Geração automatizada de drivers e stubs de teste para JUnit a partir de especificações U2TP

Luciano B. Biasi, Karin Becker

Programa de Pós-Graduação em Ciência da Computação – Faculdade de Informática
Pontifícia Universidade Católica do Rio Grande do Sul (PUCRS)
Porto Alegre – RS – Brasil

{lbiasi, kbecker}@inf.pucrs.br

Abstract. *Testing has become essential to ensure the quality of software products. Within the test process, unit testing is performed on the smallest functional part of the software with the aim of discovering defects in these units. JUnit is a unit testing tool that helps developers in test automation and verification of the results. However, much time, cost and effort are spent on the programming of the necessary drivers and stubs, minimizing the benefits expected from its use. UML 2.0 Test Profile (U2TP) allows one to specify all test artifacts using a standard, high level and visual notation, which is independent of any specific programming language. This work addresses the automated generation of test drivers and stubs for JUnit, given a set of test cases specified with U2TP. The proposed models and algorithms were applied in a case study, and all corresponding test code was correctly generated.*

Resumo. *Teste é amplamente usado para garantir a qualidade em produtos de software. O teste de unidade é realizado na menor parte funcional de um software e visa descobrir defeitos nestas unidades. JUnit é uma ferramenta de apoio ao teste unitário, a qual auxilia desenvolvedores na automação dos testes e verificação dos resultados. Porém, muito tempo e esforço são gastos para codificar os drivers e os stubs de teste necessários a esta ferramenta, minimizando os benefícios esperados. O Perfil de Teste da UML 2.0 (U2TP) permite especificar artefatos de teste em uma notação padronizada, de alto nível, gráfica, e independente de linguagem de programação. Este trabalho aborda a geração totalmente automatizada de drivers e stubs de teste para ferramenta JUnit a partir de especificações de testes modeladas com o U2TP. Os modelos e algoritmos desenvolvidos foram aplicados a um estudo de caso, gerando corretamente todo o código correspondente.*

1. Introdução

Teste de software é o processo de executar um sistema de maneira controlada, a fim de revelar seus defeitos e avaliar sua qualidade. Um teste unitário consiste em testar unidades individuais de uma aplicação a fim de descobrir defeitos nas mesmas. O nível de abstração destas unidades depende muito do tipo de sistema sendo desenvolvido. Segundo [Burnstein, 2003], o processo de teste unitário deve envolver as seguintes atividades: Planejamento, Especificação e Projeto dos casos de teste, Preparação do código auxiliar e Execução propriamente dita. O código auxiliar (*Test Harness*) é

constituído de *drivers* e *stubs* de teste. Um *driver* é uma unidade que implementa chamadas às funcionalidades testadas. *Stubs* servem para substituir funcionalidades que ainda não foram implementadas ou que estão subordinadas ao módulo sendo testado.

A automação da execução de testes tem sido uma tendência, pois diminui o tempo gasto na execução dos testes e possibilita a verificação automática dos resultados. Para teste unitário, ferramentas bastante populares são aquelas da família denominada genericamente de *xUnit,* entre elas cppUnit (C++), VBUnit (Visual Basic), NUnit (.net) e JUnit (Java). Entretanto, alguns problemas são encontrados no uso deste tipo de ferramenta, entre eles: a) o esforço necessário para preparar o código auxiliar, em particular os *drivers*, que representam implementações dos casos de teste; b) nem sempre há uma distinção clara entre as atividades de especificação dos casos de teste e de programação do código auxiliar correspondente; c) frequentemente é o próprio programador da unidade quem fica encarregado da especificação do caso de teste, muitas vezes levando em conta a programação realizada na unidade e não os requisitos da aplicação; e d) dependência da especificação do caso de teste da linguagem de programação, própria a cada ferramenta.

O Perfil de Teste da UML 2.0 [OMG, 2004], conhecido como U2TP, tem por objetivo oferecer uma notação padronizada à especificação de diversos aspectos de teste. Assim, é possível especificar os testes em um mais alto nível de abstração e independente de linguagem de programação, resultando em uma documentação adequada, objetiva e sem ambigüidades sobre os testes que devem ser realizados, independentemente de ferramentas que possam vir a ser utilizadas para automatizá-los. Neste sentido, o U2TP estabelece um mapeamento dos seus conceitos com ferramentas de testes automatizados como JUnit e TTCN-3 [Dai e Gabrowski, 2003]. Assim, através de especificações de testes modeladas com o U2TP é possível não apenas representar os artefatos utilizados no processo de teste, mas também utilizá-los como ponto de partida para a geração automatizada do código de teste para diferentes ferramentas.

A geração automatizada de testes para ferramenta JUnit tem sido a preocupação de vários trabalhos encontrados na literatura. Nestes, as especificações de teste são documentadas em UML (e.g. [Wittevrongel e Maurer, 2001] [Fraikin e Leonhardt, 2002]), ou linguagens de especificação de teste que abstraem linguagens de programação (e.g. [Cheon e Leavens, 2002] Zongyuan et al., 2004]). Nenhum dos trabalhos encontrados é baseado no U2TP, nem aborda por completo a geração automatizada de todo código de teste, incluindo *drivers* e *stubs*.

Este artigo propõe a geração automatizada de *drivers* e *stubs* de teste para a ferramenta JUnit, considerando especificações de teste unitário em U2TP, e sem interação com o usuário.

O restante deste trabalho está estruturado como segue. A Seção 2 apresenta os conceitos relevantes da ferramenta JUnit. A Seção 3 discute o U2TP, com foco em seu uso para especificação em teste unitário. Na Seção 4 é apresentada a proposta do trabalho, descrevendo o cenário para geração dos *drivers* e *stubs* de teste, bem como as convenções adotadas para especificação de testes em U2TP. Na Seção 5 são apresentados o extrator e o gerador de código. Um estudo de caso é relatado na Seção 6 e trabalhos relacionados são discutidos e comparados na Seção 7. A Seção 8 apresenta as conclusões e as direções futuras de pesquisa.

2. Junit

JUnit é um framework para teste de unidade usado por programadores que desenvolvem aplicações em linguagem Java [Beck, 2003]. Casos de teste no JUnit são constituídos por um ou mais métodos, sendo que estes podem estar agrupados em suítes de teste.

Um *driver* JUnit é constituído por uma classe principal, cujos métodos representam os casos de teste. Adicionalmente, os métodos *setup* e *teardown* permitem especificar pré e pós condições comuns a todos os casos de teste. A execução de cada caso de teste no JUnit resulta na ativação dos seguintes métodos: (1) *setup*; (2) método correspondente ao caso de teste; e (3) *teardown*. O JUnit permite o uso de diferentes assertivas para comparar os testes, tais como *assertTrue*, *assertEquals e assertFalse.* Um teste no JUnit pode apresentar como resultado três valores: *pass*, *fail* ou *error*.

