Pontifícia Universidade Católica do Rio Grande do Sul

Faculdade de Informática

Programa de Pós-Graduação em Ciência da Computação

Geração Automatizada de *Drivers* e *Stubs* de Teste para JUnit a partir de Especificações U2TP

Luciano Bathaglini Biasi

Orientadora: Profª. Drª. Karin Becker

Dissertação apresentada como requisito parcial à obtenção do grau de Mestre em Ciência da Computação.

Porto Alegre

Janeiro, 2006.

Dados Internacionais de Catalogação na Publicação (CIP)

B579g Biasi, Luciano Bathaglini
Geração automatizada de Driver e Stubs de teste para JUnit a partir de especificações U2TP / Luciano Bathaglini Biais. – Porto Alegre, 2006.
153 f. : il.

Dissertação (Mestrado) – Fac. de Informática, PUCRS, 2006.

Orientador: Profª. Drª. Karin Becker.

1. Engenharia de Software. 2. Software – Avaliação. 3. JUnit (Informática). 4. UML (Informática). I. Becker, Karin. II. Título.

CDD 005.1

Ficha Catalográfica elaborada pelo Setor de Processamento Técnico da BC-PUCRS

PUCRS
Campus Central
Av. Ipiranga, 6681 - prédio 16 - CEP 90619-900
Porto Alegre - RS - Brasil
Fone: +55 (51) 3320-3544 - Fax: +55 (51) 3320-3548
Email: bceadm@pucrs.br
www.pucrs.br/biblioteca

PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO
Reconhecido pelo Parecer No. 930/98.C.N.E. Homologação Publicada no D.O.U. de 30/12/98.

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada "***Geração Automatizada de Drivers e Stubs de Teste para JUnit a partir de Especificações U2TP***", apresentada por Luciano Bathaglini Biasi, como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, Sistemas de Informação, aprovada em 11/01/2006 pela Comissão Examinadora:

Profa. Dra. Karin Becker – PPGCC/PUCRS
Orientadora

Prof. Dr. Flávio Moreira de Oliveira – FACIN/PUCRS

Prof. Dr. Toacy Cavalcante de Oliveira – PPGCC/PUCRS

Prof. Dr. Marcelo Soares Pimenta – UFRGS

Homologada em 22/11/06, conforme Ata No. 28 pela Comissão Coordenadora.

Prof. Dr. Fernando Luís Dotti
Coordenador.

AV. IPIRANGA, 6681 - PRÉDIO 16/SALA 106
90619-900 - PORTO ALEGRE -RS -BRASIL
TELEFONE: 55 51 33203611 FAX 55 51 3320.3621
http: //www.pucrs.br/inf/pos e-mail: ppgcc@inf.pucrs.br

AGRADECIMENTOS

Primeiramente a Deus.

Agradeço, divido e dedico mais esta conquista aos meus pais, em especial a minha mãe, Asolema.

Agradeço em especial à minha orientadora Profª. Karin Becker por acreditar em mim e por toda sua dedicação, paciência e ensinamentos.

Agradeço ao convênio HP-PUCRS que viabilizou através de auxílio financeiro estes dois anos de estudos.

Aos colegas do mestrado, com quem tive a oportunidade de conviver durante estes dois anos.

Agradeço em particular aos colegas Diego e Fabiano que foram pessoas muito amigas durante o mestrado.

Agradeço aos bares da cidade baixa, onde junto aos meus colegas e amigos pude me divertir nos momentos em que precisava.

Agradeço em especial a minha amiga do coração Inná que sempre esteve presente nos momentos mais difíceis.

A turma de 2004, e aos que se tornaram amigos Rafael, Diego, Peter, Eliana, Taisa, Fabiano e Fábio.

Aos novos amigos da turma de 2005 que conquistei durante esses dois anos de mestrado Márcio, Hugo, Leandro e Gustavo.

Ao corpo docente do PPGCC, pela disposição e interação.

Aos funcionários do PPGCC, pela prestatividade e solicitude.

Por último, porém não menos importante, gostaria de agradecer ao meu time do coração Internacional por tantas alegrias e glórias no de 2005.

RESUMO

Atualmente a área de teste de software tem se tornado fundamental para garantia da qualidade dos produtos desenvolvidos. Dentro do processo de teste, o teste unitário 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, custo e esforço ainda são gastos para codificar os *drivers* e os *stubs* de teste necessários a esta ferramenta, o que muitas vezes inviabiliza o seu uso. Outro problema encontrado no processo de teste unitário é a necessidade de especificação dos casos de teste em uma linguagem de mais alto nível, que seja independente de linguagem de programação. O Perfil de Teste da UML 2.0 (U2TP) resolve este problema, pois permite representar e documentar todos artefatos utilizados no processo de teste. Esta dissertação de mestrado tem por objetivo a geração totalmente automatizada de *drivers* e *stubs* de teste para ferramenta JUnit a partir de especificações de testes modeladas com a U2TP. Um estudo de caso mostrou que os algoritmos propostos geraram corretamente todo código de teste para todos elementos explorados neste trabalho. O estudo de caso teve como principais objetivos a validação correta do código gerado, bem como uma análise quantitativa em relação ao tempo envolvido e número de linhas de código geradas.

ABSTRACT

Testing has become essential to assure the quality of software products. Within the process test, unit test is performed on the smallest funcional part of the software and it aims at discovering defects in these units. JUnit is a unit test tool, which assists developers in the automation of tests and verification of results. However, much time, cost and effort are still spent to codify drivers and stubs, which most of the time jeopardizes its use. Another problem found in the unit test is the need of cases test specification in a higher level language, which is independent of any specific programming language. The Test Profile of UML 2.0 (U2TP) solves this problem, because it allows to represent and document all artifacts used in the test process. This work aims at the fully automated generation of test drivers and *stubs* for JUnit from U2TP test specifications. A case study has shown that the developed algorithms correctly generated all test code, considering all elements explored in this work. The main goal of the case study was the validation of the correctness of the generated code, as well as the quantitative analysis of time consumed and number of code lines generated.

LISTA DE FIGURAS

LISTA DE TABELAS

LISTA DE ABREVIATURAS

CASE	*Computer-Aided Software Engineering*
DTD	*Document Type Data*
IDE	*Integraded Development Environment*
MOF	*Meta Object Facility*
OMG	*Object Management Group*
SQA	*Software Quality Assurance*
TDD	*Testing Driver Development*
UML	*Unified Modeling Language*
U2TP	*UML 2.0 Testing Profile Specification*
XML	*Extensible Markup Language*
XMI	*Extensible Metadata Interchange*
XP	*Extreme Programming*

SUMÁRIO

1 INTRODUÇÃO

Com a crescente demanda de sistemas de software cada vez mais complexos e também com o avanço da tecnologia, o processo de teste tornou-se fundamental para garantia de qualidade dos produtos de software.

Atualmente é necessário integrar o processo de teste cada vez mais cedo dentro das fases de desenvolvimento do software [DAI03]. Dessa forma, falhas de projeto e de implementação podem ser descobertas mais cedo, ajudando a diminuir o custo e o retrabalho. Entretanto, muito tempo ainda é gasto para testar produtos de software, e é nas atividades testes que se concentram os maiores esforços, custos e tempo.

Dentro do primeiro ciclo de desenvolvimento do software, a fase de codificação é freqüentemente responsável por gerar o maior percentual de defeitos em um sistema. Uma forma de reduzir esse percentual é a realização dos testes unitários. Um teste unitário é realizado na menor parte funcional de um software (uma função, uma classe, um procedimento, etc). Posteriormente ao teste unitário é realizado o teste de integração, que visa garantir a integração destas unidades e o teste de sistema, que visa garantir o funcionamento do sistema como um todo. Finalmente tem-se o teste de aceitação que visa garantir o funcionamento correto do sistema em relação aos requisitos do usuário. Para cada nível de teste são gerados casos de teste. Casos de testes servem para executar o software sob algumas condições dado um conjunto de valores de entrada e verificar as saídas correspondentes [BUR03].

A automação de testes vem sendo uma das formas de reduzir o tempo gasto nas atividades de teste. Ela está ganhando muita importância e aceitação no mercado, pois auxilia desenvolvedores a executar e verificar testes automaticamente. Existe uma classificação das ferramentas de teste de acordo com a fase do processo de teste ao qual dão apoio, por exemplo, ao planejamento de teste, projeto de casos de teste, execução de teste, entre outras atividades [LEU98]. Uma série de ferramentas para teste de unidade conhecidas como ferramentas *XUnit,* entre elas *cppUnit* (para linguagem c++) [CPP05], *dUnit* (para linguagem Delphi) [DUN05], *VBUnit* (para linguagem Visual Basic) [VBU05], JUnit (para linguagem Java) [JUN04] e *NUnit* (para linguagem .net) [NUN05], está sendo bastante utilizadas para automatização da execução dos testes. Elas diminuem o tempo para execução dos testes e proporcionam que as saídas, ou seja, os resultados possam ser verificados, garantindo assim maior confiabilidade para o testador. As diferentes ferramentas da família *XUnit* diferem

basicamente na linguagem de programação dos programas que executam. Algumas vantagens dessas ferramentas incluem:

- Verificação automatizada dos resultados e diagnóstico preciso do tipo de erro e/ou defeito;
- O código que coordena a execução do teste, conhecido como *driver*, fica armazenado em um repositório para eventuais modificações no código, favorecendo a manutenção do código.
- Satisfação do programador, uma vez que este tem a sensação de executar e verificar o resultado, resultando em um maior comprometimento do programador com a execução dos testes.

Para utilizar uma dessas ferramentas, é necessário especificar e desenvolver os casos de teste. Esses casos de testes constituem o *driver* de teste usado pela ferramenta. Também pode ser necessário especificar e desenvolver os *stubs* de teste. Um *driver* é uma classe que simula o programa principal do elemento a ser testado, ou seja, faz chamadas ao módulo a ser testado. Um *stub* serve para substituir os módulos que estejam subordinados ao módulo a ser testado. Esses elementos são códigos auxiliares desenvolvidos para dar suporte à unidade a ser testada, são os chamados *Test Harness* [BUR03]. *Test Harness* são trechos de códigos que devem ser escritos para possibilitar o teste do produto final sendo desenvolvido, mas não fazem parte deste.

Autores como [BUR03] e [PRE01] ressaltam que os *Test Harness* representam sobrecarga, já que seu desenvolvimento às vezes demanda tempo, custo e esforço, minimizando assim as vantagens da automação dos testes. Outra problema, é que os *Test Harness* são dependentes de linguagem de programação, ou seja, cada ferramenta tem uma codificação própria desses elementos.

Outra forma de reduzir o tempo e custo gasto nas atividades de teste é o uso de padrões [CAG04]. Entretanto, o processo de teste unitário muitas vezes não segue um padrão que possa ser facilmente seguido. Segundo [BUR03], o processo de teste unitário muitas vezes é executado informalmente pelo próprio desenvolvedor. Isso traz problemas graves como: (i) defeitos encontrados não são registrados; (ii) dificuldade de reuso e (iii) falta de planejamento. Por exemplo, na *Extreme Programming* através do *Test Driver Development* – TDD [BEC03], os *drivers* de testes são feitos antes mesmo do código ser desenvolvido. Isso induz a um código potencialmente livre de erros. Porém, nessa abordagem não há um planejamento das atividades de teste, nem a portabilidade dos *drivers* de teste. Assim, a

maneira como um caso de teste é especificado nessa abordagem é totalmente *ad-hoc*, pois ele é feito pelo próprio programador e sem nenhum planejamento.

Outro problema encontrado no processo de teste unitário, é que muitas vezes apesar de haver um planejamento das atividades de teste, a especificação dos casos de teste não segue um padrão de documentação dos artefatos de teste; também não existe uma linguagem universal para especificação de testes que pode ser usada independentemente do tipo de linguagem em que o sistema é desenvolvido.

Assim, para resolver esses problemas é preciso uma notação padrão para especificação de testes. Recentemente a OMG (*Object Management Group*) disponibilizou a UML 2.0 *Testing Profile Specification*, conhecida como U2TP, uma linguagem gráfica para visualização, especificação, construção e documentação de artefatos de teste para sistemas complexos de software [OMG04A]. A U2TP proporciona que todos artefatos utilizados para testar um sistema possam ser especificados e modelados em UML, auxiliando assim na documentação, entendimento e rastreabilidade dos artefatos de testes. Outra vantagem da U2TP é a definição de um conjunto de conceitos próprios à área de testes que podem ser representados e mapeados para diferentes ferramentas de testes, como por exemplo: JUnit (teste unitário) e TTCN-3 (teste de integração e sistema).

1.1 Exemplo Motivador

Um estudo para avaliar a viabilidade de uso de umas das ferramentas da família *XUnit* descritas anteriormente foi realizado em projeto de uma grande empresa de desenvolvimento de software instalada no Parque Tecnológico da PUCRS. O projeto era composto por três equipes: Projeto, Construção (*Cliente* e *Servidor*) e Teste de Integração. A Figura 1 ilustra os papéis das equipes dentro do projeto e o relacionamento entre as equipes.

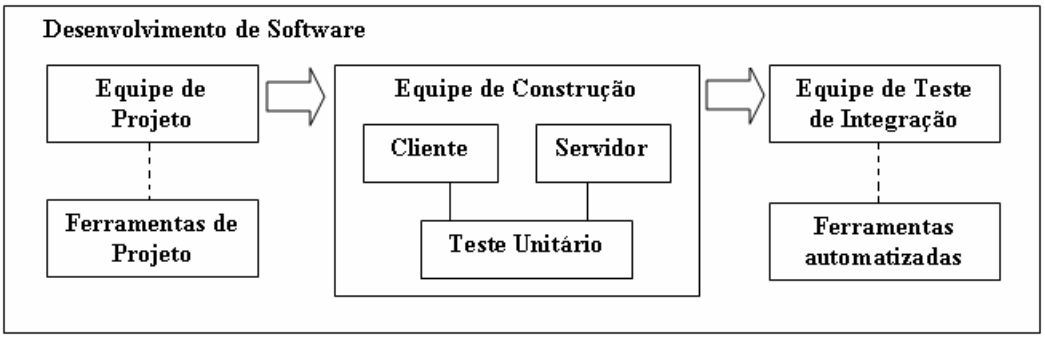


Figura 1 - Contexto de Desenvolvimento de Software.

De acordo com a Figura 1, a equipe de projeto especifica os documentos de *design* para que a equipe de construção (*Cliente* e *Servidor*) desenvolva o software. Esses documentos de *design são* descritos em pseudocódigo.

A equipe de construção *Cliente* desenvolve em linguagem *Visual Basic 6.0* e a equipe de construção *Servidor* em linguagem C++. Antes da codificação, os programadores especificam os casos de teste em nível de unidade baseado no documento de *design*. Dessa forma, uma vez codificado o documento de *design*, os programadores executam os casos de testes especificados anteriormente. A equipe de Integração é responsável por testar e integrar os módulos já testados individualmente.

Um dos problemas vivenciados por este projeto de desenvolvimento de software era o tempo gasto para execução dos casos de testes unitários. Um grande número de defeitos estava sendo descoberto pela equipe de teste de integração. Isso mostrava que os testes unitários não estavam sendo realizados com eficiência. Entre os defeitos encontrados, aproximadamente 70% estavam relacionados à equipe de construção *Cliente* e dentre esse percentual, cerca de 60% dos defeitos era relacionado a erros de tela (layout).

Para tentar minimizar esse problema, foi realizado um estudo de caso para avaliar a viabilidade de utilização de uma ferramenta para teste unitário da família *XUnit*. A ferramenta escolhida foi *VBUnit* (para linguagem de programação *Visual Basic 6.0*).

Depois de dois meses de estudo foram realizados alguns testes e um projeto piloto foi implantado. Os resultados desse estudo mostraram que os *Drivers e Stubs* de teste desenvolvidos para a ferramenta necessitavam de um esforço muito grande para codificação dos mesmos, o que inviabilizou o uso da ferramenta.

Dessa forma, o que se observou foi a necessidade de especificar e gerar *drivers* e *stubs* de teste automaticamente, o que reduziria o esforço e o tempo gasto para a codificação dos mesmos.

1.2 Objetivo do Trabalho

Como forma de minimizar os problemas descritos anteriormente, este trabalho propõe gerar automaticamente *drivers* e *stubs* de teste para ferramenta de teste unitário JUnit a partir da especificação de teste unitário modeladas com o U2TP. Com isso é possível contribuir com o uso da U2TP para especificação de teste em nível de unidade e reduzir o tempo gasto para codificação de *drivers* e *stubs* de teste.

1.2.1 Objetivos Específicos

- Investigar as possibilidades e os requisitos para conversão de uma especificação de teste unitário usando o U2TP na codificação equivalente para a ferramenta JUnit;
- Propor algoritmos para geração de *drivers* e *stubs* para ferramenta JUnit a partir da especificação do U2TP;
- Especificar e implementar um protótipo de um ambiente que dê suporte as estes algoritmos.

1.4 Organização do Trabalho

Este trabalho está organizado em nove capítulos. No capítulo um foi apresentada a introdução do presente trabalho, descrevendo a contextualização do problema bem como a proposta e seus objetivos. Os demais capítulos estão organizados como segue:

- Capítulo 2: Este capítulo apresenta o processo de teste de software, descrevendo os principais termos e conceitos bem com principais atividades e papéis envolvidos neste processo de teste. Também será abordado detalhadamente o processo de teste unitário, descrevendo os principais problemas encontrados ao realizar teste unitário. Também será apresentado como *drivers* e *stubs* de teste são desenvolvidos. Finalizando o capítulo será apresentada a ferramenta JUnit;
- Capítulo 3: Este capítulo apresenta como diagramas da UML podem ser usados para especificação de teste unitário;
- Capítulo 4: Este capítulo apresenta os principais elementos do U2TP. Também será apresentado o metamodelo baseado no MOF para testes e uma metodologia para usar o U2TP. Finalizando o capítulo serão descritos quais elementos são apropriados para teste em nível de unidade;
- Capítulo 5: Este capítulo apresenta os trabalhos relacionados. Estes estão divididos em três grupos: (i) geração de *driver*s a partir de IDEs; (ii) geração de casos de teste para JUnit e (iii) geração de *driver*s de teste a partir de modelos UML. No final do capítulo é apresentada uma tabela comparativa entre as abordagens;
- Capítulo 6: Este capítulo apresenta a descrição da proposta do presente trabalho. Serão apresentados os pressupostos para modelagem dos testes assumidos para converter os elementos na codificação equivalente a ferramenta JUnit. Também serão apresentados dois modelos de mapeamento definidos para geração do código: Modelo de Mapeamento U2TP-JUnit e Modelo de Mapeamento U2TP-XMI;

- Capítulo 7: Este capítulo descreve o funcionamento do Extrator e Gerador do Código. Serão descritos os algoritmos de mapeamento usados para capturar os elementos para classes bem como o gerador de código usado para gerar esses elementos. Finalizando o capítulo, será apresentado o protótipo desenvolvido;
- Capítulo 8: Este capítulo apresenta o estudo de caso realizado para geração automatizada dos *Drivers* e *Stubs* de teste. O objetivo do estudo de caso é validar a geração correta do código gerado, bem como comparar o tempo gasto na especificação do teste e também o número de linhas de código geradas;
- Capítulo 9: Este capítulo apresenta as considerações finais e os trabalhos futuros.

2 TESTE DE SOFTWARE

O processo de desenvolvimento de software é descrito como uma série de fases, procedimentos e passos que resultam na produção de um software. Dentro do processo de desenvolvimento de software estão outros processos, incluindo o Teste [BUR03].

Teste de software é o processo de executar o software de uma maneira controlada com o objetivo de avaliar se o mesmo se comporta conforme o especificado. Entretanto, a dificuldade em testar um sistema de software é caracterizada por alguns pontos importantes tais como [CRE04];

- o teste de software é um processo caro;
- existe uma falta de conhecimento sobre a relação custo/benefício do teste;
- há falta de profissionais especializados na área de teste;
- dificuldades em implantar um processo de teste;
- não existe um uso adequado do procedimento de teste;
- não existe um planejamento adequado das atividades de teste; e
- infelizmente existe a preocupação com as atividades de teste somente na fase final do projeto.

Assim, para que o processo de teste possa efetivamente fornecer um benefício em termos de custo, esforço e tempo, é importante que as atividades de teste sejam bem planejadas.

2.1 Atividades do Processo de Teste de Software

O processo de teste de software é composto por atividades que têm por objetivo executar um programa a fim de revelar suas falhas e avaliar sua qualidade. Ele é muito importante em toda etapa de desenvolvimento do software e, se bem aplicado, proporciona um aumento da qualidade do produto desenvolvido [PRE95].

Não existe na literatura um consenso sobre quais são as atividades chave do processo de teste. A norma 829 da IEEE [IEE98] define um conjunto de documentos (artefatos) para as atividades de teste de software. A norma separa as atividades de teste em três etapas: (1) Preparação do Teste; (2) Execução do Teste e (3) Registro do Teste.

Alguns autores também identificam tais etapas, como em [PET02], Assim, as principais atividades do processo de teste de software são descritas abaixo:

- *Planejamento*: nesta atividade é gerado um plano de teste que deve conter informações sobre a abrangência, a abordagem, os recursos e a programação da atividade de teste.
- *Projeto de casos de teste:* nesta atividade são definidos os casos de teste usados para testar o sistema. Devem ser identificadas as características específicas a serem testadas pelo projeto.
- Procedimento de Teste: nesta atividade são identificadas todas as etapas necessárias para operar o sistema e exercitar os casos de testes especificados para implementar o projeto de teste já definido;
- *Execução dos Testes*: nesta atividade os testes são executados. A execução dos testes começa em nível de unidade até integração e sistema.
- *Avaliação e/ou Resultados dos testes*: nesta atividade são relatados todos os resultados e destacadas as discrepâncias encontradas nos resultados.

Na elaboração do planejamento dos testes uma das etapas é decidir qual a estratégia de teste será usada. A estratégia de teste compreende a definição dos seguintes itens: (i) o nível de teste, ou seja, a definição da fase do desenvolvimento do software em que o teste será aplicado; (ii) a técnica de teste a ser utilizada; (iii) o critério de teste a ser adotado e (iv) o tipo de teste a ser aplicado no software [CRE04]. A Figura 2, extraída de [CRE04], ilustra os relacionamentos entre os níveis, tipos e técnicas de teste de software.

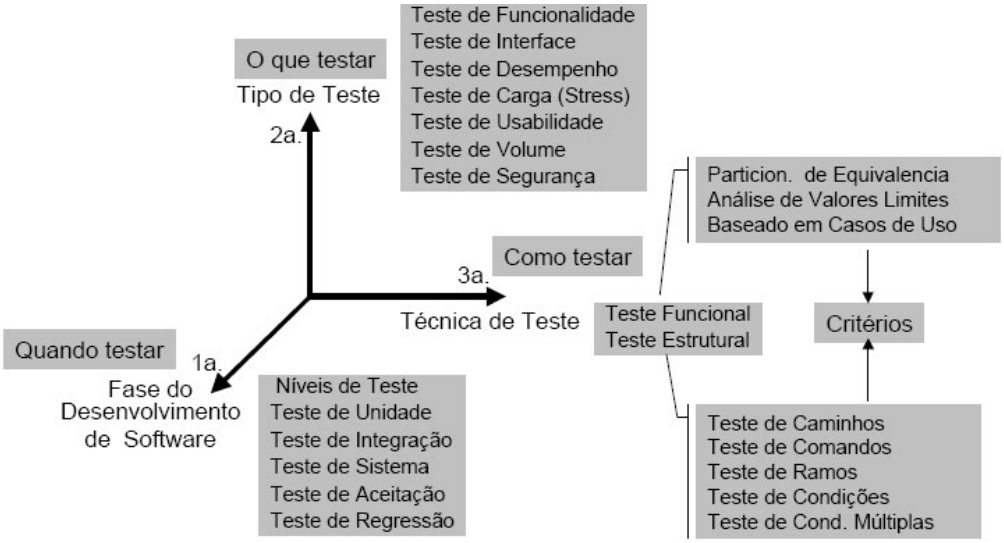


Figura 2 - Relação entre níveis, tipos e técnicas de teste.

O nível de teste depende da fase do desenvolvimento do software em que o teste poderá ser aplicado. Os três principais níveis são: Unidade, discutido detalhadamente na seção 2.3, Integração e Sistema.

A técnica de teste direciona a escolha de critérios para projetos de casos de teste, que, ao serem executados, vão exercitar os elementos requeridos pela abordagem do teste [CRE04]. As principais técnicas são: Testes Estruturais, também conhecidos como Testes de Caixa Branca e Testes Funcionais, também conhecidos como Testes de Caixa Preta. Os testes estruturais concentram-se no código fonte do sistema sob teste, tem por objetivo testar o programa internamente. Os testes funcionais concentram-se nos requisitos do software a ser testado, tem por objetivo testar as funcionalidades do sistema.

O critério de teste serve para orientar o testador na geração dos casos de teste. São definidos de acordo com a técnica de teste usada, por exemplo, se a técnica escolhida for Caixa Branca, alguns critérios para escolha dos casos de teste são: teste de caminho básico, testes de condições, loops, entre outros. Já se a técnica escolhida for Caixa Preta, alguns critérios para escolha dos casos de teste são: Análise do valor Limite, Classes de Equivalência, entre outros.

Os tipos de teste referem-se às características do software que podem ser testadas, tais como: Teste de Funcionalidade, Teste de Interface, Teste de Desempenho; Teste de Carga (*Stress*) entre outros.

2.2 Papéis e Responsabilidades no Processo de Teste

Dentro do processo de teste de software, algumas atividades são delegadas a pessoas com capacidade específicas para executá-las. Não existe um consenso na literatura sobre os possíveis papéis dentro do processo de teste e também sobre qual ou quais atividades cada papel é responsável. Entretanto, os papéis e responsabilidades comuns encontrados em algumas literaturas conforme [BUR03] e [MOL03] são:

- *Gerente de Teste:* é a pessoa central que trata de todos os assuntos relacionados a teste de software. Um gerente de teste é normalmente responsável pela definição da política de teste usada na organização, incluindo: planejamento de testes, documentação dos testes, controle e monitoramento dos testes, aquisição de ferramentas de teste, participação em inspeções, revisões do trabalho de teste, entre outros;
- *Engenheiro de Teste:* também conhecido como projetista de teste, é responsável principalmente por especificar e projetar os casos de teste.

- *Testador:* é aquele que executa o teste. Em nível de unidade o testador é freqüentemente o próprio programador.

2.3 O Processo de Teste de Unidade

O Teste Unitário ou Teste de Unidade é a primeira fase do processo de teste de software, e é considerado um dos fundamentos principais do desenvolvimento de software. É nessa fase que são descobertos o maior número de defeitos. Ele é executado para melhorar a qualidade geral do software que passa pela equipe de teste de integração, sistema e depois para o cliente.

Segundo [PRE95], um teste de unidade é realizado na menor parte funcional de um software (uma função, uma classe, um procedimento, etc...). O objetivo do teste unitário é comparar funcionalidades de um determinado módulo com aquilo que foi descrito na especificação do mesmo, de forma a assegurar que o mesmo não o contradiga [PET02].

Para [BUR03] uma unidade é considerada o menor componente de software que pode ser testado. Essa unidade pode ser caracterizada de vários modos, considerando um sistema orientado a objetos, uma unidade: (i) executa uma única função coesa; (ii) pode ser compilada separadamente e (iii) contém código que se ajusta em uma única página ou tela. A Figura 3, adaptada de [BUR03], ilustra algumas dessas unidades:

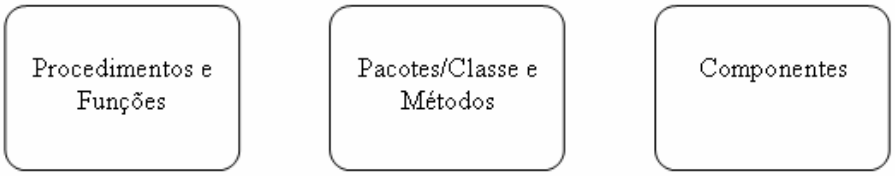


Figura 3 - Unidades de Software.

Uma unidade é tradicionalmente vista como uma função ou um procedimento considerando uma linguagem de programa procedural. Em sistemas orientados a objetos, um pacote (com uma ou mais classes), classes, objetos ou métodos são considerados unidades individuais. Uma unidade também pode ser considerada um componente COTS - *Component-of-the-shelf*, os chamados componentes de prateleira (disponíveis externamente), como também os componentes *in-house* (disponíveis internamente). Dessa forma, uma unidade é definida dentro do projeto de software que será desenvolvido. Cabe ao Engenheiro de Teste decidir qual será o nível de abstração da unidade a ser testada.

Os casos de teste em nível de unidade são normalmente feitos pela pessoa que escreveu o código. O código fonte está quase sempre disponível no teste de unidade (exceto nos componentes COTS) e os que desenvolveram estão familiarizados com os detalhes do código; conseqüentemente, eles podem fazer um uso construtivo destas informações [HEI01]. Porém, um problema comum é que o programador é quem tem que especificar e projetar os casos de teste, mas muitas vezes ele não tem o mesmo conhecimento que um engenheiro de teste. Assim, essa especificação muitas vezes é feita de maneira ad-hoc sendo que freqüentemente não há uma documentação dos casos de teste. Segundo [BUR03] testes unitários devem ser planejados. Isso evita que programadores especifiquem e desenvolvam os testes.

Para preparar um teste unitário é preciso executar as seguintes atividades [BUR03]

I. Planejar a abordagem geral para o teste unitário;
II. Especificar e projetar os casos de teste;
III. Definir os relacionamentos entre os testes;
IV. Preparar o código auxiliar necessário para o teste de unidade (*Drivers* e *Stubs*).

A atividade que pode ser automatizada dentre as descritas acima é a criação de *Drivers* e *Stubs* de teste. O desenvolvimento de *Drivers* e *Stubs* consome recursos, tempo, custo e esforço. Esses elementos podem ser desenvolvidos em vários níveis de funcionalidades. Por exemplo, um *Driver* pode ter as seguintes funções [BUR03]:

I. Fazer chamadas a uma unidade sob teste;
II. Passar parâmetros para uma unidade sob teste;
III. Mostrar parâmetros e
IV. Mostrar resultados (parâmetros de saída).

Um *Stub* também pode exibir diferentes níveis de funcionalidade [BUR03]:

I. Mostrar uma mensagem que foi chamada por uma unidade sob teste;
II. Mostrar parâmetros de entrada passados por uma unidade sob teste e
III. Passar valores para uma unidade sob teste.

A Figura 4, adaptada de [BUR03], ilustra como *Drivers* e *Stubs* são relacionados em testes unitários.

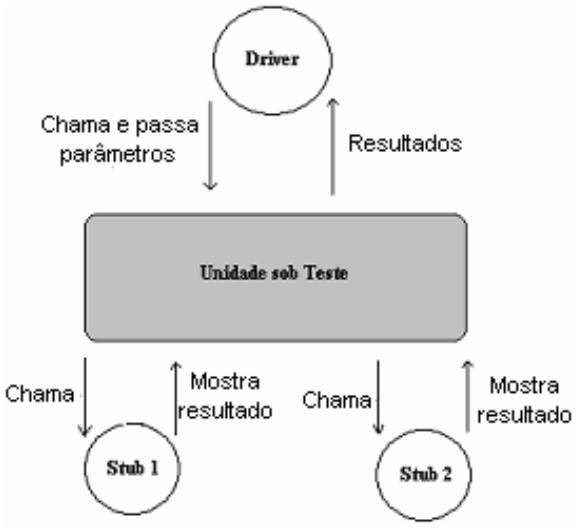


Figura 4 - *Driver* e *Stubs*.

Drivers e *Stubs* são desenvolvidos como procedimentos ou funções quando considerando um sistema baseado em linguagem procedural. Já considerando um sistema orientado a objetos, *Drivers* e *Stubs* freqüentemente oferecem meios de projetar e implementar classes especiais para executar as tarefas de teste requeridas. No exemplo da Figura 4, um *Driver*, considerando um sistema baseado em linguagem procedural, pode ser desenvolvido como um único procedimento ou um módulo principal para chamar uma unidade sob teste. Considerando um sistema orientado a objetos, um *Driver* consiste de classes de teste que exercitam as classes a serem testadas e mostram os resultados relevantes [BUR03].

Para a realização dos testes unitários, algumas abordagens como o *Test Driven Development* – TDD a qual segue os princípios da *Extreme Programming,* vem sendo muito usada em sistemas orientados a objetos [BEC98]. Nessa abordagem, os casos de teste os quais constituem o *driver* de teste são escritos antes do código a ser testado, e o desenvolvedor escreve o código de teste para todo código produzido.

Porém, como descrito em [BUR03], um problema muito comum encontrado no processo de teste unitário é a forma como ele é executado, ou seja, informalmente. Quando é feito pelo próprio programador, como no TDD, o próprio programador é responsável por projetar e codificar os casos de teste, sendo que estas duas atividades são realizadas simultaneamente e não havendo, portanto, uma especificação dos casos de teste.

Outro problema encontrado no processo de teste unitário é estabelecer uma maneira padrão para especificação dos casos de teste que possa ser facilmente usada independentemente do tipo de sistema que será desenvolvido.

Por exemplo, atualmente com o uso de uma nova notação para especificação e representação de teste de software, o U2TP, todos os artefatos produzidos no processo de teste unitário podem ser modelados e documentados, proporcionando assim uma notação padrão que pode ser usada independente do tipo de linguagem de programação que o sistema será desenvolvido. O Capítulo 4 abordará em detalhes o U2TP.

2.4 Ferramentas para Teste de Unidade

Atualmente uma gama de ferramentas para teste unitário conhecidas como família *XUnit* estão sendo bastante usadas [EBE05]. Elas estão ganhando bastante aceitação no mercado por serem fáceis de usar e por serem ferramentas de código livre.

Dentre estas ferramentas, umas das ferramentas de maior popularidade e atualmente uma das mais usadas é o JUnit. Os principais motivos de sua popularidade e uso são:

- uma ferramenta para linguagem JAVA que atualmente é uma linguagem bastante usada no desenvolvimento de sistemas orientados a objetos;
- muito usada na abordagem TDD;
- fácil de usar;
- possibilidade de integração com diversas IDE - *Integraded Development Evironment* (ambiente de desenvolvimento integrado), como por exempplo: *Eclipse*, *NetBeans* e *BlueJ*;

Por esses motivos e também pela possibilidade de mapeamento com alguns conceitos dao U2TP essa ferramenta foi escolhida para a proposta do presente trabalho. A próxima seção descreve detalhadamente o framework JUnit.

2.4.1 Framework JUnit

Junit é um *framework* para teste de unidade usado por programadores que desenvolvem aplicações em linguagem Java. A Figura 5, adaptada de [OMG04A], ilustra o diagrama de classes do *Framework* JUnit.

