Planejando e Automatizando a Geração de Casos de Teste de Unidade Utilizando Diagrama de Seqüência

Antonio Carlos Silva[1], Fabiane Barreto Vavassori Benitti[1,2]

[1] Centro de Educação de Ciências Tecnológicas da Terra e do Mar
Universidade do Vale do Itajaí (UNIVALI) – Itajaí, SC – Brasil

[2] Departamento de Sistemas e Computação
Universidade Regional de Blumenau (FURB) – Blumenau, SC – Brasil
{antonio.carlos,fabiane.benitti}@univali.br

Abstract. *One of the initial tests applied to a software is the unit test that happens right before the total codification of the system. This paper proposes the use of sequence diagram as a visual language to define the test cases, allowing to documenting them regardless of the language of implementation. This paper also proposes a tool which supports this definition along with a case study to demonstrate the proposed artifact viability.*

Resumo. *Um dos testes iniciais aplicados a um software é o teste de unidade, que acontece antes mesmo da total codificação do sistema. Este artigo propõe o uso do diagrama de seqüência como linguagem visual para definição dos casos de teste, de forma a permitir a documentação dos mesmos independentemente da linguagem de implementação. Neste artigo é apresentada ainda, uma ferramenta que suporta esta definição juntamente com um estudo de caso, a fim de demonstrar a viabilidade do artefato proposto.*

1. Introdução

O teste de software é uma das etapas no ciclo de desenvolvimento de software que tem como principal objetivo encontrar os erros presentes em um software. Esta etapa é compreendida por sistemáticas aplicações de testes ao longo de todo o processo de desenvolvimento [Macgregor e Sykes 2001]. A etapa de testes é definida por Hetzel (1987) como: "o processo de executar um programa ou sistema com a finalidade de encontrar erros. Teste é a medida de qualidade do software".

Um dos testes iniciais aplicados a um software é o teste de unidade, que acontece antes mesmo da total codificação do sistema. O teste de unidade, conhecido também por teste de componente ou teste de módulo, é definido por Thomas *et al* (2004), como um teste projetado para verificar uma única unidade de software de forma isolada das demais unidades. Em projetos orientados a objetos, uma unidade de implementação é uma classe ou um método de uma classe.

A técnica de desenvolvimento TDD (*Test Driven Development*), proposta por Beck (2002), apóia-se na aplicação de testes de unidade antes da codificação das unidades propriamente ditas. Alguns dos benefícios atribuídos ao uso da técnica

referem-se ao fato de que toda unidade de software possuirá um conjunto de testes que verifique seu funcionamento, baseado nos requisitos estabelecidos para esta unidade, assim antecipando a detecção de erros presentes na codificação. A etapa inicial envolvida na aplicação da técnica é o planejamento dos casos de teste, momento em que são definidos os detalhes envolvidos na execução dos testes, como os dados de entrada e os resultados esperados.

Para o teste de unidade a etapa de planejamento é comumente confundida com a codificação dos *drivers* e *stubs*, resultando em pouca ou nenhuma documentação para os casos de teste, restando apenas o próprio código fonte dos testes como especificação [Beck 2002]. Esta deficiência na documentação dos casos de teste pode ser explicada pela falta de uma linguagem padrão para definição dos casos de teste e uma ferramenta que apóie este processo [Biasi e Becker 2006]. Desta forma, este artigo propõe a utilização do diagrama de seqüência, definido na UML (*Unified Modeling Language*), como artefato para definição de casos de teste (detalhado na seção 3). Também apresenta, na seção 4, uma ferramenta que visa auxiliar na definição dos casos de teste, bem como automatizar a geração do código fonte de execução dos testes utilizando o *framework* JUnit. Na seção seguinte são apresentados alguns dos pontos fundamentais da técnica TDD no que se refere a testes de unidade. Por fim são apresentadas as considerações finais e direcionamentos para trabalhos futuros.

2. TDD

O TDD (*Test Driven Development*) ou *Test First* tem como base a definição e aplicação de testes de unidade antes mesmo da codificação das unidades envolvidas, fortalecendo este princípio, destaca-se que um código para uma unidade de software só pode ser desenvolvido se houver um conjunto de casos de teste para verificá-la [Astels 2003].

A técnica de desenvolvimento TDD consiste em três etapas: (i) definição dos casos de teste; (ii) codificação das classes com o objetivo de atender aos casos de teste; e (iii) refatoração. Cada uma destas etapas é aplicada a cada unidade de software [Beck 2004].

A definição dos casos de teste ganha destaque se comparado a abordagem tradicional de teste de software, uma vez que este guiará a codificação do sistema. Nesta etapa são identificados os *drivers* e *stubs* para posterior codificação [Astels 2003].

São os *drivers* que determinam a execução dos casos de teste definindo as mensagens entre os objetos envolvidos e as verificações estabelecidas a partir dos resultados esperados [Maldonado e Fabbri 2001]. Comumente este código é desenvolvido para atender a uma ferramenta que automatize a execução dos testes[1], como por exemplo, o JUnit da família xUnit, responsável pela automatização de testes desenvolvidos em linguagem Java.

Os *stubs* de testes têm por objetivo substituir outras unidades de software envolvidas em um caso de teste, eliminando a dependência entre as unidades [Maldonado e Fabbri 2001]. O uso de *stubs* permite atender ao princípio de isolamento

[1] A automatização dos testes é um dos requisitos no uso da técnica TDD, sem este a técnica poderia tornar-se inviável devido às sucessivas aplicações de testes [Beck 2004].

das unidades, em que cada unidade deve ser testada de forma isolada das demais [Thomas *et al* 2004] e [IEEE 1986].