JUnit foi adaptado a uma variedade de IDEs (*Integrated Development Evironment)* que favorecem a escrita do código para uma melhor interpretação dos procedimentos de teste. As IDEs mais conhecidas são Eclipse [Eclipse, 2005] e NetBeans [Netbeans, 2005]. No Eclipse, por exemplo, é possível gerar a estrutura do *driver* automaticamente a partir da classe a ser testada, sendo que este esqueleto deve ser completado manualmente com código para os casos de teste e a declaração de variáveis.

3. Perfil de Teste da UML 2.0

3.1. Conceitos do Perfil U2TP

O U2TP define uma linguagem para projetar, visualizar, especificar, construir e documentar artefatos de teste para sistemas [OMG, 2004]. O U2TP pode ser usado somente para a manipulação dos artefatos de teste, ou de uma maneira integrada com UML, para a manipulação conjunta de um sistema e respectivos artefatos de teste. Seus conceitos estão organizados em quatro grupos lógicos: Arquitetura, Dados, Comportamento e Tempo. Para este trabalho, foram adotados elementos pertencentes aos grupos de arquitetura de teste e comportamento de teste, por serem os mais relevantes ao teste unitário. O grupo de dados, igualmente útil, não foi considerado nesta etapa do trabalho por questões de simplificação. Os construtors destes grupos que foram utilizados neste trabalho são descritos abaixo, e representados no meta-modelo da Figura 1.

- *Arquitetura de teste*: um *TestContext* representa o agrupamento de vários casos de teste, os quais se referem a um ou mais *Sut*s *(system under test). Sut* corresponde ao que será testado. *TestCase* é uma especificação de uma situação de teste do sistema, incluindo o que testar, entradas, e resultados esperados de acordo com as condições. *TestControl* determina como o *Sut* deve ser testado em um determinado contexto de teste (*TestContext*). Um *TestComponent* é uma classe de um sistema em teste que tem por objetivo auxiliar na execução de um ou mais casos de teste. Em teste unitário, é usado para representar um *stub*.
- *Comportamento de teste*: O comportamento de um caso de teste é especificado por diagramas comportamentais da UML (e.g. diagrama de seqüência). Um caso de teste sempre deve retornar um *verdict* (i.e. *pass*, *fail, inconclusive* ou *error*).

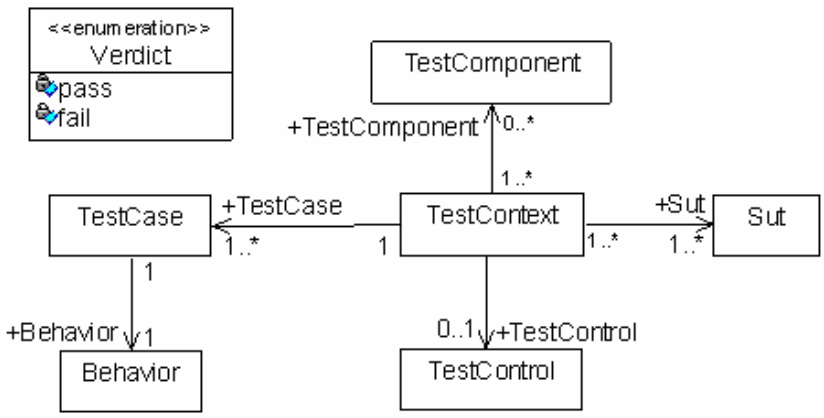

Figura 1. Construtos do U2TP adotados no trabalho.

3.2. Mapeamento do U2TP para JUnit

Os elementos descritos anteriormente possuem um mapeamento direto com os conceitos do JUnit, apresentado na Tabela 1, adaptada de [OMG, 2004].

Tabela 1. Mapeamento entre conceitos U2TP e JUnit.

U2TP	JUnit
TestContext	Subclasse de TestCase.
Sut	Qualquer classe Java.
TestCase	Uma operação pertencente a subclasse TestCase.
Behavior	Implementação de um método.
TestComponent	Uma classe Java.
TestControl	Método *runTest* que deve ser subscrito.
Verdict	Pass, fail e error.

3.3. Metodologia de Uso do U2TP

Uma metodologia para usar o U2TP é proposta em [Dai et al., 2004]. Ela assume que o U2TP é usado efetivamente a partir de um modelo de projeto UML existente, o qual deve ser enriquecido de detalhes para desenvolver as especificações de teste com o U2TP. A metodologia descreve e ilustra, passo a passo, como especificar testes para um sistema utilizando os construtos do U2TP. Para cada grupo do U2TP, são especificados os elementos mais importantes destes, e como podem ser usados. Ela também define para cada grupo do U2TP os elementos obrigatórios e opcionais. Contudo, ela não estipula o uso de cada elemento de acordo com o nível de teste (unidade, integração e sistema). Esta metodologia foi adotada neste trabalho. Com seu apoio, foram identificados os grupos do U2TP, junto com seus respectivos elementos, relevantes para o contexto de teste unitário. Após, foram definidas algumas convenções de uso destes construtos visando o mapeamento para código JUnit.

4. Uma abordagem para geração automatizada de drivers e stubs de teste para JUnit

O presente trabalho propõe uma abordagem para geração automatizada de *drivers* e *stubs* para JUnit, considerando um núcleo de conceitos do U2TP (Figura 1) e o

respectivo mapeamento destes nos conceitos do Junit (Tabela 1). A Figura 2 ilustra o cenário proposto para geração automatizada de *drivers* e *stubs* de teste.

As entradas necessárias são o projeto UML do software que será testado, junto com o projeto de teste unitário correspondente especificado com o U2TP. Ambos devem estar de acordo com uma série de pressupostos, discutidos na Seção 4.1. Estes dois modelos são gerados usando uma ferramenta de modelagem, e exportados como um documento XMI (XML Metadata Interchange). A geração dos *drivers* e *stubs* correspondentes é realizada em duas etapas: *Extração* e *Geração*. A Extração é baseada em dois modelos definidos neste trabalho: Modelo de Mapeamento U2TP-JUnit e Modelo de Mapeamento U2TP-XMI. O primeiro define quais elementos do U2TP são necessários para geração de *drivers* e *stubs* para ferramenta JUnit. O segundo define quais rótulos (*tags*) do documento XMI correspondem aos conceitos buscados e como os relacionamentos entre estes devem ser explorados. O Extrator então instancia o Modelo de Mapeamento U2TP-JUnit com base nos documentos XMI de entrada e, juntamente com um Analisador (*parser*), captura todos elementos que comporão os *drivers* e *stubs*. Finalmente, o Gerador de Código gera o código Java baseado em um *template* de *drivers* para JUnit, além de classes Java representando os *stubs*. É importante ressaltar que este processo não requer interação com o usuário.