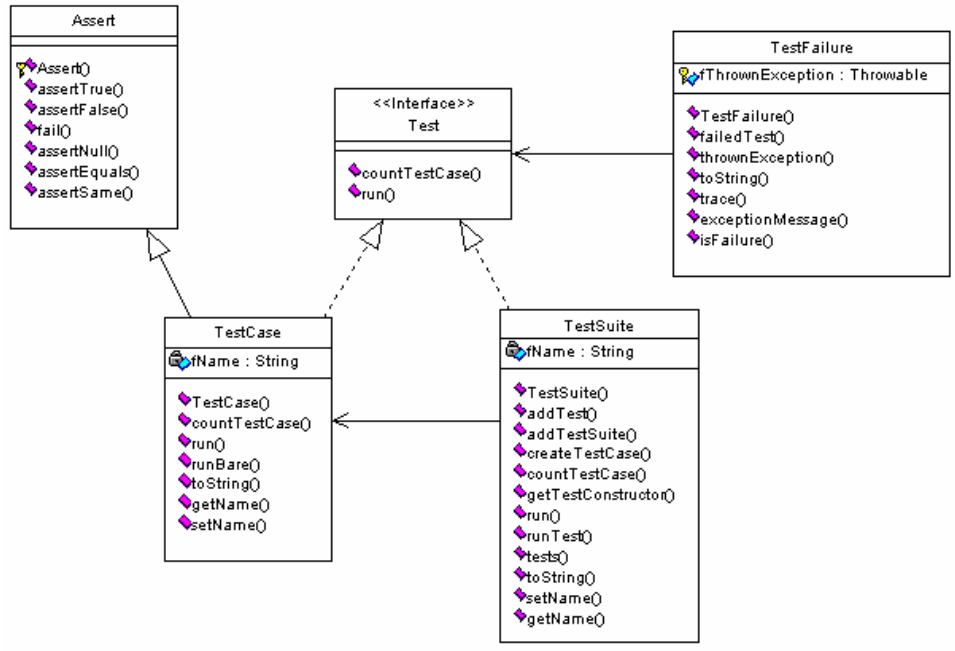


Figura 5 - Diagrama de classes do *framework* JUnit.

O JUnit possui diversos pacotes: *framework* como o pacote básico, *runner* que serve para representar algumas classes abstratas e para a execução dos testes, *textui* e *swingui* para apresentar ao usuário diferentes tipos de interfaces e *extensions* para algumas contribuições úteis ao *framework*. Apesar de fácil de usar e de ser bastante popular, o JUnit possui um código bastante complexo. Ele foi construído baseado em diversos padrões de projeto, entre eles (*Command*, *Template Method*, *Collecting Parameter*, *Observer* e *Adapter*).

As duas principais classes do JUnit são: *TestCase* e *TestSuite*. Essas classes implementam a interface *Test* que contém o método *run* responsável pela execução dos testes. A classe *TestCase* é responsável por executar um caso de teste, que corresponde a um método testado no JUnit. A classe *TestSuite* é responsável por executar um conjunto de casos de teste ou classes de teste. Toda vez que um caso de teste é executado no JUnit ele:

1. Executa o método *SetUp*(): serve para inicializar variáveis e criar objetos.
2. Executa o próprio caso de teste: serve testar um método de uma classe.
3. Executa o método *TearDown*(): serve para finalizar corretamente o método testado.

O JUnit repete esse procedimento usando reflexão até encontrar todos os métodos de teste.

Ao executar um caso de teste usando o JUnit, todas as suas expectativas são validadas através dos métodos da classe *Assert*. O JUnit registra todo o caminho percorrido durante as falhas obtidas nos métodos da classe e relata os resultados após a execução de todos os testes.

Existem duas maneiras através das quais um teste pode falhar no JUnit. Ou o teste produz o resultado errado ou lança uma exceção não esperada (por exemplo, *IndexOutOfBoundsException*). O JUnit denomina o primeiro caso como "*failure*", ou falha, e o segundo de "*error*", ou erro. Dessa forma um teste no JUnit pode apresentar como resultado três valores: *pass*, *fail* ou *error*. Assim, a classe *TestFailure* reporta todos os tipos de erros ou falhas que podem ser geradas depois da execução de um teste. Alguns dos métodos de assertivas mais usados do JUnit são mostrados abaixo:

- *assertTrue*(String errorMessage, boolean booleanExpression): Verifica se a expressão booleana é verdadeira.
- *assertFalse*(String errorMessage, boolean booleanExpression): verifica se a expressão booleana é falsa.
- *assertEquals*(String errorMessage, Object a, Object b): Verifica se o objeto "a" é igual ao objeto "b".
- *assertNull*(String errorMessage, Object o): Verifica se o objeto é nulo.

O JUnit foi adaptado a uma variedade de IDEs, que favorecem a escrita do código para uma melhor interpretação dos procedimentos de teste. As IDE's mais conhecidas com suporte ao JUnit são Eclipse [ECL05] e NetBeans [NET05].

Para usar a ferramenta JUnit deve-se:

- Crie uma classe que estenda junit.framework.TestCase
  - public class SuaClasseTest extends junit.framework.TestCase {...}
- Para cada método xxx(args) a ser testado definir um método public void testXxx() na classe de teste
  - SuaClasse:
    - `public boolean equals( Object o) { ... }`
  - SuaClasseTest
    - `public void testEquals() {...}`
- Sobrescreva o método *setUp*() para a inicialização comum a todos os métodos.

- Sobrescreva o método tearDown() se necessário, para liberar recursos como streams, apagar arquivos, etc.

2.5 Considerações Finais

Este capítulo abordou o processo de teste de software de um modo geral, descrevendo os principais termos encontrados na área de teste de software, as principais atividades, bem como os papéis envolvidos. Também foram abordados mais detalhadamente o processo de teste unitário e a ferramenta JUnit.

O processo de teste de software é composto por uma série de atividades que, em conjunto, têm por objetivo a detecção de defeitos e o aumento da qualidade do software. Dentre os níveis de teste, o teste de unidade é considerado um dos mais importante, pois é responsável por localizar e corrigir os defeitos já no início do ciclo de desenvolvimento do software. Apesar de tal importância, muitas vezes o teste de unidade não tem um planejamento adequado [BUR03].

Em algumas abordagens como o TDD, o teste é realizado antes do código ser desenvolvido e isso induz a um código potencialmente livre de erros. Porém, não existe um planejamento das atividades de teste e as especificações dos testes são feitas de maneira *ad-hoc*. Outro problema é a necessidade de estabelecer uma maneira para especificação de testes unitários utilizando um modelo como a UML para especificações de testes. Com isso é possível documentar os artefatos de testes, ajudando tanto na documentação como no entendimento desses artefatos. Também é possível abstrair os testes em um maior nível, minimizando assim a dependência dos testes em relação a linguagens de programação.

Uma gama de ferramentas para testes automatizados conhecidas como ferramentas da família *XUnit* têm contribuído para reduzir o tempo e esforço gasto na execução de testes, pois permitem a execução e verificação automatizada dos testes. Porém, a geração automatizada dos *drivers* e *stubs* de teste a partir de especificações de testes não são suportados por essas ferramentas, ou seja, o desenvolvimento desses elementos, ainda requer muito tempo, custo e esforço por parte do desenvolvedor, pois o mesmo tem que desenvolver (codificar) esses elementos.

Uma forma de minimizar esses problemas é possibilitar que o Engenheiro de Teste fique responsável por especificar os casos de teste. Isso pode ser realizado usando uma notação padrão para especificação de teste de software, como por exemplo, UML, mais especificamente o U2TP. Uma vez especificado um projeto de teste, o testador, que em nível de unidade corresponde ao desenvolvedor, fica responsável por executar estes testes.

Assim, para este trabalho dois papéis são definidos no processo de teste unitário, um Engenheiro de Teste e um Testador. Ao optar pelo uso de ferramentas como *XUnit*, a questão é minimizar o custo e esforço necessário para a geração de *drivers* e *stubs*, o que é abordado neste trabalho através da geração automatizada.

3 DIAGRAMAS DE UML E TESTE DE UNIDADE

Segundo [GRO03], as técnicas de teste de software tradicionais, como as técnicas de caixa branca (grafo de fluxo de controle, caminhos independentes, etc...) e técnicas de caixa preta (análise do valor limite, classes de equivalência, etc...) podem ser derivadas a partir de diagramas UML. A combinação de modelagem e de testes pode ser vista sobre duas perspectivas:

- *Teste Baseado em Modelos:* consiste no desenvolvimento de artefatos de teste a partir de modelos UML existentes. Em modelos UML são encontradas informações sobre testes. Modelos UML fornecem a primeira informação para o desenvolvimento de casos de teste e suítes de teste usadas para testar um sistema.
- *Modelagem de Teste:* consiste no desenvolvimento de artefatos de teste com a UML. O desenvolvimento do teste é baseado no uso da UML e segue os mesmos princípios fundamentais, como qualquer outra atividade dentro do processo de desenvolvimento do software.

A segunda perspectiva é o foco deste trabalho, onde os conceitos da U2TP são utilizados para modelar e especificar teste em nível de unidade. Porém, a primeira perspectiva é fundamental para dar suporte ao desenvolvimento, modelagem e especificação de teste com o U2TP, pois é a partir de diagramas UML existentes (projetos de software existentes) que são especificadas informações sobre o que deve ou não ser testado.

Assim, o restante desse capítulo aborda como alguns diagramas da UML podem auxiliar a abstrair informações sobre teste em nível de unidade. Primeiramente serão apresentados alguns diagramas da UML, descrevendo seus conceitos e sua utilidade dentro do processo de desenvolvimento de software. Em seguida, será abordado como estes diagramas podem ser utilizados para especificar informações sobre testes em nível de unidade. Os exemplos descritos nas próximas seções são baseados em um Sistema de Ponto de Venda, extraído de [LAR00].

3.1 Diagrama de Caso de Uso

Diagramas de caso de uso são fundamentais para a fase inicial de análise em um sistema. São através dos casos de uso que são definidas as principais funcionalidades e os requisitos de um sistema [LAR00]. Um diagrama de caso de uso é composto por:

- *Caso de Uso*: descreve a funcionalidade de um determinado sistema.
- *Atores*: uma entidade externa ao sistema que, de alguma maneira, participa da história do caso de uso.

A Figura 6, extraída de [LAR00], mostra um exemplo de um diagrama de caso de uso para o sistema de ponto de venda.

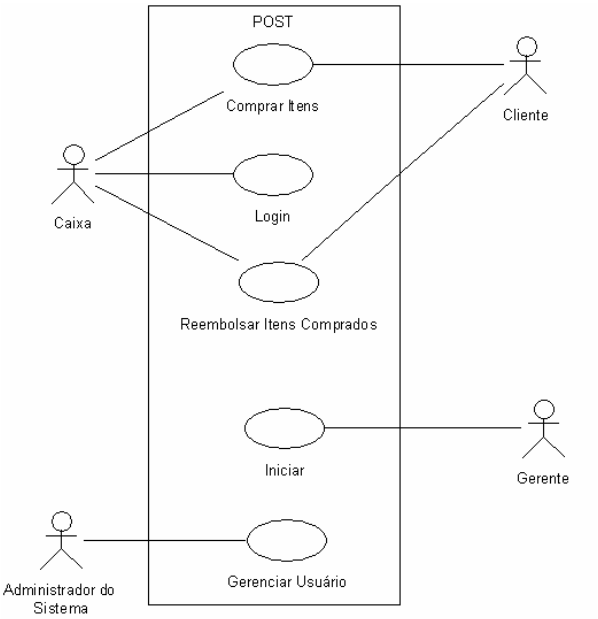


Figura 6 - Diagrama de caso de uso para um sistema de ponto de venda (extraída de [LAR00]).

Cada caso de uso pode ser descrito detalhadamente, mostrando todo o fluxo de interações que devem ser executadas para satisfazer o caso de uso, são os chamados casos de uso expandidos. Segundo [LAR00], eles são úteis para obter uma compreensão mais profunda das funcionalidades do sistema. A Figura 7, extraída de [LAR00], ilustra o caso de expandido para o caso de uso Comprar Itens.

Caso de Uso: Comprar Itens.
Atores: Cliente, caixa.
Finalidade: Capturar uma venda e seu pagamento em dinheiro.
Visão geral: Um cliente chega a um ponto de pagamento com vários itens que deseja comprar. O caixa registra os itens de compra e recebe um pagamento em dinheiro. No final, o cliente sai com os itens.
Tipo: primário e essencial.

Seqüência Típica de Eventos	
Ação do Ator	Resposta do Sistema
1. Este caso de uso começa quando um cliente chega a um ponto de pagamento equipado com um POST, com vários itens que deseja comprar	
2. O caixa registra o código de cada item	3. Determina o preço do item e acrescenta informação sobre o item à transação de vendas em andamento
4. No término da entrada de itens, o caixa indica para o POST que a entrada de itens está completa	5. Calcula e apresenta o total da venda
6. O caixa informa ao cliente o total	
7. O cliente fornece o dinenheiro (que deve ser maior que o total)	
8. O caixa registra a quantia de dinheiro recebida	9. Mostra o troco para o cliente
10. O caixa deposita o dinheiro recebido e retira o troco	11. Registra a venda
12. O cliente sai com os itens comprados	

Figura 7 - Caso de Uso expandido para o caso de uso Comprar Itens.

3.1.1 Diagrama de Caso de Uso e Teste de Unidade

Diagramas de casos de uso são muito usados em testes funcionais [GRO03]. Testes baseados em casos de uso podem ser divididos em dois grupos. O primeiro grupo é baseado no próprio caso de uso, e é a partir deste que se definem os objetivos a serem testados, ou seja, as principais funcionalidades do sistema. São testes de alto nível e são úteis em testes de integração e sistema.

O segundo grupo é baseado nos casos de uso expandidos, os quais oferecem informações sobre todos os passos de iteração para realizar uma dada funcionalidade. Por exemplo, de acordo com o caso de uso expandido descrito anteriormente, é possível saber a seqüência de passos que devem ser seguidos para satisfazer o caso de uso Comprar Itens. Por

exemplo, na seqüência 2, pode ser gerado um caso de teste para saber se o código (UCP) é conhecido do sistema. Outro caso de teste, é testar o total de uma venda, descrito na seqüência 5. Seguindo os passos, outro caso de teste que pode ser gerado é descrito na seqüência 9, ou seja, testar se o sistema está fornecendo o troco correto para o cliente.

É muito importante ressaltar que muitas vezes um caso de uso expandido, analisado isoladamente, não fornece todas as informações necessárias para gerar casos de teste. Por isso, é interessante analisar um caso de uso expandido em conjunto com outros diagramas da UML como, por exemplo, diagrama de seqüência, diagrama de classe e diagrama de atividades para então gerar os casos de teste.

Outra questão importante em relação aos casos de uso expandido são as pré e pós-condições. Sempre que elas aparecerem é comum realizar casos de testes para validá-las.

3.2 Diagrama de Classes (Fase de Projeto)

Diagrama de classes mostram a estrutura estática de um sistema. Segundo [LAR00], as informações típicas de um diagrama de classe neste nível incluem: (i) classes, associações e atributos; (ii) interfaces, com suas operações e constantes; (iii) informação do tipo de atributo; (iv) navegabilidade e (v) dependências.

Um diagrama de classes em nível de projeto enfatiza uma classe como uma entidade de software, ou seja, pode conter todas informações necessárias para que essa classe possa ser transformada em uma classe em linguagem de programação (como por exemplo, uma classe Java). A Figura 8 mostra o pacote Venda do Sistema Ponto de Vendas com duas classes: Venda e LinhadeItemVenda.

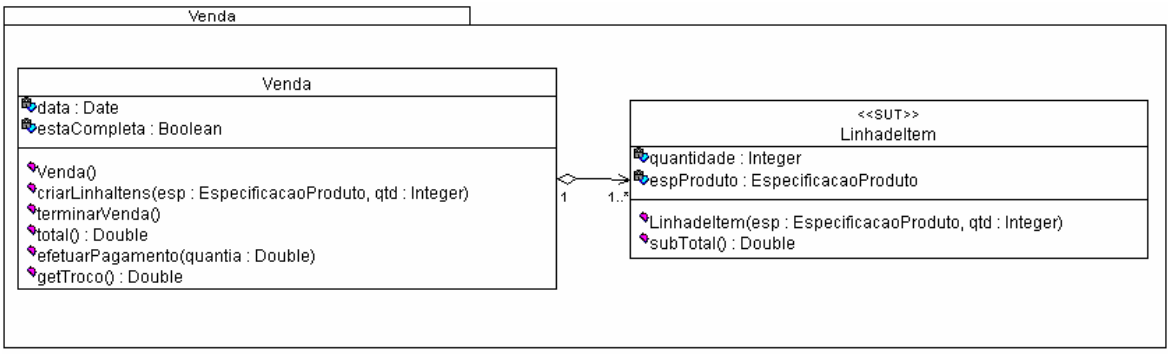


Figura 8 - Classes em nível de projeto.

3.2.1 Diagrama de Classes de Projeto e Teste de Unidade

Uma classe pode conter vários métodos. Cada método que possui uma funcionalidade relevante, por exemplo, que efetua uma operação (por exemplo, CRUD[1]), ou um cálculo, é método candidato a ser testado. Cada um desses métodos representa um caso de teste para a classe e juntos, esses casos de teste representam um suíte de teste para a classe.

As assinaturas dos métodos também são importantes para realizar os testes, pois mostram o tipo de retorno dos métodos, tipos de parâmetros e principalmente se existe alguma dependência de um método em relação a outros.

Os construtores de uma classe em nível de projeto também são usados em teste de unidade. Um construtor é um conjunto de instruções criadas para criar e inicializar uma instância (um objeto). Uma vez que os objetos são inicializados, os métodos que dependem da inicialização desses objetos podem ser testados.

No exemplo da Figura 8, e com base na descrição do caso de uso expandido Comprar Item da Figura 7, alguns casos de teste podem ser gerados. Por exemplo, o método total da classe Venda deve ser testado para saber se o mesmo está calculando corretamente a compra de determinados itens de venda. O método efetuarPagamento também da classe Venda deve ser testado para garantir que o troco seja devolvido corretamente ao cliente. O método subTotal da classe LinhadeItemVenda deve ser testado para garantir que os subtotais de um item de venda são calculados corretamente.

3.3 Diagrama de Atividades

Um diagrama de atividades descreve o fluxo de controle de um procedimento. Segundo [FOW00], um diagrama de atividades pode ser usado nas seguintes situações:

- *Para analisar um caso de uso*: através de diagramas de atividades é possível compreender que ações precisam acontecer e quais são as dependências comportamentais entre eles.
- *Para compreender um sistema wokflow*: um diagrama de atividades é muito útil para a compreensão de um sistema de negócio, mostrando como o negócio funciona e como ele pode mudar.

[1] Crud – é a abreviação para as operações *Create* (criar), *Retrieve* (recuperar), *Update* (atualizar) e *Delete* (deletar).

- *Para descrever um algoritmo complicado:* um diagrama de atividades pode mostrar todos os caminhos de execução de um algoritmo.

Um exemplo de um diagrama de atividades pode ser visto na Figura 9. Neste exemplo, é mostrado o fluxo de execução da operação efetuarPagamento da classe *Venda* (Figura 8).

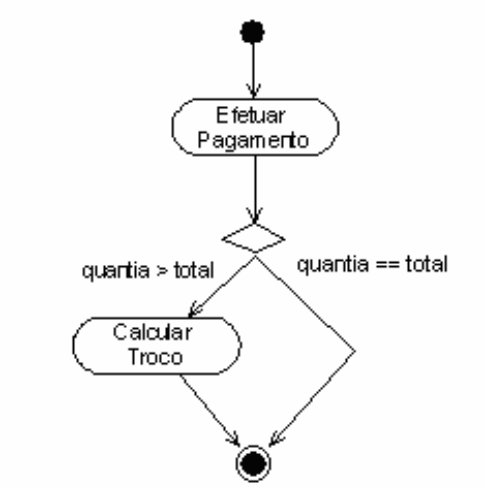


Figura 9 - Diagrama de atividades para a operação efetuarPagamento.

3.3.1 Diagrama de Atividades e Teste de Unidade

Segundo [LAR00], um diagrama de atividades é usado principalmente para especificar testes estruturais. Diagramas de atividades fornecem informações similares a um grafo de fluxo de controle (um teste tradicional de caixa branca), embora em um nível mais alto de abstração.

Para testes unitários, diagramas de atividades fornecem medidas de cobertura de código. Por exemplo, um diagrama de atividade pode mostrar todos os passos necessários para uma dada atividade. No exemplo da Figura 9, o diagrama de atividades descreve as condições para o cálculo da operação efetuarPagamento. Dessa forma, existem duas situações a serem consideradas. A primeira é quando a quantia fornecida é maior que o total. A segunda é quando a quantia fornecida é menor que o total. Portanto, estes dois caminhos, geram dois casos de teste para a operação, um teste positivo e um teste negativo.

Diagramas de atividades também são muito úteis para mostrar o fluxo de execução dos casos de teste, especificando a ordem na qual os casos de teste devem ser executados.

3.4 Diagramas de Seqüência

O objetivo de um diagrama de seqüência é mostrar uma interação, isto é, uma seqüência de mensagens trocadas entre vários objetos num determinado contexto. A Figura 10 ilustra um exemplo de um diagrama de seqüência para a operação teminarVenda, da classe Venda (Figura 8).

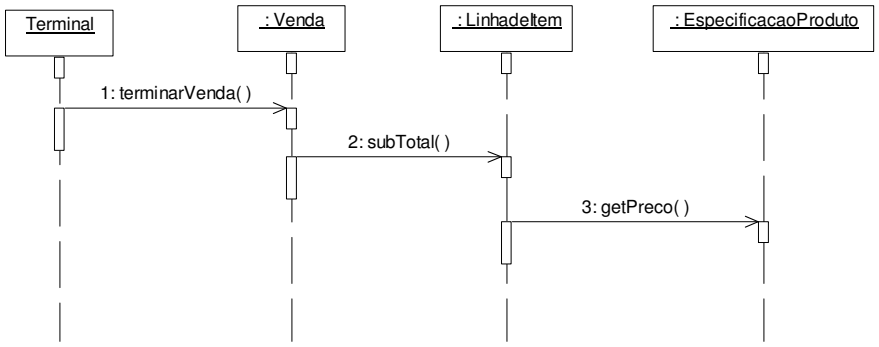


Figura 10 - Diagrama de seqüência terminarVenda.

3.4.1 Diagramas de Seqüência e Teste de Unidade

Diagramas de seqüência mostram todas as seqüências de mensagens trocadas entre os objetos. Isto é importante, pois é possível saber que mensagem está sendo enviada, em que ordem e para qual objeto.

Esse comportamento pode ser usado no contexto de teste de unitário. No exemplo do diagrama de seqüência da Figura 10, é possível verificar a seqüência de mensagens enviadas aos objetos para terminar uma venda. Estas seqüências de mensagens podem ser validadas por casos de testes.

Segundo [GRO03], alguns problemas típicos que podem ser descobertos através de testes baseados em diagramas de seqüência são:

- Saídas incorretas ou ausentes;
- Mensagem correta passada ao objeto errado;
- Mensagem incorreta passada ao objeto correto;
- Mensagem enviada para um objeto destruído (finalizado);
- Exceção correta, mas gerada pelo objeto errado;
- Exceção incorreta gerada pelo objeto correto.

Os diagramas de seqüência são muito úteis quando não se tem acesso à estrutura interna de uma classe, assim, é possível mostrar o comportamento das mensagens como uma caixa preta.

3.5 Considerações Finais

Este capítulo apresentou como alguns diagramas da UML fornecem informações úteis à especificação e projetos de teste em nível de unidade. Dessa forma, cada diagrama do modelo de projeto deve ser analisado em detalhes para transformar o mesmo em um modelo de projeto de teste.

4 O PERFIL DE TESTE DA UML 2.0

Este capítulo apresenta os principais elementos do perfil de teste da UML 2.0. É apresentado um metamodelo baseado no MOF para teste. Também serão descritos e discutidos os elementos apropriados para testes em nível de unidade.

4.1 Visão Geral

O *UML 2.0 Testing Profile Specification* (U2TP) [OMG04A] – Perfil de Teste da UML 2.0 define uma linguagem para projetar, visualizar especificar, construir e documentar artefatos de testes para sistemas [OMG04A]. A linguagem para modelagem de testes pode ser usada para tecnologias de objetos e componentes e pode ser aplicada para testes de sistemas em vários domínios de aplicação. O perfil de teste da UML 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 com seus respectivos artefatos de teste.

O U2TP é baseada na especificação da UML 2.0 [OMG04B]. Ele é definido usando a abordagem de metamodelagem da UML. Sendo assim, o U2TP foi projetada com os seguintes princípios [OMG04A]:

- Integração com a UML: o Perfil de Teste da UML é definido na base do metamodelo definido no volume da superestrutura da UML 2.0 e segue os principais Perfis da UML como definido no volume da superestrutura da UML 2.0.
- Reuso: o Perfil de Teste da UML faz uso direto dos conceitos da UML e estende e adiciona novos conceitos somente quando necessários.

O U2TP está baseada no metamodelo MOF. O MOF é o padrão da OMG para a construção de metamodelos. É uma especificação que define uma linguagem abstrata para metamodelagem e um *framework* para especificação, construção e gerenciamento de metamodelos independentes de plataforma. Exemplos de sistemas que usam o MOF incluem ferramentas de modelagem e desenvolvimento, sistemas *data warehouse*, repositórios de metadados, entre outros [OMG06].

4.2 Estrutura do U2TP

O U2TP está organizada em quatro grupos lógicos de conceitos [OMG04A]:

- *Arquitetura de Teste* - define conceitos relacionados a estrutura e configuração de teste.
- *Dados de Teste* - define conceitos para dados de teste usados em procedimentos de teste.
- *Comportamento de teste* - define conceitos relacionados aos aspectos dinâmicos dos procedimentos de teste.
- *Tempo de Teste* - define conceitos quantificados por tempo para procedimentos de teste.

Os elementos mais importantes do U2TP estão listados na Tabela 1, adaptada de [DAI03]. Esses elementos serão discutidos nas próximas seções.

Tabela 1 - Grupos e seus principais elementos (adaptado de [DAI03]).

Conceitos de Arquitetura de teste	**Conceitos de Comportamento de teste**	**Conceitos de Dados de Teste**	**Conceitos de Tempo**
Sut	*TestObjective*	*Wildcards*	*Timer*
TestComponent	*TestCase*	*DataPool*	*TimeZone*
TestContext	*Defaults*	*DataPartition*	
TestConfiguration	*Validation action*	*DataSelector*	
TestControl	*Verdicts*	*Coding Rules*	
Arbiter			
Scheduler			

4.3 Arquitetura de Teste

Este grupo define os principais elementos e seus relacionamentos envolvidos em um teste. A Figura 11, extraída de [OMG04A], ilustra os principais elementos.

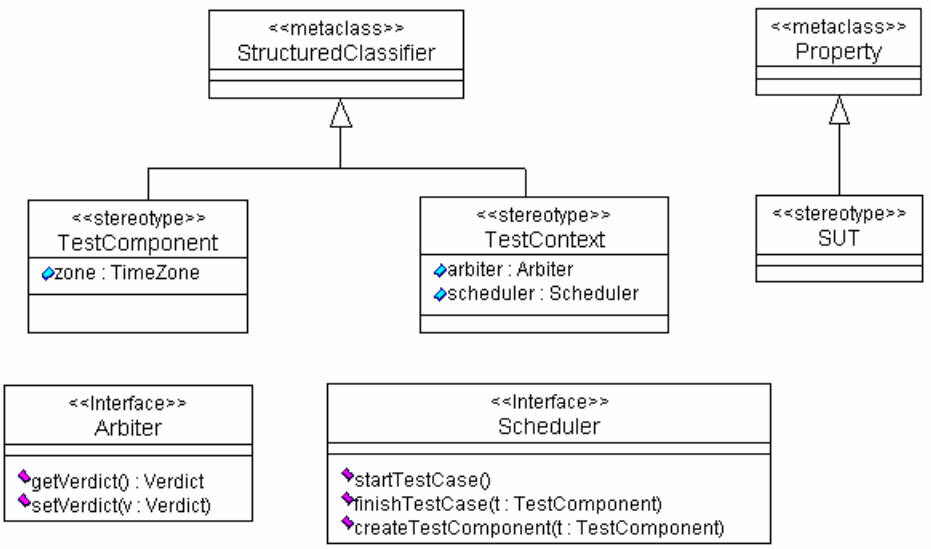


Figura 11 - Grupo de Arquitetura de Teste.

4.3.1 Sut

Descrição*:*

O *System Under Test* (*Sut*) é o sistema sob teste, ou seja, o que está será testado. Um *Sut* pode ter diferentes níveis de abstração: um sistema completo, um subsistema, um pacote, um único componente ou até mesmo uma única classe [DAI03]. Como o U2TP é direcionado somente para testes de caixa preta, o *Sut* fornece somente um conjunto de operações através das interfaces disponíveis publicamente. Nenhuma informação sobre a estrutura interna do *Sut* é disponível [OMG04A]. O *Sut* estende a metaclasse *Property* do metamodelo MOF.

Notação:

A notação para o *Sut* é nomear o que se deseja testar com o estereotipo <<Sut>>. O *Sut* é usado dentro do elemento *TestContext* (seção 4.3.2).

4.3.2 TestContext

Descrição:

Um *TestContext* (contexto de teste) é uma classe que representa o agrupamento de vários casos de teste [OMG04A], ou seja, representa o conceito conhecido como suíte de

teste. Um *TestContext* estende um *Structured Classifier*, outra metaclasse definida no metamodelo MOF.

Notação:

A notação para o elemento *TestContext* é uma classe com o estereotipo <<TestContext>>.

4.3.3 TestControl

Descrição:

O elemento *TestControl* (Controle de Teste) é uma especificação usada para determinar como o *Sut* deve ser testado para um determinado Contexto de Teste [OMG04A]. Permite especificar a ordem de execução dos casos de teste.

4.3.4 TestComponent

Descrição:

Um *TestComponent* (Componente de Teste) é uma classe de um sistema em teste. O objetivo de um *TestComponent* é ajudar a realizar o comportamento de um ou mais casos de teste [OMG04A]. *TestComponents* interagem com o *Sut* ou com outros *TestComponents* para realizar os casos de testes que são definidos dentro do contexto de teste. O elemento *TestComponent* estende um *Structured Classifier*.

Notação:

A notação para o elemento *TestComponent* é uma classe com o estereotipo <<TestComponent>>.

4.3.5 Arbiter

Descrição:

Arbiter é uma interface predefinida fornecida juntamente com o U2TP. O propósito de uma implementação *Arbiter* é determinar o *Verdict* (seção 4.4.1), ou seja, o resultado final para um caso de teste.

4.3.6 Scheduler

Descrição:

Scheduler é uma interface pré-definida fornecida com o U2TP. O propósito da implementação de um *scheduler* é controlar a execução de diferentes *TestComponents*. O *Scheduler* mantém a informação sobre qual *TestComponent* está executando e colabora com o *arbiter,* para informar o *Verdict* final para o caso de teste. Mantém o controle sob a criação e destruição do *TestComponent* e ele sabe quais *TestComponents* estão participando de cada caso de teste [OMG04A].

4.4 Comportamento de Teste

Este grupo define os conceitos necessários para representar todos os elementos que fazem parte dos aspectos dinâmicos dos procedimentos de teste. A Figura 12, extraída de [OMG04A], ilustra os principais elementos.

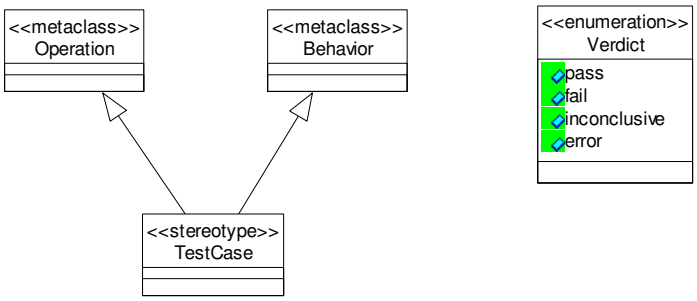


Figura 12 - Elementos *TestCase* e *Verdict*.

4.4.1 Verdict

Descrição

Verdict é um tipo de dados *enumeration* prédefinido que contém os valores: *fail*, *inconclusive*, *pass* e *error*.

- *Pass:* indica que o caso de teste é completo e que o *Sut* se comportou como esperado.
- *Fail:* descreve que o propósito do caso de teste foi violado, ou seja, o resultado esperado foi diferente do resultado real.
- *Inconclusive:* é usado quando nenhum valor *"Pass"* ou "*Fail"* pode ser fornecido.
- *Error:* é usado para indicar erros (exceções) dentro do sistema sob teste.

Um caso de teste (seção 4.4.2) sempre produz um dos *Verdicts* descritos acima. O *Verdict* de um caso de teste é calculado pela interface *Arbiter*.

4.4.2 TestCase

Descrição:

Um *TestCase* (caso de teste) é uma especificação de uma situação particular para testar o sistema, incluindo o que testar, bem como entradas, resultados, e sob quais condições [OMG04A]. Um *TestCase* estende um *Operator,* uma metaclasse definida no metamodelo MOF e um *Behavior*, outra metaclasse definida no metamodelo MOF.

Restrições:

- O tipo de retorno de um caso de teste deve ser um *Verdict*.
- O estereótipo de um caso de teste não pode ser aplicado a ambos: o comportamento e sua especificação.
- Se o estereótipo de um caso de teste for aplicado em uma operação, o *classifier* (classe) desta operação tem que ter estereótipo *TestContext* aplicado.
- Se o estereótipo de um caso de teste for aplicado a um comportamento, o comportamento desse caso de teste, tem que ter o estereótipo *TestContext* aplicado.

Notação:

A notação para um caso de teste é uma operação com o estereótipo <<TestCase>>.

4.5 Dados de Teste

Este grupo contém os conceitos necessários para descrever os elementos de dados de teste usados para exercitar os casos de teste. Esses elementos contêm os tipos de dados que são usados para execução de um ou mais casos de teste. A Figura 13, extraída de [OMG04A], ilustra os elementos principais do grupo de dados de teste.

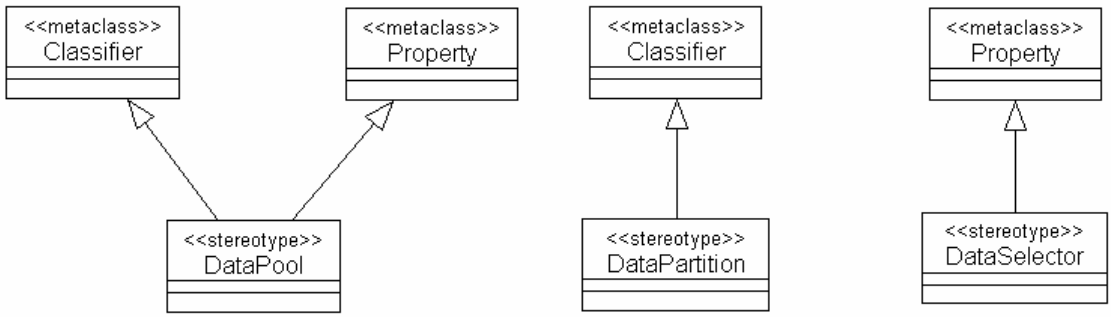


Figura 13 - Grupo de Dados de Teste.

4.5.1 DataPool

Descrição:

Um *DataPool* representa um conjunto concreto de valores de dados que são usados para produzir o resultado de um caso de teste. Um *DataPool* contém um ou vários *DataPartition* (seção 4.5.2), mas só pode estar associado a um *TestContext* ou *TestComponent*. Um *DataPool* estende as metaclasses *classifier* e *property* do MOF.

Notação:

A notação para um *DataPool* é uma classe estereotipada com <<DataPool>>.

4.5.2 DataPartition

Descrição:

Um *DataPartition* é usado para definir classes de equivalência para um determinado tipo de dados. Um *DataPartition* também estende a metaclasse *classifier* do MOF.

Notação:

A notação para o elemento *DataPartition* é uma classe com o estereótipo <<DataPartition>> aplicado.

4.5.3 DataSelector

Descrição:

Um *DataSelector* é uma operação que define como valores de dados ou classes de equivalência são selecionadas de *DataPool* ou *DataPartition* [OMG04A]. Um *DataSelector* também estende a metaclasse *operation* do MOF.

Notação:

A notação para o elemento *DataSelector* é uma operação estereotipada com <<DataSelector>>.