A codificação das classes é efetuada com base nos casos de teste estabelecidos, tendo como objetivo o êxito na execução dos testes. Na etapa de refatoração o código desenvolvido para atender aos casos de testes é avaliado de forma a verificar questões como flexibilidade, reaproveitamento de código, desempenho, clareza na solução e duplicidade no código [Astels 2003].

3. Definindo Casos de Teste

Na etapa de planejamento dos casos de teste para o teste unitário são especificados os dados de entrada e os resultados esperados, bem como todo o fluxo de execução do caso de teste como, por exemplo, as mensagens trocadas entre os objetos.

A pouca padronização na definição de casos de teste é destacada por Biasi e Becker (2006), Badri, Badri e Bourque-Fortin (2005) e Linzhang et al. (2004). Observa-se, ainda, que no padrão proposto pela IEEE (1986) para testes de unidade também não é abordada uma linguagem para definição de casos de testes. Dentre as soluções já propostas, destaca-se a de Badri, Badri e Bourque-Fortin (2005), que propõem a definição dos casos de teste tendo como base o diagrama de estados. No entanto, uma das limitações desta proposta refere-se ao fato de que os elementos deste diagrama não estão diretamente relacionados às unidades de software. A fim de contornar este problema Badri, Badri e Bourque-Fortin (2005) apresentam novos conceitos e definições ao diagrama de estados. Assim, a proposta de um modelo visual, baseado na UML e utilizando tão somente os recursos já especificados na linguagem, favorece a aplicabilidade por parte dos analistas e o suporte por ferramentas já existentes.

O diagrama de seqüência é um dos diagramas disponibilizados pela UML para modelagem das interações na etapa de projeto, no entanto, propõem-se a sua utilização para etapa de teste. A figura a seguir resume a proposta de utilização do diagrama de seqüência como linguagem visual para definição de casos de teste, destacando os principais elementos envolvidos.

Durante a elaboração do caso de teste devem ser definidas mensagens entre o *driver* (gerenciador dos casos de teste) e os objetos envolvidos neste. Para representação do *driver* utilizou-se a figura do ator, elemento já existente no diagrama de seqüência, denotando o responsável por invocar os métodos dos objetos envolvidos nos casos de teste, sendo apresentado à esquerda do diagrama, como demonstrado na Figura 1 (A).

Conforme abordado na seção 2, a utilização de *stubs* em casos de teste é fundamental para garantir o isolamento da unidade testada, desta forma, um *stub* no diagrama proposto é representado como um objeto acrescido do estereótipo <<stub>>, conforme a Figura 1 (B).

Outro recurso pertinente quando da criação de casos de teste refere-se ao *SetUp*. Este recurso permite definir objetos compartilhados por todos os casos de teste de uma classe, ou seja, considerados como de uso global. Assim, quando utilizado o *SetUp* de uma classe no diagrama de seqüência é exibida ao fim da linha de vida de cada objeto a

mensagem *from SetUp*, conforme a Figura 1 (C), evidenciando o uso de objetos compartilhados.

Para todo caso de teste definido deve ser incluído um objeto *Assert*, como demonstrado na Figura 1 (D). O elemento *Assert* representa a classe de mesmo nome do *framework* xUnit sendo responsável pela verificação do caso de teste. Ressalta-se que esta classe é padronizada nos *frameworks* da família xUnit, ou seja, a sua utilização independe da linguagem de implementação.

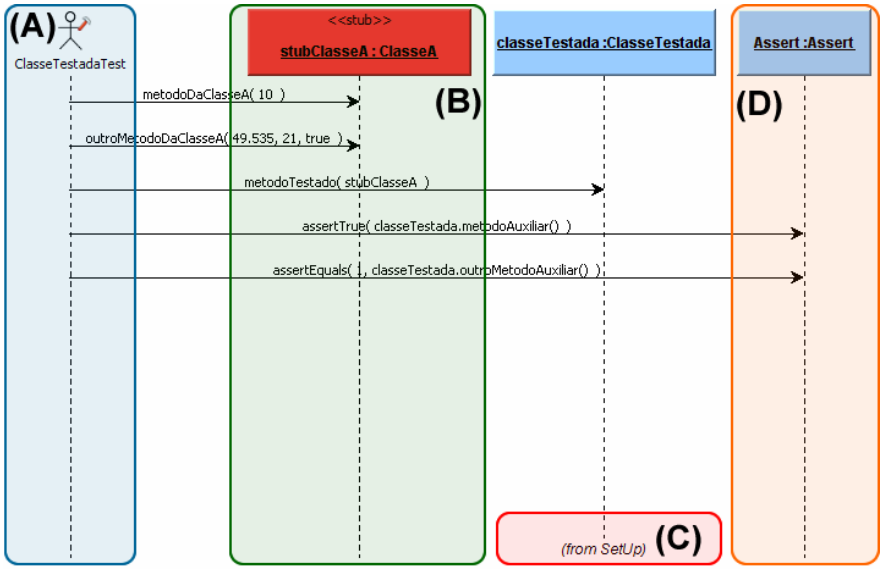

Figura 1. Diagrama de seqüência utilizado para definição dos casos de teste

Destaca-se que para definição de um caso de teste utilizando o diagrama de seqüência basta que sejam definidos os objetos envolvidos no teste e as mensagens trocadas entre estes, com isso obtêm-se um padrão baseado em uma linguagem já estabelecida e de conhecimento de muitos dos analistas e testadores envolvidos no processo de desenvolvimento de software.

