requisitos através da execução de testes. Essencialmente, os requisitos necessários para realização de testes quando são utilizados paradigmas ágeis não são muito diferentes quando comparados aos métodos de desenvolvimento tradicionais. A diferença está no grau de importância que é conferida aos testes, considerando cada uma das abordagens. Nas metodologias tradicionais, tais como cascata, espiral e prototipagem, percebe-se a existência de uma extensa lista de artefatos documentais resultantes da análise sobre o projeto que será desenvolvido. Neste contexto, os testes são vistos apenas como mais um artefato produzido pelo processo, que visa garantir a detecção de erros antes da liberação do produto. Nos processos ágeis não existe amparo documental, sendo que os testes, neste caso, passam a ter um papel importantíssimo para o sucesso da metodologia ágil utilizada.

Uma das primeiras questões que se pode pensar quando é preciso definir estratégias de teste de *software* pode estar relacionado com o uso de testes manuais ou testes automatizados. Os testes manuais, como o próprio nome sugere, são realizados pelos desenvolvedores ou pelo próprio usuário final, com o objetivo de verificar anomalias, comparando com o que havia sido combinado inicialmente (CRISPIN; HOUSE, 2005). O problema do teste manual é que dificilmente alguém consegue testar exaustivamente um sistema de forma que não reste nenhuma possibilidade de falha, pois sempre haverá a possibilidade de esquecermos algum detalhe ou possibilidade de uso que pode gerar um erro.

O aspecto positivo deste tipo de teste é que mesmo considerando tal fragilidade, são fáceis de serem aplicados, uma vez que basta utilizar o sistema como se estivesse em produção. Para Crispin e House (2005), os testes automatizados utilizam aplicativos específicos que são capazes de realizar testes exaustivos através de *scripts* de testes pré-configurados.

Recebido em 03-05-2012 • Aceito em 27-08-2012

Mesmo considerando uma quantidade de situações bem mais abrangente que um teste manual, não devemos considerar que proporcionam 100% de cobertura, pois existem falhas lógicas que só ocorrem através de combinações específicas de entrada de dados. Um aspecto que dificulta o uso dessas ferramentas é o fato de precisar de um profissional especializado que, além do domínio da ferramenta, deve possuir qualificação para o domínio dos testes. Essas características têm sido consideradas fatores complicadores sobre o uso de testes automatizados. Independente da estratégia utilizada, testes de *software* são considerados essenciais nas metodologias ágeis, pois estão diretamente relacionados com a garantia da qualidade do produto.

Resumidamente, podemos observar que as atividades de teste nas metodologias ágeis estão relacionadas aos testes de unidades e também aos testes de aceitação, pelo fato de contarmos com a participação do cliente durante o processo de desenvolvimento. É curioso notar que, embora os testes representem grande parcela do alicerce que sustenta as metodologias ágeis, pouca ou nenhuma menção a atividades de teste é encontrada, com exceção do método *Extreme Programming,* que explicitamente enfatiza essas atividades.

Do ponto de vista prático, os programadores que não usavam metodologia nenhuma pouco se importavam com esta fase quando implementavam seus projetos, desta forma deixavam que os problemas surgissem no momento da utilização e iam corrigindo-os aos poucos. Quase sempre deixavam um legado de erros sem correção ou utilizavam de remendos para acobertá-los.

O objetivo principal deste artigo é propor uma forma de testar partes de um projeto de *software* na prática, oferecendo um exemplo de um documento de teste e de uma implementação simples em um *framework* de alto nível, ou seja, utilizando uma linguagem atual de desenvolvimento.

2.TESTES DE *SOFTWARE*

Na engenharia de *software* ágil existe uma preocupação com os testes de *software*, principalmente pela característica de entregas rápidas que elas possuem. Do ponto de vista prático, os programadores que não usavam metodologia nenhuma não se importavam com esta fase quando implementavam seus projetos. Desta forma, deixavam que os problemas surgissem no momento da utilização e iam corrigindo-os aos poucos. Quase sempre deixavam um legado de erros sem correção ou utilizavam de remendos para

Recebido em 03-05-2012 • Aceito em 27-08-2012

acobertá-los(CHARETTE, 2001). Segundo Delamaro, Maldonado e Jino (2007), testar o *software* significa verificar o comportamento do sistema usando estímulos e entradas em busca de erros, verificar se o sistema foi feito dentro dos padrões pré-estabelecidos, além de analisar se os resultados de sua utilização correspondem aos processos e ao esperado. Partindo desse conceito, testar indica aumentar os custos, ou seja, para executar os processos de verificação e validação através de testes leva tempo e pode envolver vários profissionais.

Para Maldonado (1991), os testes de *software* estão divididos em duas categorias básicas: Testes Funcionais e Testes Estruturais sendo que ambos os tipos podem ser automatizados, permitindo que sejam previamente criados para testar alguns tipos de interações. A **Tabela 1** apresenta os tipos de testes e suas implicações.

