

Nesta seção você encontra artigos sobre técnicas que poderão aumentar a qualidade do desenvolvimento de software

Testes unitários de software

Aprenda a efetuar testes unitários no Delphi 2007 com nUnit e csUnit



Marcelo Santos Daibert

(marcelo@daibert.net)

é professor do Curso de Bacharelado em Ciência da Computação da FAGOC - Faculdade Governador Ozanam Coelho na graduação e pós-graduação (especialização), Mestrando e Especialista em Ciência da Computação pela Universidade Federal de Viçosa e Bacharel em Sistemas de Informação pela Faculdade Metodista Granbery. Gerente técnico da Optical Soluções em Informática.



Marco Antônio Pereira Araújo

(maraujo@devmedia.com.br)

é professor dos Cursos de Bacharelado em Sistemas de Informação do Centro de Ensino Superior de Juiz de Fora e da Faculdade Metodista Granbery, Doutorando e Mestre em Engenharia de Sistemas e Computação pela COPPE/UFRJ, Especialista em Métodos Estatísticos Computacionais e Bacharel em Informática pela UFJF, Analista de Sistemas da Prefeitura de Juiz de Fora, Editor da Engenharia de Software Magazine.

Resumo DevMan

Entre testar manualmente um sistema e testar profissionalmente, certamente fico com a melhor delas: a segunda opção. A qualidade de software é hoje uma das maiores preocupações de qualquer empresa de tecnologia envolvida no desenvolvimento de sistemas. Por isso veremos nesse artigo como criar testes unitários capazes de identificar possíveis erros em rotinas e classes.

Utilizaremos os programas nUnit e csUnit para realizar tais tarefas e chegaremos a conclusão que não é tão difícil manter o sistema bem estruturado e funcionando perfeitamente.

Nesse artigo veremos

- Testes unitários caixa branca;
- Criação de rotinas para testes;

Qual a finalidade

- Preparar o sistema para efetuar testes em classes do sistema, evitando maiores problemas no futuro;

Quais situações utilizam esses recursos?

- Os recursos vistos aqui se enquadram bem em aplicações Orientadas a Objetos e mantém as rotinas do sistema concisas e coerentes. Em qualquer sistema podemos aplicar essas técnicas;

Principalmente nos últimos anos, foi possível observar um grande avanço em todo o processo de desenvolvimento de software, motivado principalmente pelas necessidades impostas por um mercado cada vez mais competitivo. Novas técnicas de desenvolvimento de software, novos métodos, formas de documentação, linguagens, entre outros, foram desenvolvidos. A necessidade de aspectos de usabilidade nos sistemas começaram a fazer diferença e

a palavra *qualidade* cada vez mais foi se destacando nas pautas dos desenvolvedores. A qualidade no software passou a ser uma busca cada vez mais freqüente pelos desenvolvedores e empresas de software.

Uma falha no software pode ocasionar perdas financeiras, perdas de informações, que muitas vezes é até mais catastrófica do que perdas financeiras, e, dependendo da criticidade da aplicação, até perda de vidas humanas.

Neste contexto, um dos temas importantes no processo de desenvolvimento de software é a fase de testes das aplicações, onde a mesma é submetida a uma carga de testes a fim de identificar falhas antes mesmo do software entrar no ambiente de produção.

Existem algumas estratégias de teste de software e a principal utilizada atualmente em muitas empresas de desenvolvimento, principalmente as de médio e pequeno porte, é a utilização dos testes manuais. Nesta abordagem é atribuída a função de testar as aplicações desenvolvidas a um ou mais membros da equipe de forma manual, onde o sistema é executado e as informações são inseridas manualmente, verificando se os resultados esperados foram alcançados. No entanto, esta abordagem é muito ineficaz e demorada. As melhores técnicas de teste são baseadas em algum processo automatizado, onde é possível executar uma maior quantidade de testes, buscando assim testar o máximo possível dos requisitos de um software. Nesta abordagem, existem algumas estratégias, entre elas: testes unitários (o que este artigo aborda), testes funcionais, teste de desempenho, testes em banco de dados, testes em *WebServices*, entre outros.

O teste unitário, um tipo de teste caixa branca, por ser baseado na estrutura lógica do código, é responsável por testar a unidade de codificação da aplicação. Em um sistema orientado a objetos esta unidade pode ser representada pela própria classe ou pelos métodos das classes. Dada uma entrada, o teste unitário deve aferir o resultado, levando em consideração todos os possíveis caminhos do algoritmo e o seu processamento. Para definir o número mínimo de casos de teste para cobrir as possibilidades de caminhos de processamento de um trecho de código, é apresentada a métrica de software chamada complexidade ciclomática, que define o número de caminhos independentes que um algoritmo deve percorrer para efetuar todos os processamentos.