Visando demonstrar a viabilidade de utilização deste diagrama como forma de especificar casos de testes, a seção seguinte apresenta a proposta de uma ferramenta que permite a geração automática do código de teste de unidade a partir de um caso de teste, definido visualmente através do diagrama de seqüencia, elaborado conforme descrito nesta seção.

4. Ferramenta para Apoio no Planejamento dos Casos de Teste

A ferramenta tem como objetivo permitir a criação de um projeto de casos de teste, focando testes de unidade considerando o contexto de um método (unidade). Para tanto, faz-se necessário que o modelo de classes do projeto testado esteja definido, pois este compreende a base dos testes. Sendo assim, a ferramenta apóia-se nas etapas definidas no diagrama de atividades ilustrado na Figura 2.

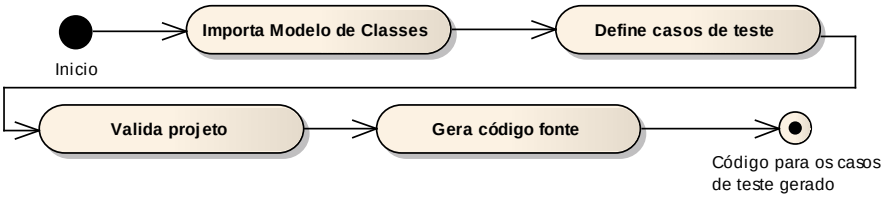

Figura 2. Diagrama de atividades para a ferramenta proposta

1. Importa modelo de classes: permite conhecer o modelo de classes a ser testado, devendo ser compatível com o padrão XMI 2.1;

2. Define casos de teste: nesta etapa, utilizando o formato do diagrama de seqüência proposto, o usuário define os casos de teste para os métodos do modelo importado;

3. Valida projeto: após a definição dos casos de teste a ferramenta permite a validação dos casos de teste, fazendo críticas a possíveis problemas encontrados nos casos de teste; e

4. Gera código fonte: por fim o usuário tem a opção de gerar o código fonte para os casos de teste do modelo, possibilitando sua execução pelo *framework* JUnit (inicialmente, para demonstrar a viabilidade do diagrama proposto, optou-se por testar sistemas desenvolvidos na linguagem Java).

Tendo em vista as etapas definidas, a Figura 3 apresenta o diagrama de casos de uso, representando as funcionalidades da ferramenta proposta.

Neste modelo de casos de uso percebe-se comportamentos básicos para operacionalizar a ferramenta, como abrir, salvar e fechar projeto (UC02, UC03 e UC04). Os demais casos de uso são o foco da ferramenta e, portanto, são detalhados na seqüência abordando sua funcionalidade juntamente com um estudo de caso.

A elaboração do caso de teste (UC05), através do diagrama de seqüência, parte da existência de um modelo de classes com seus respectivos métodos definidos. Sendo assim, considerando o modelo de classes ilustrado na Figura 4, é possível importá-lo na ferramenta (UC01) e iniciar a definição do caso de teste, através da interface ilustrada na Figura 5.

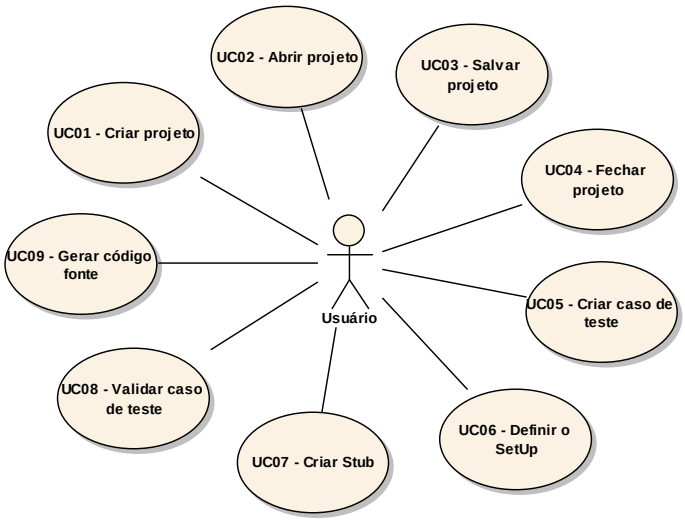

Figura 3. Casos de uso para a ferramenta proposta

O estudo de caso o qual é apresentado ao longo desta seção, a fim de demonstrar o funcionamento da ferramenta, possui como base o modelo de classes ilustrado pela Figura 4, onde o método adicionaProduto da classe CarrinhoDeCompras terá um caso de teste definido. Este caso de teste visa verificar se ao adicionar um produto ao CarrinhoDeCompras este será devidamente tratado.

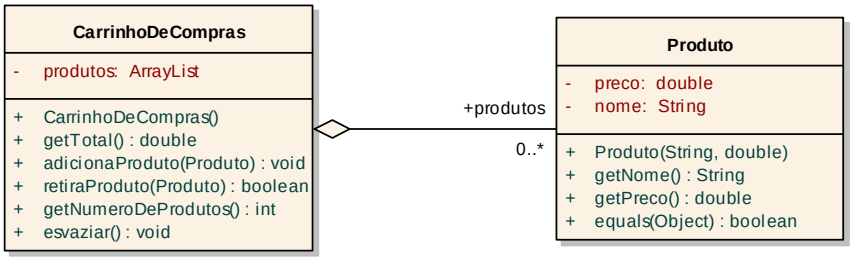

Figura 4. Diagrama de classes do estudo de caso

Alguns aspectos são importantes de serem destacados. Ao solicitar a criação de um novo caso de teste, a partir de um método presente no modelo e importado pela ferramenta (Figura 5 A), é apresentado ao usuário o diagrama de seqüência com alguns elementos pré-definidos, conforme observado em (Figura 5 B):

- ✓ Gerenciador (CarrinhoDeComprasTest): responsável por gerenciar o caso de teste e do qual partirão todas as mensagens;
- ✓ Objeto Assert, que compõe o *framework* JUnit e é responsável pela verificação do casos de teste; e
- ✓ *Stubs*: a ferramenta identifica com base na assinatura do método à ser testado, quais *Stubs* são necessários para sua chamada. Caso não exista nenhum *Stub* já criado do tipo necessário, um novo *Stub* é adicionado, devendo o usuário definir seu comportamento.