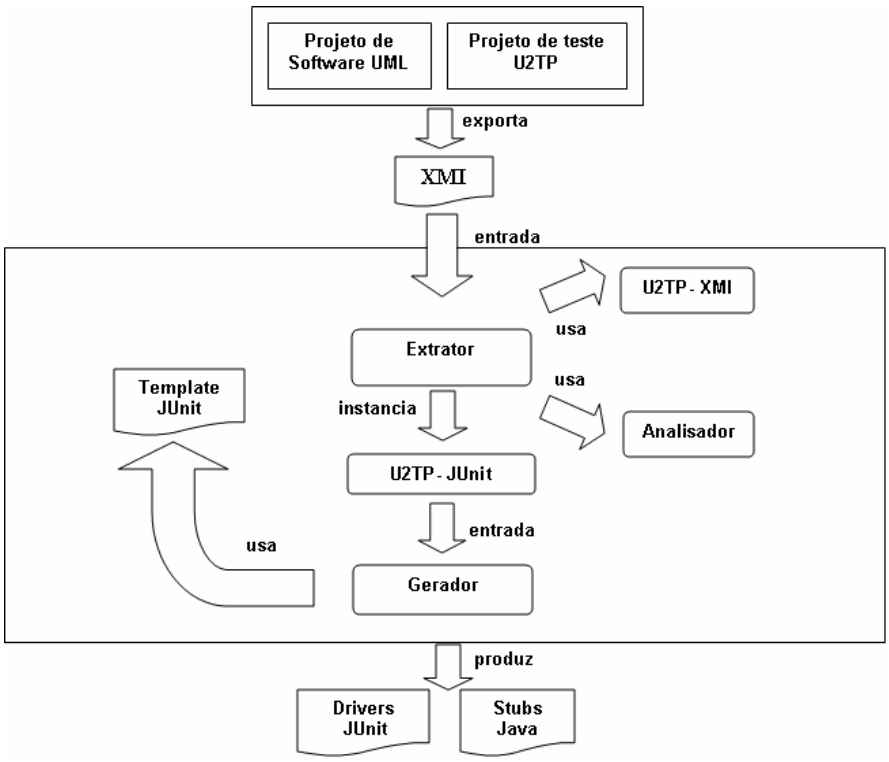

Figura 2. Cenário proposto.

4.1. Pressupostos para modelagem

Além da adoção da metodologia de [Dai et al., 2004], foram assumidos pressupostos necessários para especificar e modelar teste em nível de unidade com o U2TP. Os

pressupostos estabelecem simplificações ou padronizações sobre o uso da UML ou do U2TP, a fim de viabilizar a geração de *drivers* e *stubs*, e estão divididos em dois grupos: Projeto de Software e Projeto de Teste U2TP.

4.1.1 Projeto de Software

Assume-se a existência de um projeto de software UML correspondendo ao que será testado. Este deve conter obrigatoriamente um diagrama de classes em nível de projeto, pois define as operações das classes que podem ser testadas. Diagramas de seqüência são relevantes para a geração de *stubs*, mas não são obrigatórios. A Figura 3.(a) ilustra um projeto de software simples denominado Venda.

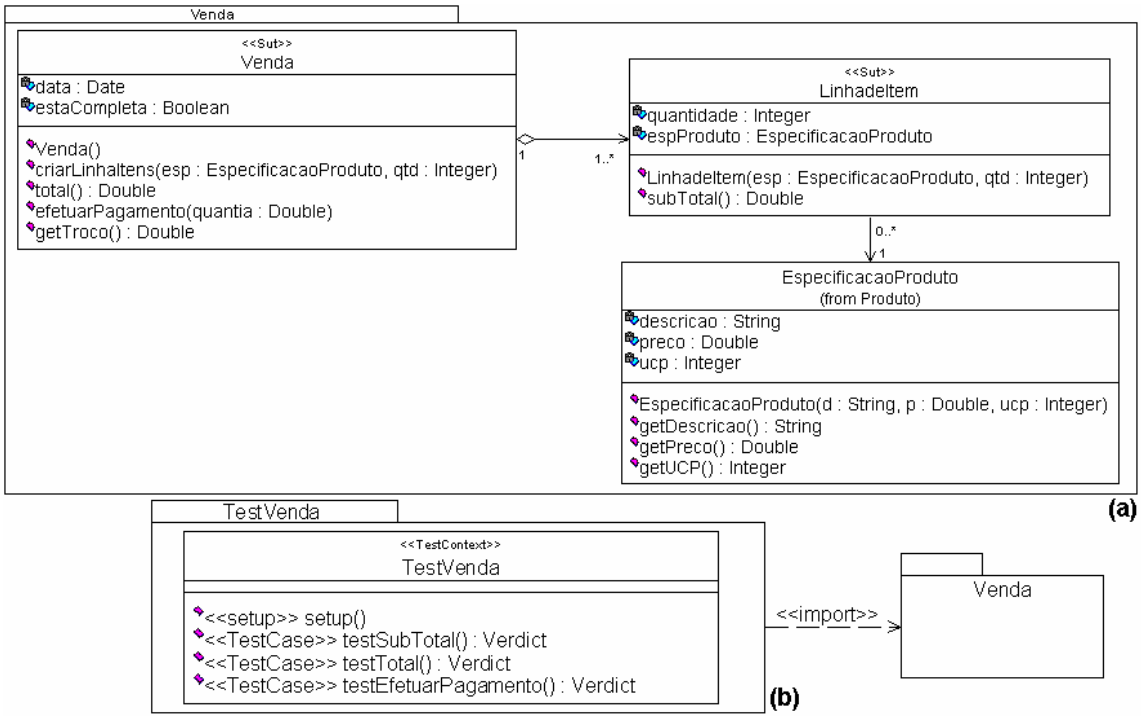

Figura 3. Projeto de Software e Projeto de Teste U2TP.

4.1.2 Projeto U2TP

Assume-se a definição de um pacote de teste referente ao projeto que será testado, como ilustrado na Figura 3.(b).