O objetivo deste artigo é apresentar de forma prática a abordagem de desenvolvimento de software baseada em testes

unitários, conceituar a técnica de programação por intenção e exibir a utilização prática da estratégia de teste unitário usando as ferramentas *nUnit* e *csUnit* no ambiente de desenvolvimento *CodeGear RAD Studio 2007* para *.NET*. Tanto o *nUnit*, quanto o *csUnit*, são compatíveis com qualquer linguagem e ambiente de desenvolvimento *.NET*.

Desenvolvimento Baseado em Testes

O *Desenvolvimento Baseado em Testes* (TDD – *Test Driven Development*) é uma abordagem que faz uso das técnicas de teste de software para minimizar a quantidade de falhas na aplicação desenvolvida. Para isso, todo o processo de desenvolvimento de software é baseado em testes, a começar pela técnica de desenvolvimento de teste antes da codificação (TFD – *Test First Development*).

Nesta técnica, o desenvolvedor deve planejar e construir o teste para um trecho ou módulo do sistema que está sendo construído, antes mesmo de sua codificação. Com isso, a qualidade do caso de teste é aperfeiçoada, já que o mesmo é feito de forma incremental, juntamente com a codificação, como pode ser visualizado em um diagrama de atividades na **Figura 1**. Esta técnica faz uso de uma estratégia chamada programação por intenção.

Na **Figura 1**, é possível identificar claramente os passos do TFD. A primeira atividade observada é a *Adicionar Teste*. Nela, o teste é escrito e então executado no segundo passo. Certamente o teste irá falhar, já que não existe codificação da função, o que justifica a terceira atividade, chamada *Modificação / Refatoração*. Nesta atividade o desenvolvedor irá codificar a função a fim de fazer o teste ser executado com sucesso. Após, na última atividade, o teste é executado novamente. Se ele passar, o desenvolvimento continua ou finaliza e, caso o teste falhe, há a necessidade de uma nova iteração na atividade *Modificação / Refatoração* até que o teste seja executado com sucesso.

A programação por intenção é uma abordagem de programação que induz a codificação usando classes, métodos



Nota do DevMan

Caixa-Preta

Técnica de teste, também chamado de Teste Funcional, em que o componente de software a ser testado é abordado como se fosse uma caixa-preta, ou seja, não se considera o comportamento interno do mesmo. Dados de entrada são fornecidos, o teste é executado e o resultado obtido é comparado a um resultado esperado previamente conhecido.

O componente de software a ser testado pode ser um método, uma função interna, um programa, um componente, um conjunto de programas e/ou componentes ou mesmo uma funcionalidade. A técnica de teste de Caixa-Preta é aplicável a todas as fases de teste - fase de teste de unidade (ou teste unitário), fase de teste de integração, fase de teste de sistema e fase de teste de aceitação.

A aplicação de técnicas de teste leva o testador a produzir um conjunto de casos de teste (ou situações de teste). A aplicação combinada de outra técnica - Técnica de Particionamento de Equivalência (ou uso de Classes de Equivalência) permite avaliar se a quantidade de casos de teste produzida é coerente. A partir das classes de equivalência identificadas, o testador irá construir casos de teste que atuem nos limites superiores e inferiores destas classes, de forma que um número mínimo de casos de teste permita a maior cobertura de teste possível

ou módulos do sistema que ainda não existem e serão criados futuramente para atender às necessidades de compilação da aplicação. No contexto de desenvolvimento baseado em testes, a programação por intenção auxilia a criação dos casos de testes, quando utilizada a técnica de desenvolvimento de testes antes da codificação, já que o teste produzido afere um método ainda inexistente. E este método, por sua vez, pode fazer uso de recursos da aplicação também inexistentes.

A idéia central desta técnica é a comunicação com as intenções do desenvolvedor ao codificar a aplicação. Mesmo gerando um código fonte não compilável, é possível definir e limitar o escopo de ação de determinado método, além de traçar os passos de desenvolvimento para atender às necessidades de compilação, inclusive utilizando técnicas de refatoração de

código-fonte. Para isso, muitas vezes é necessário utilizar os chamados objetos *Mock*, com o objetivo de simplificar a utilização da programação por intenção e a criação dos casos de teste. Objetos *Mock* são objetos falsos, ou de fachada, com o objetivo de substituir recursos não disponíveis ou inadequados, possibilitando a criação dos casos de teste e execução de testes unitários no sistema.

Independente da utilização desta abordagem de desenvolvimento baseado em testes, a utilização das técnicas de teste

se configuram como uma importante ferramenta para buscar a qualidade das aplicações e a minimização de defeitos no *software*.

Configuração do ambiente no Delphi 2007

O *nUnit* e o *csUnit* são ferramentas independentes do ambiente de desenvolvimento. Basicamente, ambas carregam uma biblioteca compilada no ambiente *.NET* e executam os testes configurados.