No caso da figura 5, tem-se os elementos propostos para o caso de teste do método AdicionaProduto da classe CarrinhoDeCompras, apresentado na Figura 4. O *Stub* de nome stubProduto, presente no caso de teste, tem por objetivo permitir que o método adicionaProduto da classe CarrinhoDeCompras possa ser invocado sem a necessidade de que seja instanciado um objeto do tipo Produto, fortalecendo assim o princípio do isolamento destacado por IEEE (2006).

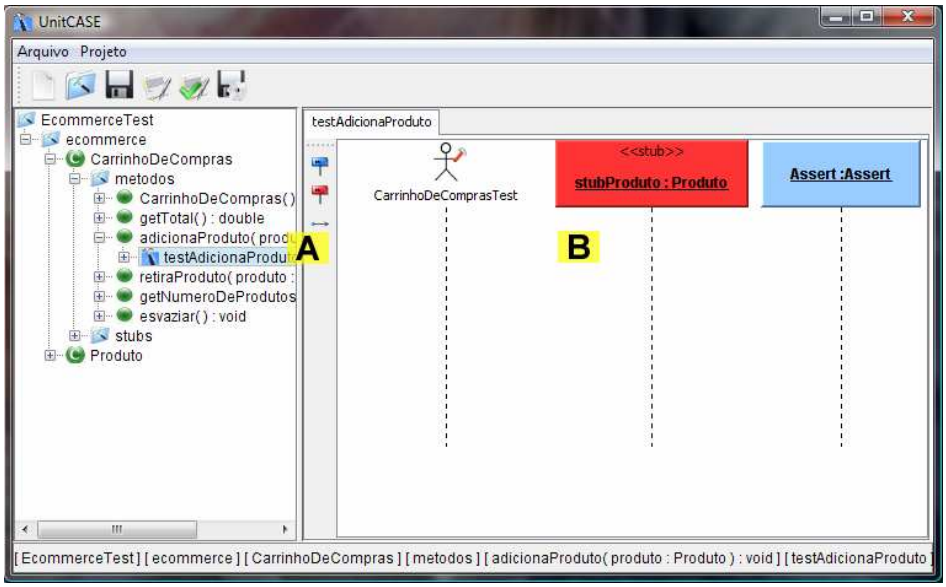

Figura 5. Interface para definição e visualização do caso de teste

O UC06 permite ao usuário a definição do *SetUp*, funcionalidade oferecida pelo *framework* JUnit (e todos os demais da família xUnit), que permite criar objetos que podem ser utilizados por vários casos de teste. No exemplo apresentado, ilustrado na Figura 6, é criado um objeto com nome de "carrinho", como instância da classe CarrinhoDeCompras, sendo que para sua criação é selecionado o construtor padrão da classe.

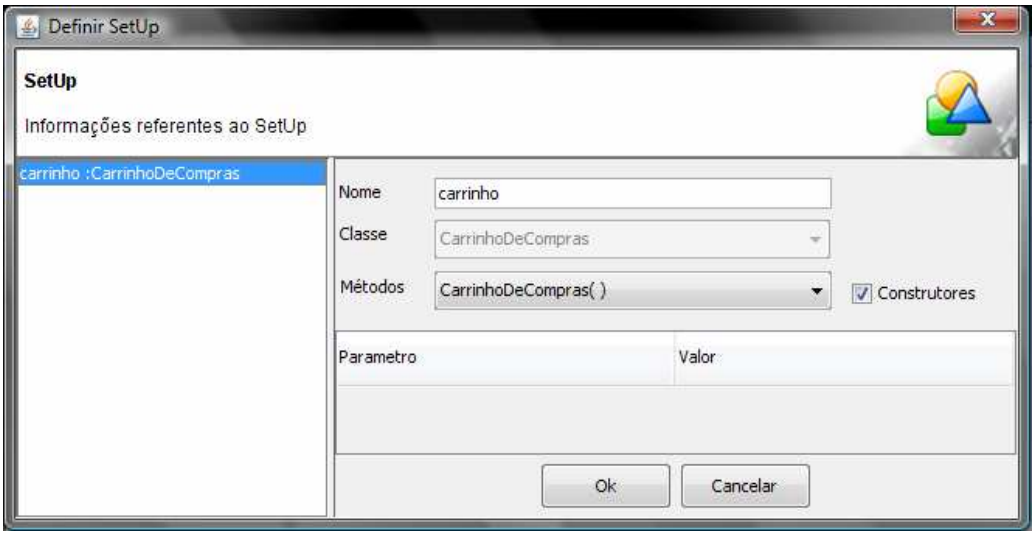

Figura 6. Interface para definição dos objetos do SetUp

Outra funcionalidade necessária é a criação de um *Stub* para uma classe (UC07). Esta funcionalidade é obtida a partir da seleção de uma das classes do modelo, quando a ferramenta apresenta a interface para criar um novo *Stub*, conforme demonstra a Figura 7a e incluí-lo em um caso de teste Figura 7b.