Tabela 1 – Lista dos tipos de testes

Testes Funcionais	Descrição do teste	Quando executar
Tratamento de Erros	Verifica se o sistema valida todas as transações e se retorna todas as mensagens de erros (feedback) no caso de receber informações erradas.	Em todo o ciclo de desenvolvimento
Requisitos	Verifica se o sistema executa todas funcionalidades conforme o elicitado no documento de requisitos.	A cada iteração
Regressão	Testa o impacto de novas funcionalidades sobre as já existentes e deve também ser realizada na documentação, principalmente se houver alterações.	A cada iteração
Suporte Manual	Testa o sistema de ajuda (help) sensível ao contexto e verifica se a documentação está completa e devidamente atualizada	Na homologação (entrega do sistema)
Controle	Testa se o processamento corresponde ao esperado principalmente em procedimentos alheios à aplicação, por exemplo: Backups e recuperação de dados	Pode ser executado a qualquer momento
Interconexão	Garante a comunicação dos módulos desenvolvidos, bem como sua integração com outros sistemas (se houver)	Pode ser executado a qualquer momento
Paralelo	Testa paralelamente o sistema em comparação ao sistema antigo, ou seja, verifica se atende as mesmas especificações, comparando resultados.	A cada iteração

Recebido em 03-05-2012 • Aceito em 27-08-2012

Testes Estruturais	Descrição do teste	Quando executar
Unidade	Testa as funções, classes e objetos do sistema, considerando a performance e a lógica utilizada.	Em todo ciclo de desenvolvimento
Execução	Analisa o desempenho do sistema com dados reais, testando a performance com múltiplos acessos simultaneamente.	No início e na homologação
Estresse	Testa os limites máximo e mínimo do sistema, afim de avaliar seu comportamento em condições adversas.	A cada iteração e na homologação.
Recuperação	Testa como um sistema pode recuperar-se de problemas físicos ou lógicos desde falhas elétricas até de componentes de hardware e rede. (contingência)	Na homologação ou a qualquer momento.
Operação	Avalia o funcionamento do sistema, comparando com os processos manuais da empresa.	A cada iteração
Conformidade	Verifica se o sistema foi feito de acordo com as normas e padrões previamente estabelecidos. (Patterns)	Em todo ciclo de desenvolvimento
Segurança	Teste de confiabilidade que assegura se o sistema está preparado para impedir acessos não autorizados ou invasões, protegendo seus dados quanto a isso.	Na homologação ou a qualquer momento.
Integração	Realizado pelo analista conferindo e validando o que foi proposto nos casos de uso e o que foi projetado.	A cada iteração
Sistemas	Testa todo o sistema utilizando cenários pré-estabelecidos a fim de confrontar resultados.	Na homologação ou a qualquer momento.
Aceitação	Testa a validação final do sistema junto ao cliente, liberando para o usuário final.	Na homologação

Fonte: Adaptado de Sbrocco e Macedo (2012)

As metodologias ágeis utilizam desses mecanismos de testes em suas práticas de desenvolvimento. Algumas optam por mais de um tipo que combinados geram resultados satisfatórios e garantem um *software* de qualidade. Segundo Sbrocco e Macedo(2012), um bom teste de *software,* independente do tipo, deve ser planejado e devidamente documentado. Recomenda-se criar um plano de testes contendo todos os testes realizados bem como seu detalhamento.

Recebido em 03-05-2012 • Aceito em 27-08-2012

3. PESQUISA SOBRE TESTES DE *SOFTWARE*

A fase de teste de *software* em um projeto pode ser considerada muito custosa, e para que seja bem feita requer muito planejamento. Para muitas empresas desenvolvedoras, o teste de *software* acaba sendo descartado deixando que o próprio usuário dê o *feedback* sobre as alterações necessárias; desta forma, o custo com o a manutenção do *software* aumenta enquanto a confiabilidade diminui. Existem inúmeras pesquisas sobre esse assunto ao redor do mundo sempre na intenção de reduzir gastos com manutenção e melhorar a qualidade dos sistemas desenvolvidos. No inicio dos anos 80 os testes em sua maioria eram manuais, feitos através de *checklists*, ou seja, o próprio desenvolvedor tratava de testar suas funcionalidades. Já em meados dos anos 90, iniciaram-se os testes automatizados pois observou-se que perdia-se muito tempo checando funcionalidades em *software* que não paravam de crescer.(ROBERT, 1999).

Para Chillarege (1999), a meta de execução de testes automatizados é de minimizar a quantidade de trabalho manual envolvido na execução do teste e obter uma maior cobertura com um maior número de casos de teste.

Segundo Robert (1999) as organizações são muitas vezes incapazes de perceber os benefícios desejados. Eles se aproximam de automação de teste a partir da perspectiva de um programa de melhoramento que envolve investimentos tradicionais em *hardware*, pessoal, *software*. No entanto, os benefícios da automação de teste podem ser estendidos ainda mais, trazendo inovações e eficiências ao longo do desenvolvimento do produto por inteiro e seu ciclo de vida. Então os testes juntamente com uma abordagem de cálculo do ROI(retorno do investimento), permitem que a estratégia de automação possa ser avaliada quantitativamente, levando as empresas desenvolvedoras de *software* a questionar o porque não testar o *software* aumentando a importância deste tipo de artigo.