Arquitetura de Teste

- *Sut*: Todas as classes de software que serão testadas devem ser estereotipadas no modelo de projeto com <<sut>>, de acordo com a metodologia de [Dai et al., 2004]. Pelo menos um elemento <<sut>> é obrigatório (Figura 3.(a)).
- *TestContext* e *TestCase*: Deve haver uma e somente uma classe no pacote de teste estereotipada com <<testcontext>>. Nesta, cada caso de teste deve ser representado através de uma operação estereotipada com <<testcase>>. Deve haver no mínimo uma operação <<testcase>>.

- *TestComponent*: Podem ser especificadas várias classes estereotipadas com <<testcomponent>>. Estas podem ser especificadas tanto no projeto de software, quanto no pacote de teste. Quando definido no projeto de software, especifica-se que as classes *sut* dependem de funcionalidades de classes <<testcomponent>>. No pacote de teste, é possível criar novas classes representando apenas aqueles aspectos necessários ao teste.

Comportamento de Teste

- *TestCase*: o comportamento de um caso de teste deve ser especificado através de diagramas de seqüência, como na Figura 4 para o caso de teste testSubTotal.

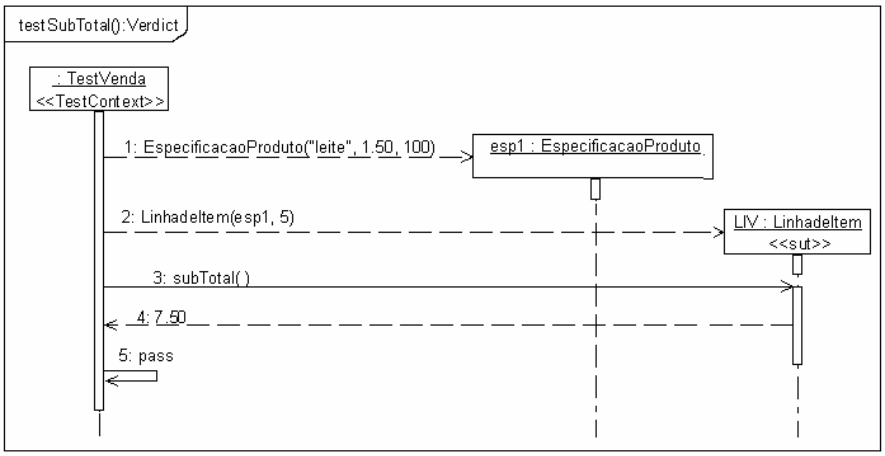

Figura 4. Comportamento do caso de teste testSubTotal.

Ainda, são assumidas as seguintes convenções para este tipo de diagrama:

1. O diagrama de seqüência deve ter o nome do caso de teste;
2. O objeto iniciador da interação do diagrama de seqüência deve ser uma instância da classe estereotipada com *TestContext*;
3. Os demais objetos do diagrama de seqüência são instâncias de classes do tipo *Sut*, *TestComponent* ou de qualquer classe do projeto de software, a qual será interpretada com um *stub* necessário. Estas instâncias devem ser nomeadas, e podem ser usadas como variáveis em outras mensagens do diagrama;
4. As mensagens representadas por setas pontilhadas partindo do objeto *TestContext* e recebidas pela classe representam a criação de um novo objeto;
5. Só é permitido um veredito por caso de teste, representado pela última mensagem do diagrama, na forma de uma auto-delegação ao objeto *TestContext*. A mensagem deve ser *pass* ou *fail*;
6. O veredito é estabelecido comparando o método representado pela antepenúltima mensagem, normalmente correspondente ao método testado, com o valor especificado para o teste. Assume-se que a penúltima mensagem corresponde ao valor especificado para o teste. O conjunto das três últimas mensagens constitui a assertiva para o caso de teste;

7. É possível declarar mensagens onde uma variável recebe o retorno de uma operação da seguinte forma: <variável> := <operação>.

Admite-se o uso de diagramas de seqüência para representar pré ou pós-condições comuns a todos os casos de teste. Para este fim, foram definidos dois novos estereótipos, <<setup>> e <<teardown>>, os quais devem ser associados a operações na classe <<testcontext>>. Pode haver no máximo uma operação <<setup>> e uma <<teardown>>, sendo que estes elementos são opcionais. Quando definidos, seu comportamento também deve ser especificado através de diagramas de seqüência.

4.2. Modelo de Mapeamento U2TP-JUnit

Na Tabela 1 foi apresentado o mapeamento do U2TP para os conceitos correspondentes na ferramenta JUnit. Além destes, foram acrescentados neste trabalho dois novos elementos para especificação de testes, a saber, *Configuration* e *Stub*. O diagrama de classes da Figura 5 estende o meta-modelo apresentado na Figura 1, apresentando todos os elementos usados para geração do código no JUnit. As classes em cor escura representam os elementos adicionados. O elemento *Stub* representa todos os *Stubs* de teste necessários à execução dos testes, os quais podem ter sido explicitamente especificados pelo projetista de software ou teste, ou ser derivados dos diagramas de seqüência do modelo de projeto. Um *testcomponent* é portanto um tipo de *stub*. *Configuration* é uma classe que representa os métodos *setup* e *teardown* do JUnit.

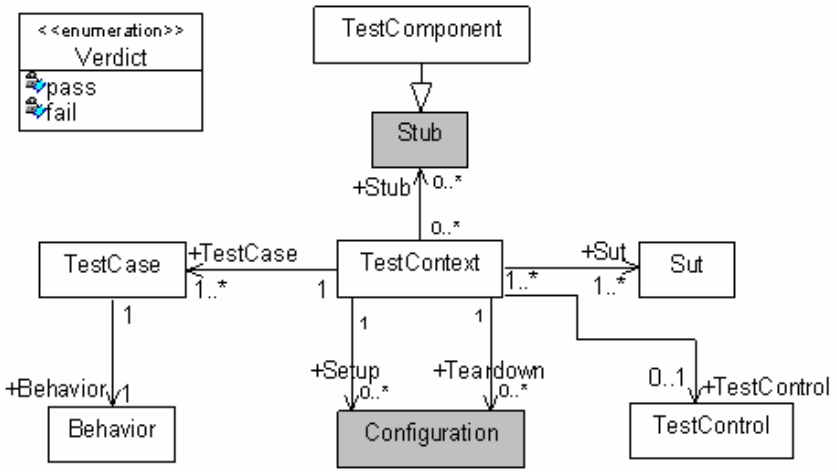

Figura 5. Modelo de mapeamento U2TP-JUnit.

4.3. Modelo de Mapeamento U2TP-XMI

Este modelo define como mapear os elementos do U2TP e extensões propostas em um documento XMI, indicando quais *tags* do documento XMI são necessários para extrair os elementos necessários à geração do código. Este mapeamento, sumarizado na Tabela 2, foi definido a partir do modelo de mapeamento U2TP-JUnit e das convenções adotadas. Cada elemento corresponde a uma ou mais *tags* XMI.