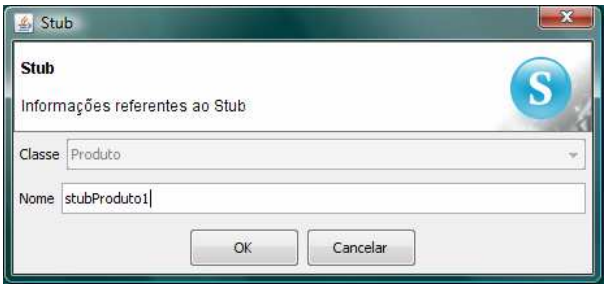

Figura 7a. Interface para criação de um novo *Stub*

Figura 7b. Interface para adicionar um Stub ao caso de teste

Com os elementos do caso de teste definidos, pode-se então estabelecer as mensagens entre o gerenciador do caso de teste e os demais objetos (comportamento previsto para o UC05). Para definição de uma mensagem em um caso de teste é utilizada a interface apresentada na Figura 8, devendo ser indicado o método que será invocado e o valor dos parâmetros (se necessário).

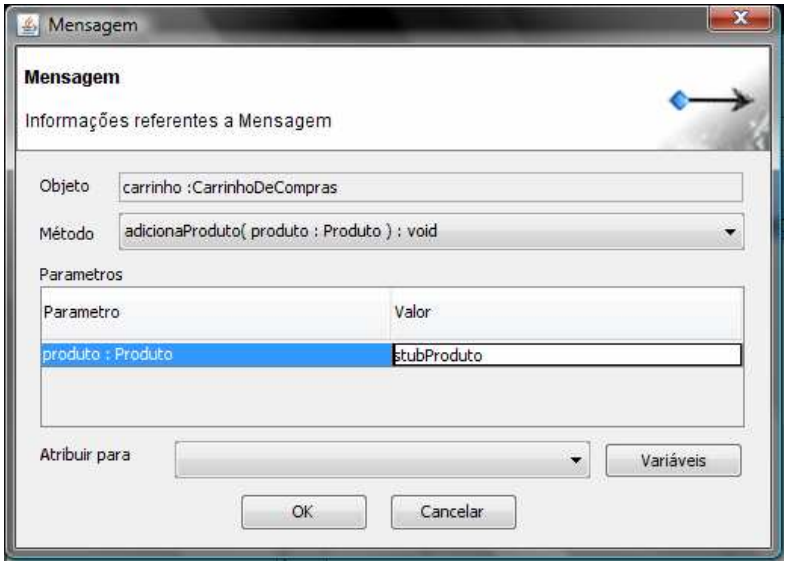

Figura 8. Interface para definição da mensagem

Para o caso de teste “testAdicionaProduto” foram definidas quatro mensagens, apresentadas na Figura 9, com o objetivo de verificar se o método responde como esperado. A primeira delas adiciona o *Stub* stubProduto ao objeto carrinho, a mensagem seguinte invoca o construtor da classe Produto, a fim de construir o objeto produto1 o qual é adicionado ao objeto carrinho na mensagem seguinte. Por fim, a última mensagem do caso de teste verifica se o número de produtos do objeto carrinho corresponde ao valor esperado.

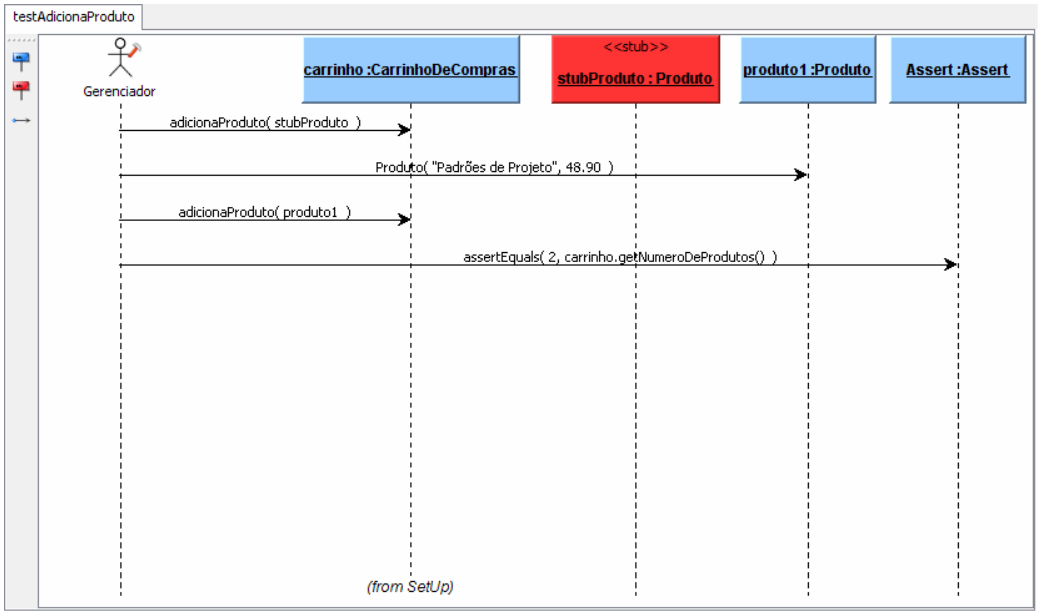

Figura 9. Caso de teste testAdicionaProduto

Utilizando a opção de validação (UC08), o sistema apresenta “críticas” quanto aos possíveis problemas encontrados nos casos de teste definidos pelo usuário. Estas críticas referem-se, por exemplo, a casos de teste que não possuem chamadas aos métodos da classe Assert do *framework* JUnit, ou métodos que não possuem casos de teste, ou mesmo casos de teste que não possuam pelo menos uma chamada ao método testado, conforme apresentado na Figura 10.