Neste contexto, para que seja possível a execução dos testes de forma automatizada pelas ferramentas de teste unitário, é necessário configurar um projeto no Delphi 2007 de forma a gerar uma biblioteca (neste caso, um arquivo *DLL*). Para isso, acesse o menu do Delphi e crie um projeto de testes unitário: *File>New>Other>Unit Test>Test Project*, como visualizado na **Figura 2**. Esta opção irá configurar o ambiente do Delphi para a geração de uma biblioteca *DLL* com os testes que serão configurados a seguir. No entanto, esta não é a única alternativa. É possível também selecionar a opção *Delphi for .NET Projects>Library*, como apresentado na **Figura 3**. Nesta opção será configurado um ambiente para a geração de uma *DLL*, o que cumpre com as necessidades para a geração dos testes e posterior execução automatizada nas ferramentas.

Com estas ações, sempre que o projeto for compilado será gerado um arquivo *.dll* com o mesmo nome do projeto (neste caso, foi dado o nome *libClasses* – o que gera uma *DLL* chamada *libClasses.dll*). É este arquivo que deve ser carregado tanto pelo *nUnit* quanto pelo *csUnit* para a execução dos testes.

Com o projeto criado, é necessário adicionar uma referência a uma biblioteca do *nUnit* para que seja possível utilizar métodos e funcionalidades disponibilizadas pela ferramenta, tornando possível assim a criação dos casos de teste. Na aba *Project Manager*, clique com o botão