Para Jenkins (2009) não há simplesmente nenhuma maneira de garantir que o *software* estará livre de falhas. Embora *software* seja matemática por natureza, não pode ser comprovado como um teorema matemático; *software* é mais como uma linguagem, com ambiguidades inerentes, com diferentes definições, diferentes premissas, diferentes níveis de significado que podem perfeitamente entrar em conflito.

Recebido em 03-05-2012 • Aceito em 27-08-2012

4.METODOLOGIA

Para ajudar na resolução da falta de testes nos *softwares* desenvolvidos, este trabalho propõe a criação de um plano de testes e casos de testes que podem ser utilizados como ferramenta de apoio durante o desenvolvimento.A metodologia desenvolvida apresentou, num primeiro momento, uma proposta de modelos de planos e casos de testes, para depois usá-los numa aplicação feita no *framework* Visual Studio 2010 versão *Professional da Microsoft Corporation*©. A razão da escolha desta ferramenta se dá ao fato de sua grande utilização nas instituições de ensino superior na área de tecnologia da informação e por considerar que o assunto deste artigo possa ser aplicado em outras linguagens de forma similar.

4.1 PLANO DE TESTES

É definido comoo documento que descreve o funcionamento e os responsáveis pelos testes no sistema, além de todos os dados das funcionalidades ou sistema a ser testado (SBROCCO; MACEDO, 2012). A **Tabela 2** mostra um modelo proposto contendo dados necessários para sua elaboração.

Tabela 2 – Lista dos tipos de testes

Plano de Testes			
Projeto:			
Data:		Versão revisada:	
Responsáveis:			

Testes a serem realizados			
1. Teste de aceitação		Responsável:	<nome responsável>
2. Teste de regressão		Responsável:	<nome responsável>
...		...	

Para cada tipo de teste listado no plano de testes deve ser providenciado um documento contendo um detalhamento do teste realizado bem como seus resultados. Na **Tabela 3** um exemplo de proposta de teste de aceitação.

Recebido em 03-05-2012 • Aceito em 27-08-2012

Tabela 3 – exemplo para teste de aceitação

Teste de aceitação		
Projeto:		
Data Realização	<data>	Responsável: <nome do analista ou coordenador>

Objetivos do teste
Testar e validar a versão final do sistema junto ao cliente, liberando para o usuário final. Verificar indicadores de boa usabilidade da versão final.

Escopo do teste:	<colocar aqui os detalhes de como vai acontecer o teste>	
Histórico:	<data>	<responsável>
	<data>	<responsável>

Itens a serem testados	
ID	Descrição
1	
2	
3	
4	
5	
...	

Itens que não serão testados	
ID	Descrição
1	
2	
3	
...	

Critérios utilizados
<incluir uma breve análise dos riscos>

Resultados dos testes	
ID	Descrição dos resultados
Aprovado por:	<nome><data> e <assinatura>

Destaca-se que para cada tipo de teste deve haver um documento correspondente, contendo detalhes e este exemplo apresentado é o conteúdo mínimo a ser preenchido.

Um tipo de teste que a maioria das metodologias ágeis utiliza é o Teste de unidade que, como citado anteriormente, garante que funcionalidades,

Recebido em 03-05-2012 • Aceito em 27-08-2012

objetos e classes trabalhem com uma lógica livre de erros. Uma das formas de realizar esse teste é criar uma implementação automatizada que, através de alguns procedimentos, conseguem aferir se o sistema está realmente trabalhando corretamente (SBROCCO; MACEDO,2012).

Para entendimento desta proposta toma-se como exemplo o desenvolvimento de uma operação/funcionalidade do sistema que calcule um parcelamento de uma dívida (**Figura 1**).

Procedimento Gera_parcelas (NumP: inteiro, DataAtual: data e hora, Valor: moeda, TabJuros: inteiro)

Figura 1 – Função de parcelamento

São passados como parâmetros o número de parcelas (NumP), a data atual (DataAtual) e a tabela de juros utilizada (TabJuros). Em seguida, assim que o método recebe esses parâmetros calcula as parcelas e as datas de vencimento de acordo com a tabela de juros indicada. Um teste unitário cria no ambiente de desenvolvimento ou anexo a ele (como no caso de *frameworks* modernos) um método que recebe os dados reais e calcula os resultados para comparar com os do sistema (**Figura 2**).

```
Procedimento TestaParcela( )
        Datahoje=05/10/2011;
        Valorhoje=R$300,00;
TabelaJuro=1;
NumeroParcelas=3;
        Parcela30=05/11/2011;
        Parcela60=05/12/2011;
Parcela90=05/01/2012;
Valor30=R$306,75;
        Valor60=R$312,88;
        Valor90=R$337,91;
        Gera_parcelas(NumeroParcelas,Datahoje,Valorhoje,TabelaJuro)
Se (Parcela30=Parcela30_obtida). E . (Parcela60=Parcela60_obtida) . E . (Parcela90=Parcela90_
obtida) então "Datas testadas estão OK" senão "Problemas nas datas"

Se (Valor30=Valor30_obtido). E . (Valor60=Valor60_obtido) . E . (Valor90=Valor90_obtido) então
"Valores testados estão OK"senão "Problemas nos valores"
```

Figura 2 – Cálculo dos resultados

Recebido em 03-05-2012 • Aceito em 27-08-2012

Os dados resultantes devem ser comparados com os dados do sistema a fim de localizar um possível erro. Obviamente este foi apenas um exemplo, podendo ter muito mais detalhes em uma implementação real. O que deve ser entendido nesse momento é a possibilidade de usar um *software* para testar outro e comparar os resultados.