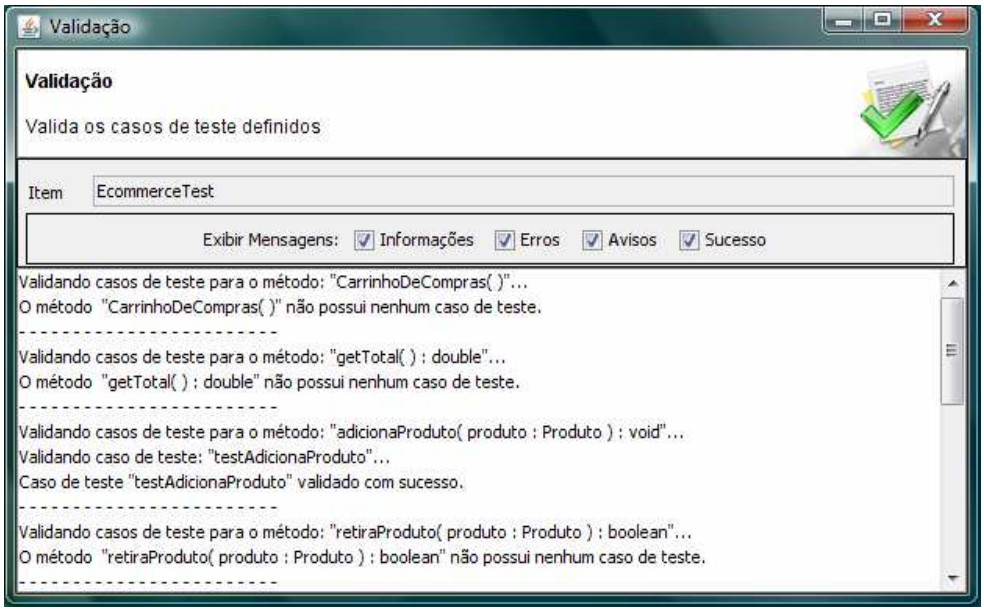

Figura 10. Interface para validação dos casos de teste

A geração do código fonte dos casos de teste definidos pelo usuário (UC09) é realiza a partir da interface demonstrada na Figura 11. No exemplo apresentado, o código gerado pela ferramenta para o caso de teste "AdicionaProduto" pode ser visualizado na Figura 12.

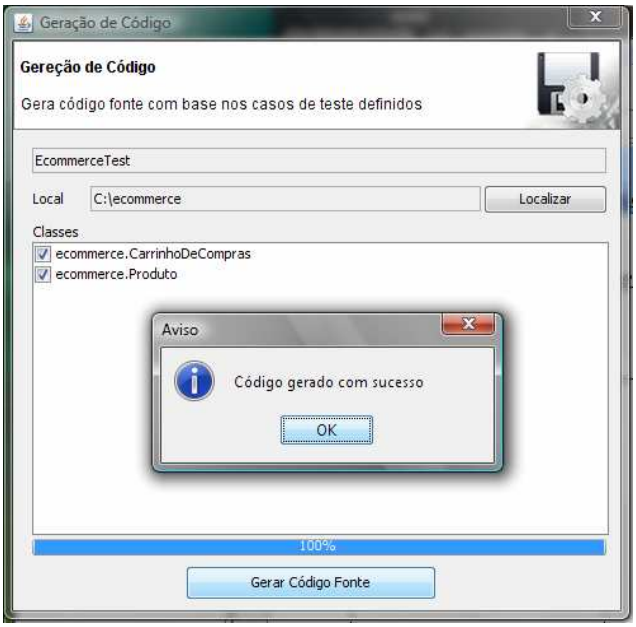

Figura 11. Interface para geração de código fonte

```
package EcommerceTest.ecommerce;
import …
public class CarrinhoDeComprasTest {
   ecommerce.CarrinhoDeCompras carrinho;
   @Before
   public void setUp() throws Exception {
      carrinho = new ecommerce.CarrinhoDeCompras(  );
   }
   @Test
   public void testAdicionaProduto() {
      ecommerce.Produto stubProduto = EasyMock.createMock(ecommerce.Produto.class);
      this.carrinho.adicionaProduto( stubProduto  );
      Produto produto1 = new Produto("Padrões de Projeto", 48.90);
      this.carrinho.adicionaProduto( produto1  );
      Assert.assertEquals( 2, carrinho.getNumeroDeProdutos()  );
   }
}
```

Figura 12. Código fonte gerado pela ferramenta

5. Exemplo de aplicação do diagrama proposto em diferentes contextos

Como base para demonstração do diagrama de seqüência para definição de casos de teste focando em testes de unidade utilizou-se de algumas das classes do sistema TeamDOC da empresa QualyTeam, que desenvolve sistemas para gestão da qualidade. O TeamDOC, que corresponde a um GED (Gerenciador Eletrônico de Documentos), permite aos seus usuários controlar através de revisões os documentos produzidos pela empresa. A figura 13 apresenta em um diagrama de classes parte das classes do sistema com alguns de seus atributos e métodos, este diagrama será utilizado a seguir como base para definição dos casos de teste.