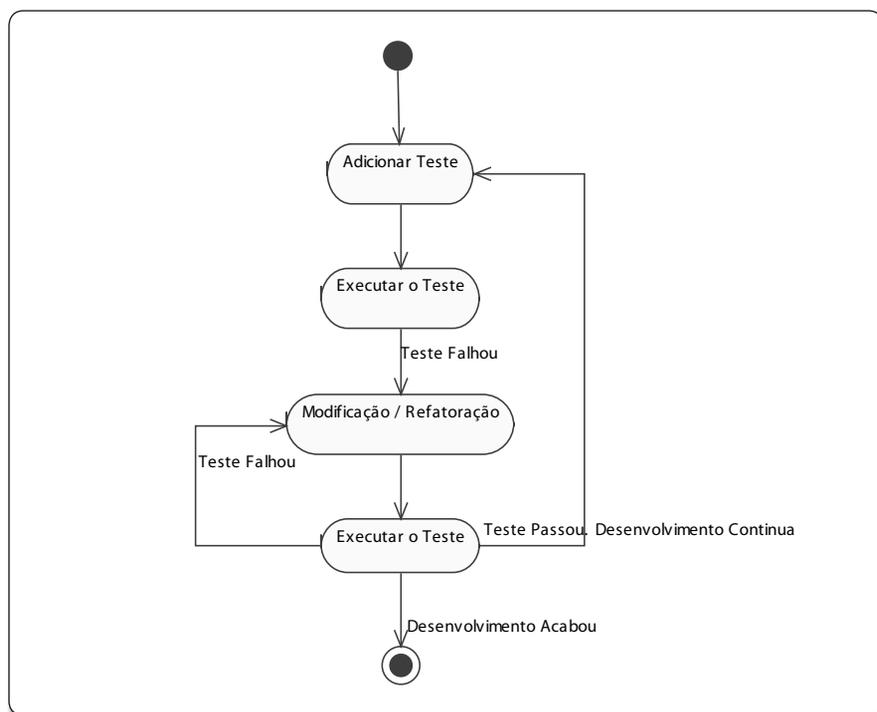


Figura 1. Passos do TFD – Test First Development

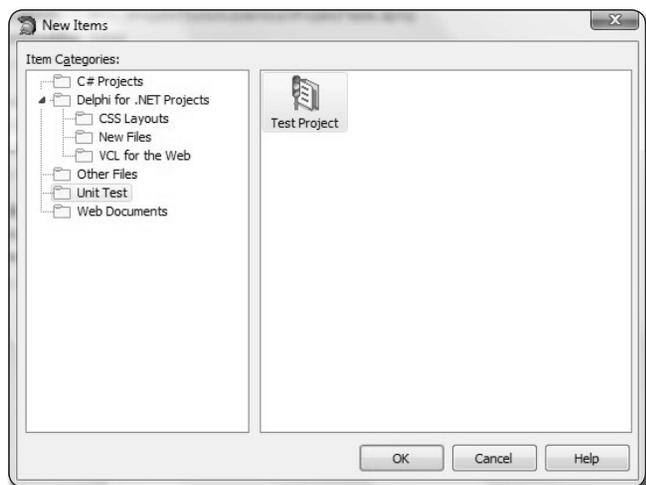


Figura 2. Projeto de Teste no Delphi 2007

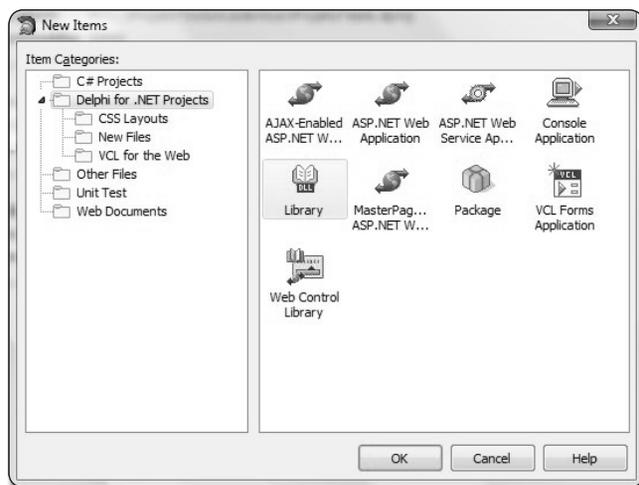


Figura 3. Biblioteca – Library no Delphi 2007

direito no item *References* e escolha a opção *Add Reference*. Ao invocar esta opção, é apresentada uma janela para escolha de referências a serem adicionadas ao projeto. A referência do *nUnit* deve ser encontrada na aba *.NET Assemblies* com o nome *nunit.framework*, como exemplificado na **Figura 4**. Caso não seja possível encontrá-la, é necessário verificar se o *nUnit* está realmente instalado (verifique o local para download da ferramenta na sessão *links* deste artigo) ou adicionar a referência de forma manual.

Para adicionar a referência manual, clique em *Browse* e localize o arquivo da biblioteca no disco. O nome do arquivo é *nunit.framework.dll* e fica armazenado dentro do diretório *bin* do diretório onde o *nUnit* foi instalado. Para finalizar, clique em *Add Reference* e *OK*. Depois de registrada no projeto, a referência da biblioteca do *nUnit* é exibida na seção *References* no *Project Manager*, como apresentado pela **Figura 5**.

É importante destacar que foi adicionado ao projeto somente uma referência à biblioteca do *nUnit*. Os casos de teste construídos irão também rodar no *csUnit*. Isso é possível, pois o *csUnit* é compatível com as diretivas de compilação e atributos do *nUnit*.

Estudo de caso e criação dos testes

Buscando exemplificar o uso das ferramentas de teste unitário, é proposto um estudo de caso criado utilizando o ambiente de desenvolvimento do *CodeGear RAD Studio 2007 for .NET*. Nele é apresentada uma classe chamada *TAluno*, com os atributos *frequência*, *nota1*, *nota2* e *provaFinal*.

Esta classe possui dois métodos: uma *procedure* chamada *setDados*, responsável por atribuir os valores aos atributos da classe, respeitando assim o princípio do *encapsulamento*, e a *function* *calcularAprovacao*, responsável por verificar se o aluno foi aprovado ou não, de acordo com sua frequência e notas. Esta classe foi codificada em uma *unit* chamada *Aluno.pas* e está sendo apresentada na **Listagem 1**.

O método *calcularAprovacao* verifica inicialmente se a frequência do aluno é menor que 75%. Caso seja, o aluno é reprovado de imediato, fazendo com que a função retorne falso. Caso contrário, é verificada a média da nota1 com a nota2. Caso essa média seja menor que 3, o aluno também é reprovado.

Caso contrário, se a média for maior ou igual a 7, o aluno é aprovado. Se a média for maior ou igual a 3 e menor que 7, o

aluno estará em prova final e será aprovado caso o valor da média de sua nota de prova final com a média anterior for superior ou igual a 5. Caso contrário, ele será reprovado.

Com a classe *Aluno* definida, o objetivo agora é criar casos de teste para o método *calcularAprovacao*, buscando testar as regras descritas pelo método. Para isso, neste estudo de caso, foi criada uma nova *unit* com o nome *AlunoTeste* e esta é apresentada na **Listagem 2**. É importante observar que esta *unit* faz referência direta à classe *TAluno* (*uses TAluno*) e à classe *nUnit.Framework* (*uses nUnit.Framework*), que contém os métodos e atributos disponibilizados pelo *nUnit* para a criação dos casos de teste.

A **Listagem 2** apresenta a codificação da *unit* *TesteAluno*, contendo uma classe chamada *TAlunoTeste*, sua definição e implementação dos métodos *testeAprovacao1* até *testeAprovacao5*, que são os casos de teste, e que representam o número de caminhos do algoritmo apresentado. Deve-se observar o uso do atributo *[TestFixture]*, antes da definição da classe *TAlunoTeste*, na linha 9. Esse atributo deve ser utilizado antes das classes definidas como classes de teste, para que o *nUnit* possa reconhecê-las dessa forma.