4.2 CASOS DE TESTE

Para cada funcionalidade a ser testada se faz necessária a criação de casos de testes, esses casos devem acompanhar cada caso de uso da modelagem a fim de checar seus resultados. A **Tabela 4** apresenta um exemplo do que deve conter um caso de teste:

Tabela 4 – Exemplo de caso de teste

Casos de teste		
ID:	<nome do Caso>	
Itens a serem testados		
1	<descrição do item>	
2	<descrição do item>	
...		
Entradas	Campo	Valor

	Campo	Valor
Saídas		

Procedimento/função – Método (sistema)	Procedimento (Teste)
Aprovação:	Data:

Recebido em 03-05-2012 • Aceito em 27-08-2012

É indiscutível a importância dos testes nas metodologias ágeis de desenvolvimento; torna-se inviável entregar um *software* sem nenhum tipo de teste. Os testes concretizam o fato de que prevenir é melhor que remediar, ou seja, o tempo gasto testando os componentes de *software* é economizado após a entrega das funcionalidades.

4.3 EXPERIMENTO

Dificilmente podem-se escrever testes unitários sem a utilização de ferramentas. Para este experimento será utilizado como ferramenta o Visual Studio 2010 versão Professional da Microsoft Corporation. Nesta IDE (*Integrated Development Environment*)ou Ambiente Integrado de Desenvolvimento estão incorporadas funções como geração de testes unitários, que podem ser realizadas de forma simples, como será demonstrado a seguir.

Inicialmente é importante que se crie um projeto específico para testes dentro do Visual Studio Professional vinculado com a aplicação que se está desenvolvendo – aquela a ser testada. Neste experimento serão aplicadas algumas fórmulas para resolução de cálculos sobre juros simples; neste regime, o percentual de juros incide apenas sobre o valor principal, sobre os juros gerados a cada período não incide novos juros. Valor Principal ou simplesmente principal é o valor inicial emprestado ou aplicado, antes de somarmos aos juros, conforme orientam Pilão e Hummel (2003).

$J = P . i . n$

Onde:
J = juros
P = principal (capital)
i = taxa de juros
n = número de períodos

Exemplo: Temos uma dívida de R$ 2000,00 que deve ser paga com juros de 5% a.m. pelo regime de juros simples e devemos pagá-la em 5 meses. Os juros que deverão ser pagos:
$J = 2000 \times 0.05 \times 5 = 500$

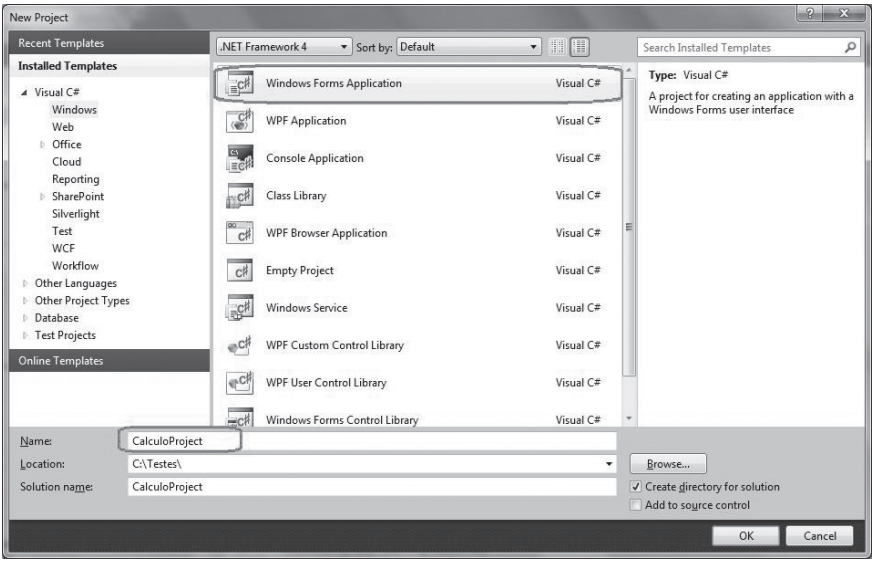
Para se obter o valor montante, soma-se os juros obtidos ao valor

Recebido em 03-05-2012 • Aceito em 27-08-2012

principal, ou seja,Montante = principal + juros. Ou ainda Montante = Principal * (1 + (i . n)). Como restrição para esse experimento não se pode ter uma taxa, um período e uma taxa de juros menor que zero, ou seja, valores negativos (PILÃO; HUMMEL,2003).

a) Criando a aplicação

Apesar do TDD(*Test Driven Development)*orientar a criação dos testes propriamente ditos e depois o código, optou-se neste artigo em realizar o inverso, ou seja, criar a classe principal responsável pelos cálculos para facilitar ao leitor o entendimento da ferramenta, para que depois de familiarizada ele possa primeiro criar os testes.Na IDE do Visual Studio 2010 será criado o projeto CalculoProject com o *template Windows Forms Application* como ilustra a **Figura 3**:



Fonte: *Microsoft Visual Studio*[1] (2010)
Figura3 – Windows Form Application

[1] Microsoft Visual Studio - é marca registrada ou marca comercial da Microsoft Corporation nos Estados Unidos e/ou outros países.