4.6 Grupo de Tempo de Teste

Este último grupo define um conjunto de conceitos para especificar restrições de tempo, e/ou de observações de tempo [OMG04A]. O objetivo principal desse grupo é definir conceitos para restrições e controle do comportamento dos testes no que diz respeito ao tempo. Os dois elementos principais desse grupo são: *Timer* e *TimeZone*.

4.6.1 Timer

Descrição:

Timer é uma interface predefinida fornecida no U2TP. Um *Timer* é um mecanismo que pode gerar um *timeout* quando um valor especificado de tempo acontece. Isto pode ser quando um intervalo de tempo pré-especificado expira em um determinado momento (normalmente no momento em que o *Timer* é inicializado).

4.6.2 Timezones

Descrição:

Timezones servem como mecanismos de agrupamento para *TestComponents* em teste de sistema. Cada *TestComponents* pertence a no máximo um *timezone*. *TestComponents* com o mesmo *timezone* possuem a mesma percepção de tempo, por exemplo, possuem tempos sincronizados.

4.7. Metamodelo baseado no MOF para Teste

Esta seção apresenta um metamodelo padrão para o Perfil de teste da UML 2.0, o qual é uma instância do MOF. Este metamodelo é limitado aos elementos da arquitetura e comportamento de teste e é apresentado na Figura 14 extraída de [OMG04A]. A maioria dos elementos do Perfil de Teste apresentados anteriormente também estão presentes no metamodelo.

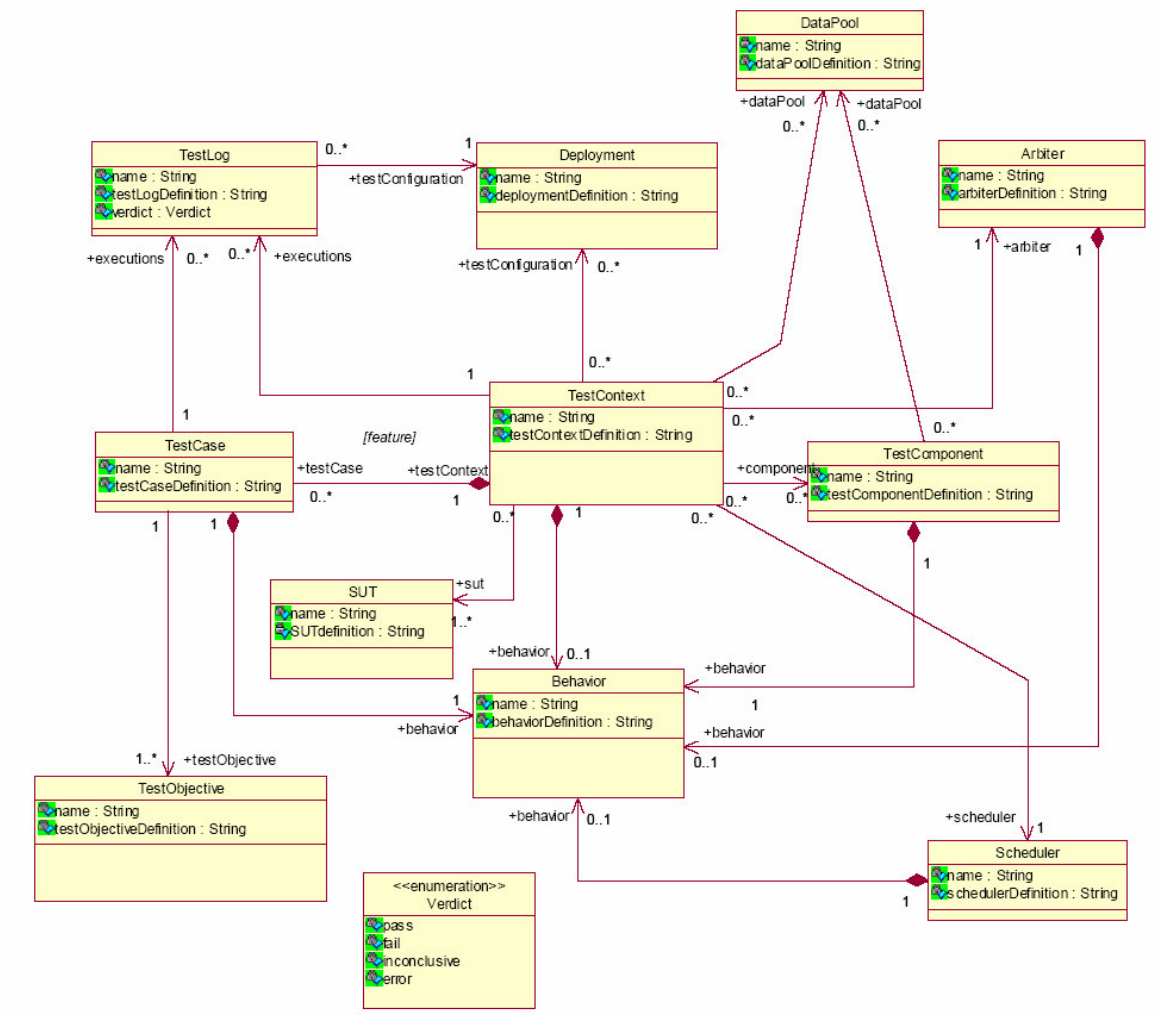


Figura 14 - Arquitetura e Comportamento de Teste do Metamodelo baseado no MOF.

O elemento central dessa figura é o elemento "*TestContext*", que tem associação com todos os demais elementos. Ele representa o passo inicial para especificação da parte de arquitetura de teste, e a partir dele, os demais elementos são instanciados e utilizados. Cada elemento da Figura 14, como apresentado anteriormente, possui uma descrição, uma notação e tem um objetivo específico dentro do perfil.

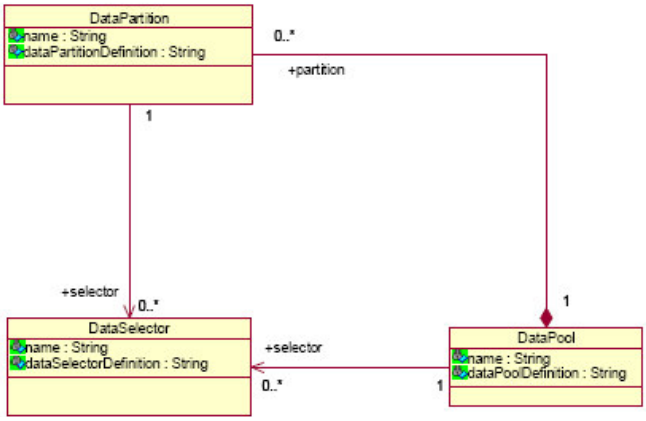


Figura 15 - Grupo Dados de teste do Metamodelo baseado no MOF.

A Figura 15, extraída de [OMG04A], mostra os elementos principais do grupo de dados de teste. O elemento central desse grupo é o *DataPool*. O elemento *DataPartition* é usado obrigatoriamente sempre que o elemento *DataPool* for usado, pela associação de agregação entre esses dois elementos. O elemento *DataSelector*, deve estar associado a no máximo um *DataPool* e um *DataPartition*.

A Figura 16, extraída de [OMG04A], fornece três interfaces que tecnicamente não fazem parte do metamodelo baseado no MOF, são elas: *Timer*, *Arbiter* e *Scheduler*.

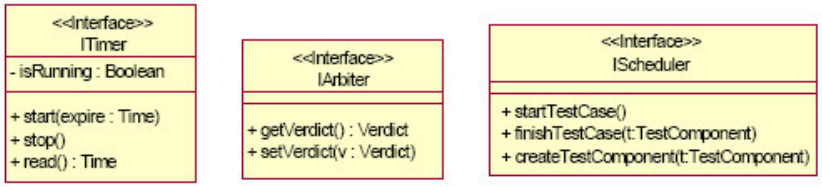


Figura 16 - Interfaces predefinidas: *Timer*, *Arbiter* e *Sheduler*.

4.8 U2TP e Teste Unitário

Os elementos apresentados nas seções anteriores são usados em todos os níveis de teste. Em nível de unidade, os elementos apropriados e explorados nesse trabalho são apresentados na Figura 17.

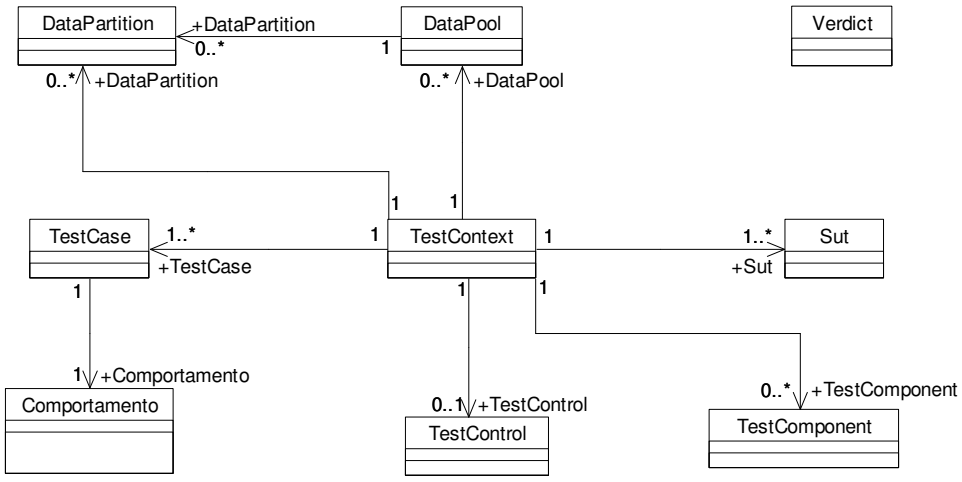


Figura 17 - Elementos do U2TP para teste em nível de unidade.

A Figura 17 ilustra o metamodelo para representação de teste em nível de unidade para os elementos pertencentes aos grupos de Arquitetura de Teste (*Sut*, *TestContext*, *TestComponent*, *TestCase*, *TestControl*), Comportamento de Teste (Comportamento e *Verdict*) e Dados de teste (*DataPool* e *DataPartition*). Como podemos observar pela Figura 17, a cardinalidade de algumas associaçõess mudaram, como é o exemplo da associação de *TestContext* com *Sut*, onde agora, um *TestContext* deve obrigatoriamente ter um ou mais *Sut* associados. Assim, como este metamodelo é focado somente para teste de unidade, algumas restrições foram acrescentadas as associações entre os elementos.

Sut: Esse elemento é especificado para representar a unidade sob teste. Sempre haverá no mínimo um *Sut*.

TestContext: Esse elemento é especificado para representar uma suíte de testes para a unidade a ser testada. Assim, um *TestContext* tem que conter no mínimo um *Sut* e um *Testcase*.

TestCase: Esse elemento é especificado para representar os casos de teste em nível de unidade. Sempre haverá no mínimo um caso de teste para testar o *Sut*.

TestComponent: Esse elemento é especificado como sinônimo de um "*stub*" em teste unitário, ou seja, é usado para simular serviços explicitamente requisitados pelo *Sut*. Pode ser usado sempre que um ou mais casos de teste necessitar de serviços que não estejam disponíveis no projeto de teste.

TestControl: Esse elemento deve especificado para indicar a ordem de execução dos casos de teste sempre que for necessário.

Comportamento: Esse elemento é especificado para representar o comportamento de um caso de teste. Precisa ser especificado para cada caso de teste.
Verdict: Esse elemento é especificado para especificar o veredito dos casos de teste.
DataPool: Esse elemento pode ser usado para representar um conjunto de valores concretos usados por um ou mais casos de teste para testar um determinado *Sut*.
DataPartition: Esse elemento pode ser usado para particionar os valores de DataPool quando for necssário.

4.9 Mapeamento dos Conceitos para Junit

Alguns elementos apresentados anteriormente possuem um mapeamento direto dos conceitos para a ferramenta JUnit. A Tabela 2, adaptada de [OMG04A], apresenta esse mapeamento.

Tabela 2 - Mapeamento dos conceitos para JUnit.

U2TP	JUnit
Arquitetura e Comportamento de Teste	
TestContext	Um contexto de teste no JUnit é realizado como uma especialização (ou subclasse) de *TestCase*.
Sut	Qualquer classe que possa ser testada no JUnit.
TestComponent	Não há um mapeamento explicito desse elemento.
TestCase	Um caso de teste é realizado no JUnit como uma operação pertencente a uma subclasse de *TestCase*.
TestControl	Um *TestControl* é implementado no JUnit sobrecarregando o método *runTest*.
Verdict	No JUnit os valores predefinidos para *Verdict* são: *pass*, *fail* e *error*.
Dados de Teste	
DataPool	Qualquer classe com operações de acesso aos dados dessa classe.
DataPartition	Qualquer classe com operações de acesso aos dados dessa classe.
DataSelector	Uma operação de acesso a um datappol ou datapartition.

4.10. Uma Metodologia para usar o U2TP

Uma metodologia para usar o U2TP é proposta em [DAI03] e [DAI04A]. Os autores assumem que o U2TP é usado efetivamente a partir de um modelo de projeto UML existente. Quanto mais enriquecido de detalhes de informações for este modelo, mais fácil será aplicar o perfil para especificar os casos de teste para este modelo. Também definem para cada grupo do U2TP quais elementos são obrigatórios e quais são opcionais. Entretanto, na metodologia não há uma divisão desses elementos entre os níveis de teste (unidade, integração e sistema). A metodologia está baseada na arquitetura MDA - *Model Driver Architecture* [OMG03] e [DAI04B].

A idéia principal da metodologia é reaproveitar sempre que possível diagramas da UML do projeto de sistema para especificação de teste. Dessa forma foram descritos alguns pressupostos de como usar as notações da U2TP para cada grupo.

4.10.1 Grupo de Arquitetura de Teste

Para o grupo de arquitetura de teste deve-se especificar dois elementos que são obrigatórios: *Sut* e *TestContext*. O primeiro é especificado nomeando uma classe ou um objeto do modelo de sistema, como *Sut*, definindo o que será testado. O segundo é especificado criando uma nova classe e nomeando essa como *TestContext*, listando todos seus atributos e casos de teste.

Outro elemento que pode ser especificado, mas que não é obrigatório, é o elemento *TestComponent*. Os autores ressaltam que em nível de unidade esse elemento não é necessário.

Para definir a ordem de execução dos casos de teste sempre que necessário deve-se especificar o elemento *TestControl*. Se existirem diagramas de atividades no modelo de projeto de sistema, cada atividade ilustra um caso de teste, e o fluxo das atividades descrevem o fluxo dos casos de teste na especificação do *TestControl*. Se existirem diagramas de casos de uso no modelo de projeto de sistema, cada caso de uso descreve um caso de teste que pode ser especificado junto no *TestControl*. Esse elemento é opcional para o grupo de arquitetura de teste.

4.10.2 Comportamento de Teste

Para especificação do comportamento dos casos de teste são usados diagramas de interação do modelo do sistema. Com isso pode-se renomear ou agrupar instâncias e nomear com o estereótipo de acordo com seus papéis (por exemplo, *TestComponent* ou *Sut*). É preciso nomear o elemento *Verdict* no final de cada especificação do caso de teste. Normalmente um *verdict* em um caso de teste é configurado com "*pass*".

Pode-se definir os objetivos de teste para cada caso de teste. Em testes de integração ou sistema, um objetivo de teste pode ser um caso de uso. Esse elemento não é obrigatório.

Timers podem ser derivados de especificações de restrições de tempo com diagramas de seqüência ou diagrama de máquina de estados. O autor ressalta que em nível de unidade *timers* não são usados. Esse elemento também não é obrigatório.

De acordo com os autores, a grande idéia de transformar um projeto de sistema em um projeto de teste é fortalecer a inclusão do teste cada vez mais cedo dentro das fases de desenvolvimento de software e o reuso, quando possível, de alguns diagramas do modelo de projeto de sistema para o modelo de projeto de teste.

4.11 Considerações Finais

Esse capítulo apresentou os principais elementos do U2TP. Cada elemento possui uma descrição, uma semântica e uma notação. O metamodelo apresentado na seção 4.7 define como esses elementos se relacionam.

A seção 4.8 apresentou quais elementos do U2TP são apropriados para teste em nível de unidade, os quais serão explorados neste trabalho para geração automática de *driver*s e *stubs* para JUnit. A geração proposta é baseada no refinamento do mapeamento dos conceitos do U2TP para os conceitos do JUnit, definido pelo próprio U2TP.

Esse capítulo também apresentou uma metodologia para usar o U2TP. Essa metodologia assume que o U2TP é usado efetivamente a partir de um modelo de projeto existente. Assim, usando o U2TP é possível transformar um modelo de projeto existente em um modelo de projeto de teste. Esta metodologia será adotada neste trabalho.

O U2TP foi desenvolvido para facilitar a especificação, construção, documentação e visualização dos artefatos utilizados no processo de teste. Ela é capaz de capturar todas informações que podem ser necessárias para representar diferentes processos de teste.

Por se tratar de um modelo UML, o U2TP segue um padrão que pode ser usado independente de linguagem de programação. Com isso, é possível especificar teste em um nível mais alto de abstração. Permite que testes sejam especificados como uma caixa preta.

U2TP permite a integração com diferentes ferramentas, e em diferentes domínios de aplicação. Através de ferramentas de modelagem com suporte a exportação/importação de documentos como XMI (*XML Metadata Interchange*) é possível à troca, ou seja, o intercâmbio entre modelos, o que facilita a manutenção e portabilidade dos artefatos de teste.

Outra vantagem do U2TP é a possibilidade de geração de código, ou seja, possibilita a geração de testes a partir de aspectos estruturais (estáticos) e comportamentais (dinâmicos) de modelos UML. Assim, é possível usar seus conceitos para fazer o mapeamento dos mesmos com diferentes ferramentas de testes e para diferentes linguagens de programação.

5 TRABALHOS RELACIONADOS

Os trabalhos relacionados apresentados neste capítulo estão divididos em três grupos: (i) geração automatizada de *drivers* a partir de ambientes integrados de desenvolvimento de Software (IDE); (ii) geração automatizada de casos de teste para JUnit e (iii) geração automatizada de *drivers* a partir de modelos UML. Estes são discutidos e comparados ao final do capítulo.

5.1 Geração de Drivers a partir de IDEs

Atualmente o uso de IDEs (*Eclipse*, *NetBeans*, *JBuilder*) como ambiente de desenvolvimento de software está bastante difundido. Alguns dos motivos incluem: suporte a vários padrões, portabilidade, flexibilidade e também possibilidade de integração com outras ferramentas, como por exemplo, o JUnit.

A ferramenta JUnit, foi adaptada a uma variedade de IDEs que permitem armazenar todo o código de teste em um repositório facilitando assim a manutenção e documentação do código desenvolvido. Essas IDEs possibilitam que a estrutura do *driver* de teste possa ser gerada automaticamente. Segundo [OLA03], o principal objetivo das IDEs é fornecer meios simples e interativos para criar métodos e classes de teste, sem a necessidade de escrever o *driver* de teste. Já em [DAI03], o autor ressalta que a grande dificuldade em relação à geração do *driver* de teste é a codificação automatizada da parte dinâmica dos casos de teste, a qual não é contemplada nestas IDEs.

Por exemplo, no *Eclipse* [ECL05] e *NetBeans* [NET05], para gerar o *Driver* de teste o desenvolvedor precisa selecionar a classe que deseja testar e os métodos desta. Uma vez selecionados, a estrutura estática do *driver* é gerada automaticamente. A Figura 18 mostra esse procedimento no Eclipse, e o esqueleto do *driver* correspondente gerado. Entretanto, não existe uma especificação do comportamento dos casos de teste, ou seja, o próprio desenvolvedor é o responsável por especificar e codificar o comportamento dos casos de teste. A geração dos *Stubs* de teste também não é suportada por essas IDEs ficando, portanto, de responsabilidade do programador gerar esse elemento.

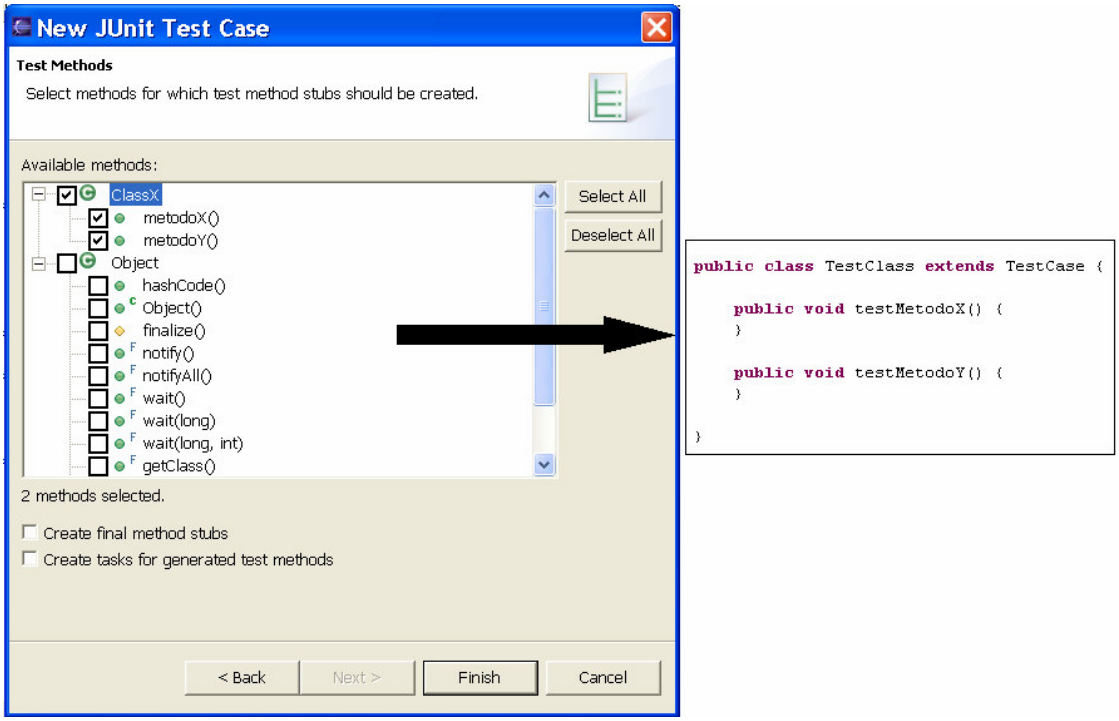


Figura 18 - Geração de *Driver* no Eclipse.

5.2 Geração Automatizada de Drivers para Junit

5.2.1 Especificações JML

Uma abordagem para geração automatizada de *drivers* de teste para a ferramenta JUnit é encontrada em [CHE02]. A abordagem usa uma linguagem de especificação formal (JML – *Java Modeling Language*) para verificar assertivas em tempo de execução.

Nessa abordagem, ao invés de especificar as saídas esperadas (oráculos de teste) e comparar então com aquelas produzidas pela execução do teste, o comportamento do método testado é monitorado para decidir se o teste passou ou falhou. Este monitoramento é feito usando a linguagem de especificação formal JML. A Figura 19, extraída [CHE02], ilustra um exemplo de especificação JML para uma classe denominada *Person*.

```
public class Person{
      private /*@ spec_public @*/ String name;
      private /*@ spec_public @*/ int weight;
      // @ public invariant name != null && name.length() > 0 &&
      weight >= 0;

      /*@ public behavior
            @ requires n != null && name.lenght() > 0;
            @ assignable name , weight;
            @ ensures n.equals(name) && weight = = 0;
            @ signals (Exception e) false;
      public Person (String n) {name = n; weight = 0;}

      /*@ public behavior
            @ assignable weight;
            @ ensures Kgs >= 0 && weight = = \old(weight  + Kgs);
            @ signals (IllegalArgumentException e) Kgs < 0;
      @*/
      public void addKgs(int Kgs) {weight + = Kgs; }

      /*@ public behavior
            @ ensures \result = = weight;
            @ signals (Exception e) false;
      @*/
      public /*@ pure @*/ int getWeight() {return weight; }

      /* . . . */
}
```

Figura 19 - Especificação em JML para a classe *Person*.

Especificações formais incluem invariantes, pré e pós-condições. Dessa forma, é assumido que tais especificações são suficientes para mostrar o comportamento desejado para o caso de teste. Porém, a qualidade gerada dos oráculos de teste dependerá da qualidade das especificações das pós-condições, que são de responsabilidade do usuário (programador).

Como podemos observar pela Figura 19, para cada método testado, é especificado o comportamento correspondente em JML. Assim, através das variáveis: *requires*, *assignable* e *ensures* são especificadas as condições para a execução de cada caso de teste. A variável *signals* é responsável por lançar uma exceção no caso de alguma condição for violada. Dessa forma, o comportamento do método testado é monitorado e os *drivers* de teste são gerados automaticamente, bem como as saídas esperadas para cada caso de teste. Porém, a configuração dos valores de entrada (por exemplo, configuração do método *setup* do Junit) é feita manualmente pelo programador. A geração de *stubs* de teste também não é contemplada nessa abordagem.

5.2.2 Especificações AOTDL

Em [ZON04] é proposta a geração automatizada de casos de testes a partir de programas orientados a aspectos AOP – *Aspect-Oriented Programming*. Essa abordagem usa a linguagem *AspectJ*, uma extensão para Java de linguagem orientada a aspectos. *AspectJ* adiciona alguns novos conceitos na linguagem Java como: *join points, pointcuts*, *advice* e *aspect.* Também é definida uma linguagem para teste denominada AOTDL (*Aspect-Oriented Test Description Language*) – linguagem de descrição de teste orientada a aspectos.

A idéia de usar AOP é dividir aspectos genéricos da linguagem em aspectos específicos da aplicação, como por exemplo, aspectos de pré e pós-condições, aspectos de monitoração, aspectos de comportamento, etc. Dessa forma foi definida a AOTDL como aspecto para teste. A AOTDL envia mensagens em tempo de execução à unidade testada (métodos de classes) para identificar oráculos de teste. A Figura 20, extraída de [ZON04], ilustra a visão geral da proposta. Uma ferramenta denominada JAOUT foi desenvolvida para prova de conceitos.

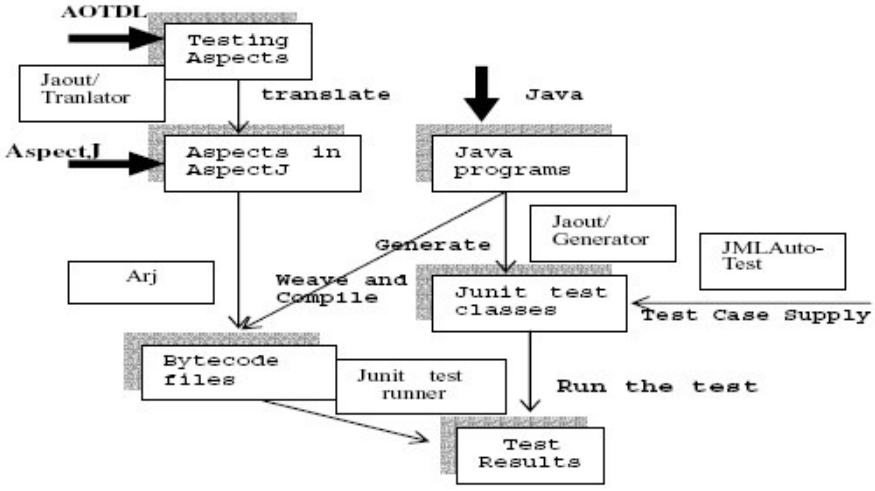


Figura 20 - Visão geral da proposta.

De acordo com a Figura 20, a AOTDL é traduzida para a ferramenta JAOUT. Em seguida, os aspectos são compilados para arquivos Java (arquivos de classes). A ferramenta então gera classes de testes correspondentes para a ferramenta JUnit para a unidade a ser testada. Os arquivos de classes podem servir como oráculos de teste. Finalizando, abastecida com as entradas de testes geradas automaticamente pela classe JMLAutoTest, a unidade testada é executada, e os resultados são verificados em tempo de execução através do

lançamento de exceções. A Figura 21, extraída de [ZON04], ilustra exemplos de AOP e AOTDL para uma classe denominada *Stack*.

```
class Stack{
  public void init() {...}
  public void push (Node n) {…}
}
Aspect TempLogic{
  protected Boolean isInitialized = false;
  //method push is called
  pointcut pushReached(Stack st):
    target (st) &&  call(void
 Stack.push (Node));
  //method init is called
  pointcut initReached (Stack st):
    target (st) &&  call(void
 Stack.init (void));
  //advice after init is called
  after (Stack st): initReached(st){
    isInitialized = true;
 }
 //advice before push is called
 before(Stack st)
      throws NotInitializedException:
 pushReached (st){
  if (!isInitialized)
    throw new NotInitializedException();
 }
}
```

Exemplo de AspectJ para Classe Stack

```
TestingAspect TempLogic{
  //all pointcuts and other utilitary advice are
declared
  //in the Utility unit
 Utility{
   protected Boolean isInitialized = false;
   //push is reached
   pointcut pushReached (Stack st):
    target (st) && call (void
  Stack.push(Integer));
   //init is reached
   pointcut pushReached (Stack st):
    target(st) && call (void Stack.init()void));
   after (Stack st): initReached(st){
     isInitialized = true;
  }
 }
 MeaninglessCase Advice{
 //advice for specifying criteria of
 //meaningless test cases
 before(Stacke s):
   pushReached(s):
      s.getSize()>=MAX:"Overflow";
 . . .
 }
 Error Advice{
 //advices for specifying criteria of
 //errors
 before(Stack s):
   pushReached(s):
      !isInitialized:"Not Initialized";
 . . .
 }
}
```

Exemplo de aspecto de teste para a classe Stack

Figura 21 - Exemplos de AOP e AOTDL para a classe *Stack*.

5.3 Geração de Drivers de Teste a partir de Modelos UML

5.3.1 TestExpert

Outro tipo de abordagem, sendo esta baseada em modelos UML, é um *Add-in* que a ferramenta *Rational Rose* disponibiliza, denominado *TestExpert* [IBM05]. Este *Add-in* propõe a geração automatizada da estrutura do *Driver* de teste para as ferramentas JUnit e *cppUnit,* através da geração de classes de teste baseado em modelos UML usando *templates* de teste (das ferramentas JUnit e *Cppunit*). Porém, o comportamento do teste descrevendo a lógica de execução do teste não é gerado. A Figura 22 mostra a visão geral da abordagem (geração para *cppUnit*).

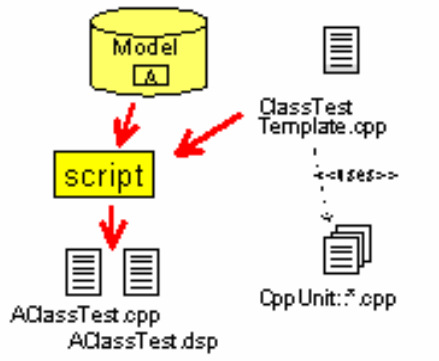


Figura 22 - TestExpert.

De acordo com a Figura 22, baseado nas informações contidas no modelo *Rose*, um *script* extrai classes de código de teste. Esse *script* então preenche o *template* das classes de teste (JUnit e *CppUnit*). O *template* usa as classes internas do JUnit e *cppUnit* para gerar o código de teste automaticamente. Para geração do código de teste, uma ou mais classe do modelo *Rose* devem ser selecionadas e para cada uma destas, as operaçõea a serem testadas. Feito isso, os *drivers* são gerados automaticamente.

Uma vantagem dessa abordagem é a possibilidade de geração da parte estática do *driver* de teste para ambas as ferramentas: JUnit e *CppUnit*. Porém, a parte dinâmica dos casos de teste não é gerada automaticamente. A geração de *stub*s também não é contemplada nessa abordagem.

5.3.2 Scentor

Outra abordagem que utiliza a UML para gerar parcialmente *Drivers* de teste para JUnit é encontrada em [WIT01]. Nesta abordagem, o cenário para geração dos *Drivers* de teste utiliza diagramas de seqüência.

A Figura 23, extraída de [WIT01], ilustra o cenário proposto para geração dos *Drivers* de teste. Primeiramente os casos de teste são modelados com diagramas de seqüência, assumindo que casos de teste são tipicamente baseados em um conjunto de mensagens enviadas por um único objeto do diagrama de seqüência. O modelo UML é então exportado para um arquivo XMI que é carregado para a ferramenta SCENTOR. SCENTOR é uma ferramenta desenvolvida para validar o mapeamento proposto nessa abordagem.Também é possível carregar uma especificação de teste já previamente especificada (no formato XML). É necessário adicionar valores concretos para cada método chamado pelo objeto e especificar

o resultado esperado. Também é necessário especificar os métodos *setUp* e *tearDown*. Finalizando, o *Driver* de teste é compilado e executado.

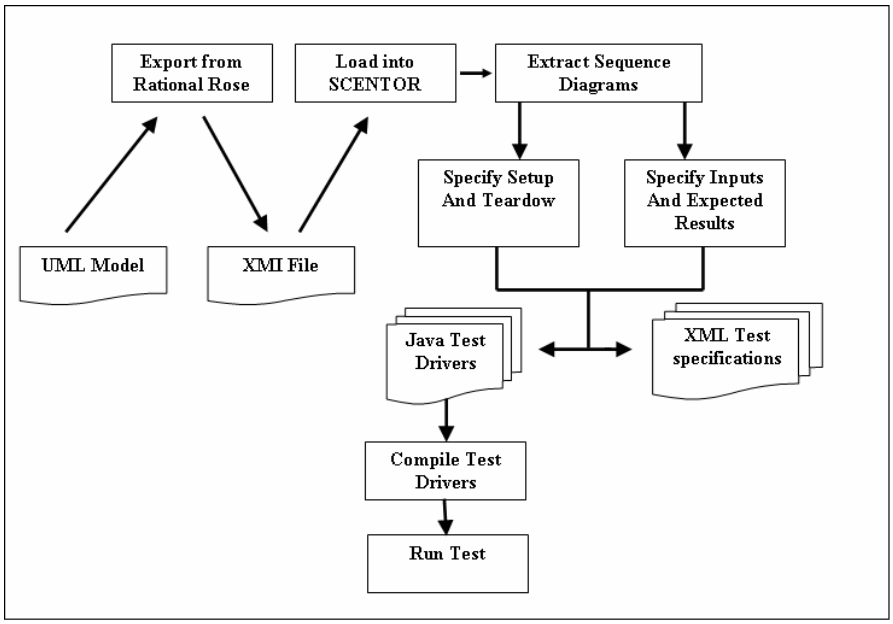


Figura 23 - Cenário para geração parcial dos *Drivers* de teste a partir de Diagramas de Seqüência.

Uma vantagem dessa abordagem é poder gerar junto com os casos de teste, seus respectivos comportamentos, embora isso não seja totalmente automatizado, pois é preciso a interação do usuário para preencher as entradas e as saídas para cada objeto testado. Também é importante ressaltar que esse mapeamento não explora os conceitos específicos de teste do U2TP. Essa abordagem também não contempla a geração de S*tubs* de teste.

5.3.3 Seditec

Em [FRA02], também é proposto um trabalho muito parecido com a abordagem de [WIT01]. Nessa abordagem é apresentado um conceito para geração de testes automatizados para aplicações orientadas a objetos através de uma ferramenta denominada SEDITEC, a qual implementa esses conceitos para aplicações Java. O autor usa diagramas de seqüência para geração de testes automatizados. Nesta abordagem, toda classe e seus respectivos métodos, têm seus comportamentos especificados por um ou mais diagrama de seqüência, o autor descreve sete itens que ele considera fundamental para tornar um diagrama de seqüência "testável", ou seja, para especificações de teste. Os diagramas de seqüência são completados

com um conjunto de dados de casos de teste que consistem de valores de entradas e valores de retornos para os métodos chamados no diagrama. Diagramas de seqüência também podem ser combinados, dessa forma, *Stubs* de teste também podem ser gerados. Porém, a geração de *stub*s nessa abordagem está relacionada com as pré-condições para a execução dos testes. Por exemplo, em um diagrama de seqüência um objeto pode ser criado com seus valores iniciais, e em outro diagrama, este mesmo é testado através da seqüência de mensagens enviadas a ele. Essa abordagem também permite a geração dos casos de teste para a ferramenta JUnit, embora não haja um mapeamento direto destes testes com os conceitos do JUnit. Para validação da abordagem, foi desenvolvida uma ferramenta denominada SEDITEC que suporta a execução de diagramas de seqüência testáveis para aplicações Java. A ferramenta importa extensões de ferramentas CASE, como por exemplo, *Together*[2], ou arquivos no formato XML. Dessa forma é possível: (i) combinar diagramas de seqüência; (ii) especificar um conjunto de dados de valores de entrada para cada diagrama de seqüência e (iii) executar suítes de testes.