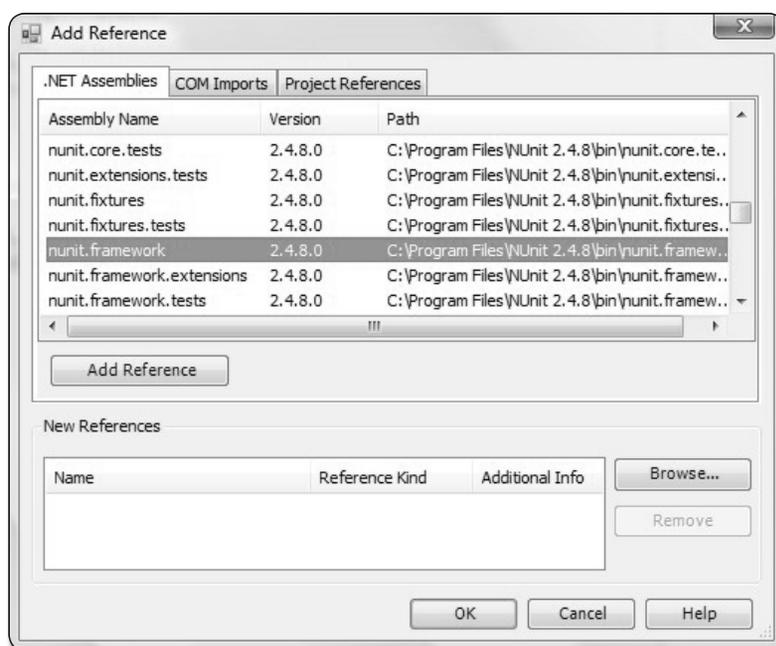


Figura 4. Adicionando a referência ao nUnit



Figura 5. Referência ao nUnit adicionada

O *setUp* é responsável pela inicialização do teste, como a instanciação do objeto, por exemplo. É importante destacar o uso do atributo *[SetUp]* antes da sua definição, como pode ser visto na linha 31. Esse atributo indica ao *nUnit* que o método definido a seguir deve ser executado antes de qualquer caso de teste. Já o *TearDown* é executado ao finalizar a execução de cada caso de teste. Ele pode ser utilizado, por exemplo, para liberar a memória ocupada pelo objeto instanciado. Nele é utilizado o atributo

[TearDown] como exemplificado na linha 37, indicando ao *nUnit* que o método definido a seguir deve ser executado ao final de cada caso de teste.

Os demais métodos listados são os casos de teste. Para esses, é usado o atributo *[Test]* antes de sua definição, indicando ao *nUnit* que o método a seguir é um caso de teste.

No primeiro caso de teste, é definido um aluno que foi aprovado na prova final, tendo *nota1* igual a 3, *nota2* igual a 3, *notafinal* igual a 7 e *frequência* em 75%,

como pode ser visualizado a partir da linha 43. Já, na linha 47, é invocado o *IsTrue* da classe *Assert*. Essa classe é responsável por definir o valor esperado do teste para comparação com o valor obtido na sua execução. Neste caso, o valor esperado para retorno do método *calcularAprovacao* é verdadeiro (*IsTrue*).

O segundo caso de teste cobrirá a opção do aluno ser reprovado por frequência. Para isso, é definida a variável *frequência* para 74%. Os demais valores são definidos com 0, já que o aluno é reprovado de imediato. Para esse caso de teste, o valor esperado é falso, ou seja, *IsFalse*.

O terceiro caso de teste apresenta um aluno com 75% de frequência, primeira nota com valor 3, segunda nota com 2.9 e prova final com 0, pois não importa o seu valor nessa situação. Para esse caso é esperado o valor falso (*IsFalse*).

Para o quarto caso de teste é configurado o aluno com 75% de frequência, primeira e segunda notas com valor 3 e prova final com valor 6.9. Espera-se que o aluno seja reprovado, logo a classe *Assert* é configurada para invocar o *IsFalse*.

Por fim, o quinto e último caso de teste, o aluno é definido com 75% de frequência e notas com valores iguais a 7 para a primeira e segunda nota. A nota final foi definida com 0. O valor esperado de retorno para o caso de teste é verdadeiro. Observe que a média é igual a 7.

A classe *Assert* apresenta outros métodos para definição de valores esperados além dos apresentados. Dentro os principais, são citados:

- *IsFalse*: se o parâmetro definido é falso;
- *IsTrue*: se o parâmetro definido é verdadeiro;
- *AreEqual*: se os parâmetros são iguais;
- *IsNotNull*: se o parâmetro não é nulo;
- *IsNull*: se o parâmetro é nulo.