Recebido em 03-05-2012 • Aceito em 27-08-2012

A classe Calculo, apresentada na **Figura 4**, possui 04 campos privados (propriedades automáticas na linguagem C#) responsáveis pelo armazenamento dos juros, valor principal (capital), taxa de juros e número de períodos; também possui dois métodos: a) calcularJuros Simples() e b) calcularMontante() responsáveis pelos cálculo dos juros simples e do valor do capital, respectivamente. Nos dois métodos são verificadas as restrições utilizando-se blocos de tratamento de erros (*try-catch*).

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace CalculoProject
{
/* Legenda
    * j = juros
    * p = principal (capital)
    * i = taxa de juros
    * n = número de períodos
    *
   *
   */
/// <summary>
/// Classe Calculo: Reponsável pelo calculo de juros simples
/// e de valor principal (capital).
/// </summary>
publicclassCalculo
   {
public decimal j { set; get; }
public decimal p { set; get; }
public decimal i { set; get; }
public int n { set; get; }
/// <summary>
/// Método responsável pelo Cálculo dos Juros simples
/// </summary>
```

Figura 4 – Código de exemplo – parte I

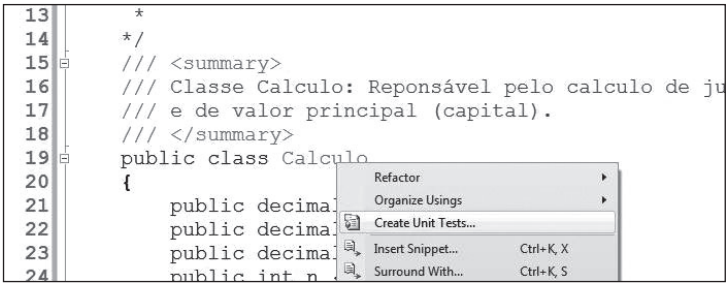
Recebido em 03-05-2012 • Aceito em 27-08-2012

```
/// <returns>Valor dos Juros</returns>
public decimal calcular JurosSimples()
{
try
        {
if (this.p< 0 || this.i< 0 || this.n< 0)
throw new ArgumentException("Valores negativos não são permitidos");
this.j = this.p * this.i * this.n;
returnthis.j;
        }
catch (ArgumentException ex)
        {
throw ex;
}
    }
/// <summary>
/// Método responsável pelo Cálculo do Valor principal
/// </summary>
/// <returns>Valor Principal</returns>
public decimal calcularMontante()
    {
try
        {
if (this.p< 0 || this.i< 0 || this.n< 0)
throw new ArgumentException("Valores negativos não são permitidos");
this.p = this.p * (1 + (i * n));
return this.p;
        }
catch (ArgumentException ex)
        {
throw ex;
        }
    }
  }
}
```

Figura 4 – Código de exemplo – parte II

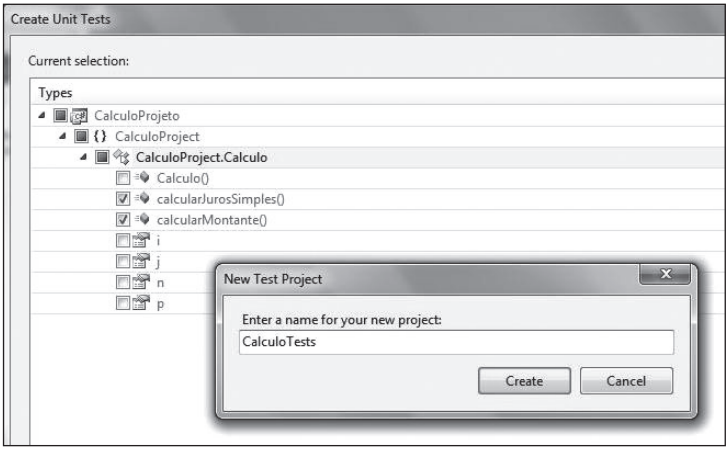
Recebido em 03-05-2012 • Aceito em 27-08-2012

Para criação automática do *template* para testes unitários na IDE do Visual Studio pode-se acessar com um clique com o botão direito do mouse, sobre o nome da classe criada, selecionando a opção *Create Unit Tests*, como mostra a **Figura 5**.



Fonte:*Microsoft Visual Studio*(2010)
Figura 5 – Código da classe cálculo

Devem-se marcar os métodos para os quais se pretende criar testes, o Visual Studio irá gerar uma estrutura do teste automaticamente. Deve-se então preencher o *template* gerado com a lógica de verificação. No caso desse experimento, devem ser selecionados apenas os dois métodos a serem testados. Como mostra a **Figura 6**:



Fonte:*Microsoft Visual Studio*(2010)
Figura 6 - Criando unidade de testes

Recebido em 03-05-2012 • Aceito em 27-08-2012

A seguir, na **Figura 7**, criada automaticamente pelo Visual Studio. É importante observarmos algumas informações:
i) A classe de teste possui um atributo chamado *TestClass*. Atributo que indica esta como uma classe especial que contém as rotinas que serão testadas.
ii) Todos os métodos de testes possuem um atributo chamado *TestMethod.* Indica o método como um teste.
iii) A classe *Assert* que se apresenta na listagem é a classe mais usada quando se trabalha com testes. Possui uma coleção de métodos estáticos que disparam mensagens em uma janela conhecida por *Test Results*. A **Figura 7** apresenta a descrição dos métodos estáticos da classe.