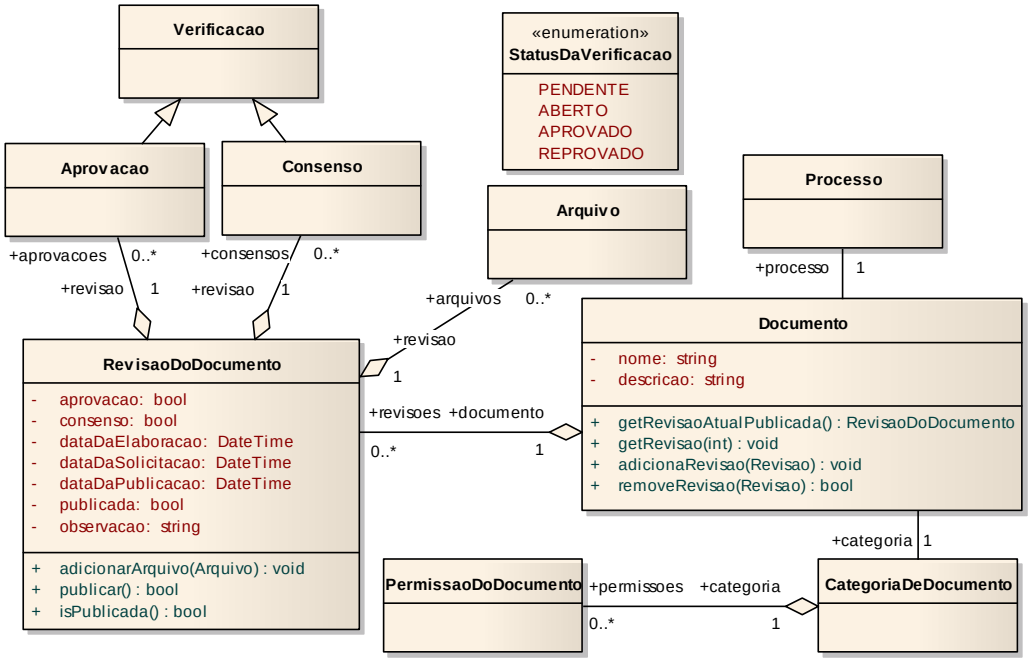

Figura 13. Diagrama de classes do sistema TeamDOC

Com base no diagrama de classes são apresentados casos de teste para os métodos getRevisaoAtualPublicada(), getRevisao(int) da classe Documento, já a classe RevisaoDoDocumento() terá um caso de teste para o método publicar.

O caso de teste para o método getRevisao(int) adiciona duas revisões ao documento e verifica se a revisão retornada pelo método corresponde ao índice especificado, este caso de teste é apresentado na figura 14.

Figura 14. Caso de teste para o método getRevisao(int)

O método getRevisaoAtualPublicada() da classe Documento retorna sempre que invocada a revisão mais atual publicada para o documento em questão, caso não exista nenhuma revisão publicada este retorna null. O caso de teste apresentado na figura 15 verifica seu funcionamento adicionando duas revisões ao documento e comparando o retorno do método testado com o valor esperado.

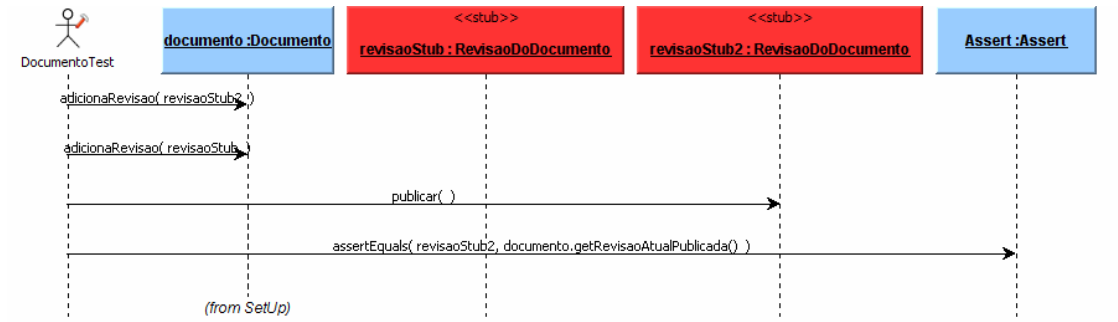

Figura 15. Caso de teste para o método getRevisaoAtualPublicada()

Cada documento no sistema TeamDOC possui uma única revisão publicada, as demais revisões de um documento podem encontrar-se em elaboração, que corresponde ao estado inicial de uma revisão ou cancelada, estado assumido após a publicação de uma revisão mais recente para o documento. O método publicar() da classe RevisaoDoDocumento deve alterar o estado da revisão para publicada, caso esta esteja em elaboração e avisar ao documento sobre a alteração, para que este altere o estado da revisão anterior, caso exista, para cancelada. O caso de teste apresentado na figura 16 verifica este comportamento adicionando a um documento duas revisões, inicialmente uma destas revisões é publicada, verifica-se se o estado desta e, em seguida, é publicada uma nova revisão e novamente verifica-se o estado das revisões do documento.