Execução dos testes

Ao finalizar a programação dos casos de teste, deve-se complicar o projeto para que a DLL seja gerada. Após isso, a mesma deve ser carregada na interface do *nUnit* e *csUnit* para que ocorra a execução dos testes definidos. No *nUnit* o carregamento da DLL deve ser feito

Listagem 1. Unit Aluno

```
unit Aluno;

interface

type
  TAluno = class
  private
    frequencia, nota1, nota2, provafinal: Real;
  public
    procedure setDados (pFrequencia, pNota1, pNota2, pProvaFinal: real);
    function CalcularAprovacao: boolean;
    constructor Create;
  end;

implementation

function TAluno.CalcularAprovacao: boolean;
var
  vMedia: real;
begin
  if self.frequencia < 75 then
  begin
    Result := False;
  end
  else
  begin
    vMedia := (self.nota1 + self.nota2) / 2;
    if vMedia < 3 then
    begin
      Result := False;
    end
    else
    begin
      if vMedia > 7 then
      begin
        Result := True;
      end
      else
      begin
        if (vMedia + self.provafinal) / 2 < 5 then
        begin
          Result := False;
        end
        else
        begin
          Result := True;
        end;
      end;
    end;
  end;
end;

constructor TAluno.Create;
begin
  inherited Create;
end;

procedure TAluno.setDados(pFrequencia, pNota1, pNota2, pProvaFinal: real);
begin
  self.frequencia := pFrequencia;
  self.nota1 := pNota1;
  self.nota2 := pNota2;
  self.provafinal := pProvaFinal;
end;

end.
```

acessando o menu *File>Open Project*. No *csUnit* o processo é semelhante: menu *Assembly>Add*. Após, solicite à ferramenta para que execute os testes.

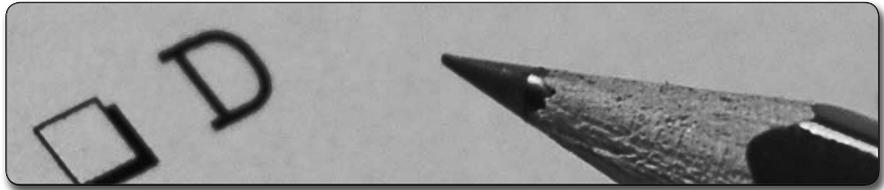
As **Figuras 6 e 7** apresentam, respectivamente, o resultado da execução dos testes nas ferramentas *nUnit* e *csUnit*. Observe que o último caso de teste falhou. Isso se deve ao fato que se esperaria que alunos com média 7 fossem aprovados e, na realidade, estão sendo reprovados. De acordo com o método *calcularAprovacao*, quem possuir média entre 3 e 7 (inclusive) deve fazer prova final. Como este aluno possui valor 0 na sua prova final, o mesmo foi reprovado, sendo retornado o valor falso pelo método, onde o valor esperado deveria ser verdadeiro, falhando assim o caso de teste.

Com isso foi possível identificar uma falha no código-fonte, uma vez que o aluno com média de valor 7 deveria ser aprovado. Deve-se então fazer uma alteração no código-fonte do método *calcularAprovacao* para que sejam aprovados os alunos com média superior ou IGUAL a sete. A alteração deve ser realizada na linha 34 da **Listagem 1**, modificando a condição para *vMedia >= 7*. A atribuição dos valores para os testes deve ser feita com muito cuidado. Por exemplo, se o caso de teste *TesteAprovacao5* utilizasse o valor 8 no lugar de 7 (linha 74 da **Listagem 2**), não seria possível encontrar o defeito no código-fonte da aplicação. Assim, sugere-se que sempre sejam utilizados os valores mais próximos das condições a serem testadas.

A **Figura 8** exibe a interface do *nUnit* com os resultados sem falhas após a alteração proposta. E a **Figura 9**, a execução dos testes no *csUnit*. Como foi observado, ambas as ferramentas têm a mesma funcionalidade e a sugestão é testar qual delas se adapta melhor para o projeto e ao profissional que a está utilizando.

Conclusão

Uma questão importante numa abordagem como essa é a definição do número de casos de teste necessários para avaliar os caminhos possíveis de um algoritmo. Como é impossível testar todas as situa-



Listagem 2. Unit AlunoTeste

```

01 unit AlunoTeste;
02
03 interface
04
05 uses
06   Aluno, NUnit.Framework;
07
08 type
09   [TestFixture]
10   TAlunoTeste = class
11   private
12     objAluno: TAluno;
13   public
14     constructor Create;
15     procedure Setup;
16     procedure TearDown;
17     procedure TesteAprovacao1;
18     procedure TesteAprovacao2;
19     procedure TesteAprovacao3;
20     procedure TesteAprovacao4;
21     procedure TesteAprovacao5;
22   end;
23
24 implementation
25
26 constructor TAlunoTeste.Create;
27 begin
28   inherited Create;
29 end;
30
31 [SETUP]
32 procedure TAlunoTeste.Setup;
33 begin
34   objAluno := TAluno.Create;
35 end;
36
37 [TEARDOWN]
38 procedure TAlunoTeste.TearDown;
39 begin
40   objAluno.Free;
41 end;
42
43 [TEST]
44 procedure TAlunoTeste.TesteAprovacao1;
45 begin
46   objAluno.setDados(75, 3, 3, 7);
47   Assert.IsTrue(objAluno.CalcularAprovacao, 'Revise o Método de Aprovação');
48 end;
49
50 [TEST]
51 procedure TAlunoTeste.TesteAprovacao2;
52 begin
53   objAluno.setDados(74, 0, 0, 0);
54   Assert.IsFalse(objAluno.CalcularAprovacao, 'Revise o Método de Aprovação');
55 end;
56
57 [TEST]
58 procedure TAlunoTeste.TesteAprovacao3;
59 begin
60   objAluno.setDados(75, 3, 2.9, 0);
61   Assert.IsFalse(objAluno.CalcularAprovacao, 'Revise o Método de Aprovação');
62 end;
63
64 [TEST]
65 procedure TAlunoTeste.TesteAprovacao4;
66 begin
67   objAluno.setDados(75, 3, 3, 6.9);
68   Assert.IsFalse(objAluno.CalcularAprovacao, 'Revise o Método de Aprovação');
69 end;
70
71 [TEST]
72 procedure TAlunoTeste.TesteAprovacao5;
73 begin
74   objAluno.setDados(75, 7, 7, 0);
75   Assert.IsTrue(objAluno.CalcularAprovacao, 'Revise o Método de Aprovação');
76 end;
77
78
79 end.