```
using CalculoProject;
using Microsoft.VisualStudio.TestTools.UnitTesting;
using System;
namespace CalculoTests
{
/// <summary>
///This is a test class for CalculoTest and is intended
///to contain all CalculoTest Unit Tests
///</summary>
   [TestClass()]
public class CalculoTest
   {
private TestContext testContextInstance;

/// <summary>
///Gets or sets the test context which provides
///information about and functionality for the current test run.
///</summary>
public TestContext TestContext
      {
get
         {
Return testContextInstance;
         }
set
```

Figura 7 – Código de exemplo – parte I

Recebido em 03-05-2012 • Aceito em 27-08-2012

```
        {
testContextInstance=value;
        }
    }

/// <summary>
///A test for calcularMontante
///</summary>
    [TestMethod()]
public void calcularMontanteTest()
    {
Calculo target=new Calculo(); // TODO: Initialize to an appropriate value
Decimal expected=new Decimal(); // TODO: Initialize to an appropriate value
Decimal actual;
actual= target.calcularMontante();
Assert.AreEqual(expected, actual);
Assert.Inconclusive("Verify the correctness of this test method.");
    }
/// <summary>
///A test for calcularJurosSimples
///</summary>
    [TestMethod()]
public void calcularMontanteTest()
    {
Calculo target=new Calculo(); // TODO: Initialize to an appropriate value
Decimal expected=new Decimal(); // TODO: Initialize to an appropriate value
Decimal actual;
actual= target.calcularJurosSimples();
Assert.AreEqual(expected, actual);
Assert.Inconclusive("Verify the correctness of this test method.");
    }
/// <summary>
///A test for calcularJurosSimples
///</summary>
    [TestMethod()]
public void calcularJurosSimplesTest()
```

Figura 7 – Código de exemplo – parte II

Recebido em 03-05-2012 • Aceito em 27-08-2012

```
        {
Calculo target=new Calculo(); // TODO: Initialize to an appropriate value
Decimal expected=new Decimal(); // TODO: Initialize to an appropriate value
Decimal actual;
actual= target.calcularJurosSimples();
Assert.AreEqual(expected, actual);
Assert.Inconclusive("Verify the correctness of this test method.");
}
    }
}
```

Figura 7 – Código de exemplo – parte III

Tabela 5 – Métodos estáticos

Método	Descrição
AreEqual()	Compara a igualdade entre dois valores do mesmo tipo.
AreNotEqual()	Compara se dois valores são diferentes.
IsFalse()	Verifica se o valor passado por parâmetro é falso.
IsTrue()	Verifica se o valor passado por parâmetro é verdadeiro.
AreSame()	Compara se dois objetos são iguais.
AreNotSame()	Compara se dois objetos são diferentes.
Fail()	Retorna falha.
Inconclusive()	Retorna teste inconclusivo.
IsInstanceOfType()	Verifica se um objeto é de um determinado tipo.
IsNotInstanceOfType()	Verifica se um objeto não é de um tipo específico.
IsNull()	Verifica se o objeto é nulo.
IsNotNull()	Verifica se o objeto não é nulo.

Antes de realizar os testes é necessário que alteremos os métodos de testes com valores válidos e inválidos,para verificar se o teste foi bem sucedido. Para isso deve-se alterar os métodos carcularJurosSimplesTest() e calcularMontanteTest() de acordo com a **Figura 8.**

Recebido em 03-05-2012 • Aceito em 27-08-2012

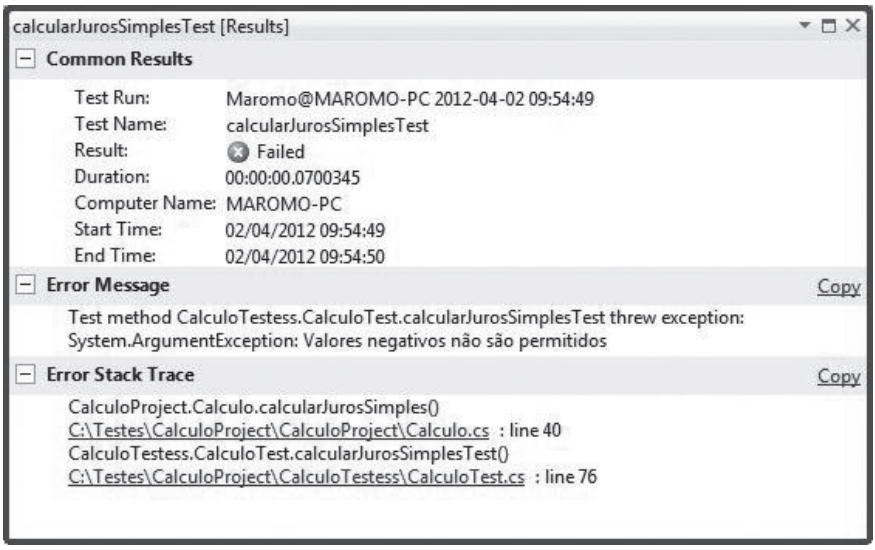
```
[TestMethod()]
publicvoidcalcularJurosSimplesTest()
        {
Calculo target = new Calculo(); // TODO: Initialize to an appropriate value
            Decimal expected = new Decimal(); // TODO: Initialize to an
appropriate value
           Decimal actual;
expected = 750;
target.p = 5000;
target.i = (Decimal)0.05;
target.n = -3;
actual = target.calcularJurosSimples();
Assert.AreEqual(expected, actual);
        }