O principal objetivo dessa abordagem é introduzir o teste já nas primeiras fases de desenvolvimento do software, através da validação do comportamento das classes especificadas por diagramas de seqüência. Essa abordagem gera os *drivers* de teste com os respectivos comportamentos sem nenhum tipo de interação, embora o *driver* gerado seja específico para a ferramenta SEDITEC. Também como em [WIT01], as especificações de testes não exploram conceitos específicos de teste do U2TP.

5.4 Considerações Finais

O que se pôde observar nos trabalhos descritos neste capítulo, é que eles não oferecem meios eficientes para gerar todo código de teste. Os seguintes critérios foram utilizados para analisar cada uma das propostas descritas:

- *Usa diagramas UML para especificação do teste*: quais diagramas UML são usados?;
- *Interação do Usuário*: é necessário algum tipo de interação do usuário para geração do código de teste?;
- *Geração total do código*: todo código de teste é gerado automaticamente, incluindo *Drivers* e *Stubs,* com o respectivo comportamento dos casos de teste?;

[2] Together é uma ferramenta de modelagem da Borland que tem por finalidade fornecer suporte nas áreas de análise de negócio, projeto, arquitetura e desenvolvimento.

- *Utiliza conceitos específicos do U2TP*: são utilizados conceitos do U2TP para especificação do teste?;
- *Geração para JUnit*: a geração do código tem como alvo a ferramenta JUnit?;
- *Geração de Stubs de teste*: suporta a geração de *stub*s?

Tabela 3 - Comparação entre as abordagens.

Abordagens	Usa diagramas UML?	Interação com usuário?	Geração total do código?	U2TP?	Geração para JUnit?	Geração de *Stubs*?
IDE	Não	Sim	Não, apenas a estrutura	Não	Sim	Não
TestExpert	Sim, diagramas de classe	Sim	Não, apenas a estrutura	Não	Sim	Não
Especificações JML	Não	Sim	Não, é necessário especificar os valores de entrada	Não	Sim	Não
Especificações AOTDL	Não	Sim	Não, é necessário especificar os valores de entrada	Não	Sim	Não
SCENTOR	Sim, diagramas de seqüência	Sim	Parcial, é necessário especificar os valores de entrada	Não	Sim	Não
SEDITEC	Sim, diagramas de seqüência	Não	Sim	Não	Não	Sim

Através da análise da Tabela 3, as seguintes considerações podem ser formuladas:

- A maioria das abordagens usam diagramas da UML para geração do código, exceto as IDEs e especificações JML e AOTDL. Todas abordagens requerem algum tipo de

interação do usuário para gerar o código de teste automaticamente, exceto em SEDITEC, onde a ferramenta gera automaticamente todo código de teste;

- Apenas em SEDITEC é gerado todo o *driver* de teste, incluindo o comportamento dos casos de teste e *stub*s de teste, contudo, esta não enfoca a ferramenta JUnit;
- Nenhuma das abordagens descritas utiliza os conceitos próprios do U2TP para geração do código de teste;

Assim, uma vantagem do presente trabalho, não encontrada em nenhum trabalho descrito anteriormente, é a possibilidade de gerar todo código de teste incluindo os *Drivers* e *Stubs* sem requerer nenhum tipo de interação do usuário. Outra vantagem é que o *stub* de teste gerado no presente trabalho é somente o necessário para a execução do teste, diferente de SEDITEC, que gera *stubs* representando apenas pré-condições para a execução dos testes.

6 DESCRIÇÃO DA PROPOSTA

A automação de testes já é uma realidade atualmente no processo de teste, pois ajuda a reduzir o tempo gasto para execução dos testes. Ferramentas para automação de testes estão bem difundidas, especialmente ferramentas para teste de unidade. As ferramentas da família *XUnit* vêm sendo cada vez mais utilizadas no processo de teste unitário, pois ajudam a diminuir o tempo gasto para execução dos testes e possibilitam a verificação automatizada dos resultados. A ferramenta JUnit em particular, foi adaptada a uma variedade de IDEs que facilitam seu uso. Entretanto, um problema comum encontrado nessas ferramentas é a necessidade de codificação dos *drivers* e *stubs* de teste, pois muito custo e esforço ainda são gastos para codificar esses elementos, o que muitas vezes inviabiliza o uso dessas ferramentas.

Uma forma de resolver esse problema é a possibilidade de gerar esses elementos automaticamente a partir de algum tipo de especificação em mais alto nível. O U2TP, por possuir um conjunto de elementos e conceitos específicos para teste, permite o mapeamento para diferentes ferramentas como JUnit e TTCN-3. Além de permitir esse mapeamento, ele também permite representar o processo de teste em um nível mais alto de abstração, através de notações UML. Com isso, é possível usar esta notação para representação de testes voltados a diferentes linguagens de programação, além de favorecer o entendimento e a documentação dos artefatos de teste.

Sendo assim, o objetivo desse trabalho é gerar automaticamente *drivers* e *stubs* de teste para ferramenta de teste unitário JUnit a partir de especificações de teste unitário modeladas com a U2TP.

As contribuições deste trabalho incluem:

- Diminuir o custo e esforço gasto para codificação de *drivers* e *stubs* de teste, uma vez que esses elementos são gerados automaticamente;
- Fortalecer o uso do U2TP para especificação de teste, especialmente testes unitários;
- Aumentar a qualidade do código de teste gerado, uma vez que a especificação do teste é feita em um maior nível de abstração (UML);

- Fortalecer a abordagem sugerida na literatura para realização de teste unitário [BUR03], a qual difere a pessoa que especifica e projeta o teste (engenheiro de teste), daquela que executa o teste (testador). Isto diminui problemas graves como: falta de documentação dos artefatos de teste e falta de conhecimento do programador para especificação do teste.

Com base nos objetivos descritos anteriormente, esse capítulo apresenta a proposta do presente trabalho.

6.1 Visão Geral

A Figura 24 ilustra a visão geral da proposta do presente trabalho.

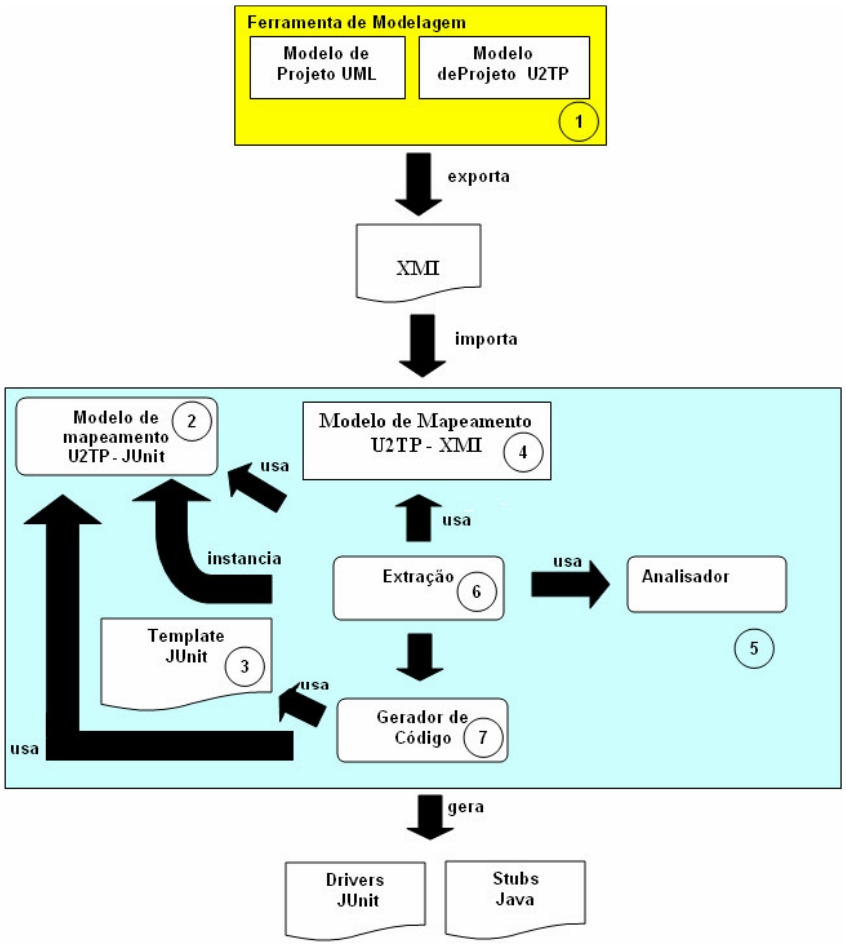


Figura 24 - Visão Geral da propsota.

Como podemos observar na Figura 24, como entrada para a geração do código tem-se o modelo de projeto do sistema em UML e o modelo de projeto de teste em U2TP (Figura 24,

item 1). Os modelos de projeto e de teste devem respeitar uma série de pressupostos assumidos neste trabalho, discutidos na Seção 6.2. Esses dois modelos são gerados a partir de uma ferramenta de modelagem. Esta deve dar suporte à importação/exportação de documentos XMI, tanto para diagramas estruturais como comportamentais da UML. A ferramenta exporta um documento XMI representando estes dois modelos. A geração dos *drivers* e *stubs* correspondentes é realizada em duas etapas, Extração e Geração. A extração (Figura 24, item 6) é baseada em dois modelos adotados neste trabalho, a saber Modelo de Mapeamento U2TP-JUnit e Modelo de Mapeamento U2TP-XMI. O Modelo de Mapeamento U2TP-JUnit (Figura 24, item 2) define quais elementos do U2TP são necessários para geração de *drivers* e *stubs* para ferramenta JUnit. O Modelo de Mapeamento U2TP-XMI (Figura 24, item 4) 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 no documento XMI recebido como entrada e juntamente com o Analisador (Figura 24, item 5) captura todos elementos que comporão os *drivers* e *stubs* para a ferramenta JUnit. Então, o Gerador de Código (Figura 24, item 7) gera os elementos seguindo o template do JUnit (Figura 24, item 3). Como saída tem-se a geração automatizada de um *driver* para ferramenta JUnit e *stubs* de teste na linguagem Java. É importante ressaltar que o processo requer o mínimo de interação com o usuário e a geração do código e totalmente automatizada.

As próximas seções descrevem detalhadamente cada um das etapas da Figura 24.

6.2 Pressupostos para Modelagem

Este trabalho adota a metodologia de [DAI03] (Seção 4.10). Além disto, define quais são os pressupostos necessários para especificar e modelar teste em nível de unidade com o U2TP a fim de gerar automaticamente os *drivers* e *stubs* de teste. Estes pressupostos estão divididos em dois grupos: Projeto de Software e Projeto de teste.

6.1.1 Pressupostos para Projeto de Software

Assume-se a existência de um modelo de projeto UML correspondente ao que será testado. Este deve conter obrigatoriamente um diagrama de classes em nível de projeto, pois a partir deste, métodos de classes podem ser testados. Outros diagramas são opcionais

(diagramas de caso de uso expandido, diagramas de seqüência e diagramas de atividade). Entretanto, se cada classe do modelo de projeto tiver seu comportamento especificado por diagramas de seqüência, estes poderão ser utilizados para gerar *Stubs* de teste. Também, deve-se mencionar que para especificar os casos de teste, embora não seja obrigatório, é interessante que o modelo de projeto de sistema contenha diagramas de casos de uso expandido, para a partir destes, derivar os métodos mais relevantes a serem testados.

6.1.2 Pressupostos para o Projeto de Teste

Assume-se a definição de um pacote de teste. Na explanação que segue, será apresentado um exemplo baseado em um modelo de projeto de software extraído de [LAR00]. Em seguida, esse modelo de projeto de software será transformado em um modelo de projeto de teste, sobre o qual serão discutidos e exemplificados os pressupostos assumidos. A Figura 25 ilustra o pacote Vendas do sistema.

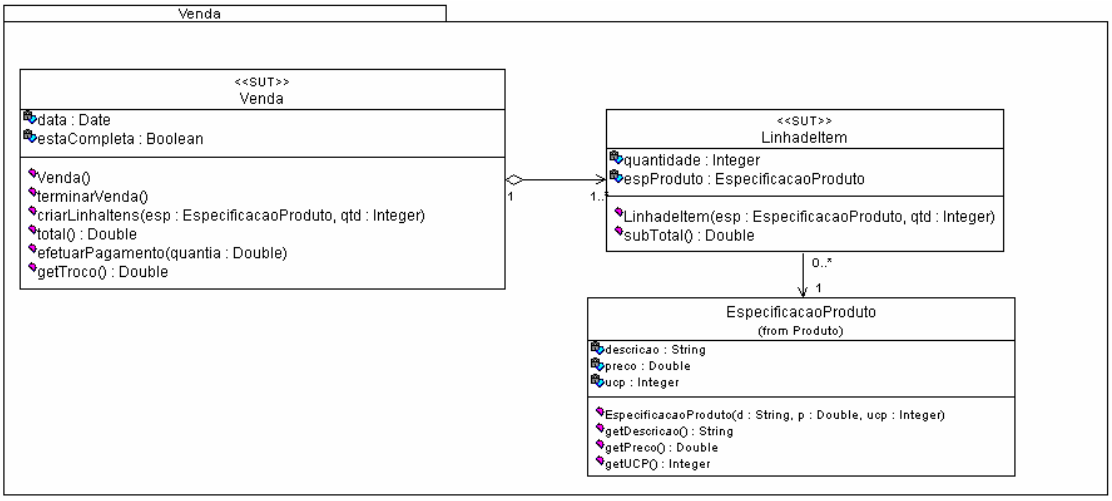


Figura 25 - Pacote Vendas.

6.2 Arquitetura de Teste

Os elementos especificados para o grupo de arquitetura de teste são especificados no modelo de projeto de software (*Sut* e *TestComponent*) e no modelo de projeto de teste (*TestContext*, *TestCase*, *TestControl* e *TestComponent*).

6.2.1 Sut

Todas as classes do modelo de projeto de software que serão testadas devem ser estereotipadas com <<Sut>>, como no projeto ilustrado na Figura 25.

6.2.2 TestContext, TestCase e TestComponent

Deve haver uma e somente uma classe *TestContext* estereotipada com <<testcontext>>

Deve haver no mínimo um caso de teste especificado dentro da classe, cada qual representado através de uma operação estereotipada com <<testcase>>.

A Figura 26 ilustra o novo pacote para teste unitário denominado TestVenda. No pacote foi especificado um *testcontext*, denominado TestVenda com três casos de teste: testSubtotal, testTotal e testEfetuarPagamento e um *testcomponent* denominado EspProduto.

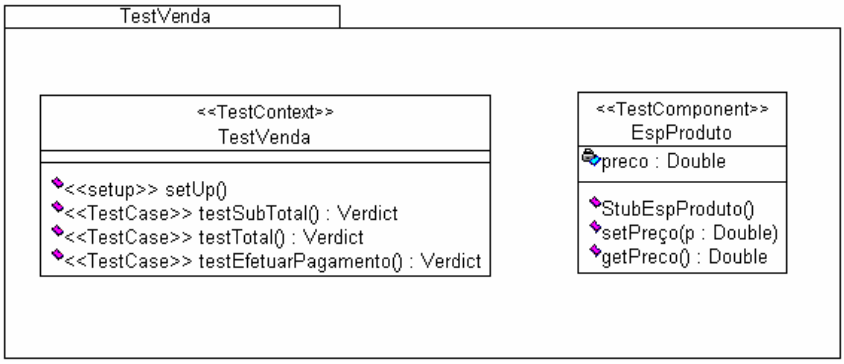


Figura 26- *TestContext* e *TestCase*.

O elemento *TestComponent* pode ser especificado tanto no modelo de Projeto de Software como no modelo de Projeto de Teste:

a) Projeto de Teste: se um caso de teste necessitar de uma funcionalidade que não esteja definida nos *Sust* (exemplo do caso de teste testSubTotal), pode-se especificar uma nova classe (desde que a mesma não esteja no projeto de software) com o mínimo de funcionalidades necessárias para auxiliar na execução de um ou mais casos de teste. Esta classe também deve ser estereotipada com <<testcomponent>>, como ilustrado na Figura 26. Somente o atributo preço foi definido, juntamente com seus métodos acessores e modificadores. Vale ressaltar que para o exemplo da Figura 25, não é necessário especificar o elemento *TestComponent*, pois a classe

EspecificacaodeProduto já está no modelo de projeto de software e pode ser usada para auxiliar na execução do caso de teste testSubTotal. Assim, o *testcomponent* foi especificado no exemplo da Figura 26 somente para descrever o contexto em que ele poderia ser usado.

b) Projeto de Software: pode-se estereotipar classes do modelo de projeto de software com <<testcomponent>> toda vez que a mesma for usada para auxiliar na execução de um ou mais casos de. Dessa forma, a classe EspecificacaodeProduto da Figura 25 também poderia ser um *testcomponent*, e poderia ser estereotipada com <<testcomponent>>. Neste exemplo, ela não foi especificada no modelo de projeto de software porque a classe EspecificacaodeProduto será interpretada como um *Stub* de teste.

Cabe salientar que se no modelo de projeto de sistema não tiver diagramas de seqüência, não será possível gerar os *Stubs* de teste, exceto no caso em que o mesmo é gerado a partir do elemento *TestComponent*.

6.2.3 TestControl

O elemento *TestControl*, se adotado, deve ser especificado através de um diagrama de atividades. A Figura 27 ilustra um exemplo de como usar esse elemento.

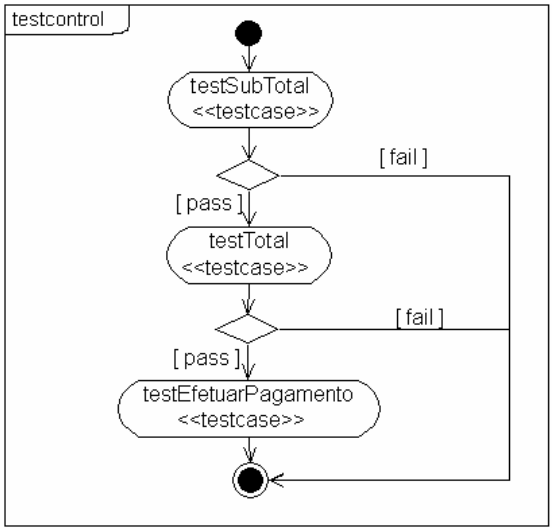


Figura 27 - Especificando o elemento *TestControl*.

Cada atividade do diagrama deve conter o nome igual ao respectivo caso de teste. Também devem possuir o estereótipo <<testcase>>.

Se o elemento *TestControl* não for adotado, os casos de testes serão executados na mesma ordem em que eles foram definidos na classe *TestContext*.

6.3 Comportamento de Teste

Para o grupo de comportamento de teste são especificados os elementos *testcase* e *verdict*, os quais são especificados no modelo de projeto de teste.

6.3.1 TestCase

Para especificar o comportamento de um caso de teste devem ser usados diagramas de seqüência. A Figura 28 ilustra o comportamento para o caso de teste *subTotal* da classe LinhadeItemVenda.

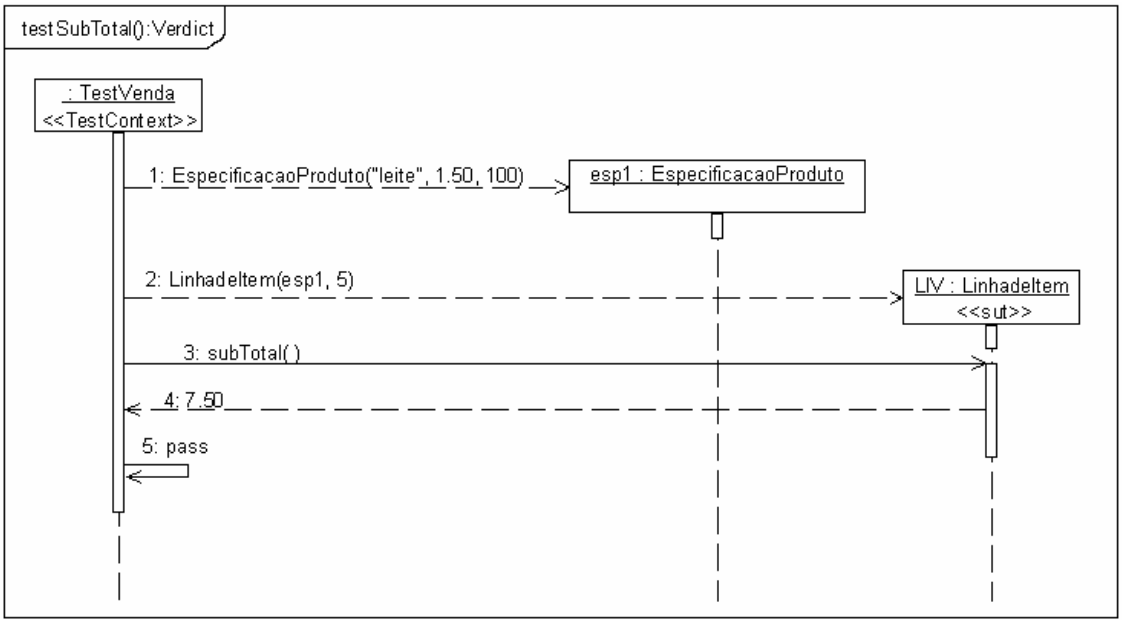


Figura 28 - Comportamento do caso de teste testSubTotal.

Os pressupostos necessários para modelar o comportamento de um caso de teste através de um diagrama de seqüência são descritos abaixo:

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*, pois essa classe contém todos os casos de teste associados e ela será responsável por atribuir o *Verdict* final para o caso de teste.

3. Os demais objetos do diagrama de seqüência são instâncias das classes do tipo *Sut*, *TestComponent* ou de qualquer classe do modelo de projeto, que será interpretada com um *TestComponent*. Estas instâncias devem ser nomeadas.
4. As mensagens representadas por setas pontilhadas partindo do objeto *TestContext* 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 autodelegação ao objeto *TestContext* com a mensagem *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.
7. É possível declarar uma mensagem onde uma variável recebe o retorno de uma operação da seguinte forma:
    a. <variável> := <operação>

Assume-se que pode haver diagramas de seqüência que representam as pré e pós-condições para todos os casos de teste, como descrito na Seção 6.3.2.

6.3.2 Pré e Pós Condições Comuns aos Casos de Teste

Quando as pré-condições forem comuns a todos os casos de teste, é possível representá-las em um diagrama de seqüência separado. Para isso, é necessário definir uma nova operação na classe *TestContext* e estereotipá-la com <<Setup>>, como ilustrado na Figura 25. Este estereótipo não é fornecido no U2TP. Neste caso é necessário nomear o nome do diagrama de seqüência com o mesmo nome da operação, nesse exemplo, *setUp*. Duas pré-condições necessárias para a execução do caso de teste *testSubTotal* são ilustrados na Figura 29.

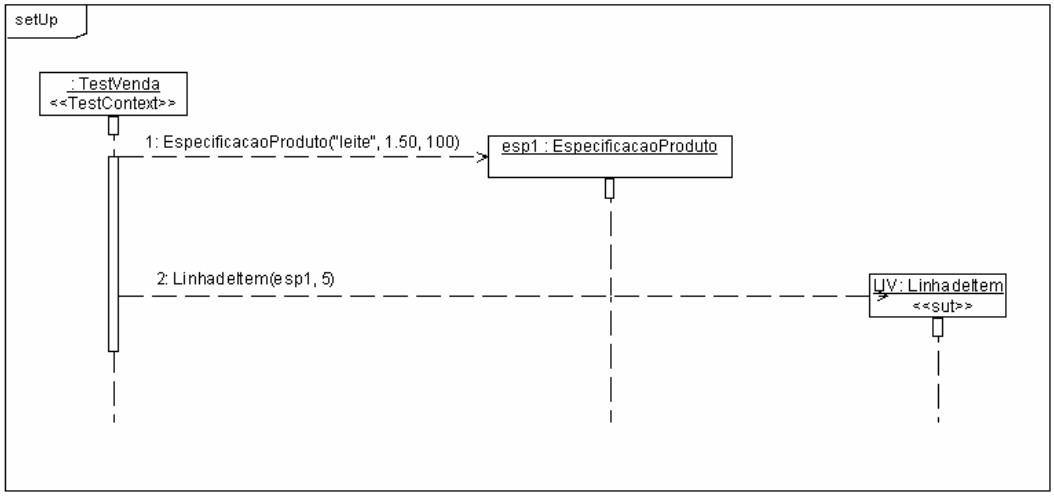


Figura 29 - Definindo as pré-condições.

É importante ressaltar que através das novas notações da UML 2.0, é possível representar as pré-condições com referências a outros diagramas de seqüência. Mas como a maioria das ferramentas de modelagem atuais ainda não suportam a UML 2.0, essa notação não foi adotada neste trabalho. A Figura 30 mostra esta notação para o comportamento do caso de teste testSubtotal (Figura 28) com referência ao diagrama da Figura 29.

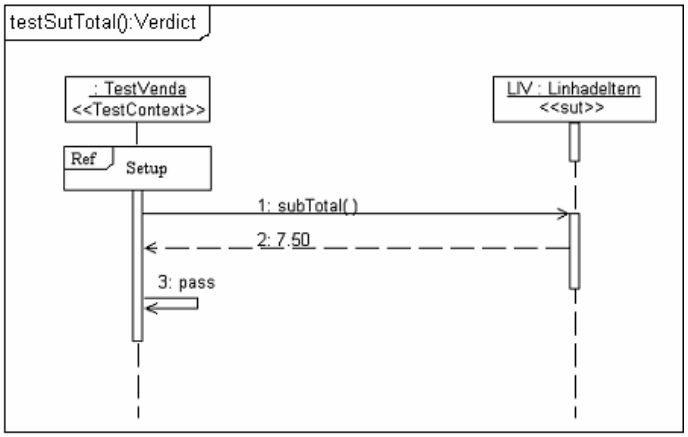


Figura 30 - Comportamento do caso de teste subTotal usando uma referência a outro diagrama de seqüência.

Também é possível representar as pós-condições para os casos de teste. Da mesma forma que as pré-condições, é preciso definir uma nova operação na classe *TestContex,* e estereotipá-la com <<teardown>> (estereotipo também não fornecido pelo U2TP). As pós-condições também são representadas por um diagrama de seqüência que também deve ter o mesmo nome da operação estereotipada com <teardown> associado ao diagrama.

6.3 Mapeamento dos Elementos da U2TP para JUnit

A Seção 4.8 descreve quais elementos do U2TP são apropriados para especificação do teste unitário. Este trabalho adota os elementos pertencentes à arquitetura de teste (*TestContext*, *Sut*, *Testcomponent*, *TestCase* e *TestControl*) e Comportamento de Teste (*TestCase* e *Verdict*).

O grupo Dados de Teste, embora importante, não foi adotado neste trabalho pela complexidade de gerar esses elementos no código correspondente à ferramenta JUnit.

A Tabela 2, apresentada no Capítulo 4, mostra como os elementos do U2TP são mapeados para os conceitos correspondentes na ferramenta JUnit. Além destes, foram acrescentados neste trabalho dois novos elementos ao U2TP para especificação de testes, a saber *Configuration* e *Stub*.

O diagrama de classes da Figura 31 apresenta todos os elementos usados para geração de código no JUnit. As classes na cor branca representam os conceitos adotados do U2TP e as classes em cor escura representam os novos elementos.

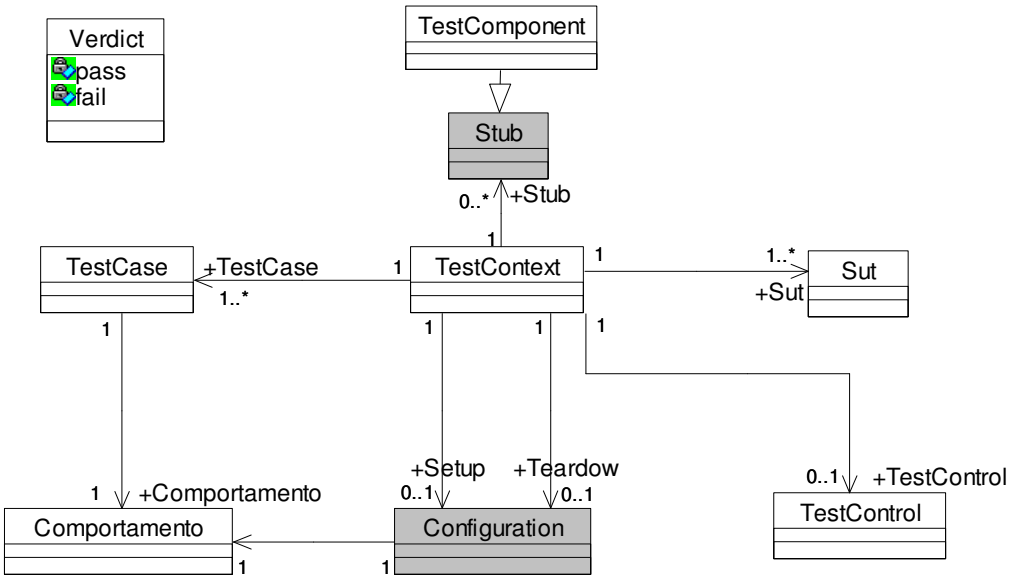


Figura 31 – Modelo de mapeamento do U2TP para JUnit.

Para cada elemento apresentado na Figura 32, a Tabela 4 descreve como esses elementos são representados no JUnit.

Tabela 4 - Representação dos elementos no JUnit.

U2TP	JUnit
TestContext	é representado como uma subclasse de *TestCase* do JUnit.
Sut	é representado como um atributo, uma variável de instância no JUnit.
TestComponent	é representado como um atributo, uma variável de instância no JUnit.
Setup	representa todos os objetos criados dentro do método *setUp*.
Teardown	representa a finalização dos objetos dentro do método *Teardown*.
TestCase	é representado como um método no JUnit. Esse método deve obrigatoriamente iniciar com a *String* "test".
TestControl	é representado no JUnit sobrecarregando o método *runTest*.
Verdict	é representado através de assertivas usadas nos casos de teste.
Comportamento	define o comportamento de cada caso de teste, e também o comportamento do elemento *SetUp e Teardown*.
Stub	é representado como um atributo, uma variável de instância no JUnit.

Como podemos observar pela Tabela 4, cada elemento do U2TP é representado de uma maneira no JUnit. Assim, para fazer o mapeamento desses elementos para o código correspondente no JUnit, é usado um *template* que define a representação da estrutura do *Driver* do JUnit.

Estes elementos foram adotados para este trabalho por possuírem esse mapeamento direto para o conceitos do JUnit. Entretanto, os elementos pertencentes ao grupo Dados de Teste (*DataPool* e *DataPartition*) não foram definidos no Modelo de Mapeamento para ferramenta JUnit. O motivo pelo qual não foi possível fazer o mapeamento destes elementos para o código equivalente a ferramenta JUnit foi a complexidade para gerar os mesmos.

6.4 Template do JUnit

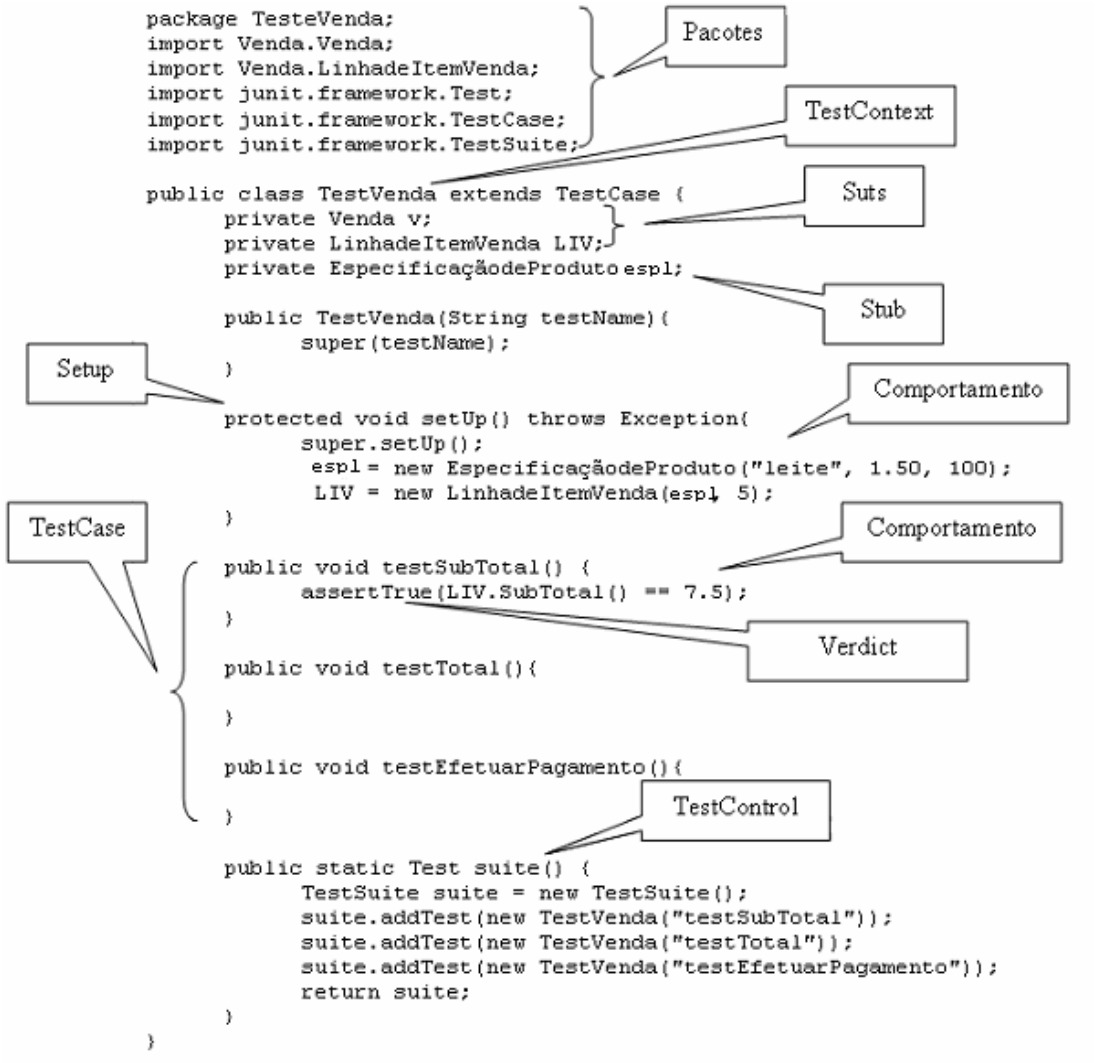
A Figura 32 descreve o *template* do JUnit definido neste trabalho.

```
(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 32 - Template do JUnit.

De acordo com o template da Figura 32, a linha 1 representa o pacote do elemento *TestContext*. A linha 2 representa o pacote do próprio JUnit. As linhas 3, 4 e 5 representam os pacotes das classes *Sut*, *TestComponent* e *Stub* respectivamente. As linhas 7 e 8 representam os pacotes do JUnit (esses pacotes são gerados se o elemento *testcontrol* for especificado). A linha 9 representa o elemento *TestContext*. As linhas 10, 11 e 12 representam a declaração dos elementos *Sut*, *TestComponent* e *Stub* respectivamente (somente o elemento *Sut* é obrigatório). As linhas 13 a 16 representam o método construtor do *TestContext* que também só é gerado se o elemento *testcontrol* for especificado. As linhas 16 a 20 representam o método *Setup* do JUnit. Da mesma forma, as linhas 21 a 24 representam o método *Teardown* do JUnit. As linhas 25 a 29 representam um caso de teste. Dentro do corpo do caso de teste, as linhas 26 a 28 representam seu comportamento. Finalizando, as linhas 30 a 35 representam o elemento *TestControl*, que no JUnit é implementado através do método TestSuite.

O *driver* de teste apresentado na Figura 33 ilustra a representação dos elementos descritos na Tabela 4 para o código correspondente para o JUnit baseado neste *template*, considerando o exemplo do sistema Venda apresentados nas Figuras 25, 26, 27, 28 e 29.