```

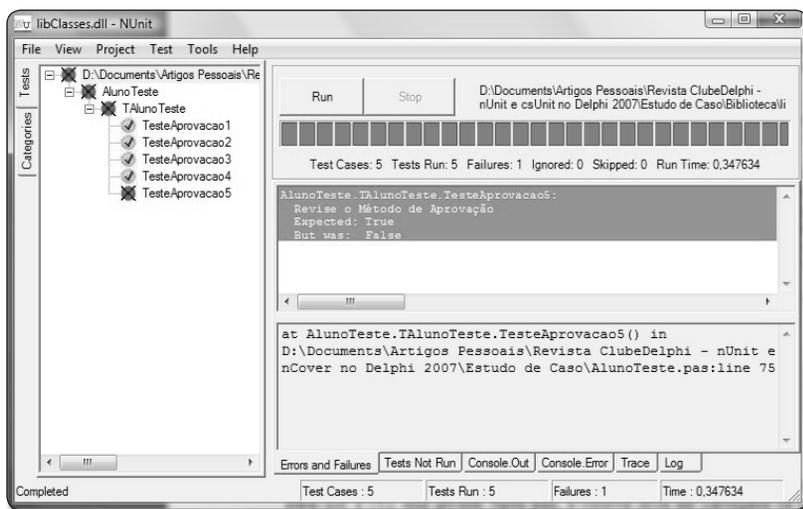


Figura 6. Resultados dos Testes no nUnit

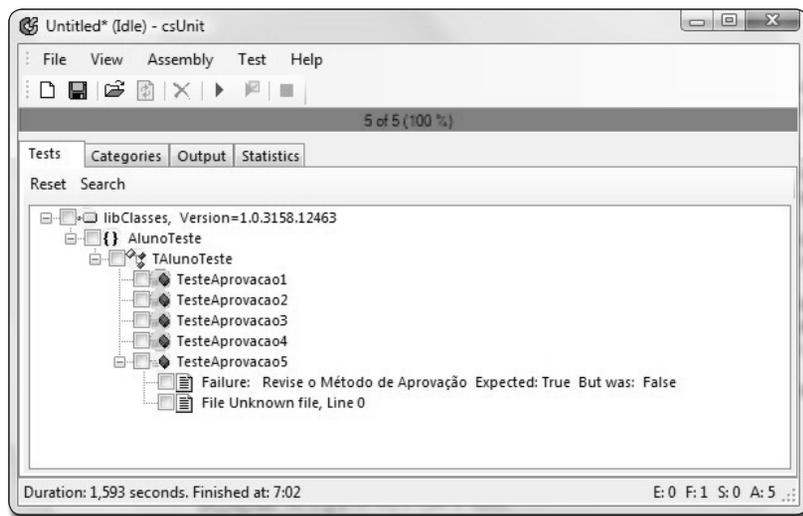


Figura 7. Resultados dos Testes no csUnit

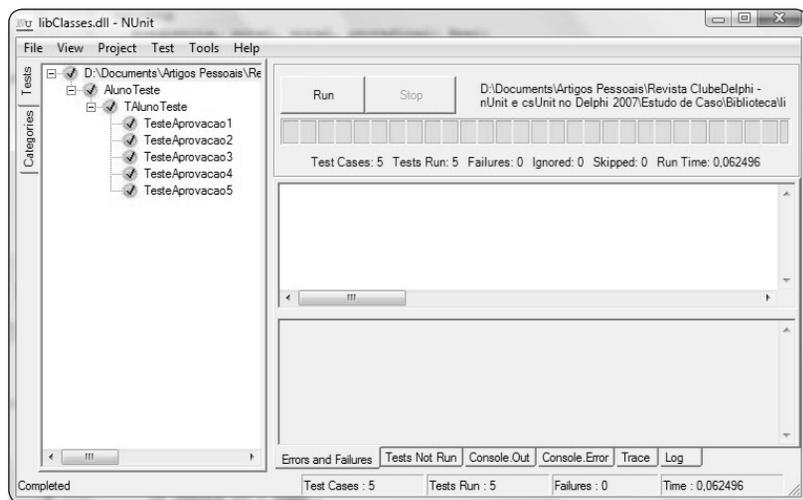


Figura 8. Resultados dos Testes no nUnit após correção do código

ções, torna-se primordial não construir casos de teste desnecessários, testando as mesmas situações, e nem deixar de construir um caso de teste para uma situação que necessite ser testada.

Uma importante técnica neste sentido é conhecida como Complexidade Ciclométrica de McCabe, que avalia o número de caminhos possíveis em um algoritmo, indicando o número de casos de teste necessários para avaliar as suas possibilidades.

Entretanto, essa técnica apenas determina o número de casos de teste, não os valores que devem ser utilizados no mesmo. Para isso, uma outra técnica conhecida como Particionamento por Classes de Equivalência, auxiliada por outra técnica chamada Análise do Valor Limite, são úteis, mas fogem ao escopo deste artigo.

Entretanto, o sucesso de uma abordagem de desenvolvimento apoiada por testes dá-se em função da qualidade dos casos de teste. Se um sistema não falha nos testes planejados, fica a dúvida se o sistema é de boa qualidade ou se os casos de teste é que são de baixa qualidade. Portanto, investir num bom planejamento de testes é fundamental para esta estratégia.

O ambiente de execução de testes deve ser cuidadosamente planejado. A execução de testes em dias diferentes, por exemplo, pela simples mudança da data do sistema operacional, pode não representar a mesma situação a ser testada. Assim, é importante garantir que o ambiente seja o mesmo a cada execução do teste, preocupando-se, por exemplo, com o estado das informações no banco de dados.