[TestMethod()]
public void calcularMontanteTest()
        {
Calculo target = new Calculo(); // TODO: Initialize to an appropriate value
            Decimal expected = new Decimal(); // TODO: Initialize to an
appropriate value
Decimal actual;
expected = 5750;
target.p = 5000;
target.i = (Decimal)0.05;
target.n = 3;
actual = target.calcularMontante();
Assert.AreEqual(expected, actual);
        }
```

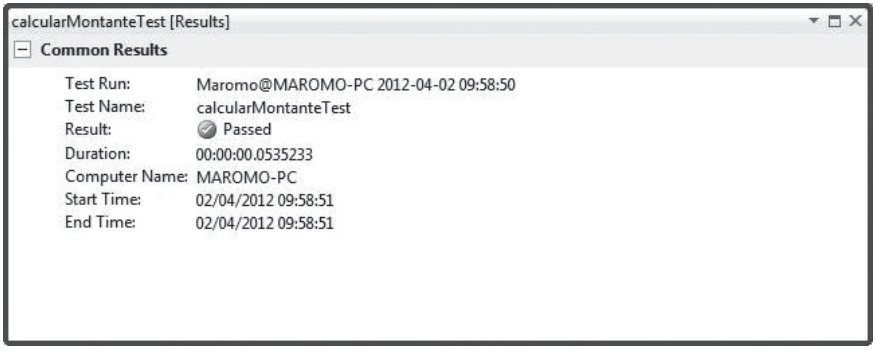
Figura 8 – Código de exemplo

Ao se executar os testes, no primeiro caso teremos uma falha, já que o valor (-3) é negativo. No segundo teste obteremos êxito, pois os valores passados como parâmetros resultam do cálculo o valor esperado. As **Figuras 9 e 10** mostram os resultados na janela de testes dos métodos calcular JurosSimplesTest() e calcular MontanteTest(), respectivamente.

Recebido em 03-05-2012 • Aceito em 27-08-2012



Fonte: Microsoft Visual Studio (2010)
Figura 9 – Juros simples



Fonte: Microsoft Visual Studio (2010)
Figura 10 – Montante

4.4 RESULTADOS E DISCUSSÃO

O *software* pode ser testado em vários estágios de desenvolvimento, com vários graus de rigor e, como qualquer atividade de desenvolvimento, os testes consomem custos e esforço. Os desenvolvedores

Recebido em 03-05-2012 • Aceito em 27-08-2012

devem planejar entre 30% e 40% do tempo a serem gastos em atividades de verificação e validação, incluindo testes de *software* (JENKINS, 2009).

Do ponto de vista da economia, o nível de testes apropriados para uma determinada organização e aplicação de *software* dependerá das potenciais consequências de erros não detectados. Tais consequências podem variar de um pequeno inconveniente de ter de perder um trabalho todo por um erro de diversas fontes.

Muitas vezes ignorada pelos desenvolvedores de *software* (mas não pelos clientes), pode causar dano a longo prazo para a credibilidade de uma organização que oferece *software* para usuários com *bugs*, e o impacto negativo em negócios futuros. Por outro lado, a reputação de *software* confiável irá ajudar uma organização a obter melhores negócios no futuro.

Segundo Pressman (2006), estudos apontam que realizar atividades de teste ao longo do processo de desenvolvimento de software pode ser eficaz, reduzindo custos na produção. Sendo assim, eficiência e qualidade são melhores servidas por um *software* testado desde o início de seu ciclo de vida; os testes devem ser usados como prática, com testes de regressão completa sempre que alterações forem feitas. Quanto mais tarde um erro for encontrado, maior o custo de corrigi-lo, por isso é uma considerada uma boa prática identificar e corrigir os erros o mais cedo possível. O ato de projetar os testes irá ajudar a identificar erros, mesmo antes dos testes serem efetivamente executados.

Na prática, a concepção de cada nível de teste de *software* deve ser desenvolvido através de um número de camadas, cada uma adicionando mais detalhes para os testes. Cada nível de testes deve ser concebido antes da aplicação atingir um ponto, que poderia influenciar a concepção de testes de tal forma a ser prejudicial para a objetividade dos mesmos. Lembrando que o *software* deve ser testado contra o que está especificado para fazer, e não contra o que ele realmente observou a fazer. Os testes mostram a presença e não a ausência de defeitos (SOMMERVILLE, 2004).

A eficácia do esforço de teste pode ser maximizado pela seleção de uma estratégia de ensaio adequada, boa gestão do processo de teste e uso adequado de ferramentas para apoiar esse processo. O resultado líquido será um aumento na qualidade e uma diminuição dos custos, os quais só podem ser benéficos para uma empresa de desenvolvedores de *software*.

Recebido em 03-05-2012 • Aceito em 27-08-2012

CONSIDERAÇÕES FINAIS