```
package TesteVenda;
import Venda.Venda;
import Venda.LinhadeItemVenda;
import junit.framework.Test;
import junit.framework.TestCase;
import junit.framework.TestSuite;

public class TestVenda extends TestCase {
      private Venda v;
      private LinhadeItemVenda LIV;
      private EspecificaçãodeProduto espl;

      public TestVenda(String testName){
            super(testName);
      }

      protected void setUp() throws Exception{
            super.setUp();
             espl = new EspecificaçãodeProduto("leite", 1.50, 100);
             LIV = new LinhadeItemVenda(espl, 5);
      }

      public void testSubTotal() {
            assertTrue(LIV.SubTotal() == 7.5);
      }

      public void testTotal(){

      }

      public void testEfetuarPagamento(){

      }

      public static Test suite() {
            TestSuite suite = new TestSuite();
            suite.addTest(new TestVenda("testSubTotal"));
            suite.addTest(new TestVenda("testTotal"));
            suite.addTest(new TestVenda("testEfetuarPagamento"));
            return suite;
      }
}
```

Figura 33 - Mapeamento do exemplo Sistema Venda para o template do JUnit.

O código correspondente ao elemento *Stub* é ilustrado na Figura 34.

```
public class EspecificaçãodeProduto {
	private String descricao;
	private double preco;
	private int ucp;
	public EspecificaçãodeProduto(String d, double p, int ucp) {
		this.descricao = d;
		this.preco = p;
		this.ucp = ucp;
	}
	public String getDescricao(){
		return descricao;
	}
	public double getPreco(){
		return preco;
	}
	public int getUcp(){
		return ucp;
	}
}
```

Figura 34 - Código *Stub*.

6.5 Modelo de Mapeamento dos Elementos U2TP para XMI

Esta seção apresenta o modelo de mapeamento dos elementos do U2TP para o documento XMI.

Um documento XMI - *XML Metadata Interchange* é um tipo de arquivo XML - *Extensible Markup Language* que permite a codificação de modelos UML em XML. É baseada no MOF e ambos são padrões adaptados pela OMG. Um documento XMI utiliza o MOF como metamodelo e XML como linguagem padrão para formatação dos dados a serem transferidos, mapeando de MOF para XML. Como a UML também é baseada na MOF, o XMI apresenta-se como um formato de intercâmbio [OMG05]. Portanto, um documento XMI é um documento escrito em XML, que utiliza *tags* definidas pela especificação do XMI para representar as informações.

Um documento XMI possui um arquivo DTD - *Document Type Data* (definição do tipo de documento) o qual define a sua estrutura. Cada DTD usado por um documento XMI deve satisfazer os seguintes requisitos [OMG05]:

- Todo elemento XML definido por uma especificação XMI deve ser declarado em um DTD;
- Cada construção de metamodelo (classe, atributo e associação) deve ter uma declaração de elemento correspondente;

- Qualquer elemento XML que represente extensões do metamodelo deve ser declarado em um DTD externo ou interno.

A partir do XMI, foram identificados quais elementos XML estavam envolvidos na descrição dos elementos utilizados neste trabalho tanto do modelo de projeto de sistema quanto no modelo de projeto de teste com o U2TP e extensões propostas. Estes são apresentadas na Figura 35.

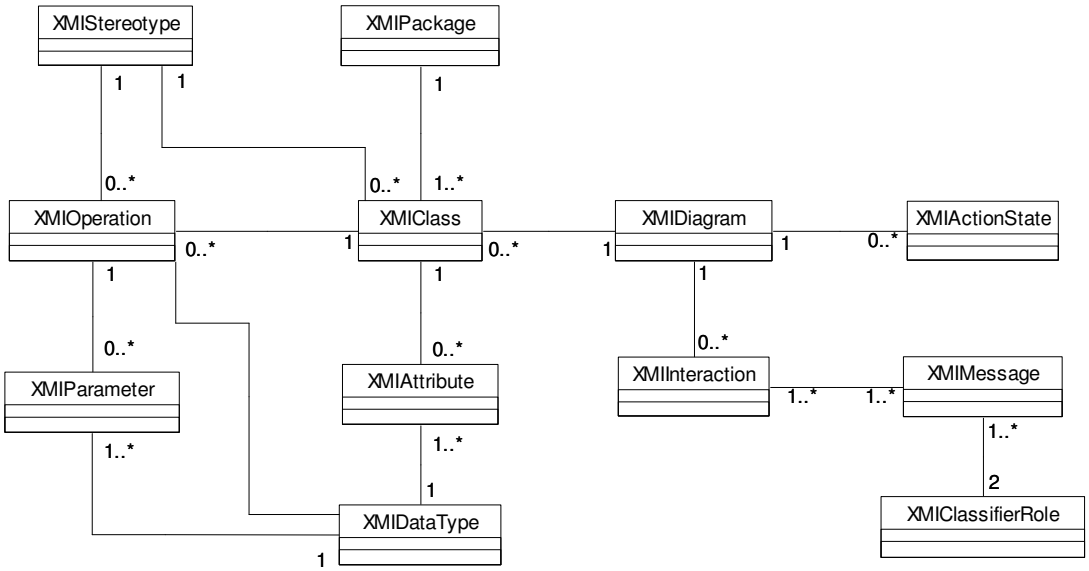


Figura 35 - Modelo de mapeamento dos elementos do U2TP para XMI.

O modelo apresentado na Figura 35 descreve todas as *tags* (rótulos) e seus relacionamentos que são usados para mapear os elementos do U2TP. A Tabela 5 descreve como os elementos do U2TP são mapeados para o modelo XMI.

Tabela 5 - Mapeamento da U2TP para XMI.

Elementos U2TP e Extenções	XMI	Descrição
Sut	XMIStereotype	identifica o elemento *Sut*
	XMIClass	representa o nome do *Sut*
	XMIPackage	representa o nome do pacote do *Sut*
	XMIClassifierRole e XMIDiagram	representa as instâncias do *Sut*
TestContext	XMIStereotype	identifica o elemento *TestContext*
	XMIClass	representa o nome do *TestContext*
	XMIPackage	representa o nome do pacote do *TestContext*
TestComponent	XMIStereotype	identifica o elemento *TestComponent*
	XMIClass	representa o nome do *TestComponent*
	XMIPackage	representa o nome do pacote do *TestComponent*
	XMIClassifierRole	representa as instâncias do *TestComponent*
Stub	XMIStereotype	identifica o elemento *TestComponent*
	XMIClass	representa o nome do *TestComponent*
	XMIPackage	representa o nome do pacote do *TestComponent*
	XMIClassifierRole	representa as instâncias do *TestComponent*
TestCase	XMIStereotype	identifica o elemento *Testcase*
	XMIOperation	representa o nome do *Testcase*
	XMIInteraction	representa a interação do *Testcase* (diagrama de seqüência)
	XMIMessage	representa as mensagens do diagrama de seqüência
	XMIClassifierRole	Mapeia os objetos destinatários das mensagens
Setup e *Teardown*	XMIStereotype	identifica o elemento *Setup* e *Teardown*
	XMIOperation	representa o identificador da mensagem do *Setup* e *Teardown*
	XMIInteraction	representa a interação do *Setup* (diagrama de seqüência)
	XMIMessage	representa as mensagens do diagrama de seqüência
	XMIClassifierRole	representa os objetos destinatários das mensagens
TestControl	XMIActionState	representa o nome das atividades do diagrama de atividades

7 EXTRATOR E GERADOR PARA JUNIT

Este capítulo descreve a transformação dos elementos contidos no documento XMI para os conceitos do JUnit. Essa transformação é composta por algoritmos extratores que têm por objetivo mapear os elementos do JUnit para classes e, através do gerador de código, gerar os *drivers* e *stubs* de teste. Primeiramente será apresentado o analisador (*parser*) usado neste trabalho, depois será apresentada uma descrição dos elementos do modelo XMI. Finalizando o capítulo, é apresentado o funcionamento do extrator de código, dos algoritmos de mapeamento e do gerador do código.

7.1 Analisador

Um analisador, também conhecido como *parser*, faz a análise em um determinado conteúdo e/ou documento para ver se ele está de acordo com as regras de uma determinada linguagem e/ou estrutura. Em arquivos XML, um *parser* verifica se a sintaxe está correta, ou seja, se as *tags* estão montadas de forma correta.

Atualmente para manipulação de documentos XML os *parsers* mais usados atualmente são: *DOM*, *SAX*, *STaX*, *JAXB* e *Beans* [DOE05].

Para este trabalho foi usado o parser DOM - *Document Object Model*. Ele é o mapeamento para Java do padrão DOM do W3C. Como os elementos do JUnit estão distribuídos por todo o documento XMI sem nenhuma ordem pré-definida, o parser DOM permite armazenar esses elementos e um modelo orientado a objetos representando os mesmos em memória para que estes possam ser manipulados. Assim a idéia básica é transformar um documento XML em objetos. O analisador DOM transforma o documento em uma árvore de "nós" (*nodos*) que reproduz a estrutura de itens do documento. A Figura 36, adaptada de [W3C05], ilustra o funcionamento básico do analisador DOM.

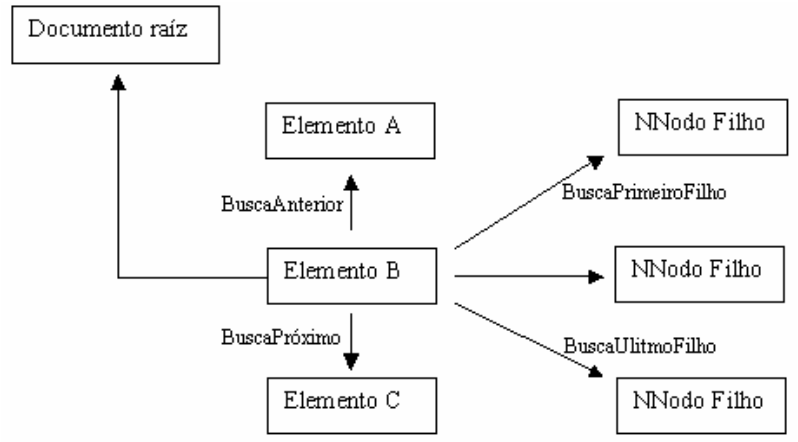


Figura 36 - Funcionamento básico do parser DOM.

Conforme ilustra a Figura 36, vários métodos na interface *Nodo* do DOM define relacionamentos entre os nodos que promovem formas de movimentação pelas conexões entre eles. No exemplo da Figura 37 é ilustrado um nodo (objeto) equivalente a *tag* <UML:Class> e seus relacionamentos.

```
- <UML:Package xmi.id="S.298.1247.52.25" name="TesteVenda" visibility="public" isSpecification="false" isRoot="false"
    isLeaf="false" isAbstract="false" namespace="G.0">
  - <UML:Namespace.ownedElement>
      <!--  ==================== VendaSetub::TesteVenda::TestVenda    [Class] ====================   -->
    - <UML:Class xmi.id="S.298.1247.52.26" name="TestVenda" visibility="public" isSpecification="false" isRoot="true"
        isLeaf="true" isAbstract="false" isActive="false" namespace="S.298.1247.52.25">
      - <UML:Classifier.feature>
          <!--  ==================== VendaSetub::TesteVenda::TestVenda::testSubTotal    [Operation]
          ====================   -->
        - <UML:Operation xmi.id="S.298.1247.52.27" name="testSubTotal" visibility="public" isSpecification="false"
            ownerScope="instance" isQuery="false" concurrency="sequential" isRoot="false" isLeaf="false"
            isAbstract="false" specification="">
          - <UML:BehavioralFeature.parameter>
              <UML:Parameter xmi.id="XX.26.1247.53.21" name="testSubTotal.Return" visibility="public"
                isSpecification="false" kind="return" type="G.102" />
            </UML:BehavioralFeature.parameter>
          </UML:Operation>
```

Figura 37 - Relacionamento entre os nodos.

Note que o nodo <UML:Class>, tem como pai o nodo <UML:Package> e como filho o nodo <UML:Operation>. Se existir mais de uma *tag* "<UML:Class>" no documento XMI, cada *tag* corresponde a um *nodo* (objeto). Dessa forma, um conjunto de *nodos* relacionados a uma mesma *tag* forma uma coleção de *NodeList*, ou seja, uma lista de *nodos* rotulados com uma dada *tag*. Por exemplo, uma NodeListPackage é uma lista de nodos Node Package.

7.2 Descrição dos Elementos do Modelo XMI

Esta seção descreve os elementos do modelo XMI apresentado na Seção 6.5, ou seja, serão descritos quais atributos serão utilizados para capturar os elementos do documento XMI. Assim, cada *tag* juntamente com os atributos usados para capturar os elementos são descritos a seguir:

- <UML:Package>: representa todos os nodos pacotes declarados no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
xmi.id	identificador do pacote	NodePackage	NodeListPackage
name	nome do pacote		
namespace	identificador do pacote o qual o pacote pertence		

- <UML:Class>: representa todos os nodos classes declarados no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
xmi.id	identificador da classe	NodeClass	NodeListClass
name	nome da classe		
namespace	identificador do pacote à qual a classe pertence		

- <UML:Diagram>: representa todos os nodos diagramas declarados no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
xmi.id	identificador do diagrama	NodeDiagram	NodeListDiagram
name	nome do diagrama		
diagramType	tipo de diagrama		
onwer	identificador dos objetos declarados dentro do diagrama		

- <UML:DiagramElement>: representa todos os elementos filhos do nodo Diagram. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
subject	identificador objetos do diagrama	NodeDiagramElement	NodeListDiagramElement

- <UML:Interaction>: representa todos os nodos interações dos diagramas de seqüência definidos no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
xmi.id	identificador da interação do diagrama (diagrama de seqüência)	NodeInteraction	NodeListInteraction
name	nome da interação (nome do diagrama de seqüência)		

- <UML:ClassifierRole>: representa todos nodos objetos declarados nos diagramas de seqüência definidos no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
xmi.id	identificador do objeto	NodeClassifierRole	NodeListClassifierRole
name	nome do objeto		
base	identificador da classe que o objeto instancia		

- <UML:Message>: representa todos os nodos mensagens dos diagramas de seqüência definidos no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
name	nome da mensagem	NodeMessage	NodeListMessage
sender	identificador do objeto que envia a mensagem		
receiver	identificador do objeto que recebe a mensagem		

- <UML:Stereotype>: representa todos os nodos estereótipos declarados no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
xmi.id	identificador do estereotipo	NodeStereotype	NodeListStereotype
name	nome do estereotipo		
extendElement	Identificador da classe e/ou operação que o estereotipo estende		

- <UML:DataType>: representa todos os nodos tipos de dados declarados no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
xmi.id	identificador do tipo de dados	NodeDataType	NodeListDataType
name	nome do tipo de dados		
visibility	visibilidade do tipo de dados		

- <UML:Operation>: representa todos os nodos operações de classes declaradas no documento XMI. A tabela abaixo os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
xmi.id	identificador da operação	NodeOperation	NodeListOperation
name	nome da operação		
visibility	visibilidade da operação		

- <UML:Parameter>: representa todos os nodos parâmetros das operações definidos no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
xmi.id	identificador do parâmetro	NodeParameter	NodeListParameter
name	nome do parâmetro		
kind	esse atributo tem dois tipos: inout (não retorna nada, é um void) e return (retorna alguma coisa).		
type	identificador do tipo de dados do parâmetro		

- <UML:Attribute>: representa todos nodos atributos das classes definidos no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
xmi.id	identificador do atributo	NodeAttribute	NodeListAttribute
name	nome do atributo		
type	identificador do tipo de dados do atributo		
visibility	visibilidade do atributo		

- <UML:ActionState>: representa todos os diagramas de atividades declarados no documento XMI. A tabela abaixo ilustra os atributos usados deste nodo:

Atributos Usados	Descrição	Nodo	Lista de Nodos
name	nome da atividade	NodeActionState	NodeListActionState

Cabe salientar que os algoritmos de mapeamento desenvolvidos são dependentes dos documentos XMI gerados pela ferramenta de modelagem *Rationale Rose* e também do analisador DOM.

Para fazer o mapeamento dos elementos para o código, foi definido um modelo de classes em nível de projeto. Cada classe do diagrama é responsável por armazenar os elementos do JUnit, exceto as classes *Algoritmo* e *Parser*. A Figura 38 descreve o diagrama de classes.

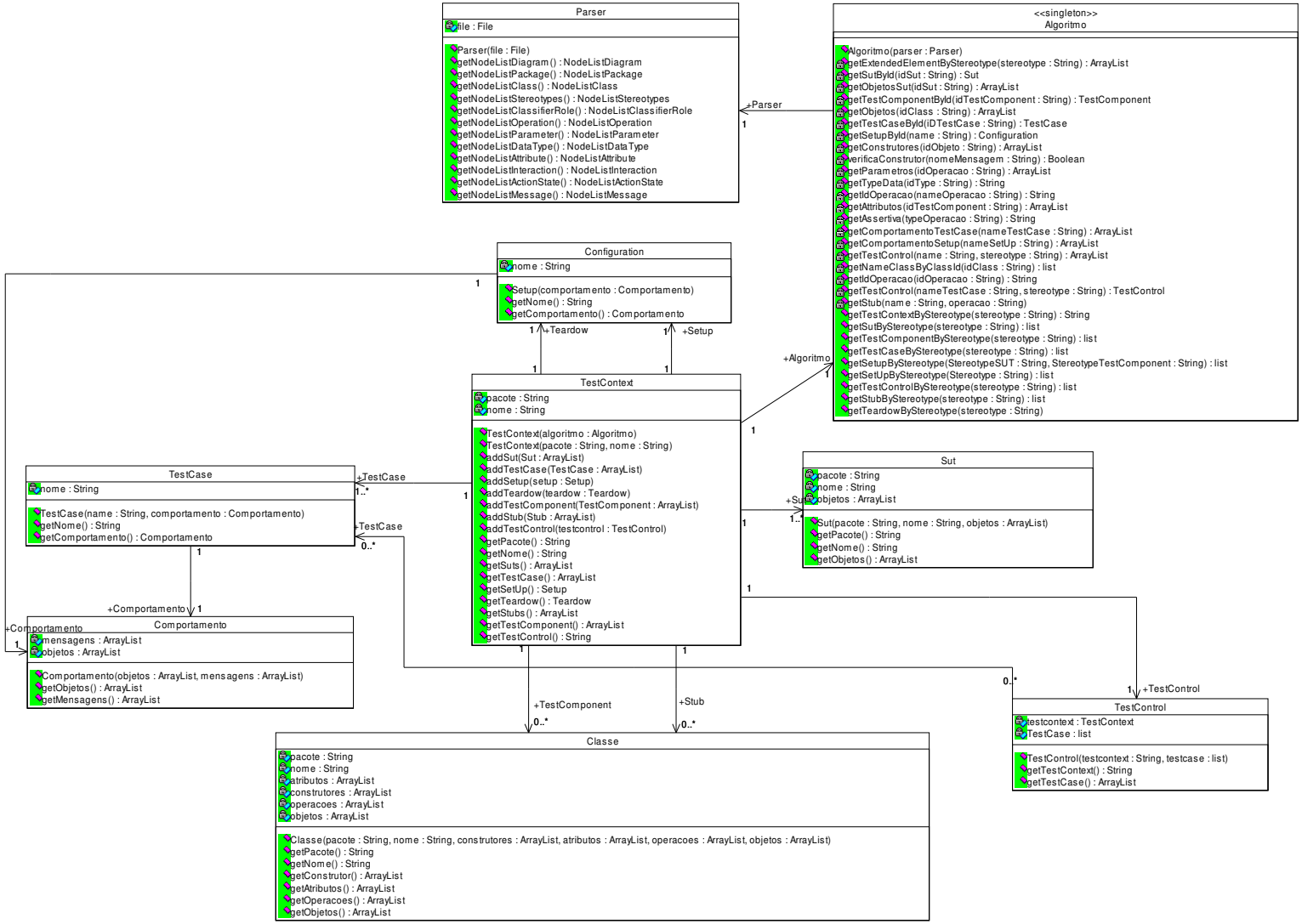


Figura 38 - Mapeamento dos elementos do JUnit para diagrama de Classes.

As classes *Sut*, *TestCase*, *TestControl*, *Comportamento*, *Setup, Teardown, TestComponent e Stub* são classes que armazenam os elementos que serão gerados pelo gerador de código. Essas classes contêm apenas operações de acesso (*get*).

7.3 Funcionamento do Extrator de Código

Para desenvolver os algoritmos três classes principais estão envolvidas: *Parser*, *Algoritmo* e *TestContext*:

7.3.1 Classe Parser

Faz a análise no documento XMI e armazena as *tags* (nodos) que serão usadas para mapear os elementos. A Figura 39 descreve a classe parser.

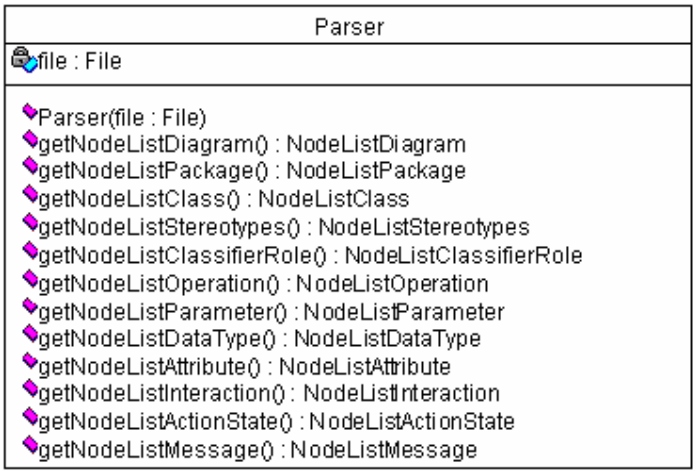


Figura 39 - Classe Parser.

A classe *parser* tem como atributo um *file* do tipo *File*. Esse atributo é usado para a declaração do arquivo XMI. O método construtor da classe é responsável por realizar a análise no documento XMI, através do arquivo passado como parâmetro. Cada método público da classe *Parser* retorna uma lista de nodos, conforme o tipo de *tag* que busca.

7.3.2 Classe Algoritmo

Define um conjunto de métodos públicos e privados usados para extrair os elementos do XMI necessários à geração dos *drivers* e *stubs* no JUnit. A Figura 40 descreve a classe Algoritmo.

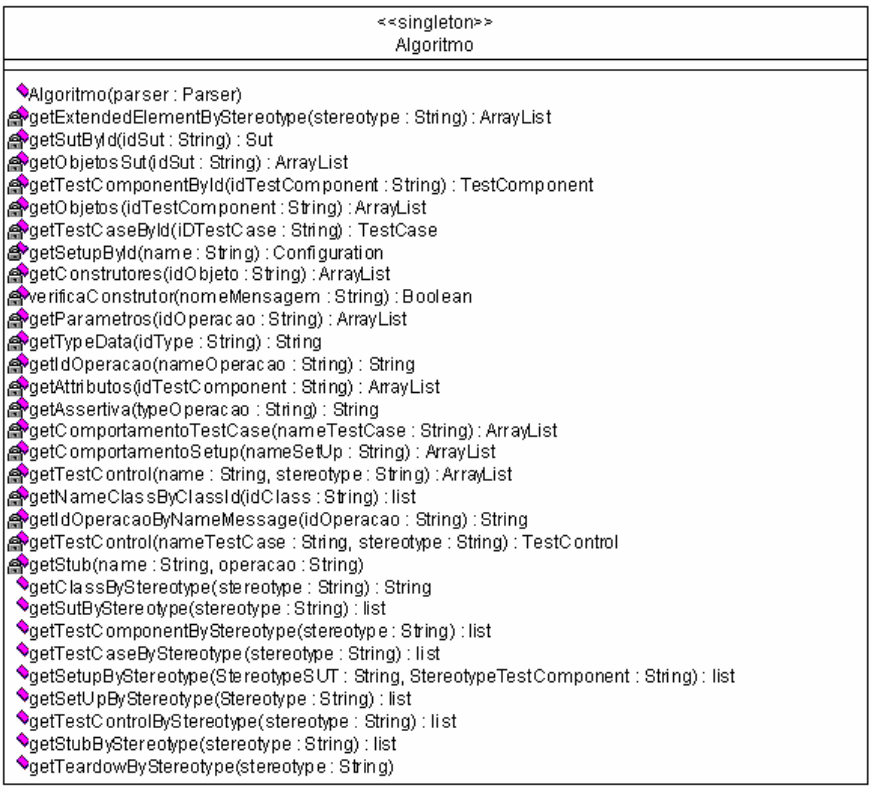


Figura 40 - Classe algoritmo.

A classe *Algoritmo* implementa o padrão *singleton*. Dessa forma só é permitida uma instância desta classe. Ela tem como atributo uma instância da classe *Parser* de tal forma que seus métodos possam usar os métodos públicos de *Parser* de acordo com suas necessidades. Uma descrição dos métodos públicos e privados de desta classe é descrito no Anexo I.

7.3.3 Classe TestContext

Esta classe se relaciona com todos os elementos do JUnit, já que ela é responsável por gerar esses elementos. A Figura 41 descreve a classe *Testcontext*.

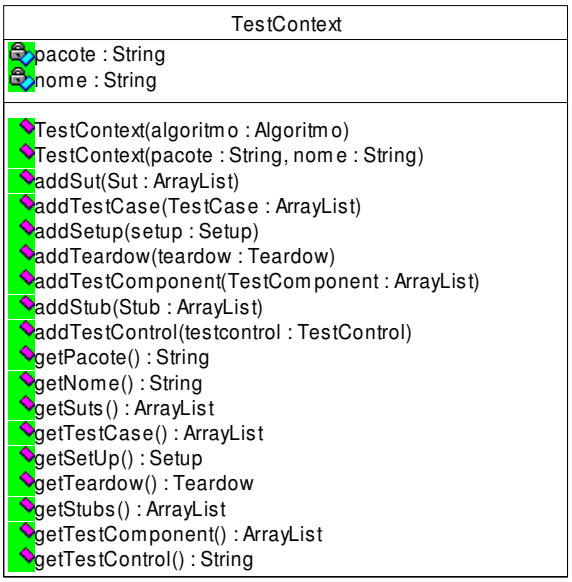


Figura 41 - Classe *TestContext*.

7.4 Algoritmos de Mapeamento

Os algoritmos de mapeamento identificam a maioria dos elementos a partir de estereótipos, mapeando-os para o diagrama de classes da Figura 38. A Tabela 6 ilustra para cada elemento o estereótipo usado para buscá-lo no documento XMI e o nodo que o estereótipo estende.

Tabela 6 - Estereótipos usados para buscar os elementos no documento XMI.

Elemento	Estereótipo	Nodo
TestContext	<<testcontext>>	<UML:Class>
Sut	<<sut>>	<UML:Class>
TestComponent	<<testcomponent>>	<UML:Class>
TestCase	<<testcase>>	<UML:Operation>
Setup	<<setup>>	<UML:Operation>
Teardown	<<teardown>>	<UML:Operation>
TestControl	<<testcase>>	<UML:Operation>
Stub	<<tescase>>	<UML:Operation>

O método que retorna uma lista com os "extendedElement" dos respectivos estereótipos é o método getExtendedElementsByStereotype (String stereotype). Esse método é usado para buscar os "xmi.id" de todas classes e/ou operações que estão estereotipadas. A Figura 42 descreve o algoritmo deste método. O método decompoeElemento (linha 5) quebra o atributo "extendedElement" dos elementos (classes ou operações) estendidos pelo estereótipo e armazena em uma lista.

Algoritmo: getExtendedElementByStereotype;
Entrada: stereotype;
Saída: listId;
1 Iterator nodeListStereotype := parser.getNodeListStereotype().iterator; 2 **enquanto** NodeListStereotype.hasNext() **faça** 3 node := NodeListStereotype.next(); 4 **se** node.name == stereotype **então** 5 listId:= **decompoeElemento**(node.extendedElement); 6 **return** listId;

Figura 42 - Algoritmo getExtendedElementsByStereotype

Considere o nodo <UML:Stereotype> da Figura 43. Se fosse passado o estereótipo "*Sut*" como parâmetro para o método getExtendedElementsByStereotype, este retornaria uma lista contendo os "xmi.id" de classes que o estereótipo estende. Como podemos observar o atributo "extendedElement" estende duas classes :"S.298.1247.52.2" e "S.298124752.20".

```
<UML:Stereotype xmi.id="S.298.1247.53.10" name="Sut" visibility="public"
  isSpecification="false" isRoot="false" isLeaf="false" isAbstract="false" icon=""
  baseClass="Class" extendedElement="S.298.1247.52.2 S.298.1247.52.20" />
```

Figura 43 - Nodo <UML:Stereotype>.

Como mencionado anteriormente, uma coleção de nodos equivale a um *NodeList*. Assim, na maioria das vezes, para percorrer estas listas são utlizados iteratores (*iterator*), assumindo-se que existe um método *iterator* para a conversão da lista em um *iterator*.

Para descrever os algoritmos envolvidos na busca de cada elemento, foram usados diagramas de seqüência. Dessa forma, é possível identificar quais métodos são usados para buscar cada elemento e em que ordem esses métodos são executados. As próximas seções apresentam os diagramas com os respectivos métodos usados para buscar cada elemento.

7.4.1 Buscando o elemento TestContext

Buscar o pacote e o nome do elemento *TestContext* a partir da classe estereotipada com <<testcontext>>. Assume-se que sempre existe um e somente um elemento *TestContext*, como especificado na Seção 6.2.2. Considere o exemplo da Figura 26 (Seção 6.2.2). O objetivo é buscar o nome e o pacote do elemento *TestContext*, que no exemplo da Figura 26 corresponde à TestVenda nos dois casos. A Figura 44 mostra o diagrama de seqüência com os métodos usados para buscar esse elemento.

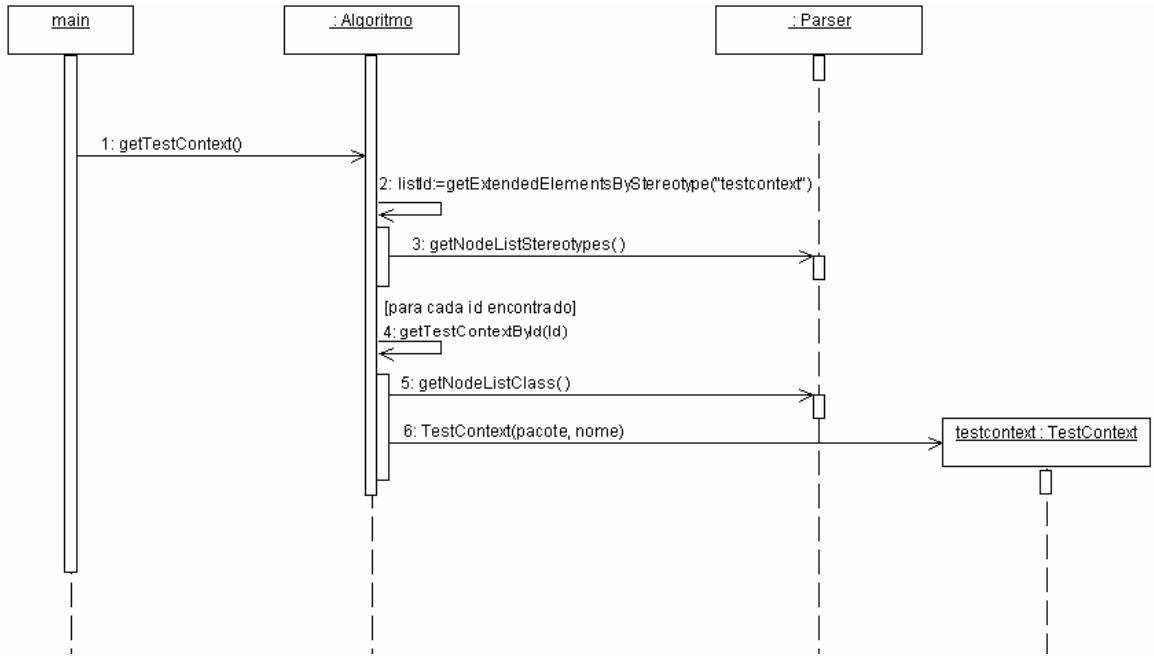


Figura 44 - Buscando o elemento *TestContext*

O elemento *testContext* é buscado através do método getTestContext(). Este retorna um objeto *Testcontext*. A Figura 45 descreve o algoritmo deste método.

Algoritmo: getTestContext();
Saída: testcontext;
1 Iterator listId := **getExtendedElementByStereotype**(“testcontext”).iterator; 2 id := listId.next(); 3 idClass := id.xmiId; 4 **return getTestContextById**(idClass);

Figura 45 - Algoritmo getTestContext.

Sempre haverá um único idClass para o estereótipo *TestContext*, e este é passado para o método getTestContextById, que retorna uma instância da classe *Testcontext*. A Figura 46 descreve o algoritmo deste método.

Algoritmo: getTestContextById;
Entrada: idClass;
Saída: testcontext;
1 Iterator nodeListClass := parser.getNodeListClass().iterator; 2 **enquanto** nodeListClass.hasNext() **faça** 4 node := nodeListClass.next(); 5 **se** node.xmiId == idClass 6 pacote := node.getParenteNode().name; 7 TestContext testcontext = new TestContext(pacote, node.name) 8 **return** testcontext;

Figura 46 - Algoritmo getTextContextById.

Quando o elemento *TestContext* é criado, este por sua vez gera os demais elementos, como descrito na Figura 47.

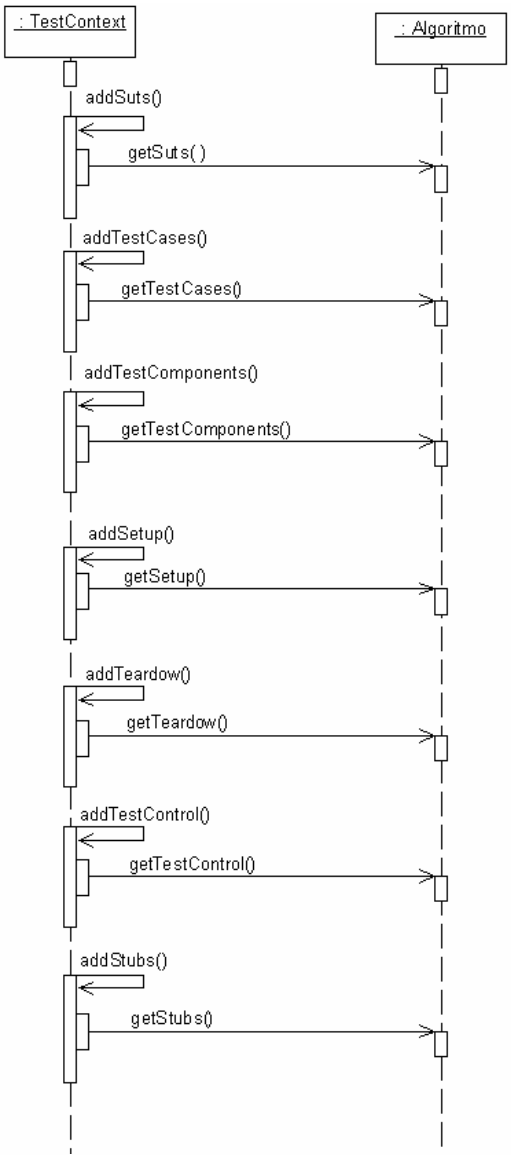


Figura 47 - *TestContext* buscando os demais elementos.

Uma vez extraídos todos os elementos, o gerador de código descrito na Seção 7.5 mostra como esses elementos são gerados.

7.4.2 Buscando os elementos Suts

Buscar o pacote, o nome, bem como as instâncias das classes estereotipadas com <<sut>>. Pode ter um ou mais *Sut*. Considere o exemplo da Figura 25 (Seção 6.2.1). O objetivo é buscar os nomes e os pacotes do *Sut*, que no exemplo correspondem a: Venda e LinhadeItemVenda, ambos do pacote Venda. Considere ainda o exemplo da Figura 28 (Seção 6.3.1). Nesta Figura, tem-se uma instância do *Sut* LinhadeItemVenda, denominada LIV. Dessa forma, a idéia é buscar todas as instâncias de cada um dos *Sut*s. A Figura 48 mostra o diagrama de seqüência com os métodos usados para buscar os *Sut*s.

Figura 48 - Buscando os elementos *Sut*s.

O elemento *Sut* é buscado através do método getSuts(). Cada *Sut* encontrado é adicionado na lista listSut. A Figura 49 descreve o algoritmo deste método.