Tabela 2. Mapeamento dos elementos do U2TP e extensões para tags XMI.

Elemento U2TP e Extensões	XMI	Descrição
Sut, TestComponent, Stub e TestContext	XMIStereotype	identifica o tipo de elemento.
	XMIClass	representa o nome do elemento.
	XMIPackage	representa o nome do pacote onde se encontra o elemento.
	XMIClassifierRole e XMIDiagram	representa as instâncias do elemento usadas em diagramas de seqüência.
TestCase, Setup e Teardown	XMIStereotype	identifica o tipo da operação.
	XMIOperation	representa o nome da operação.
	XMIInteraction	representa a interação do testcase, setup e teardown.
	XMIMessage	representa as mensagens referentes àquela operação em diagramas de seqüência.
	XMIClassifierRole	representa os objetos destinatários das mensagens.
TestControl	XMIActionState	representa o nome das atividades do diagrama de atividades.

5. Extração e Geração de Drivers e Stubs

O extrator é responsável por identificar os elementos necessários contidos no documento XMI e armazená-los em uma instância do modelo de mapeamento U2TP-JUnit. O mecanismo de extração foi baseado no *parser* DOM – *Document Object Model* e no XMI gerado pela ferramenta Rational Rose. O *parser* DOM permite percorrer o documento XMI sem nenhuma ordem pré-definida. Dessa forma, é possível armazenar os elementos contidos no documento XMI em memória e manipulá-los mais facilmente. Para identificar e extrair os elementos necessários à geração de código, foram definidos algoritmos de mapeamento, os quais identificam os elementos no XMI com base no modelo U2TP-XMI, e os mapeiam para uma instância do modelo de mapeamento U2TP-JUnit.

Por fim, o gerador é responsável por gerar os elementos de código Java. No caso de *drivers*, esta geração segue um template da ferramenta alvo Junit. No caso de *stubs*, classes Java com os métodos necessários são geradas. O restante desta seção detalha os aspectos importantes da extração e geração de *drivers* e *stubs*.

5.1. Extração de Drivers

O parser DOM permite recuperar coleções de elementos possuindo um mesmo rótulo. Desta forma, para extrair os elementos necessários da especificação e instanciar o modelo U2TP-Junit são percorridas todas as *tags* correspondentes a cada elemento e através de um conjunto de métodos, extraídas as informações pertinentes.

O funcionamento do extrator será ilustrado considerando o caso de teste *testSubTotal*, detalhado no diagrama de seqüência da Figura 4. De acordo com a Tabela 2, cinco *tags* estão envolvidas. A *tag* que representa as interações envolvidas no detalhamento deste caso de teste através de um diagrama de seqüência é <UML:Interaction>, como ilustra a Figura 6.

```
- <UML:Interaction xmi.id="G.21" name="{Use Case View::Comportamento}testSubTotal"
    visibility="public" isSpecification="false">
  - <UML:Interaction.message>
      <UML:Message xmi.id="G.16" name="EspecificaçãodeProduto("leite", 1.50, 100)"
        visibility="public" isSpecification="false" sender="G.10" receiver="G.15" message3="G.17"
        communicationConnection="G.12" action="XX.26.1247.53.38" />
      <UML:Message xmi.id="G.17" name="LinhadeItemVenda(esp1, 5)" visibility="public"
        isSpecification="false" sender="G.10" receiver="G.14" message3="G.18"
        predecessor="G.16" communicationConnection="G.11" action="XX.26.1247.53.39" />
      <UML:Message xmi.id="G.18" name="subTotal( )" visibility="public" isSpecification="false"
        sender="G.10" receiver="G.14" message3="G.19" predecessor="G.17"                    antepenúltima
        communicationConnection="G.11" action="XX.26.1247.53.40" />
      <UML:Message xmi.id="G.19" name="7.50" visibility="public" isSpecification="false"
        activator="G.18" sender="G.14" receiver="G.10" message3="G.20" predecessor="G.18"
        communicationConnection="G.11" action="XX.26.1247.53.41" />                         penúltima
      <UML:Message xmi.id="G.20" name="pass" visibility="public" isSpecification="false"
        sender="G.10" receiver="G.10" predecessor="G.19" communicationConnection="G.13"
        action="XX.26.1247.53.42" />                                                        última
    </UML:Interaction.message>
  </UML:Interaction>
```

Figura 6. Tag <UML:Interaction>.

O atributo *name* da *tag* <UML:Interaction> identifica o nome do caso de teste. As *tags* <UML:Message> representam as mensagens trocadas por objetos neste caso de teste. Para cada mensagem, existe um objeto que recebe (atributo *receiver*) e outro que envia a mensagem (atributo *sender*). Na Figura 6, o objeto que recebe a primeira mensagem é representando pelo identificador "G.15". Dessa forma, para saber o nome da instância do objeto que recebe esta mensagem, são percorridas todas *tags* <UML:ClassifierRole> até encontrar aquela com identificador igual a "G.15". De acordo com a Figura 7, o nome desta instância é esp1.

```
- <UML:ClassifierRole xmi.id="G.15" name="esp1" visibility="public" isSpecification="false"
    isRoot="false" isLeaf="false" isAbstract="false" base="S.298.1247.52.12"
    availableFeature="S.298.1247.52.16" message1="G.16">
  - <UML:ClassifierRole.multiplicity>
    - <UML:Multiplicity>
      - <UML:Multiplicity.range>
          <UML:MultiplicityRange xmi.id="id.2991547.18" lower="1" upper="1" />
        </UML:Multiplicity.range>
      </UML:Multiplicity>
    </UML:ClassifierRole.multiplicity>
  </UML:ClassifierRole>
```

Figura 7. Tag <UML:ClassifierRole> correspondente ao objeto esp1.

Para extrair o comportamento do caso de teste são buscadas no diagrama de seqüência todas as mensagens, com os respectivos destinatários, à exceção das três últimas mensagens. O formato da mensagem extraída varia conforme o tipo de mensagem: instanciação de classe (e.g. esp1 = new EspecificaçãodeProduto("leite", 1.50, 100)) ou mensagem à instância (e.g. LIV.subTotal()).

As três últimas mensagens são usadas para montar a assertiva para o caso de teste, como ilustra a Figura 8. A última mensagem corresponde ao veredito do caso de teste (*pass* ou *fail*), a qual equivale ao tipo de assertiva usada no caso de teste. A antepenúltima mensagem normalmente corresponde ao método testado, que será igual à mensagem representada pelo caso de teste. A penúltima mensagem corresponde ao valor esperado da execução da operação.