Assim, a utilização de técnicas de teste proporciona aos desenvolvedores um menor índice de defeitos no código-fonte e, conseqüentemente, uma maior qualidade e confiabilidade do sistema no ambiente de produção.

Neste sentido, tanto o *nUnit*, quanto o *csUnit*, são *frameworks* para execução de testes unitários para o ambiente de desenvolvimento .NET, que buscam automatizar o processo de testes, tornando possível a implantação de uma política de testes na prática e sem agregar custos com ferramentas ao projeto, já que ambas são gratuitas. ●

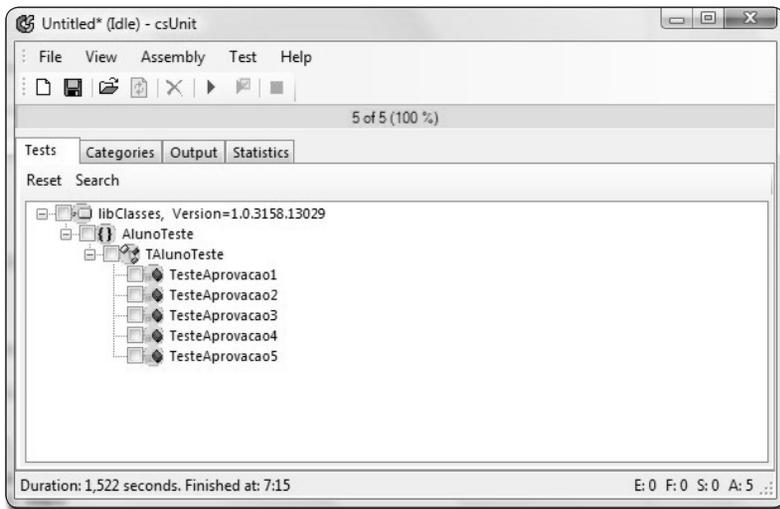


Figura 9. Resultados dos Testes no csUnit após correção do código

Links

Site Oficial do nUnit
www.nunit.org

Site Oficial do csUnit
www.csunit.org

Dê seu feedback sobre esta edição!

A Clubedelphi tem que ser feita ao seu gosto. Para isso, precisamos saber o que você, leitor, acha da revista!

Dê seu voto sobre este artigo, através do link:

www.devmedia.com.br/clubedelphi/feedback



A EDIÇÃO QUE VOCÊ PRECISA
 ESTÁ ESGOTADA?

SEUS PROBLEMAS
 ACABARAM!!

Seja um assinante Gold!

Com a assinatura Gold você já pode consultar online todos os artigos publicados na sua revista desde a edição nº 1.



Saiba Mais! Acesse:

www.devmedia.com.br/assgold

Para mais informações:

www.devmedia.com.br/central

Assinatura

Gold

Atenção: Já encontram-se disponíveis as assinaturas GOLD das revistas WebMobile e .net Magazine.