```
Algoritmo: getSuts();
Saída: listSut;
1 Iterator listId := getExtendedElementByStereotype("sut").iterator;
2 enquanto listId.hasNext()faça
3    id := listId.next();
4    idClass := id.xmiId;
5    listSut.add(getSutById(idClass));
6 return listSut;
```

Figura 49 - Algoritmo getSuts.

Para cada idClass encontrado, um *Sut* é criado através do método getSutById(String idClass). A Figura 50 descreve o algoritmo deste método.

```
Algoritmo: getSutById(String idClass);
Entrada: idClass;
Saída: sut;
1 Iterator nodeListClass := parser.getNodeListClass().iterator;
2 enquanto nodeListClass.hasNext() faça
3    node := nodeListClass.next();
4    se node.xmiId == idClass então
5       pacote := node.getParenteNode().name;
6       Sut sut = new Sut(pacote, node.name, getObjetosSut(node.xmiId));
7       return sut;
```

Figura 50 - Algoritmo getSutById.

Para buscar as instâncias do *Sut*, é usado o método getObjetosSut(String IdClass), passando o "xmi.id" do *Sut* como parâmetro. Para entender como são identificadas estas instâncias, considere o nodo <UML:Class> da Figura 51, o qual corresponde à classe LinhadeItemVenda da Figura 25 (seção 6.2.1). Através de seu identificador, é preciso buscar suas instâncias, nos diagramas de seqüência correspondentes aos casos de teste. Um exemplo é mostrado na Figura 28, Seção 6.3.1. Este nome de instâncias é buscado na *tag* <UML:ClassifierRole>. Como podemos observar na Figura 51, o valor do atributo xmi.id = "S.298.1247.52.20" é igual ao valor do atributo base = "S.298.1247.52.20" do nodo <UML:ClassifierRole>, como ilustra a Figura 52.

```
- <UML:Class xmi.id="S.298.1247.52.20" name="LinhadeItemVenda" visibility="public" isSpecification="false"
    isRoot="true" isLeaf="true" isAbstract="false" isActive="false" namespace="S.298.1247.52.1">
  - <UML:Classifier.feature>
```

Figura 51 - Nodo <UML:Class> referente a classe LinhadeItemVenda.

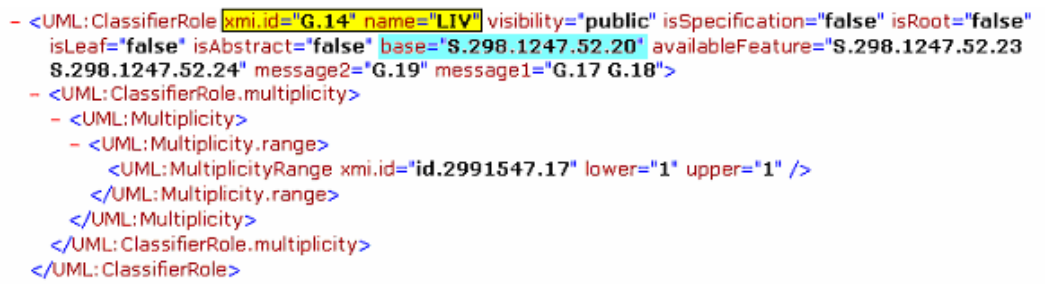


Figura 52 - Nodo <UML:ClassifierRole> referente ao objeto LIV.

Dessa forma, são identificadas todas as instâncias referentes aos objetos dos diagramas de seqüência correspondentes ao comportamento de casos de teste. No exemplo da Figura 53 é mostrado o trecho do documento XMI correspondente ao comportamento do caso de teste testSubTotal (referente ao exemplo da Figura 28, Seção 6.3.1). De acordo com a Figura 53, o atributo name = "testSutTotal" e o atributo diagramType = "SequenceDiagram" identificam que este nodo corresponde ao diagrama de seqüência equivalente ao comportamento do caso de teste. Também como podemos observar, o atributo subject = "G.14" é uma das instâncias do *Sut* declaradas naquele diagrama. Porém, nem todas as instâncias são do *Sut*. Dessa forma, deve-se buscar todas as instâncias, sendo estas representadas pelo atributo "subject" e depois verificar quais delas são do *Sut*. Voltando ao exemplo da Figura 53, o atributo subject = "G.14", corresponde ao atributo "xmi.id" do nodo <UML:ClassifierRole> denominado "LIV", como mostra a Figura 52 .

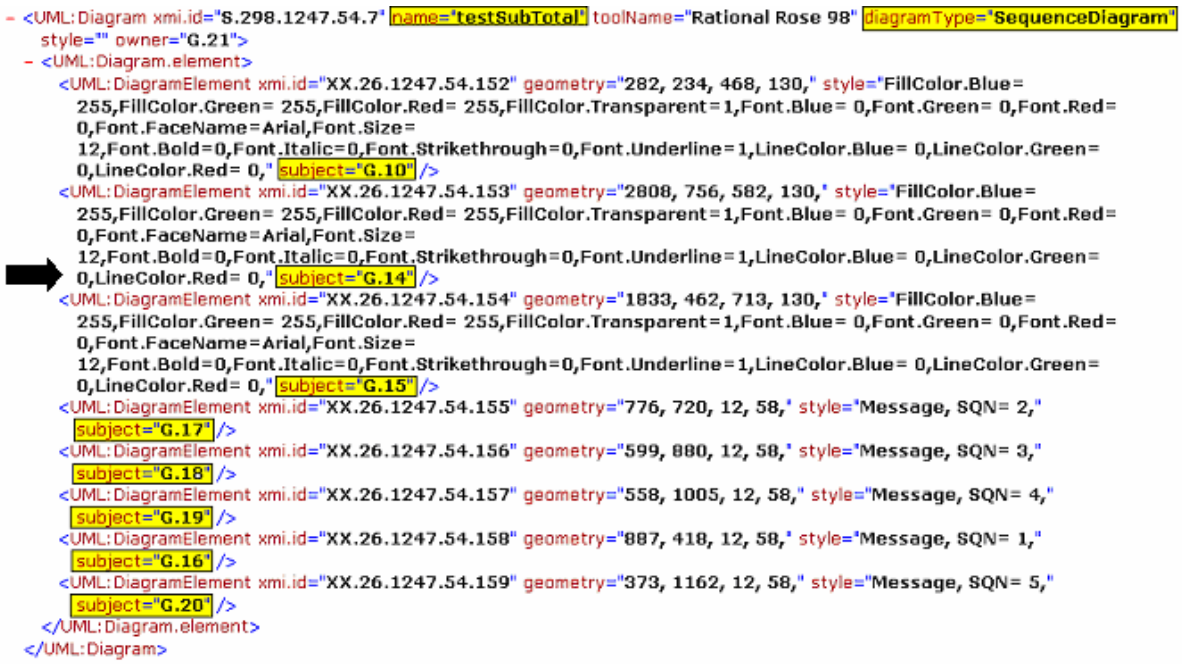


Figura 53 - Nodo <UML:Diagram> referente ao caso de teste testSubTotal.

A Figura 54 descreve o algoritmo que busca as instâncias do *Sut*.

```
Algoritmo: getObjetosSut;
Entrada: idClass;
Saída: listObjetosSut;
1 nodeListDiagram := parser.getNodeListDiagram();
2 nodeListOperation := parser.getNodeListOperation();
3 nodeListClassifierRole := parser.NodeListClassifierRole();
4 para i de 0 até tamanho de nodeListDiagram faça
5   nodeDiagrama := nodeListDiagram[i];
6   se nodeDiagrama.DiagramType == "SequenceDiagram" então
7     nomeDiagram := nodeDiagrama.name;
8     Iterator listId := getExtendedElementByStereotype("TestCase").iterator;
9     enquanto listId.hasNext() faça
10      id := listId.next();
11      idTestCase := id.xmiId;
12      para j de 0 até tamanho de nodeListOperation faça
13        nodeOperation := nodeListOperation[j];
14        se nodeOperation.xmiId == idTestCase então
15          nomeCasoTeste := nodeOperation.name;
16          se nomeCasoTeste == nomeDiagram então
17            nodeListDiagramElement := nodeDiagrama. getChildNodes();
18            para k de 0 até tamanho de nodeListDiagramElement faça
19            nodeDiagramElement := nodeListDiagramElement[k];
20            listSubject.add(nodeDiagramElement.subject);
21 Iterator listId := listSubject.iterator;
22 enquanto listId.hasNext() então
23   lista := listId.next();
24   nome := getObjeto(lista.subject)
25   listObjetosSut.add(nome); return listObjetosSut;
```

Figura 54 - Algoritmo getObjetosSut.

Primeiramente são percorridos todos os nodos <UML:Diagram> para verificar se o tipo de diagrama é de seqüência (linhas 4 a 7). Para cada diagrama encontrado, é armazenado o "name" na variável *nomeDiagram*. Em seguida, são identificados os nomes dos casos de teste e armazenados na variável *nomeTestCase* (linhas 8 a 15). É verificado se o nome do caso de teste é igual ao nome do diagrama (linha 16), se verdadeiro, são identificados os nodos filhos do diagrama de seqüência. Para cada nodo encontrado, é armazenado o atributo "subject" na lista listSubject (linhas 18 a 20). A lista listSubject é percorrida e cada "subject" é passado para método getObjeto que retorna o nome do objeto correspondente (linha 24). Assim, cada nome encontrado é adicionado na lista listObjetosSut. A Figura 55 descreve o algoritmo.

```
Algoritmo: getObjeto;
Entrada: idClass;
Saída: nome;
1 Iterator nodeListClassifierRole := parser.getNodeListClassifierRole().iterator;
2 enquanto nodeListClassifierRole.hasNext() faça
3   node := nodeListClassifierRole.next();
4   se node.base == idClass então return node.name;
```

Figura 55 - Algoritmo getObjeto.

O algoritmo da Figura 55 busca o “name” de um objeto a partir de seu “xmi.id”. Voltando ao exemplo da Figura 52, considerando xmi.id = “G.14”, retornará o “name” igual a “LIV”.

7.4.3 Buscando os elementos TestCases

Buscar o nome e o comportamento dos elementos *TestCases*. Assume-se a existência de uma ou mais operações estereotipadas com <<testcase>> na classe estereotipada com *TestContext*. Considere o exemplo da Figura 26 (Seção 6.2.2). O objetivo é buscar os casos de teste, que no exemplo correspondem aos casos de teste testSubTotal, testTotal e testEfetuarPagamento. Agora considere o exemplo da Figura 28 (Seção 6.3.1). Nesta Figura, tem-se o comportamento do caso de teste testSubTotal. Assim, para cada caso de teste também são identificados seus respectivos comportamentos. A Figura 56 mostra o diagrama de seqüência com os métodos usados para buscar os *testcase*s.

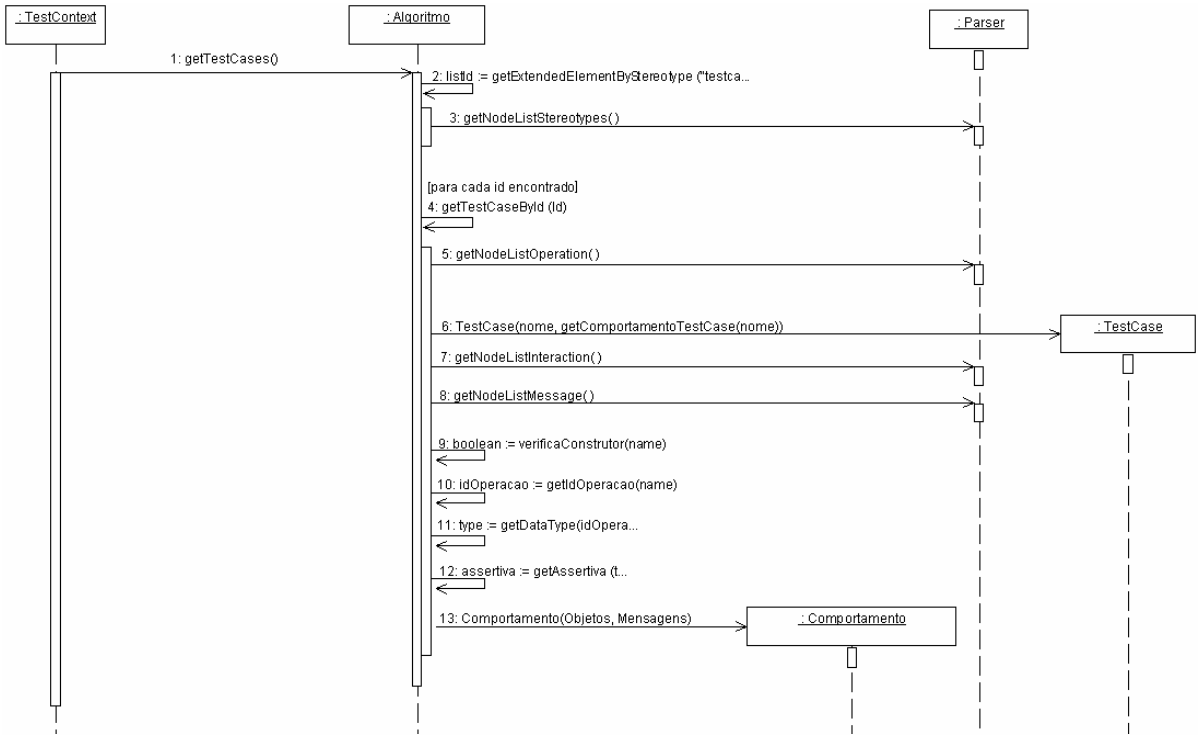


Figura 56 - Buscando os elementos *TestCase*s.

O elemento *TestCase* é buscado através do método getTestCase(). Este retorna uma lista de objetos *testcase* com o nome e o respectivo comportamento. A Figura 57 descreve o algoritmo deste método.

```
Algoritmo: getTestCase();
Saída: listTestCase;
1 Iterator listId := getExtendedElementByStereotype("testcase").iterator;
2 enquanto listId.hasNext() faça
3    id := listId.next();
4    idTestCase := id.xmiId;
5    listTestCase.add(getTestCaseById(idTestCase));
6 return listTestCase;
```

Figura 57 - Algoritmo getTestCase.

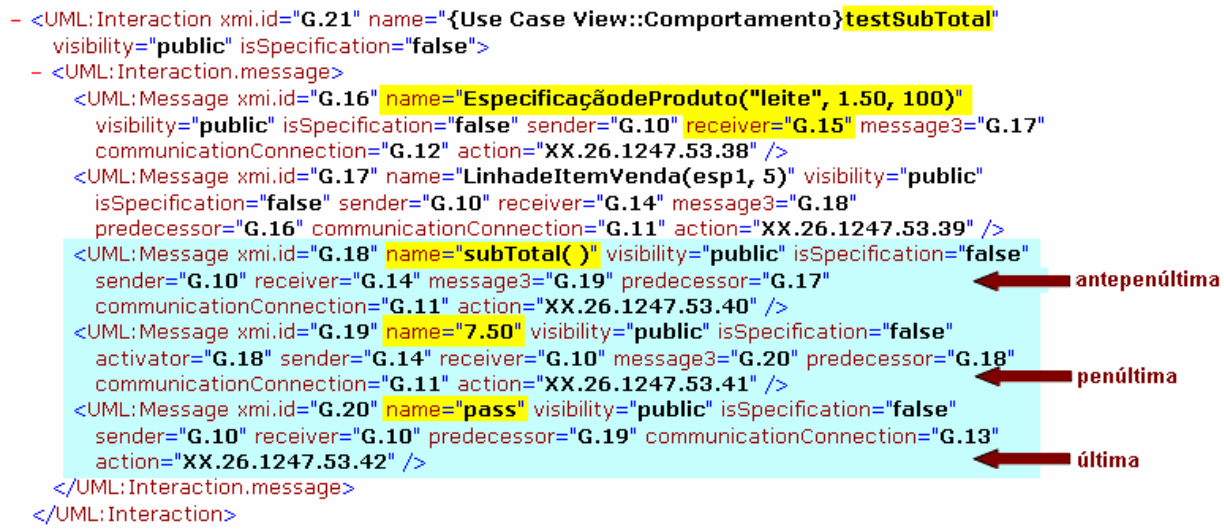
Para cada IdTestCase do *TestCase* encontrado, um objeto *TestCase* é criado através do método getTestCaseById(String idTestCase). A Figura 58 descreve o algoritmo deste método.

```
Algoritmo: getTestCaseById(String IdTestCase);
Entrada: idTestCase;
Saída: testcase;
1 Iterator nodeListOperacoes := parser.getNodeListOperacoes().iterator;
2 enquanto nodeListOperacoes.hasNext() faça
3   node := nodeListOperacoes.next();
4   se node.xmiId == idTestCase então
5     nome := node.name;
6     TestCase testcase = new TestCase(nome,getComportamentoTestCase(nome));
7     return testcase;
```

Figura 58 - Algoritmo getTestCaseById.

Assim, para cada caso de teste é criada uma instância passando o nome e o comportamento, sendo este último buscado através do método getComportamentoTestCase.

O comportamento de um caso de teste é composto basicamente por objetos e mensagens, podendo também conter declaração de variáveis locais. Para explicar o algoritmo é importante entender como ele é representado no documento XMI. A *tag* que representa o comportamento de um caso de teste é a <UML:Interaction>. Nesta *tag* são definidas todas as mensagens do diagrama de seqüência. As Figuras 59 e 60 ilustram o comportamento do caso de teste *testSubTotal* , referente ao caso de teste da Figura 28 (Seção 6.3.1) .




```
- <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"
        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" />
      <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" />
    </UML:Interaction.message>
  </UML:Interaction>
```

Figura 59 - Nodo <UML:Interaction> representando o comportamento do caso de teste testSubTotal.

```
- <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 60 - Nodo <UML:ClassifierRole> que corresponde ao objeto esp1.

Para cada nodo <UML:Interaction> é verificado se o atributo “name” é igual ao nome do caso de teste. Caso verdadeiro, são identificados os filhos da interação, ou seja, as mensagens, representadas pelos nodos <UML:Message>. Cada mensagem contém um identificador do objeto que recebe a mensagem (atributo “receiver”) e um identificador do objeto que envia a mensagem (atributo “send”). Estes correspondem aos “xmi.id” dos objetos representados pelos nodos<UML:ClassifierRole>.

O objeto que recebe a primeira mensagem “EspecificaçãodeProduto(“leite”, 1.50, 100)” (Figura 59) é o objeto que possui o atributo xmi.id = “G.15” denominado “esp1” conforme mostra a Figura 60. Dessa forma, a idéia para buscar o comportamento de um caso de teste é a seguinte: São identificadas todas mensagens (junto com os objetos destinatários

das mesmas) exceto as três últimas mensagens. O exemplo das Figuras 59 e 60 correspondem o seguinte código em um *driver* JUnit:

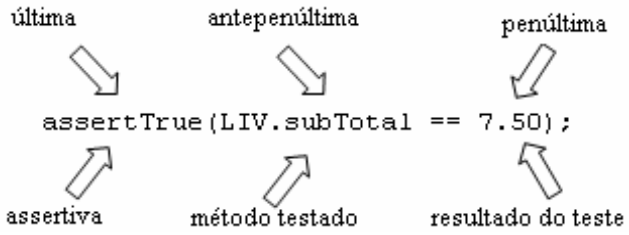
```
esp1 = new EspecificacaodeProduto("leite", 1.50, 100);
LIV = new LinhadeItemVenda(esp1, 5);
```

Essas mensagens são geradas antes de montar a assertiva para o caso de teste. Neste exemplo específico, as mensagens correspondem aos construtores das classes (EspecificaçãodeProduto e LinhaItemVenda). Porém, caso não fossem mensagens equivalentes a construtores de classes, o mapeamento seria feito da seguinte forma:

```
objeto.mensagem;
```

concatenado o nome do objeto que recebe a mensagem com o caracter "." mais o nome da mensagem.

Conforme os pressupostos descritos na Seção 6.3, as três últimas mensagens são usadas para montar a assertiva para o caso de teste, como segue:



A última mensagem corresponde ao veredito do caso de teste (Por exemplo, "*pass*" ou "*fail*"). Esta mensagem 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 correspondente do resultado do caso de teste.

O algoritmo da Figura 61 descreve detalhadamente como é buscado o comportamento do caso de teste:

```
Algoritmo: getComportamentoTestCase(String nomeTestCase);
Entrada: nomeTestCase;
Saída: comportamento;
1 nodeListInteraction := parser.getNodeListInteraction();
2 para i de 0 até tamanho de nodeListInteraction faça
3   nodeInteraction := nodeListInteraction [i];
4   se nodeInteraction.name == nomeTestCase então //busca filhos
5     nodeListMessage:= nodeListInteraction. getChildNodes();//filhos
6     antepenultima := nodeListMessage.Length()-3;
7     penúltima := nodeListMessage.Length()-2;//guarda posição penultima mensagem
8     ultima := nodeListMessage.Length()-1;//guarda posição ultima mensagem
9     para j de 0 até tamanho de nodeListMessage faça
10      nodeMessage := nodeListMessage[j];
11      nameMessage := nodeMessage.name;
12      receiver := nodeMessage.receiver;
13      nomeObjeto := getObjeto(receiver);
14      ndClass := getIdClass(receiver);
15      se j < antepenultima então //diferente das ultimas três
16        se (verificaConstrutor(nameMessage)) == true então
17          nistComportamento.add(nameObjeto + "= new" + nameMessage);
18          se (verificaSetup() == false) então
19            nomeClasse := getClassById(ndClass);
20            listComportamento.add(nomeClasse+nameObjeto+"=new"+nameMessage);
21      se j == antepenultima então
22        idOperacao := getIdOperacao(nameMessage, base);
23        type := getDataType(idOperacao);
24        metodoTestado :=  nameObjeto + "." + nameMessage;
25      se j == penultima então
26        resultado := nameMessage;
27      se j == ultima então
28        se nameMessage == "pass" então
29          assertiva := getAssertiva(type);
30          se nameMessage == "fail" então
31            assertiva := "assertFalse";
32      se assertiva == "assertTrue" ou "assertFalse" então
33          listComportamento.add(assertiva + metodoTestado + "==" + resultado);
34        se assertiva == "assertEquals" então
35          listComportamento.add(assertiva+ "0," +metodoTestado+ "," + resultado);
36        Comportamento c = new Comportamento(listComportamento):
37      return comportamento;
```

Figura 61 - Algoritmo getComportamentoTestCase.

Primeiramente, é percorrida a lista com os nodes <UML:Interaction> até encontrar o atributo "name" igual ao caso de teste passado por parâmetro (linhas 2 à 4). Se encontrado, são identificados todos os filhos de <UML:Interaction>, isto é, as mensagens (linha 5). São armazenadas as posições das três últimas mensagens (linhas 6 a 8). Para cada mensagem é armazenado o "name" na variável *nameMessage* (linha 11), o nome do objeto que recebe a mensagem na variável *nomeObjeto* (linha 13), sendo este buscado através do método getObjeto (ver Figura 55) e o identificador da classe que o objeto instancia na variável *idClass* (linha 14), sendo este buscado através do método getClassById (ver Anexo I, algoritmo 1).

Se mensagem for menor que antepenúltima (linha 15), significa que ainda não é o método testado, então é verificado se a mensagem é igual ao construtor de uma classe, que pode ser um *sut*, *testcomponent* ou uma classe do modelo de projeto que será interpretada como um *TestComponent*, através do método verificaConstrutor (ver Anexo I, algoritmo 2).

Se a mensagem for um construtor de uma classe e se a mesma é comum a todos casos de teste, então ela é gerada dentro do método *Setup* como segue:

```
esp1 = new EspecificacaodeProduto("leite", 1.50, 100);
LIV = new LinhadeItemVenda(esp1, 5);
```

Caso contrário, se o nome da mensagem não for comum a todos casos de teste, então ela é gerada como segue:

```
EspecificacaodeProduto esp1 = new
EspecificacaodeProduto("leite", 1.50, 100);
```

Isto significa que na hora de gerar o código, esta mensagem é declarada como uma variável local, dentro do corpo do caso de teste.

Seguindo o algoritmo, se mensagem é igual à antepenúltima (linha 21), ou seja, se é a mensagem que corresponde ao método testado, então é buscado o tipo da mensagem (*String*, *double*, *int*, etc...) e armazenado na variável "type". O tipo da mensagem é buscado pelos métodos getIdOperation e getDataType (ver Anexo I, algoritmos 3 e 4 respectivamente). Este é buscado para poder gerar a assertiva correta para o caso de teste. Se um caso de teste retornar um tipo primitivo (exemplo, *int*, *double*, etc...), é usada a assertiva *assertTrue*. Já se um caso de teste retornar um objeto de uma classe ou uma *String*, é usada a assertiva *assertEquals*. Também é armazenado o método testado na variável *metodoTestado*, sendo este composto pelo objeto e a mensagem (linha 24).

Seguindo o algoritmo, se mensagem for igual a penúltima (linha 25), ou seja, se é a mensagem que corresponde ao resultado do caso de teste, a mesma é armazenada na variável *resultado*. Esta variável será usada mais adiante para montar a assertiva para o caso de teste.

Se mensagem for igual a última (linha 27) então é buscado o veredito para o caso de teste. Assim, se o nome da mensagem for igual a "pass" então é buscado o tipo de assertiva através do método getAssertiva (ver Anexo I, algoritmo 5). Já, se o nome da mensagem for igual a "fail" é usada a assertiva *assertFail*.

Finalizando o algoritmo, uma vez buscada a assertiva (linha 25 ou 27), o método testado (linha 24) e o resultado do teste (linha 26), os mesmos são adicionados na lista listComportamento. Porém são consideradas duas situações para adicionar esses elementos na lista. Se assertiva for igual a *assertTue* a assertiva é montada como segue:

```
<assertiva> (<metodoTestado> "==" <resultado>
      assertTrue(LIV.subTotal == 7.50);
```

Já se assertiva for igual a *assertEquals*, ela é montada como segue:

```
<assertiva> (<metodoTestado> "," <resultado>
    assertEquals(0, LIV.subTotal , 7.50);
```

7.4.4 Buscando o elemento Setup

Buscar o comportamento do elemento *Setup*. Assume-se a existência de uma e somente uma operação estereotipada com <<setup>>. Considere o exemplo da Figura 29 (Seção 6.4.2). O objetivo é gerar as mensagens do diagrama de seqüência juntamente com os objetos dentro do método *setup* da ferramenta JUnit. A Figura 62 mostra o diagrama de seqüência com os métodos usados para buscar este elemento.

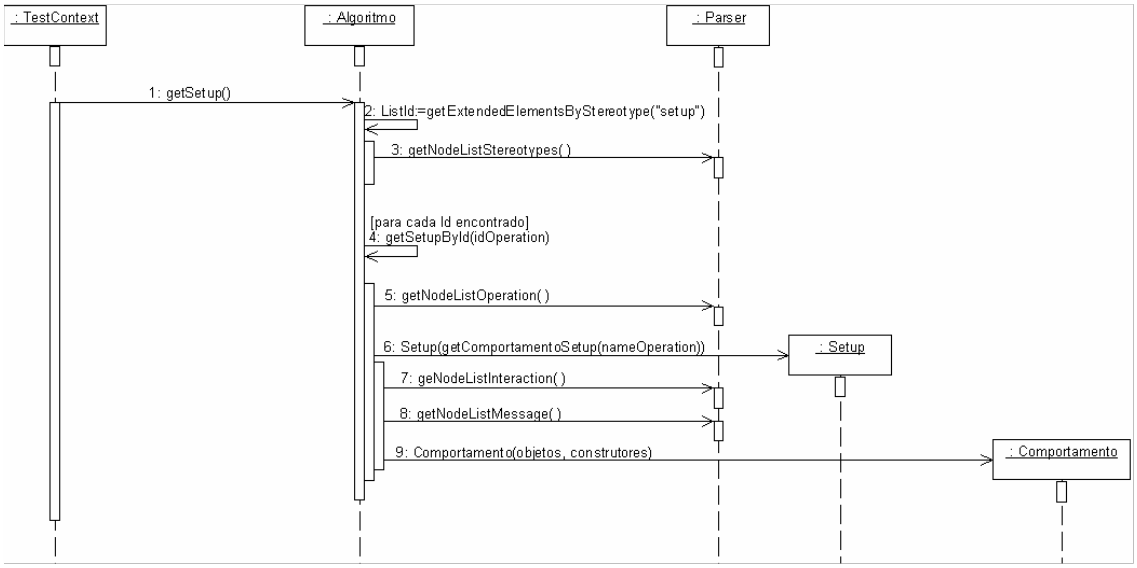


Figura 62 - Buscando o elemento *Setup*.

O elemento *Setup* é buscado através do método getSetup(). Este retorna o objeto *setup* com o comportamento. A Figura 63 ilustra o algoritmo deste método.

```
Algoritmo: getSetup();
Saída: setup
1 Iterator listId := getExtendedElementByStereotype("setup").iterator;
2   id := listId.next();
3   idOperation := id.xmiId;
4   setup := getSetupById(idOperation);
5   return setup;
```

Figura 63 - Algoritmo getSetup.

Sempre haverá um único idOperation para o estereótipo *setup*. O mesmo é passado para o método getSetupById que retorna uma instância da classe *setup*. A Figura 64 descreve o algoritmo deste método.

```
Algoritmo: getSetupById(String IdOperation);
Entrada: idOperation;
Saída: setup
1 Iterator nodeListOperacoes := parser.getNodeListOperacoes().iterator;
2 enquanto nodeListOperacoes.hasNext() faça
3   node := nodeListOperacoes.next();
4   se node.xmiId == idOperation então
5     name := node.name;
6     Setup setup = new Setup (getComportamentoSetup(name));
7     return setup;
```

Figura 64 - Algoritmo getSetupById.

Assim, uma vez criada uma instância de *Setup*, é passado o comportamento, sendo este buscado através do método getComportamentoSetup. O algoritmo para buscar o comportamento do elemento *Setup* é igual ao algoritmo que busca o comportamento do caso até a parte em que são identificadas as três últimas mensagens. A Figura 65 ilustra o algoritmo deste método.

```
Algoritmo: getComportamentoSetup(String nameOperation);
Entrada: nameOperation;
Saída: comportamento;
1 nodeListInteraction := parser.getNodeListInteraction();
2 para i de 0 até tamanho de nodeListInteraction faça
3   nodeInteraction := nodeListInteraction[i];
4   se nodeInteraction.name == nameOperation então
5     nodeListMessage:= nodeListInteraction. getChildNodes();//filhos
6     para j de 0 até tamanho de nodeListMessage faça
7       nodemessage := nodeListMessage[j];
8       nameMessage := nodemessage.name;
9       receiver := nodemessage.receiver;
10      nomeObjeto := getObjeto(receiver);
11      listComportamento.add (nomeObjeto + "= new" + nameMessage);
12 Comportamento comportamento = new Comportamento(listComportamento);
13 return comportamento;
```

Figura 65 - Algoritmo getComportamentoSetup.

Uma vez encontrada a interação, são identificados os filhos, isto é, as mensagens. Para cada mensagem, é armazenado o "name" na variável *nameMessage*, e o objeto que recebe e mensagem, na variável *nomeObjeto*, sendo este buscado pelo método getObjeto(ver Figura 55). Uma vez encontrado o nome da mensagem e o nome do objeto, os mesmos são adicionados na lista listComportamento (linha 11). É criada uma instância de Comportamento passando a lista (linha 12). Abaixo é mostrado o mapeamento para o código:

```
<objeto> = "new" <mensagem>
```

Para buscar o elemento *Teardown*, o procedimento é o mesmo, mudando apenas como ele é representado no JUnit (ver template, Seção 6.4).

7.2.5 Buscando os elementos TestComponents

Buscar o pacote, nome, construtores, atributos, métodos e instâncias de cada elemento *TestComponent*. Podem existir várias classes estereotipadas com <<testcomponent>>. Considere o exemplo da Figura 26 (Seção 6.1.2). O objetivo é buscar e gerar a classe estereotipada com <<TestComponent>>, que no exemplo, corresponde à classe EspProduto. O *TestComponent* é referenciado dentro do *driver* de teste (ver template Seção 6.4) e também fora do *driver* de teste como uma nova classe JAVA e seus respectivos, construtores, atributos e métodos. A Figura 66 mostra o diagrama de seqüência com os métodos usados para buscar este elemento.

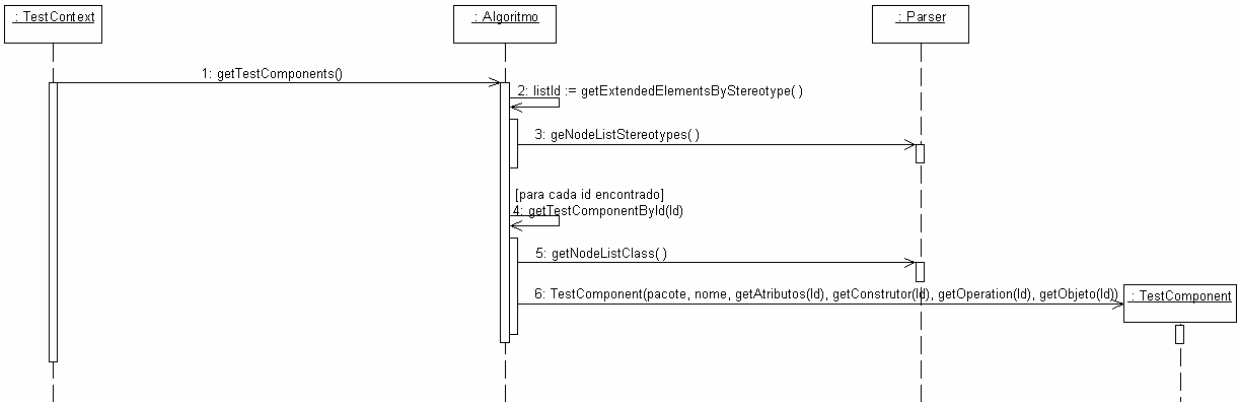


Figura 66- Buscando os elementos *TestComponents*.