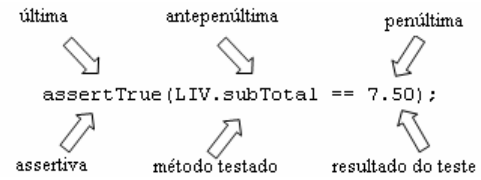

Figura 8. Três últimas mensagens constituem a assertiva.

Cabe salientar que neste exemplo foi extraído o comportamento do caso de teste testSubTotal. Na busca do nome do caso de teste, conforme a Tabela 2, estão envolvidas as *tags* <UML:Stereotype> e <UML:Operation>. Assim, o procedimento para buscar os demais elementos é o mesmo, mudando apenas as *tags* envolvidas na extração de cada elemento.

5.2. Gerador de Código para Driver

O template Junit define a estrutura básica de um *driver* de teste, descrita na Figura 9. Considerando este template, o nome e o comportamento do caso de teste testSubtotal, o código correspondente é gerado como segue:

```
public void testSubTotal(){
       esp1 = new EspecificaçãdeProduto("leite", 1.50, 100);
       LIV = new LinhadeItemVenda(esp1, 5);
       assertTrue(LIV.subTotal() == 7.50);}
```

```
(1)package pacote_testcontext; //obrigatório
(2)import junit.framework.TestCase;
(3)import nome_pacote_sut.nome_classe_sut;
(4)import nome_pacote_testcomponent.nome_classe_testcomponent;
(5)import nome_pacote_stub.nome_classe_stub;
...
(6)if (testcontrol != "){
(7)import junit.framework.TestSuite;
(8)import junit.framework.Test;
}

(9) public class nome_testcontext extends TestCase{ //obrigatório
(10)     private nome_classe_sut nome_objeto_sut; //obrigatório
(11)     private nome_classe_testcomponent nome_objeto_testcomponent;
(12)     private nome_classe_stub nome_objeto_stub;

(13)     public nome_testcontext(String testName){
(14)          super(testName);
(15)     }

(16)     protected void setup() throws Exception{
(17)          super.setup();
(18)          nome_objeto = new nome_construtor(parâmetros);
(19)          nome_classe nome_objeto = new nome_construtor(parâmetros);
(20)     }

(21)     protected void teardown() throws Exception{
(22)          super.teardown();
(23)          nome_objeto = null;
(24)     }

(25)     public void test_nome_testcase(){
(26)          nome_objeto.nome_mensagem;
(27)          //declaração de variáveis locais;
(28)          //assertiva, ex: asserTrue(método_testado == resultado_teste);
(29)     }
         ...
(30)     public static Test suite(){
(31)          TestSuite suite = new TestSuite();
(32)          suite.addTest(new nome_testcontext("nome_testcase"));
(33)          . . .
(34)          return suite;
(35)     }
(36)}
```

Figura 9. Template JUnit.

5.3. Extração e Geração de Stubs de Teste

Os s*tubs* são extraídos: a) a partir de especificações do elemento *testcomponent,* definidas tanto no projeto de software, como na especificação do teste U2TP; b) quando não explicitamente definidos, através de outros diagramas de seqüência do projeto de software. No primeiro caso, o extrator busca as *tags* correspondentes ao componente de teste, como definido na Tabela 2. No segundo, o extrator deve percorrer os diagramas de seqüência. A Figura 10 mostra um diagrama de seqüência do projeto de software do Sistema Venda referente ao comportamento do método *subTotal*.

Para extrair os *stubs*, são percorridas todas as *tags* que representam as interações dos diagramas de seqüência (exceto aquelas correspondentes ao comportamento de casos de teste) até encontrar uma mensagem correspondente ao método testado no caso de teste. Se o objeto que receber esta mensagem por sua vez enviar uma mensagem para um ou mais objetos, então é instanciado um *stub* para cada classe correspondente a estes objetos. Cada *stub* possui apenas um construtor e os métodos invocados. Cabe salientar que os *Stubs* são incluídos tanto no código do *driver* de teste, na forma de variáveis (Figura 9), como fora do *driver*, como classes Java.

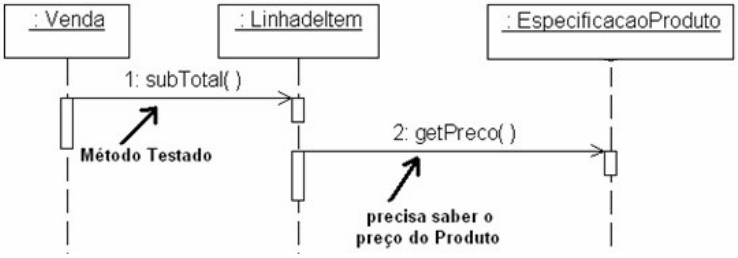

Figura 10. Diagrama de seqüência do método subTotal.

6. Estudo de Caso

Um estudo de caso foi realizado no centro de pesquisa em teste de software – CPTS, localizado na PUCRS. Para a geração dos *drivers* e *stubs* de teste foi usado um sistema para locações de DVDs desenvolvido em Java, o qual foi originalmente desenvolvido para experimentos de teste de software. Este sistema contém uma série de artefatos que foram usados para especificar os casos de teste usando o U2TP, entre eles: diagrama de classes, diagrama de casos de uso, casos de uso expandido, bem como alguns *drivers* de teste que já haviam sido previamente codificados por um testador do CPTS.

Para possibilitar a comparação entre o código produzido manualmente, e aquele gerado automaticamente, foram especificados apenas os testes referentes aos *drivers* previamente codificados no CPTS, os quais correspondem a métodos de 3 classes do sistema: CatalogoCategoria, Categoria, e CatalogoCliente. Juntas, estas classes contemplavam todas as situações contempladas pelos modelos e algoritmos propostos neste trabalho, como resumido na Tabela 3.

Tabela 3. Construtos U2TP encontrados em cada classe do estudo de caso.

Elementos/Classes	Categoria	CatalogoCategoria	CagalogoCliente
TestCase	4	4	3
TestComponent/Stub	0	3	2
Setup	0	1	1
Teardown	0	1	1
TestControl	0	0	1

A especificação dos casos de teste para estas classes utilizando o U2TP foi baseada na documentação recebida, e de acordo com os pressupostos descritos na Seção 4.1. A Figura 11 ilustra a especificação de teste unitário referente à classe CatalogoCategoria. A especificação para o comportamento do caso de teste testNew_config é mostrado na Figura 12.