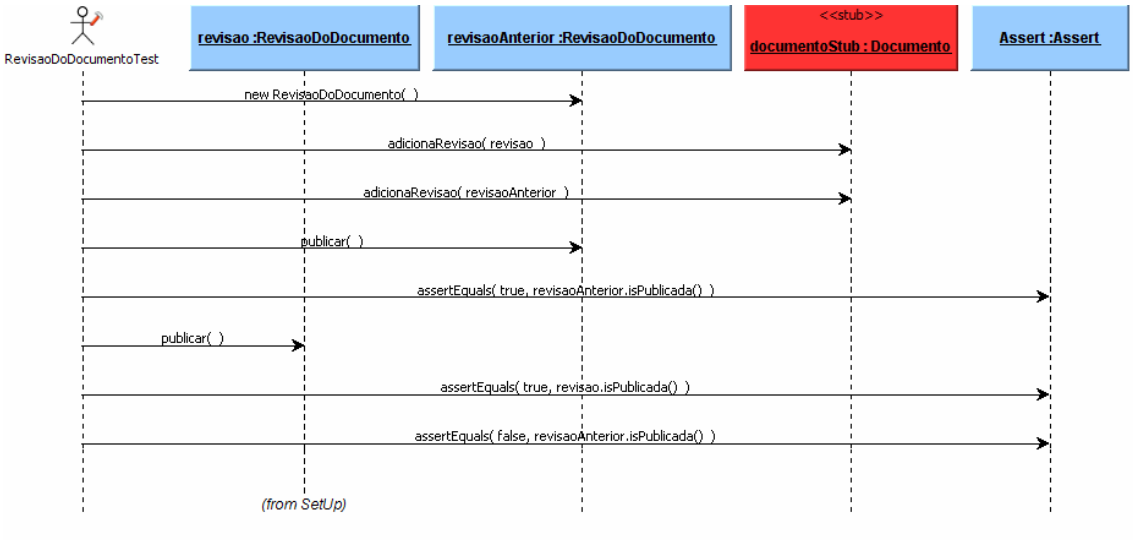

Figura 16. Caso de teste para o método publicar

Com base nos casos de teste definidos para o sistema TemDOC apresentados nesta seção pode-se verificar a adequação do diagrama de seqüência proposto, diagramas estes que foram definidos com o auxílio da ferramenta apresentada neste documento.

6. Considerações Finais

A principal contribuição deste artigo é apresentar uma proposta para utilização do diagrama de seqüência como artefato de definição de casos de teste visando auxiliar, por exemplo, a aplicação da técnica TDD.

O uso do diagrama de seqüência na definição dos casos de teste, ganha destaque à medida que este não foi concebido para este fim. Apesar disso, sua adequação se deu de forma natural, não exigindo a criação de qualquer elemento ou comportamento em especial, apenas utilizando de recursos já existentes na UML, como o estereótipo, utilizado para identificação do *stub*.

Biasi e Becker (2006) apresentam alguns problemas na utilização de ferramentas para automatização de: (i) o esforço necessário para preparar o código auxiliar; (ii) nem sempre há distinção clara entre as atividades de especificação dos casos de teste e de programação do código auxiliar; e (iii) dependência do caso de teste da linguagem de programação. Desta forma, o diagrama de seqüência como proposta para definição de casos de teste juntamente com a ferramenta desenvolvida se configuram como solução para os problemas citados em [Biasi e Becker 2006], dado que o código auxiliar é gerado automaticamente pela ferramenta. Os casos de testes podem ser definidos de forma visual sem qualquer contato com a linguagem de implementação, tendo como base apenas o modelo de classes do projeto.

O uso do diagrama de seqüência para definição dos casos de teste se demonstrou compatível com os recursos necessários para tal aplicação, dado que a ferramenta desenvolvida com base nesta definição pode gerar o código fonte para os casos de teste, tendo como base tão somente o diagrama de seqüência definido pelo usuário na própria

ferramenta. No entanto, esta ferramenta necessita de melhorias para sua aplicação em um cenário corporativo, podendo-se citar como prioritárias as seguintes adequações: (i) permitir a execução dos testes integrado à ferramenta; e (ii) viabiliza a semi-automatização da geração do diagrama de casos de teste a partir de um diagrama de seqüência de projeto.

Por fim, acredita-se que o uso de uma linguagem visual para definição de casos de teste – linguagem esta que já é de domínio de parte dos analistas - permite uma melhor comunicação e interação entre os envolvidos no processo de desenvolvimento.

Referências

Astels, D. (2003) Test Driven development: a practical guide, Prentice Hall.

Badri, M., Badri, L. e Bourque-Fortin, M.; (2005) Generating unit test sequences for aspect-oriented programs: towards a formal approach using UML state diagrams. In *3rd International Conference on Information and Communications Technology*. p. 237 – 253. Coimbatore, India.

Beck, K. (2002) Test driven development: by example, Addison-Wesley.

Biasi, L. e Becker, K. (2006) Geração automatizada de drivers e stubs de teste para JUnit a partir de especificações U2TP. In *Proceedings of Brazilian Symposium of Software Engineering (SBES)*, Florianópolis – Santa Catarina.

Hetzel, W. (1987) Guia completo do teste de software, Campus.

IEEE (1986) IEEE standard for software unit testing.

IEEE (2006). A survey of unit testing practices. In *IEEE Software*, v 23, Issue 4, p. 22 – 29, July-Aug.

Linzhang, W.; Jiesong Y.; Xiaofeng Y.; Jun H.; Xuandong L.; Guoliang Z. (2004). Generating test cases from UML activity diagram based on Gray-box method. In 11th *Software Engineering Conference*. p. 284 – 291. Asia-Pacific.

Macgregor, J. e Sykes, D. (2001) A Practical guide to testing object-oriented software, Addison-Wesley.

Maldonado, J. C. e Fabbri, S. C. P. F. (2001) Teste de software. *In*: Rocha, A. R. C. da; Maldonado, J. C.; Weber, K. C. (Coord.). Qualidade de software: teoria e prática. São Paulo: Prentice Hall, p. 73-84.

Thomas, J., Young, M., Brown, K. e Glover, A. (2004) Java testing patterns, John Wiley.