Os elementos *Testcomponent* são identificados através do método getTestComponents(). Este retorna uma lista de objetos *testcomponent* com o pacote, nome, construtores, atributos, métodos e instâncias. A Figura 67 descreve o algoritmo deste método.

```
Algoritmo: getTestComponent();
Saída: listTestComponent;
1 Iterator listId := getExtendedElementByStereotype("testcomponent").iterator;
2 enquanto listId.hasNext()faça
3    id := listId.next();
4    idClass := id.xmiId;
5    listTestComponent.add(getTestComponentById(idClass));
6 return listTestComponent;
```

Figura 67 - Algoritmo getTestComponent.

Para cada idClass encontrado, um *testcomponent* é criado através do método getTestComponent (String idClass). A Figura 68 descreve o algoritmo deste método.

```
Algoritmo: getTestComponentById(String idClass);
Entrada: idClass;
Saída: testcomponent;
1 Iterator nodeListClass := parser.getNodeListClass().iterator;
2 enquanto nodeListClass.hasNext() faça
3   node := NodeListClass.next();
4   se node.xmiId == idClass então
5     id := node.xmiId;
6     pacote := node.getParentNode();//busca nodo pai
7        TestComponent testcomponent = new TestComponent(pacote, node.name,
getConstrutores(id),getAttribute(id),getOperation(id) ,getObjeto(id));
8     return testecomponent;
```

Figura 68 - Algoritmo getTestComponentById.

Os métodos getConstrutores (Anexo I, algoritmo 6), getAttribute (Anexo I, algoritmo 7), getOperation (ver Anexo I, algoritmo 8) e getObjeto (ver Figura 55), buscam os construtores, atributos, operações e instâncias de cada *testcomponent* respectivamente.

7.4.6 Buscando os elementos Stubs

Os elementos *Stubs* são identificados a partir de outros diagramas de seqüência do modelo de projeto do sistema. Para compreender como funciona o algoritmo para gerar esse elemento, é importante saber o contexto no qual ele é usado. Um *Stub* é um tipo de *TestComponent*, porém diferente do *testcomponent* que é gerado a partir do modelo de projeto de teste e de Software, o *Stub* é gerado somente a partir do modelo de projeto do software.

No exemplo explorado no capítulo 6 (Sistema Venda), um dos casos de teste envolve o método subTotal da classe LinhadeItemVenda (Figura 28 Seção 6.3.1). Porém, para testar esse método, é preciso saber o preço de cada produto. Observando a Figura 69 que representa o comportamento do método subTotal() da classe Venda, verificamos que existe uma dependência da classe LinhadeItemVenda com a classe EspecificaçãodeProduto. Assim, como a classe EspecificaçãodeProduto não é um *Sut* e não foi especificada como um *TestComponent*, ela é gerada como um *Stub* de teste.

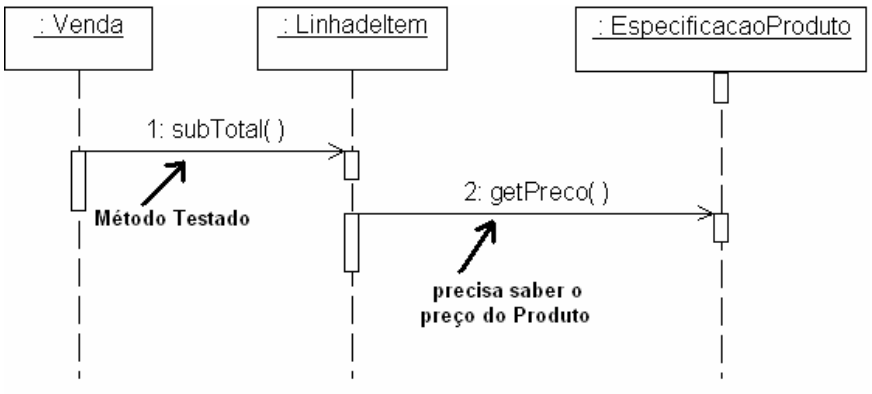


Figura 69 - Comportamento do método subTotal().

A representação destas mensagens no documento XMI é ilustrado na Figura 70 através do nodo <UML:Interaction>.

```
- <UML:Interaction xmi.id="G.63" name="{Use Case View::Comportamento}subTotal"
    visibility="public" isSpecification="false">
  - <UML:Interaction.message>
      <UML:Message xmi.id="G.61" name="subTotal( )" visibility="public" isSpecification="false"
        sender="G.56" receiver="G.58" message3="G.62" communicationConnection="G.57"
        action="XX.11.040.26.91" />
      <UML:Message xmi.id="G.62" name="getPreco( )" visibility="public" isSpecification="false"
        sender="G.58" receiver="G.60" predecessor="G.61" communicationConnection="G.59"
        action="XX.11.040.26.92" />
    </UML:Interaction.message>
  </UML:Interaction>
```

método testado

próxima mensagem

Figura 70 - Nodo <UML:Interaction>, representando o comportamento do método subTotal.

Dessa forma a idéia geral do algoritmo é a seguinte: São identificados todos diagramas de seqüência diferentes daqueles que representam o comportamento de casos de teste, como descrito na Figura 70. Para cada diagrama de seqüência, é verificado se existe alguma mensagem do diagrama igual ao método a ser testado, situação esta descrita através da primeira mensagem da Figura 70. Se existir, é preciso verificar se o objeto que recebe esta mensagem envia uma outra mensagem para outro objeto, situação descrita na Figura 70

ilustrado pelos atributos receiver = “G.58” e sender = ‘G.58”. Em caso verdadeiro, é verificado se o objeto que recebe esta mensagem (atributo receiver = “G.60”) é uma instância de *Sut*. Se não for uma instância de *Sut*, é buscada a classe que o objeto instância e essa classe será mapeada como um *Stub* de teste.

A Figura 71 mostra o diagrama de seqüência com os métodos usados para buscar este elemento.

Figura 71 - Buscando os elementos *Stubs*.

Os elementos *Stub*s são identificados através do método getStubs(). Este retorna uma lista de objetos *stub*s com o pacote, nome, construtores, atributos, métodos e instâncias. A Figura 72 ilustra o algoritmo deste método.

Algoritmo: getStubs();
Saída: listStubs;
1 Iterator listId := **getExtendedElementByStereotype**(“TestCase”).iterator; 2 **enquanto** listId.hasNext() **faça** 3 id := listId.next(); 4 idTestCase := id.xmiId; 5 TestCase testcase := **getTestCaseById**(idTestCase); 6 listStubs.add(**getSt** **ubByTestCase**(testcase.getName())); 7 **return** listStubs;

Figura 72 - Algoritmo getStubs.

Cada caso de teste encontrado é passado para o método getStubByTestCase que retorna um lista com todas classes *Stub*s. A Figura 73 ilustra o algoritmo deste método.

```
Algoritmo: getStubByTestCase(String nameTestCase);
Entrada: nameTestCase;
Saída: stub;
1 Iterator NodeListDiagram := parser.getNodeListDiagram().iterator;
2 NodeListClass := parser.getNodeListClass();
3 enquanto NodeListDiagram.hasNext() faça
4   nodeDiagram := NodeListDiagram.next();
5  se nodeDiagram.type == "SequenceDiagram" então
6    se nodeDiagram.name != nameTestCase então
7      ListOnwer.add(nodeDiagram.onwer);
8 NodeListIteraction := parser.getNodeListIteraction();
9 para i de 0 até tamanho de NodeListInteraction faça
10   nodeInteraction := NodeListInteraction[i];
11  enquanto ListOnwer.hasNext() faça
12    onwer := ListOnwer.next();
13    se nodeInteraction.xmiId == onwer então
14      //busca filhos de interaction, as mensagens
15      Iterator NodeListMessage := NodeListIteraction.getChildNodes();
16      para j de 0 até tamanho de NodeListMessage faça
17        nodeMessage := NodeListMessage[j];
18       se nodeMessage.name == nameTestCase então
19         receiver := nodeMessage.receiver;
20         se nodeMessage.sender == receiver então
21           receiverAux := nodeMessage.receiver;
22           para k de 0 até tamanho de NodeListClassifierRole faça
23             nodeClassifierRole := NodeListClassifierRole[k];
24            se nodeClassifierRole.xmiId == receiverAux então
25              base := nodeClassifierRole.base;
26              para l de 0 até tamanho de NodeListClass faça
27                nodeClass := NodeListClass[l]
28               se nodeClass.xmiId == base então
29                 Id := nodeClass.xmiId;
30                 se verificaSut(Id) == false
31                   Node n := NodeListClass.getParentNode();
32                            Stub stub = new Stub(n.name, NodeListClass.next.name, getConstrutores(id),
getAtributos(id), getOperation(id), getobjetos(id));
33                   return stub;
```

Figura 73 - Algoritmo getStubByTestCase.

Primeiramente, a lista com os nodos <UML:Diagram> é percorrida até encontrar aqueles que sejam diagramas de seqüência. Para cada nodo correspondente ao diagrama de seqüência encontrado, é verifica se o "name" do diagrama é diferente dos casos de teste. Se diferente, é armazenado na lista ListOnwer o atributo "onwer" dos diagramas de seqüência (linhas 3 à 7). O próxima passo é percorrer os nodos <UML:Interaction> e a ListOnwer (linhas 9 e 11) e comparar se o atributo "xmi.id" da interação é igual ao atributo "onwer" da ListOnwer. Caso iguais, são identificados os filhos da interação, isto é, as mensagens (linha 14). Para cada mensagem, é verificado se a mesma é igual ao nome do caso de teste passado por parâmetro. Se forem iguais, é armazenado o atributo "receiver" da mensagem (linhas 18 e

19). Uma vez armazenado o identificador do objeto que recebe a mensagem, é verificado se a próxima mensagem é envidada pelo mesmo objeto que recebeu a mensagem anterior, ou seja, aquela de nome igual ao caso de teste (linha 20). Se verdadeiro, é armazenado o atributo "receiver" desta mensagem na variável *receiverAux* (linha 21). Assim, a lista com os nodos <UML:ClassifierRole> é percorrida até encontrar o atributo "xmi.id" igual a variável *receiverAux* e quando encontrado, é armazenado o atributo "base" na variável *base* (linhas 22 a 25), ou seja, é buscado o identificador da classe que o objeto instancia. Finalizando o algoritmo, a lista com os nodos <UML:Class> é percorrida até encontrar o atributo "xmi.id" da classe igual a variável *base* e se encontrado, é criado um novo objeto *Stub* denominado *stub* passando o nome do pacote, o nome da classe, os construtores, os atributos, os métodos e os objetos (linhas 26 a 30). Os algoritmos que buscam os construtores, atributos, métodos e objetos são os mesmos usados para o elemento *testcomponent*, mudando apenas o identificador passado por parâmetro.

7.4.7 Buscando o elemento TestControl

Buscar os nomes das atividades na ordem em que elas aparecem no diagrama de atividades especificado para o elemento *TestControl*. É assumido no máximo um elemento *testcontrol*. Considere o exemplo da Figura 27, Seção 6.2.3. O objetivo é buscar o nome de cada atividade do diagrama de atividades na ordem em que elas aparecem, no exemplo: testSubTotal, testTotal e testEfeturarPagamento, respectivamente. A Figura 74 mostra o diagrama de seqüência com os métodos usados para buscar o elemento *TestControl*.

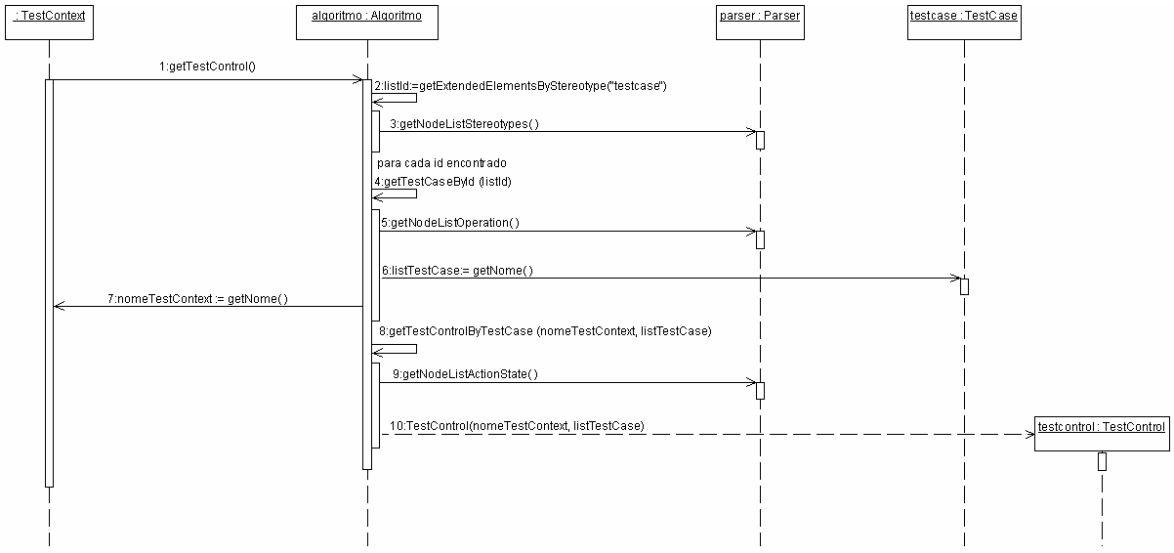


Figura 74 - Buscando o elemento *TestControl*.

O elemento *TestControl* é buscado através do método getTestControl(). Este retorna um objeto *testcontrol*, com o nome do elemento *TestContext* e o nome das atividades do diagrama de atividades. A Figura 75 ilustra o algoritmo deste método.

Algoritmo: getTestControl();
Saída: testControl;
1 Iterator listId := **getExtendedElementByStereotype**(“testcase”).iterator; 2 **enquanto** listId.hasNext() **faça** 3 id := listId.next(); 4 idTestCase := id.xmiId; 5 TestCase testcase := **getTestCaseById**(idTestCase)); 6 listTestCase.add(**getTestControlByTestCase**(testcase.getName())); 7 TestContext testcontext := new TestContext(); 8 nomeTestContext := testcontext.getNome(); 7 TestControl testcontrol = new Testcontrol (nomeTestContext, listTestCase) 9 **return** testControl;

Figura 75 - Algoritmo getTestControl.

Para cada IdTestcase encontrado, é buscado o nome do caso de teste através do método getTestCaseById (linha 5). Cada nome é passado para o método getTestControlByTestCase que retorna o nome da atividade do diagrama de atividade na ordem em que ela aparece. A Figura 76 ilustra o algoritmo deste método.

Algoritmo: getTestControlByTestCase(String nameTestCase);
Entrada: nameTestCase;
Saída: nomeAtividade;
1 Iterator nodeListActionState := parser.getNodeListActionState().iterator; 2 **enquanto** nodeListActionState.hasNext() **faça** 3 nodo := nodeListActionState.next(); 4 **se** nodo.name == nameTestCase **então** 5 **return** nodo.name;

Figura 76 - Algoritmo getTesControlByTestCase.

A lista com os nodos <UML:ActionState> é percorrida, e para cada atributo "name" é verificado se o nome da atividade é igual ao nome do caso de teste passado como parâmetro. Caso iguais, a mesma é retornada.

7.5 Gerador do Código

O gerador do código é responsável por gerar os elementos de acordo com o template apresentado na Seção 6.4. Seja *testcontext* uma instância de *TestContext* criada como explorado na Seção 7.2, abaixo é descrito como cada elemento é gerado:

1 - Gerando o pacote do elemento *testcontext*:

```
"package" + textcontext.getPacote();
```

2 - Gerando o pacote dos elementos *Suts*, *TestComponent* e *Stubs*:

```
para i de 0 até textcontext.getSus().size faça
       "import"+textcontext.getSUTs().get(i).getPacote()+"."+textcontext.getSUTs().
       get(i)).getNome();

para i de 0 até textcontext.getTestComponents().size faça
    "import"+    textcontext.getTestComponents().get(i).getPacote()    +    "."    +
textcontext.getTestComponents().get(i)).getNome();

para i de 0 até textcontext.getStubs().size faça
    "import"   +   ""   +   textcontext.getStubs().get(i).getPacote()   +   "."   +
textcontext.getStubs().get(i)).getNome();
```

3 - Gerando o nome do elemento *TestContext*:

```
"public class" + textcontext.getNome() + " extends TestCase{"
```

4 - Gerando os elementos *Sut*, *Testcomponent* e *Stub* como variáveis globais:

```
para i de 0 até textcontext.getSus().size faça
       Objetos := textcontext.getSuts().get(i)).getObjetos();
       para j de 0 até Objetos.size faça
              nomeObjeto := Objetos[j];
             "private" + textcontext.getSuts().get(i).getNome()+""+ nomeObjeto;

para i de 0 até textcontext. getTestComponents().size faça
       Objetos := textcontext.getTestComponents().get(i)).getObjetos();
       para j de 0 até Objetos.size faça
              nomeObjeto := Objetos[j];

"private"+textcontext.getTestComponents().get(i).getNome()+""+nomeObjeto;

para i de 0 até textcontext. getStubs().size faça
       Objetos := textcontext.getStubs().get(i).getObjetos();
       para j de 0 até Objetos.size faça
              nomeObjeto := Objetos[j];
             "private" + textcontext.getStubs().get(i).getNome()+""+ nomeObjeto;
```

5 - Gerando o elemento *Setup*:

```
textcontext.getSetup().getComportamento();
```

6 - Gerando o elemento *Teardown*:

```
textcontext.getTeardown().getComportamento();
```

7 - Gerando os elementos *TestCase*s:

```
para i de 0 até textcontext.getTestCases().size faça
       "public void" + "" + textcontext.getTestCases().get(i).getNome() + "(){";
       comportamento := textcontext.getTestCases().get(i).getComportamento();
       para j de 0 até comportamento.size faça
              mensagem := comportamento[j];
              mensagem + "\n";
"}"
```

8 - Gerando o elemento *TestControl*:

```
"TestSuite suite = new TestSuite(){";
ListTestCase := textcontext.getTestControl().getTestCase();
para i de 0 até ListTestCase.size() faça
    nomeTestCase := ListTestCase[i];
```

```
       "suite.addTest(new "+ textcontext.getTestControl().getTestContext()
       +"("+nomeTestCase+"))";
```

9 – Gerando os elementos *TestComponent* e *Stubs* como classes Java:

```
para i de 0 até textcontext.getTestComponents().size faça
     "package " + textcontext.getTestComponents().get(i)).getName();
     "public class " + textcontext.getTestComponents().get(i).getNome()+"{";

ListAtributos := textcontext.getTestComponents().get(i).getAttributos();
            para j de 0 até ListAtributos.size faça
            nome : = ListAtributos[j];
        nome + ";";
ListaConstrutores:=textcontext.getTestComponents().get(i).getConstrutores();
     para k de 0 até ListaConstrutores.size faça
         nome := ListaConstrutores[k];
         nome + ";";
ListOperacoes := textcontext.getTestComponents().get(i).getOperacoes();
            para l de 0 até ListOperacoes.size faça
            nome : = ListOperacoes[k];
        nome + ";";
"}"
```

O exemplo (item 9) acima mostrou a geração da classe Java para o elemento *TestComponent*. Para a classe correspondente ao elemento *Stub* o procedimento é o mesmo.

7.6 Protótipo

Para validação dos algoritmos desenvolvidos foi implementado um protótipo, sendo este feito na linguagem de programação Java. Um dos motivos da escolha dessa linguagem está relacionado com o próprio JUnit, que é todo baseado nesta linguagem. Outro motivo é que esta linguagem disponibiliza diversas APIs que têm suporte à manipulação de documentos XML, como por exemplo, DOM, que foi a API que usada.

Para geração do documento XMI, algumas ferramentas de modelagem foram analisadas, entre elas: *Rational Rose* (IBM), *Enterprise Architect* (*Sparx Systems), Jude* (ferramenta *free*), *Together Architect* (*Borland*), *argoUML* (ferramenta *free*), *Visual Paradigm* (ferramenta *free*), *Poseidon for UML* (ferramenta *free*) e *ObjectDomain* (ferramenta *free*). Um requisito fundamental era que as ferramentas suportassem a geração de arquivos XMI tanto de diagramas estruturais como diagramas comportamentais da UML. As ferramentas que suportaram estes requisitos foram: *Rational Rose, Enterprise Architect,*

Together Architect e Poseidon for UML. Entre estas ferramentas a *Rational Rose* foi escolhida. A escolha desta ferramenta teve como critérios: (1) ser uma ferramenta muito conceituada e conhecida na modelagem de sistemas, tanta na área acadêmica como na área industrial e (2) aderência do XMI ao padrão OMG.

A interface do protótipo é simples. A interação com o usuário é mínima e a geração dos *Drivers* e *Stubs* é totalmente automatizada. A Figura 77 ilustra o protótipo com o *Driver* e *Stub* gerado no exemplo do sistema Venda, explorado no capítulo 6.

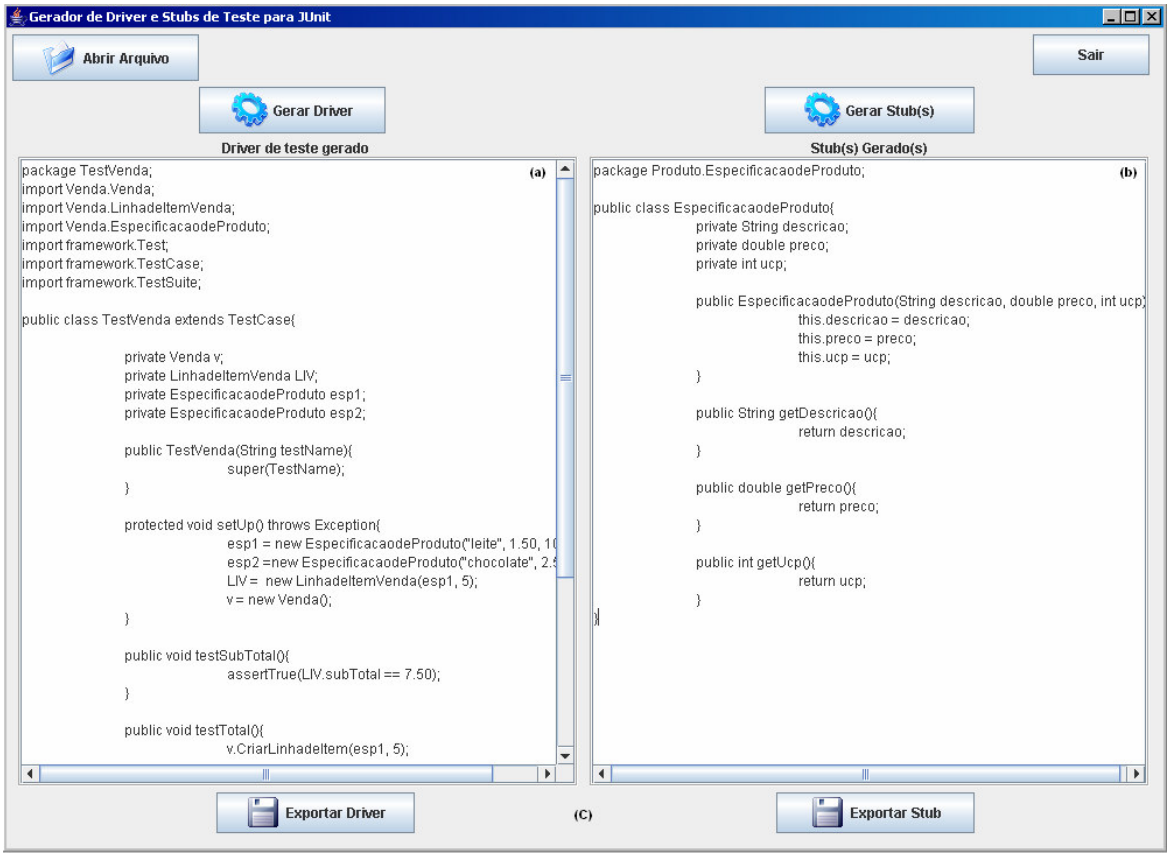


Figura 77 - Protótipo.

Como mencionado anteriormente, o objetivo é proporcionar o mínimo de interação do usuário para geração do código. Dessa forma, para gerar o código, basta clicar no botão “abrir arquivo”, escolher o arquivo XMI e clicar no botão “Gerar *Driver*”. O *driver* é gerado automaticamente, como mostrado no lado esquerdo da tela (Figura 77(a)). Se existirem *Stubs* e/ou *testcomponents* para este *driver*, aparecerá uma mensagem informando o usuário que existem *Stub*s para serem gerados. Assim, basta o usuário clicar no botão “Gerar *Stub*” que o mesmo é gerado automaticamente (Figura 77(b)).

O código pode ser exportado tanto em um arquivo txt, como também pode ser exportado para um arquivo Java (Figura 77 (c)). Assim, é possível importar esse arquivo para o ambiente de programação que o sistema está sendo desenvolvido e executar os testes.

8 ESTUDO DE CASO

8.1 Descrição do Estudo de Caso

Este capítulo descreve o estudo de caso realizado para validar a proposta do presente trabalho. Para realizar o estudo de caso, foi utilizado um sistema desenvolvido pelo CPTS – Centro de Pesquisa em Teste de Software da PUCRS. O sistema refere-se a um sistema de locações de DVDs. Para este, existe um projeto em UML, uma implementação na linguagem Java, bem como vários *driver*s de teste codificados para ferramenta Junit.

Além da existência dos artefatos já mencionados, a complexidade dos diferentes cenários de teste permitiram explorar especificações de teste em U2TP com todos os elementos deste perfil adotados neste trabalho, em especificações de diferentes complexidades. Com isso, espera-se testar de maneira bastante completa a geração automatizada proposta neste trabalho.

Este estudo de caso baseou-se nos seguintes documentos recebidos do CPTS:

- Modelo de Projeto de Software;
- Diagrama de casos de uso e diagramas de casos de uso expandidos e
- Vários *drivers* de teste (com os respectivos *stub*s, quando pertinentes), sendo que apenas três estavam completos.

Os *driver*s foram implementados no Junit por um testador do CPTS. Utilizando o Eclipse, o mesmo gerou automaticamente a estrutura de cada *driver* a partir do código Java das classes testadas. Este esqueleto foi então completado com a implementação de cada caso de teste. O tempo gasto para esta especificação foi fornecido pelo testador.

O estudo de caso centrou-se em dois objetivos:

- Verificar a correção do código gerado e
- Levantar dados quantitativos referentes ao tempo envolvido no processo de teste unitário e a qualidade do código gerado.

Assim, as próximas seções descrevem o modelo de projeto do sistema e o modelo de projeto de teste. No final do capítulo, será feita uma análise dos resultados.

8.2 Modelo de Projeto

O projeto do sistema de Locação de DVDs é composto por um conjunto de diagramas da UML, entre eles, Diagramas de Casos de Uso (incluindo os casos de uso expandidos) e Diagramas de Classe. A Figura 78 ilustra o diagrama de classes do sistema, tal como recebido do CPTS.

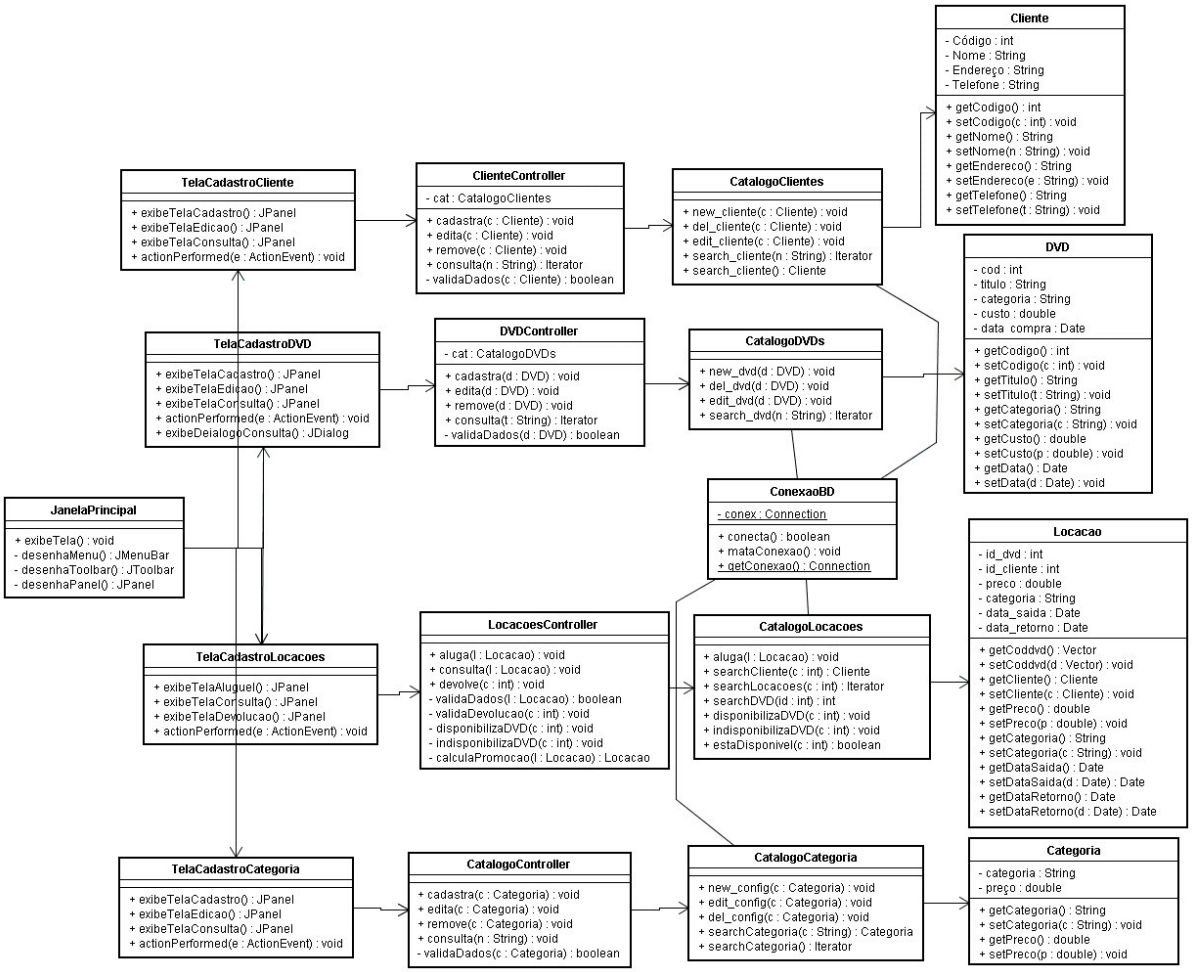


Figura 78 - Diagrama de classes do modelo de projeto.

As classes escolhidas como foco para os testes unitários foram: Categoria, CatalogoCategoria e CatalogoCliente. Dois foram os motivos desta escolha:

- No conjunto, contemplam todas as possíveis situações consideradas neste trabalho para geração automatizada de *drivers* e *stubs*;
- As mesmas correspondem aos três *driver*s completamente codificados recebidos do CPTS.

Assim, para cada classe serão especificados os testes e conseqüentemente os *drivers* gerados. Os *driver*s destas mesmas classes codificados pelo testador são apresentados no Anexo III.

Para especificação dos testes, foi usado o diagrama de classes do modelo de projeto e os diagramas de casos de uso e casos de uso expandidos, sendo este último muito importante para compreender o funcionamento do sistema. Cada operação da classe foi considerada um caso de teste.

8.3 Projeto de Teste

Esta seção descreve a transformação do modelo de projeto do software para o projeto de teste em U2TP, respeitando os pressupostos descritos na Seção 6.2. Foram gerados três *drivers* de teste referentes às classes Categoria, CatalogoCategoria, CatalogoCliente.

8.2.1 Especificação de teste Unitário para a Classe Categoria

A Figura 79 mostra a arquitetura do projeto de teste, e os diagramas das Figuras 80, 81, 82 e 83 definem o comportamento dos casos de teste. Os casos de teste especificados foram referentes às quatro operações da classe Categoria.

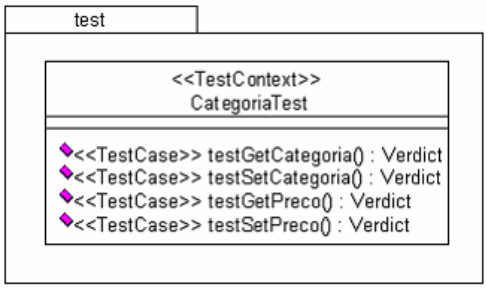


Figura 79 - Pacote de Teste Unitário para classe Categoria.

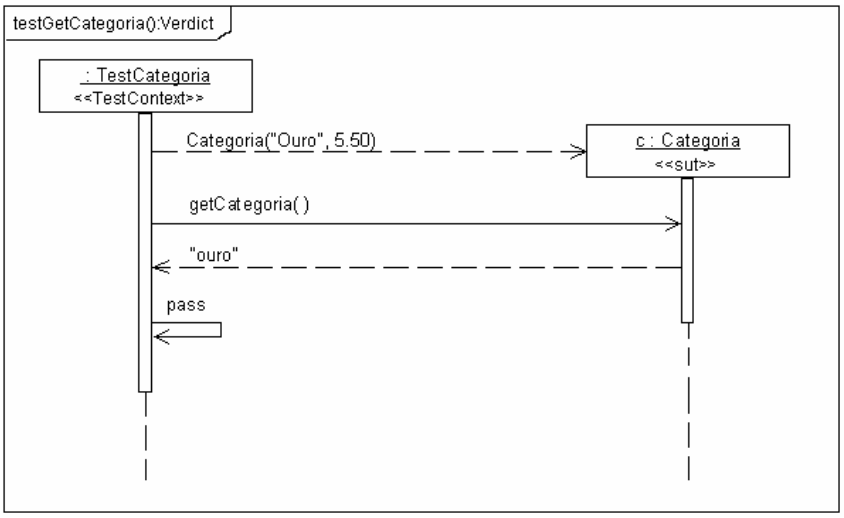


Figura 80 - Comportamento do caso de teste testGetCategoria.

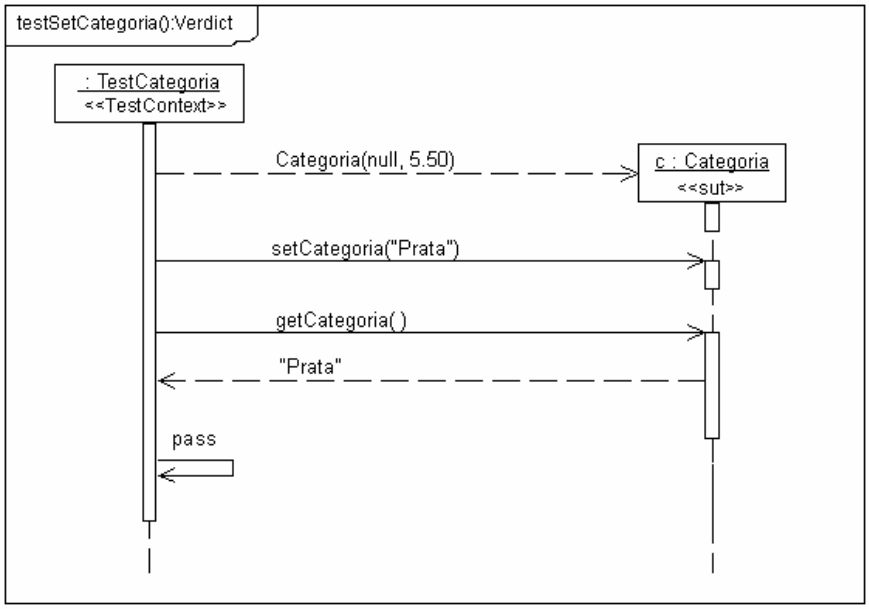


Figura 81 - Comportamento do caso de teste testSetCategoria.