Após o experimento podemos constatar que as ferramentas de desenvolvimento atuais oferecem recursos que facilitam os testes de *software*, indicando não haver razões para não faze-lo. Erros durante o desenvolvimento de um *software* sempre irão ocorrer, independente do paradigma adotado. Isso ocorre pelo fato de estarmos interagindo com pessoas e também usando ferramentas, que podem estar com *bugs* (erros no *software*), entre outras razões. Assim, considerando que as metodologias ágeis não são orientadas a documentação (justamente para colaborar com a desburocratização do processo de desenvolvimento), é comum refletirmos sobre a adequação do uso dessas metodologias quando decidirmos pelo uso de um modelo de qualidade. Na engenharia de *software*, o custo gerado por um erro tende a aumentar na medida em que esse erro se propaga pelo ciclo de vida do *software*. Dessa forma, quanto mais tarde se descobrir um erro, maior será o prejuízo. Quando pensamos em qualidade de *software*, devemos deixar claro que ela é obtida quando percebemos que o produto desenvolvido está em conformidade com os requisitos funcionais e não funcionais. Portanto, a ocorrência de não conformidades com os requisitos representa a falta de qualidade. Também existem diversas perspectivas sob as quais podemos analisar a qualidade do *software*. Sob o ponto de vista do usuário final, s*oftware* com qualidade é aquele que atende suas necessidades, se mostrando fácil de operar, além de demonstrar ser confiável. Já para um desenvolvedor, software com qualidade é aquele que permite uma manutenção rápida para atender as necessidades do cliente. No caso do cliente, a qualidade do *software* pode ser medida pelo valor que ele agrega à organização e pelo seu ROI (*Return of Investment*). Certamente podemos incluir outros elementos a serem considerados para avaliar a qualidade de um *software* como, por exemplo, o fato do *software* ter ou não sido entregue no prazo e custos previstos. De uma maneira resumida, podemos dizer que um fator decisivo para o sucesso de qualquer projeto de *software* é garantir um mínimo de qualidade, o que normalmente está diretamente relacionado com os investimentos feitos em prol do controle de qualidade, normalmente divididos entre atividades de prevenção de erros, avaliação e identificação de falhas.

REFERÊNCIAS BIBLIOGRÁFICAS

BACH, J.. **Heuristic Risk-Based Testing -** First published in *Software* Testing and Quality Engineering Magazine, 11/99 Copyright ,1999

Recebido em 03-05-2012 • Aceito em 27-08-2012

CHARETTE, R.. **Fair Fight? Agile versus Heavy Methodologies** Cutter Consortium E--Project – Management Advisory Service – 2 – 13, (2001)

CHILLAREGE, R.. **Software Testing Best Practices -** Center for Software Engineering IBM Research, 1999.

CRISPIN, L.; HOUSE, T.. **Testing Extreme Programming**, Addison Wesley, 2005

DELAMARO, M.; MALDONADO, J.; JINO, M..**Introdução ao Teste de Software**, Campus, 2007.

JENKINS, N.. **A Software Testing Primer, An Introduction to Software Testing -** This work is licensed under the Creative Commons (Attribution-Non Commercial-Share A like) 2.5 License. – 2008

MALDONADO, J.. **Critérios Potenciais Usos:** Uma Contribuição ao Teste Estrutural de Software. PhD thesis, DCA/FEE/UNICAMP, Campinas, SP, Julho 1991.

PILÃO, N. E. ; HUMMEL, P. R. V.. **Matemática Financeira e Engenharia Econômica –** Teoria e a prática da análise de projetos de investimentos. Editora: Cengage Learning, 2003

PRESSMAN, R. S.. **Engenharia de Software.** / Roger S. Pressman; - 6ª ed. - São Paulo Editora: McGraw.Hill. 2006.

ROBERT, V. B.. **Testing Object-Oriented Systems**. Addison Wesley Professional, 1999.

SBROCCO, J. H.T.C; MACEDO, P. C. **Metodologias** Ágeis **- Engenharia de Software sob medida**, Ed. Erica , 2012

SOMMERVILLE, I.. **Engenharia de Software. 7ª** Ed; Pearson Addison. Wesley. São. Paulo, 2004.

Recebido em 03-05-2012 • Aceito em 27-08-2012