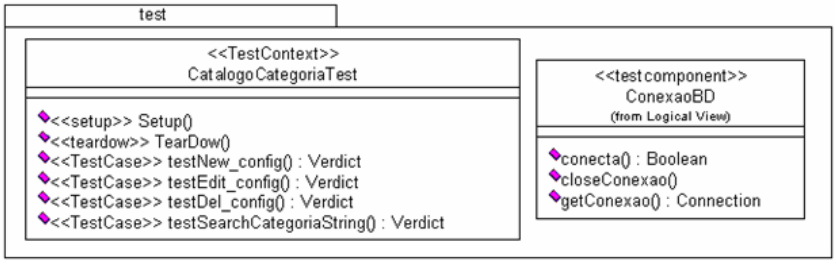

Figura 11. Pacote de teste unitário.

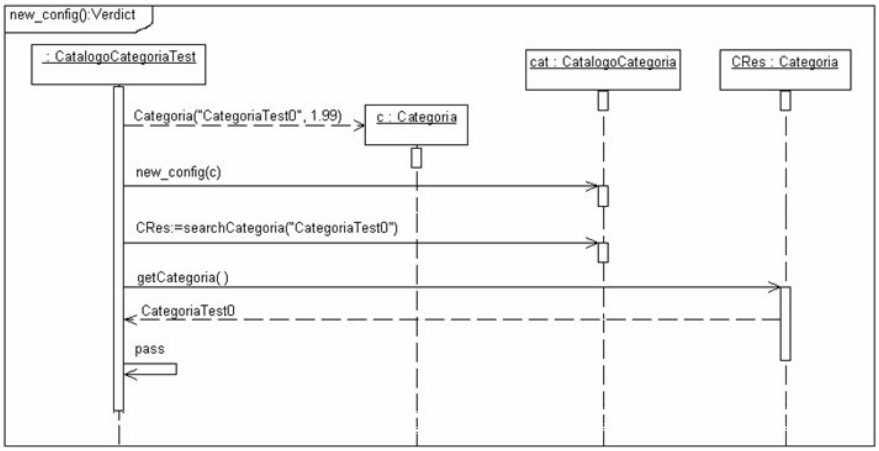

Figura 12. Comportamento para o caso de teste testNewConfig.

Foi especificado um elemento *TestComponent* no projeto de teste U2TP denominado ConexaoBD (Figura 11). Este foi especificado, para conexão com o banco de dados, uma vez que esta classe não existia no projeto de software. Já no projeto de software, a classe Categoria foi estereotipada com <<testcomponent>>, pois é necessária à execução dos casos de teste. A Figura 12 o uso do componente de Categoria no comportamento de um caso de teste.

O *driver* Junit gerado para os testes modelados com o U2TP é mostrado na Figura 13.(a). Também foram gerados os *stubs* de teste referente às classes Categoria e ConexaoDB, juntamente com seus métodos e atributos.

```
(1)package tmm.test;
(2)import tmm.bdi.CatalogoCategoria;
(3)import tmm.entities.Categoria;
(4)import junit.framework.TestCase;

(5)public class CatalogoCategoriaTest extends TestCase{

(6)    private CatalogoCategoria cat;

(7)    protected void setUp() throws Exception {
(8)          super.setUp();
(9)           cat = new CatalogoCategoria( );
(10)          cbd = new conecta( );
(11)   }

(12)   protected void tearDown() throws Exception {
(13)         super.tearDownn();
(14)          cat = null;
(15)   }

(16)   public void testNew_config(){
(17)          Categoria c = new Categoria("CategoriaTest0", 1.99);
(18)          cat.new_conf(c);
(19)         CatalogoCategoria CRes = cat.searchCategoriaString("CategoriaTest0");
(20)          assertEquals(0, CRes.getCategoria( ),"CategoriaTest0");
(21)   }

(22)   public void testDel_config(){
(23)          Categoria c = new Categoria("CategoriaTest1", 1.99);
(24)          cat.new_conf(c);
(25)          cat.del_conf(c);
(26)          CatalogoCategoria CRes = cat.searchCategoriaString("CategoriaTest1");
(27)          assertEquals(0, CRes,null);
(28)   }

(29)   public void testEdit_config(){
(30)          Categoria c = new Categoria("CategoriaTest0", 2.99);
(31)          cat.new_conf (c);
(32)          c.setPreco(1.0);
(33)          cat.edit_conf (c);
(34)          CatalogoCategoria CRes = cat.searchCategoriaString("CategoriaTest0");
(35)          assertTrue(CRes.getPreco( ) == 1.0);
(36)   }

(37)   public void testSearchCategoriaString(){
(38)          Categoria c = new Categoria("CategoriarTest0", 1.99);
(39)          cat.new_conf(c);
(41)          CatalogoCategoria CRes = cat.searchCategoriaString("CategoriaTest0");
(42)          String nomeCategoria = CRes.getCategoria( );
(42)          assertEquals(0, c.getCategoria( ),"nomeCategoria");
(43)}  }                                                                    (a)
```

```
(17)  public void testNew_config() {
(18)          Categoria c = new Categoria("CategoriaTest0", 1.99);
(19)          cat.new_config(c);
(20)          String res = "CategoriaTest0";
(21)          Categoria cRes = cat.searchCategoria("CategoriaTest0");
(22)          assertEquals("", cRes.getCategoria(), res);
(23)   }

(24)  public void testDel_config() {//funcionoiu
(25)          Categoria c = new Categoria("CategoriaTest1", 1.99);
(26)          cat.new_config(c);
(27)          cat.del_config(c);
(28)          Categoria cRes = cat.searchCategoria("CategoriaTest1");
(29)          assertNull("E null??", cRes);
(30)   }                                                                   (b)
```

Figura 13. Driver de teste gerado para classe CatalogoCategoria.

6.1. Resultados

Os resultados deste estudo de caso foram bastante satisfatórios, pois foi gerado 100% do código necessário para os *drivers* e *stubs*. O estudo de caso foi bastante abrangente, pois explorou todos os elementos propostos nos modelos, exercitando assim diferentes aspectos dos algoritmos desenvolvidos.

Outro aspecto importante está relacionado à qualidade do código gerado. A Figura 13.(a) mostra o *driver* gerado automaticamente, enquanto que a Figura 13.(b) mostra uma porção do código produzido pelo testador referente a dois casos de teste , a saber, testNew_config e testDel_config. Observa-se que o código produzido automaticamente é equivalente em termos de funcionalidade, mas levemente menor.

A principal vantagem deste trabalho está na qualidade da documentação dos artefatos de teste. Além de ser uma documentação padronizada, visual e independente de linguagem de programação, ela facilita a manutenção dos testes, bem como testes de regressão. Em comparação, a única documentação produzida pelo testador neste estudo de caso era o próprio código-fonte, o que dificulta a manutenção dos testes, e seu reaproveitamento em caso de implementação do sistema em outras linguagens de programação. Outro fator é que a codificação é fortemente dependente da habilidade e experiência do programador do *driver*, o que muitas vezes pode resultar em um código fonte de baixa qualidade.

Observou-se ainda uma pequena redução do *workload* para este estudo de caso. O testador gerou a partir da IDE Eclipse o esqueleto de cada *driver*, e dispunha-se do tempo gasto por ele para programar o comportamento dos casos de teste de cada *driver* . No total, o testador despendeu cerca de 70 minutos na codificação dos 3 *drivers*. Para razões de comparação, contabilizou-se o tempo para modelagem dos mesmos casos de teste com o U2TP, partindo do diagrama de classes e da especificação dos casos de uso, o que totalizou menos de 60 minutos. Portanto, neste estudo a redução de tempo foi de aproximadamente 15%. Contudo, devem ser realizados mais experimentos para saber se este tipo de ganho é generalizável, em que proporção, e sob quais condições.

7. Trabalhos Relacionados

A geração de *driver*s de teste para JUnit é proposta em vários artigos. Entretanto, nenhuma abordagem propõe a geração totalmente automatizada de *driver*s, incluindo o comportamento dos casos de teste, tampouco a geração dos *stubs* de teste. Por exemplo, algumas IDEs como *Eclipse* e *NetBeans* possibilitam gerar automaticamente a estrutura do *driver* de teste para JUnit, porém é necessário completar esta com as variáveis, importações de pacotes e código referente ao comportamento dos casos de teste. O mesmo dá-se para o *addin* TestExpert [IBM, 2005] da ferramenta Rational Rose.

Outras abordagem geram parcialmente *drivers* e *stubs* de teste para JUnit a partir de modelos UML. Em [Wittevrongel e Maurer, 2001], uma ferramenta denominada *Scensor* gera *drivers* de teste para JUnit a partir de diagramas de seqüência. Entretanto, esta abordagem não adota os conceitos do U2TP e também não gera *stubs* de teste. Já em [Fraikin e Leonhardt, 2002] também é proposta a geração automatizada de código de teste para aplicações Java, através de uma ferramenta denominada *Seditec*. Esta abordagem também usa diagramas de seqüência para geração de testes automatizados, incluindo *stubs* de teste. Porém, o código gerado é específico da ferramenta *Seditec*, de uso muito mais restrito que o JUnit. Duas outras limitações desta ferramenta são: a) não considera o U2TP; e b) os *stubs* de teste gerados estão relacionados com as pré-condições para a execução dos testes, não considerando a geração de *stubs* a partir do relacionamento com outras classes, como no presente trabalho.

Outras abordagens propõem a geração de casos de teste para JUnit a partir de notações que são abstrações de linguagens de programação [Cheon e Leavens, 2002] [Zongyuan et al., 2004]. Nestas abordagens é necessário codificar os casos de teste usando as linguagens JML e AspectJ, respectivamente, não permitindo beneficiar das vantagens preconizadas na adoção do U2TP para especificação de teste.

8. Considerações Finais

Este trabalho propôs uma abordagem para geração automatizada de *drivers* e *stubs* de teste para JUnit a partir de especificações de testes modeladas com o U2TP. Com a adoção do U2TP, é possível especificar testes em diferentes níveis usando uma linguagem visual e padronizada. Desta forma, problemas referentes à falta de documentação, entendimento das especificações, bem como dependência de linguagem de programação são resolvidos.

Em relação aos trabalhos relacionados, as principais contribuições desta proposta são: a) considerar especificações padronizadas em U2TP como entrada para a geração

automática; b) gerar *drivers*, incluindo o código para os casos de teste; c) gerar *stubs*, considerando tanto a especificação explícita quanto implícita destes; d) não necessitar de intervenção do usuário neste processo. Além disto, o código gerado é padronizado, não sendo dependente de habilidade, estilo ou experiência do codificador do *driver*.

Entre as principais limitações deste trabalho estão: a) dependência dos algoritmos desenvolvidos do *parser* DOM e do código XMI produzido pela ferramenta Rational Rose; e b) não considerar o grupo de Dados de Teste do U2TP.

Como trabalhos futuros, podem ser citados, entre outros: a) a extensão desta proposta para outras ferramentas da família xUnit (e.g. cppUnit, n.Unit); b) a consideração dos conceitos do U2TP para Dados de Teste; c) a extensão da proposta para outros níveis de teste, com a geração de código correspondente a ferramentas de teste próprias a estes níveis, etc.

9. Referências

Beck, K (2003). "Test-Driven Development". Addison-Wesley, 220p.

Burnstein, I (2003). "Practical software testing: a process-oriented approach". Springer-Verlag, 709p.

Cheon, Y.; Leavens, G. T (2002). "A Simple and practical approach to unit testing: the JML and JUnit way". In: Proceedings of 16th European Conference Object-Oriented Programming (ECOOP), 231-255p.

Dai, J. Grabowski, A. R (2003). "The UML 2.0 Testing Profile and its Relation to TTCN-3". In: 15th IFIP International Conference on Testing of Communicating Systems (TestCom), pp. 79-94.

Dai, Z.R., Grabowski, J., Neukirchen, H., Pals, H (2004). "From Design to Test with UML". In: 16th IFIP International Conference on Testing of Communicating Systems (TestCom), pp. 33-49.

Eclipse (2005). "Eclipse". http://www.eclipse.org, May.

Fraikin, F.; Leonhardt, T (2002). "SeDiTeC – Testing Based on Sequence Diagrams." In: 17th IEEE International Conference on Automated Software Engineering (ASE), pp. 262-266.

IBM (2005). Test Expert Rose AddIn. http://www-128.ibm.com/developerworks/rational/content/03July/2500/2834/Rose/rational_rose.html, June.

NetBeans (2005). "NetBeans". http://www.netbeans.org, April.

OMG (2004). "UML 2.0 Testing Profile Specification". Technical Report, OMG. http://www.omg.org/docs/ptc/04-04-02.pdf, June 2004, 114p.

Wittevrongel, J.; Maurer, F (2001). "SCENTOR: Scenario-Based Testing of E-Business Applications". In: 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), pp. 41-48.

Zongyuan, Y.; Guoqing, X; Haitao, H; Qian, C; Ling, C; Fengbin, X (2004). "JAOUT: Automated Generation of Aspect-Oriented Unit Test". In: 11th Asia-Pacific Software Engineering Conference (APSEC), pp. 374-381.