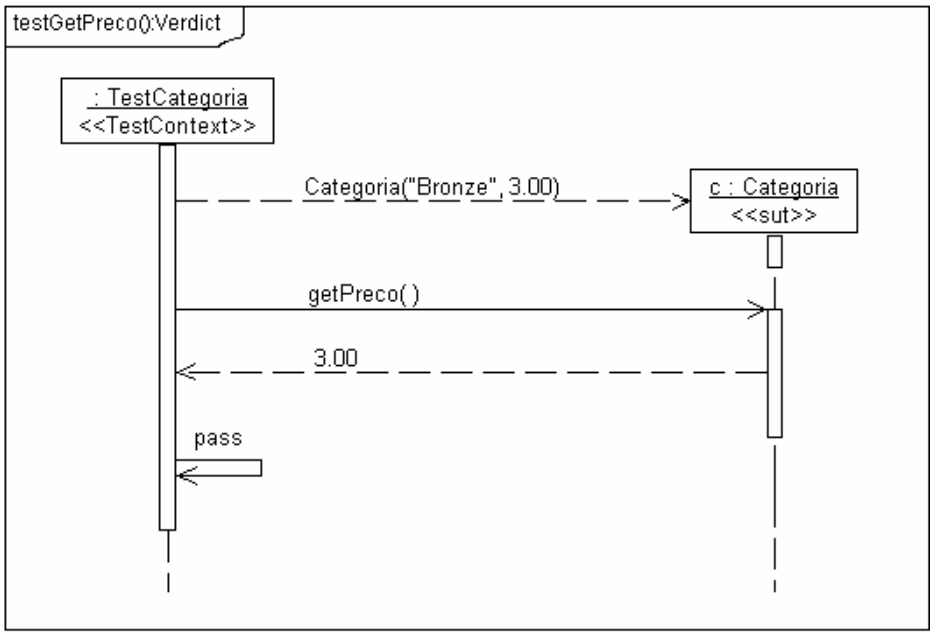


Figura 82 - Comportamento do caso de teste testGetPreco.

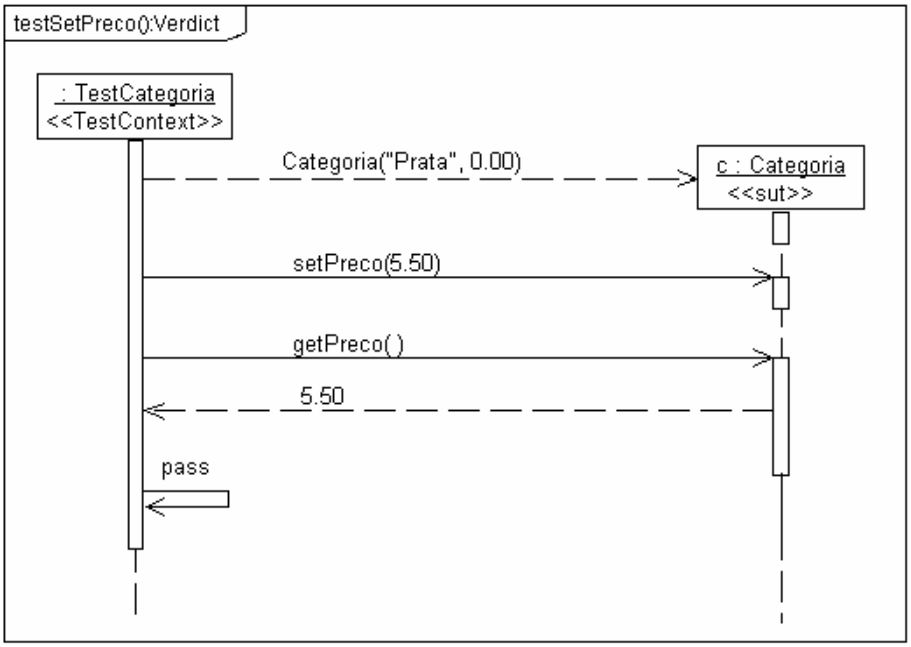


Figura 83 - Comportamento do caso de teste testSetPreco.

Considerando esta especificação, a Figura 84 mostra o *driver* de teste gerado automaticamente para a classe Categoria.

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

(4)public class CategoriaTest extends TestCase{

(5)   public void testGetCategoria(){
(6)         Categoria c = new Categoria("Ouro", 5.50);
(7)         assertEquals(0, c.getCategoria( ),"ouro");
(8)   }

(9)   public void testSetCategoria(){
(10)        Categoria c = new Categoria(null, 5.50);
(11)        c.setCategoria("Prata");
(12)        assertEquals(0, c.getCategoria( ),"Prata");
(13)  }

(14)  public void testGetPreco(){
(15)        Categoria c = new Categoria("Bronze", 3.00);
(16)        assertTrue(c.getPreco( ) == 3.00);
(17)  }

(18)  public void testSetPreco(){
(19)        Categoria c = new Categoria("Prata", 0.00);
(20)        c.setPreco(5.50);
(21)        assertTrue(c.getPreco( ) == 5.50);
(22)  }
(23)}
```

Figura 84 - *Driver* gerado para a classe Categoria.

Como podemos observar pela Figura 84, não foram gerados os métodos *Setup* e *Teardown*, pois os mesmos não foram especificados. Também pode-se observar que o *Sut* não foi declarado como uma variável global, pois cada caso de teste define um novo objeto *Sut* localmente, justificando assim o correto funcionamento dos algoritmos que geram esses elementos. Em relação aos pacotes gerados (linhas 1 e 2), pode-se observar que foi gerado todo o caminho correto na declaração dos pacotes. Neste exemplo, não foi gerado nenhum *Stub* de teste.

Em relação às assertivas algumas observações são ressaltadas:

- Foram geradas duas assertivas *assertEquals* referentes aos casos de teste testGetCategoria e testSetCategoria (linhas 7 e 12), pois o tipo de retorno é uma *String;*
- Foram geradas duas assertivas *assertTrue* referentes aos casos de teste testGetPreco e testSetPreco (linhas 16 e 21), pois o tipo de retorno é um *double*, validando dessa forma o algoritmo que busca as assertivas usadas nos casos de teste.

8.2.2 Especificação de Teste Unitário para a classe CatalogoCategoria

A Figura 85 mostra a arquitetura do projeto de teste da classe CatalogoCategoria. Nesta especificação foram definidos os elementos *Setup* e *Teardown*. Os diagramas das Figuras 86 e 87 mostram o comportamento dos métodos *Setup* e *Teardown* respectivamente. Os diagramas das Figuras 88, 89, 90 e 91 definem o comportamento dos casos de teste.

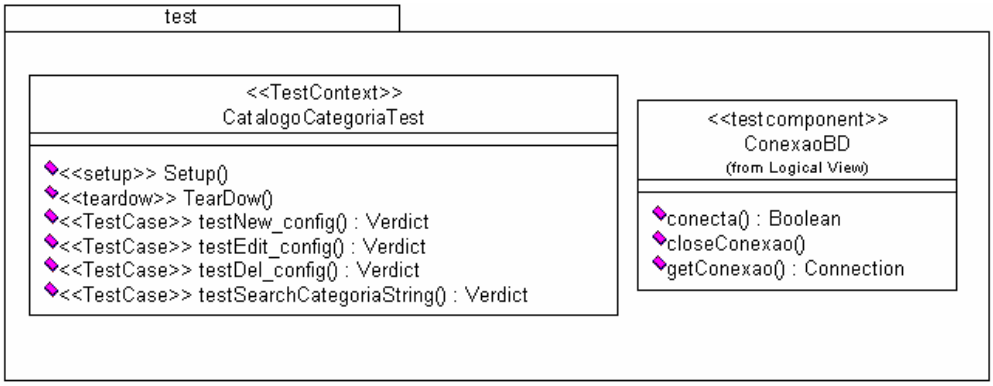


Figura 85 - Pacote de Teste Unitário para classe Catalogocategoria.

Note que no pacote de teste foi definido um componente de teste, denominado ConexaoBD. Este foi especificado para fazer a conexão com o banco de dados. Também foi especificado outro componente de teste denominado Categoria, sendo este especificado no modelo de projeto do software. A identificação destes foi feita com base na documentação recebida.

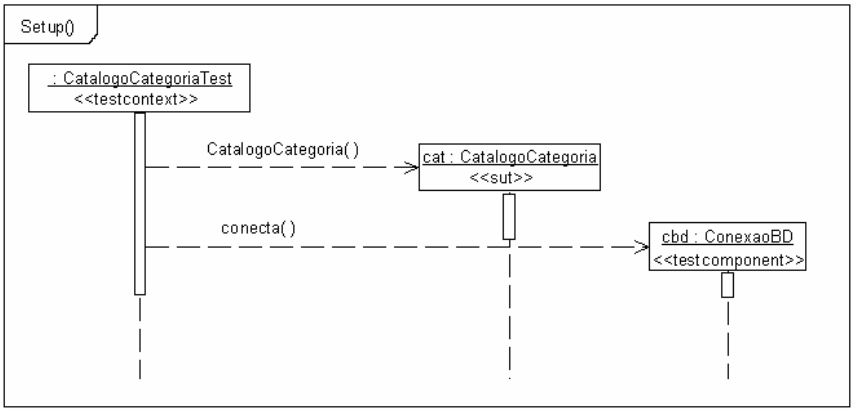


Figura 86 - Comportamento do método *Setup*.

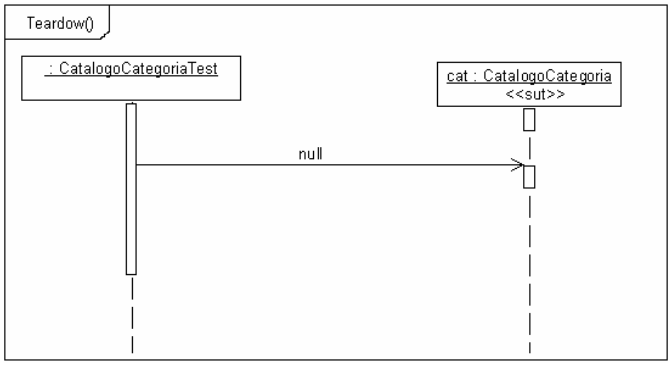


Figura 87 - Comportamento do método *teardown*.

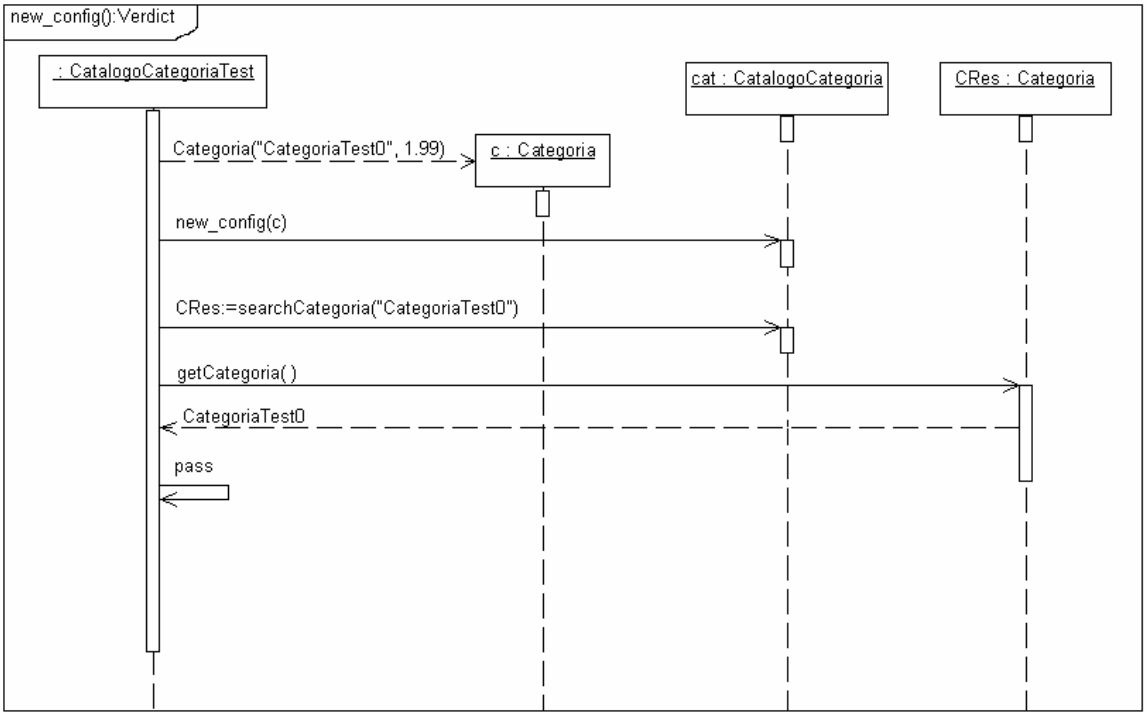


Figura 88 - Comportamento do caso de teste testNew_config.

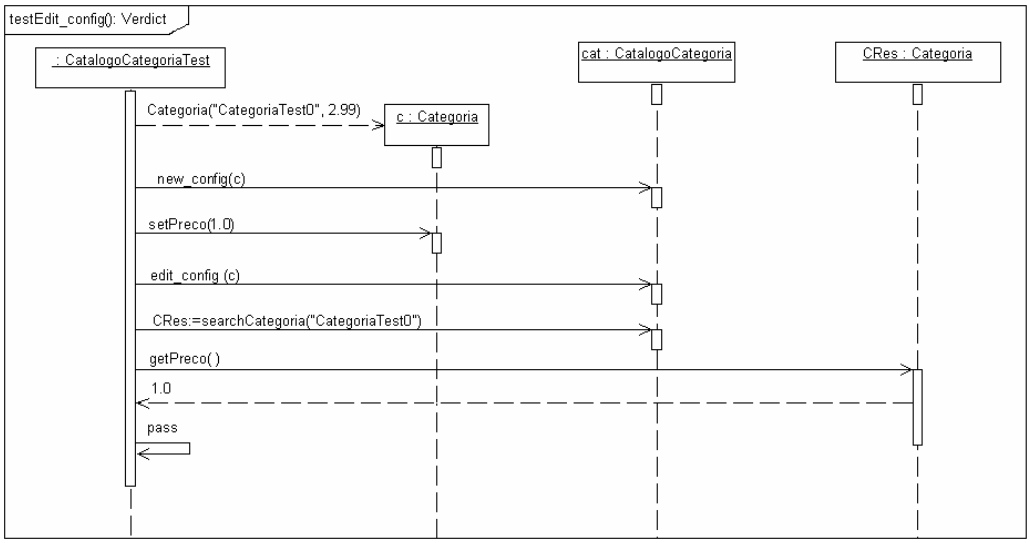


Figura 89 - Comportamento do caso de teste testEdit_config.

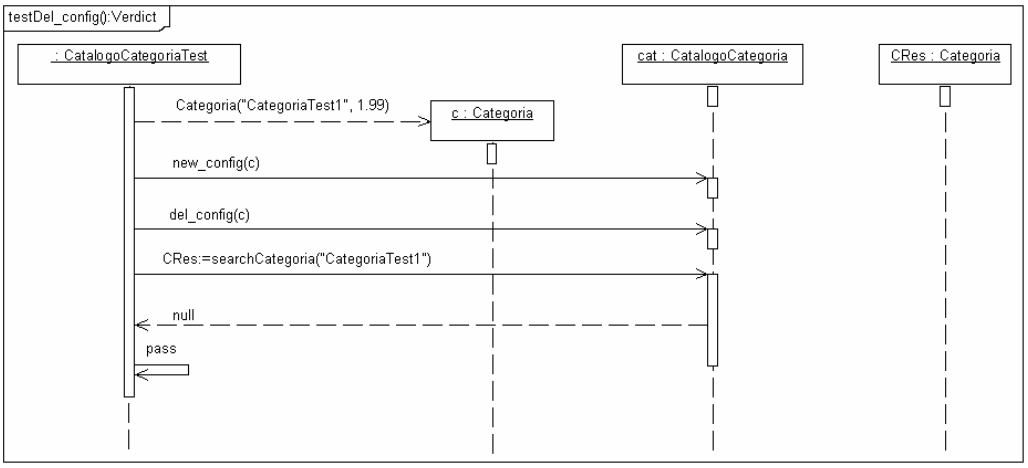


Figura 90 - Comportamento do caso de teste testDel_config.

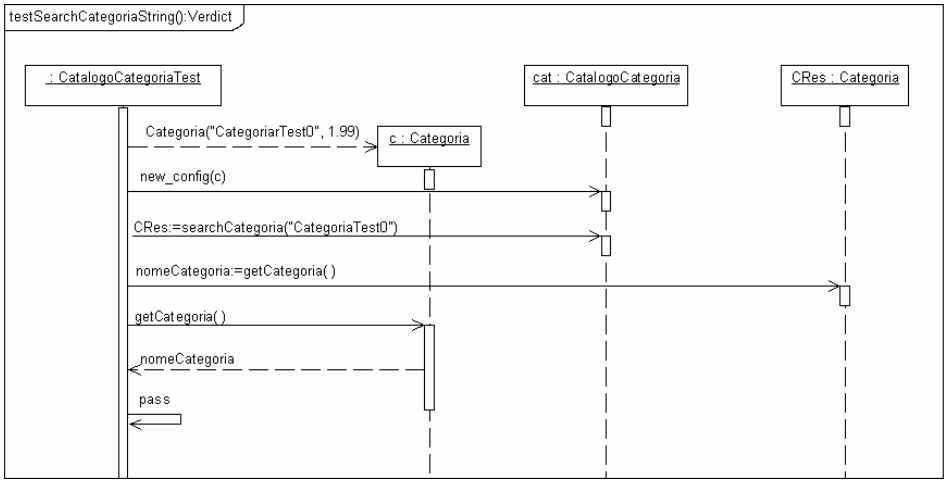


Figura 91 - Comportamento do caso de teste testSearchCategoriaString.

Considerando esta especificação, a Figura 92 ilustra o *driver* de teste gerado automaticamente para a classe CatalogoCategoria.

```
(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.tearDown();
(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)   }
(44)}
```

Figura 92 - *Driver* gerado para classe CatalogoCategoria.

Como podemos observar pela Figura 92, o *sut* foi declarado como uma variável global (linha 6), pois o mesmo é comum a todos casos de teste. Já o *TestComponent* Categoria, foi definido localmente para cada caso de teste (linhas 17, 23, 30 e 38). Em relação aos métodos

setup e *teardown,* pode-se observar que eles foram gerados corretamente conforme especificado nos diagramas de seqüência.

Outra observação em relação ao *driver* gerado é a declaração de novas variáveis locais. Veja o exemplo do caso de teste testNew_config (linhas 16 a 21). Neste, foi declarado o atributo CRes do tipo CatalogoCategoria, justificando o funcionamento correto do algoritmo que busca o tipo de dados de um método. Da mesma forma, para o caso de teste testSearchCategoriaString (linhas 37 a 42) também foi gerada corretamente a variável nomeCategoria do tipo *String* (linha 42).

Em relação às assertivas algumas observações são ressaltadas:

- Foram geradas três assertivas *assertEquals* referentes aos casos de teste testNew_config (comparando uma String), testDel_config (comparando se o objeto é nulo) e testSearchCategoriaString (comparando uma String);
- Foi gerada uma assertiva *assertTrue* referente ao caso de teste testDdit_config (comparando um double).

As Figuras 93 e 94 ilustram o código Java gerado para os *testcomponents*: Categoria e ConexaoBD respectivamente.

```
(1)package tmm.entities.Categoria;
(2)public class Categoria{
(3)    private String categoria;
(4)    private double preco;
(5)   public Categoria(String categoria, double preco){
(6)      this.categoria = categoria;
(7)      this.preco = preco;
(8)   }
(9)   public void setCategoria(String categoria){
(10)     this.categoria = categoria;
(11)  }
(12)  public String getCategoria(){
(13)     return categoria;
(14)  }
(15)  public void setPreco(double preco){
(16)    this.preco = preco;
(17)  }
(18)  public double getPreco(){
(19)    return preco;
(20)  }
(21)}
```

Figura 93 - *TestComponent* Categoria.

```
(1)package tmm.test;
(2)public class ConexaoBD{
(3)     private Connection conex;
(4)     public boolean conecta(){
(5)     }
(6)     public void closeConexao(){
(7)     }
(8)     public Connection getConexao(){
(9)         return conex;
(10)    }
(11)}
```

Figura 94 - *TestComponent* ConexaoBD.

Como podemos observar pelas Figuras 93 e 94, foi gerado somente a estrutura (esqueleto) destas classes. O primeiro componente de teste foi especificado no modelo de projeto do sistema, e como podemos observar pela Figura 93, gerou o pacote com todo o caminho (Figura 93, linha 1). Já o segundo, foi especificado no modelo de projeto de teste, gerando o pacote pertencente ao pacote de teste (Figura 94, linha 1).

8.2.3 Especificação de Teste Unitário para a classe CatalogoCliente

A Figura 95 mostra a arquitetura do projeto de teste. Da mesma forma que a classe CatalogoCategoria, nesta especificação foram definidos os elementos *Setup* e *Teardown*.

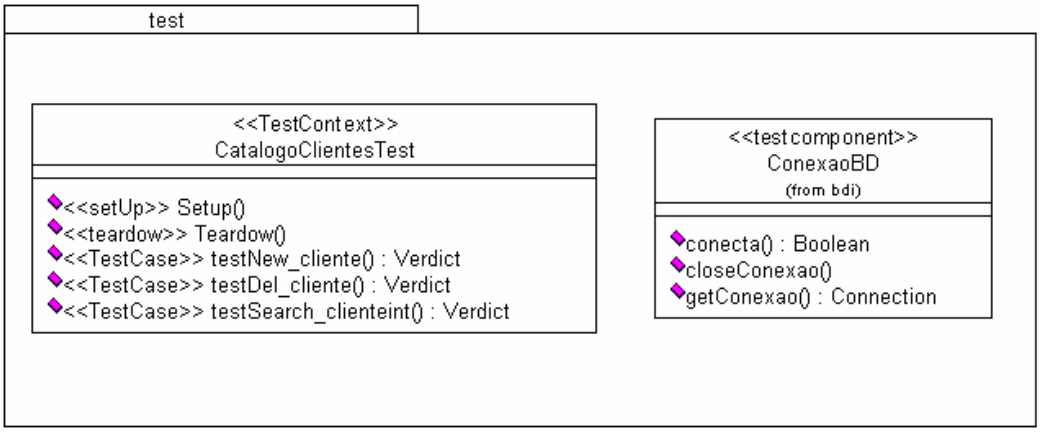


Figura 95 - Pacote de teste para classe CatalogoCliente.

Para especificação deste último *driver*, foi modificado o modelo de projeto do sistema, e acrescentando a este um diagrama de seqüência referente ao cadastramento de um novo Cliente. O objetivo da criação deste novo diagrama de seqüência é gerar o *Stub* de teste para o *driver* da classe CatalogoCliente. A Figura 96 descreve o diagrama de seqüência especificado.

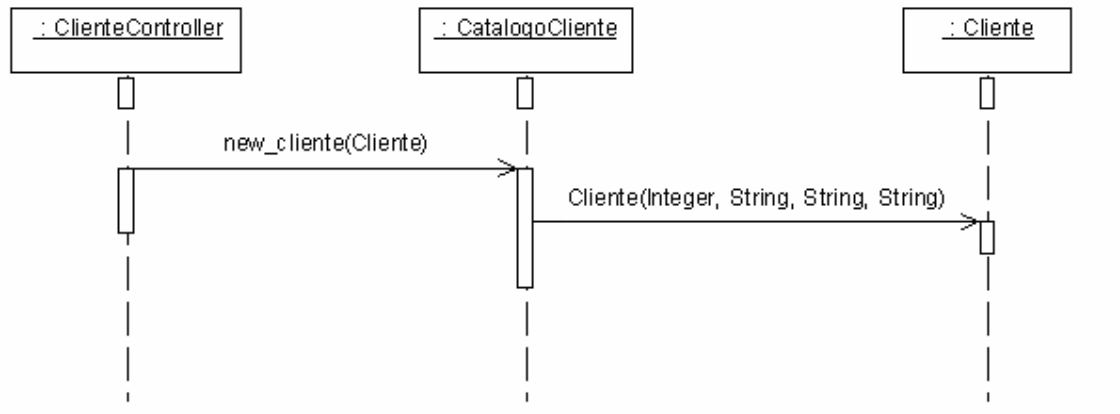


Figura 96 - Diagrama de seqüência especificando o cadastramento de um novo cliente.

Como podemos observar pela Figura 96, para incluir um novo cliente é necessário criar primeiramente um novo cliente, ou seja, a um relacionamento entre CatalagoCliente que é um *Sut* com a classe Cliente. Dessa forma, a classe cliente será gerada como um *Stub* de teste.

Outro elemento especificado neste exemplo é o elemento *testcontrol*. Embora ele não tenha sido especificado e codificado pelo testador do CTPS, neste exemplo ele será usado. O objetivo é verificar se o mesmo é gerado corretamente. A Figura 97 ilustra o elemento *Testcontrol* descrevendo a ordem de execução dos casos de teste para classe CatalogoCliente.

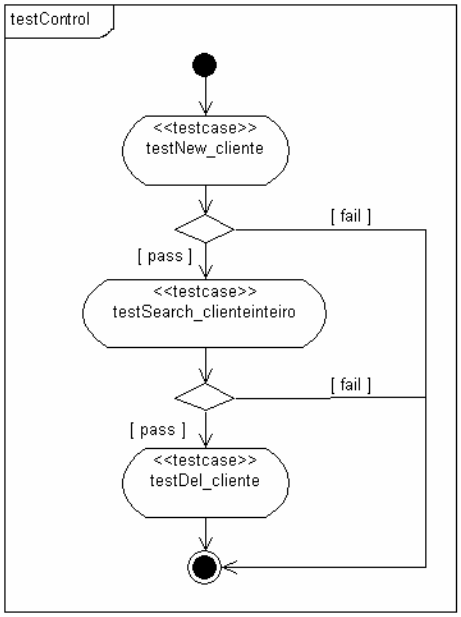


Figura 97 - Elemento *Testcontrol* para classe CatalogoCliente.

As especificações dos comportamentos dos métodos *setup* e *teardown* são descritas abaixo nas Figuras 98 e 99 respectivamente. Os diagramas das Figuras 100, 101 e 102 definem o comportamento dos casos de teste.

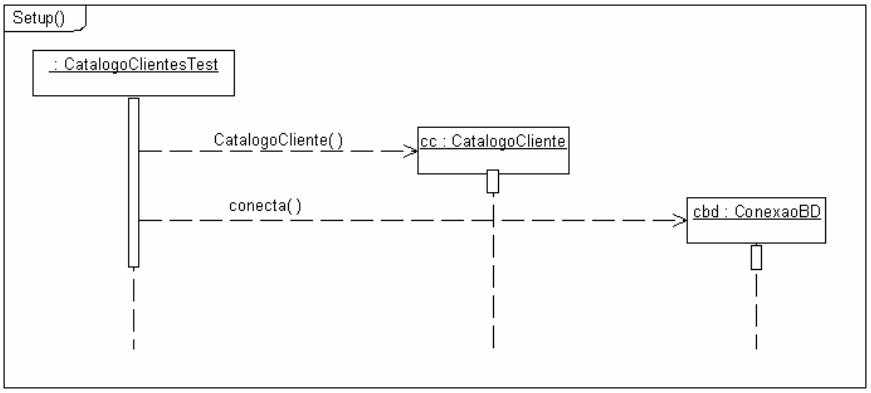


Figura 98 - Comportamento do método *Setup*.

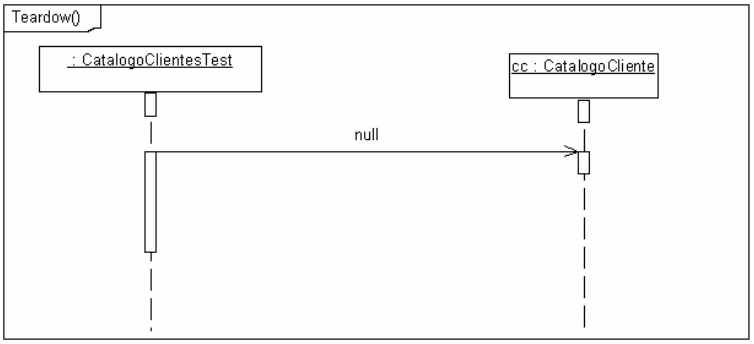


Figura 99 - Comportamento do método *Teardown*.

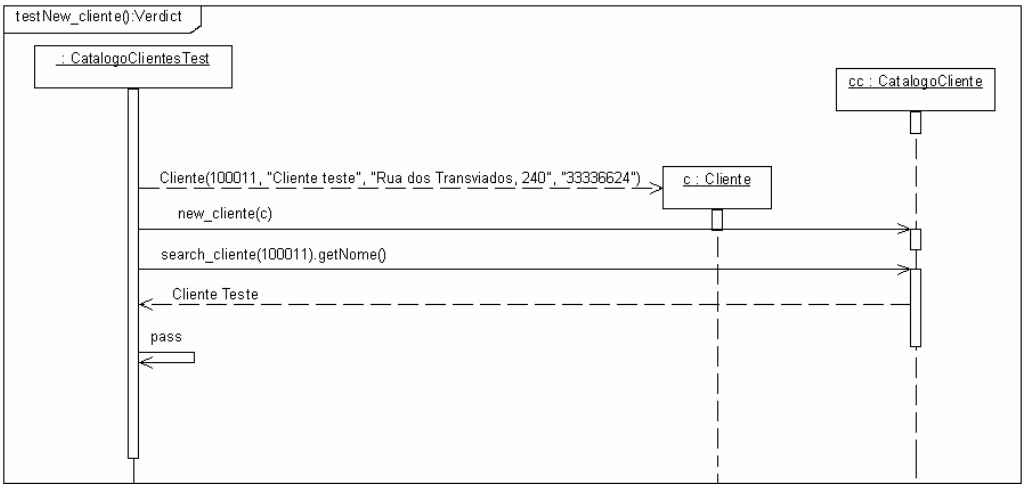


Figura 100 - Comportamento do caso de teste new_cliente.

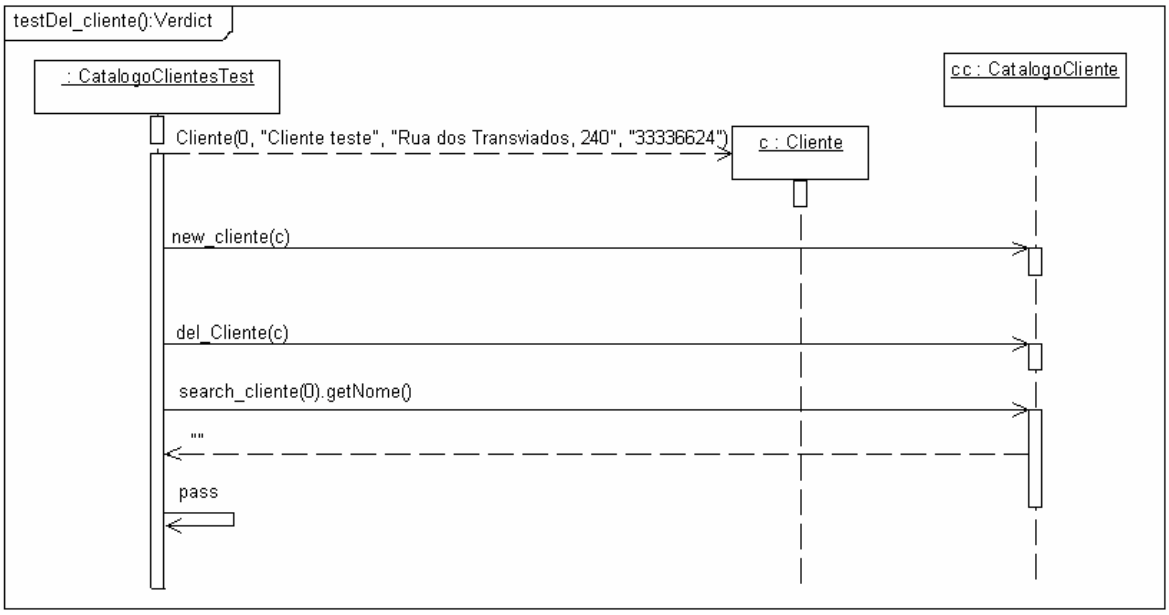


Figura 101 - Comportamento do caso de teste testDel_cliente.

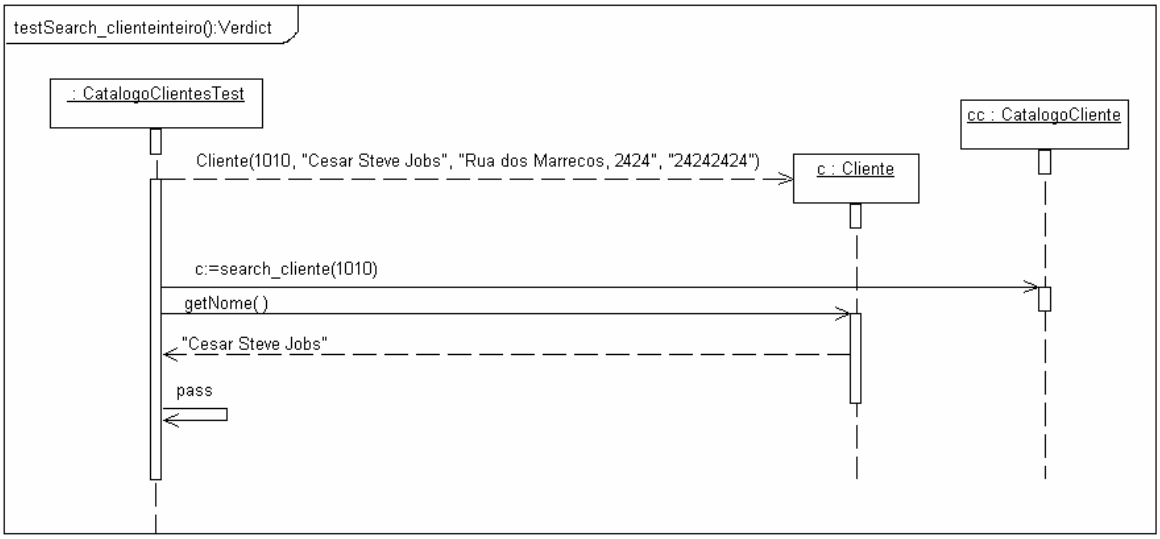


Figura 102 - Comportamento do caso de teste search_cliente.

Considerando esta especificação, a Figura 103 ilustra o *driver* de teste gerado automaticamente para a classe CatalogoCliente.

```
(1)package tmm.test;
(2)import tmm.bdi.CatalogoCliente;
(3)import tmm.entities.Cliente;
(4)import junit.framework.TestCase;
   import junit.framework.Test;
   import junit.framework.TestSuite;

(5)public class CatalogoClientesTest extends TestCase{
(6)    private CatalogoCliente cc;

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

(12)   protected void teardown() throws Exception {
(13)        super.teardown ();
(14)         cc = null;
(15)   }

(16)   public void testNew_cliente(){
(17)         Cliente c = new Cliente(100011, "Cliente teste", "Rua dos Transviados,
240", "33336624");
(18)         cc.new_cliente(c);
(19)         assertEquals(0,        cc.search_clienteByCod(100011).getNome(),"Cliente
Teste");
(20)   }

(21)   public void testDel_cliente(){
(22)         Cliente c = new Cliente(0, "Cliente teste", "Rua dos Transviados, 240",
"33336624");
(23)         cc.new_cliente(c);
(24)         cc.del_Cliente(c);
(25)         assertEquals(0, cc.search_clienteByCod(0).getNome(),"");
(26)   }

(27)   public void testSearch_clienteinteiro(){
(28)         Cliente c = new Cliente(1010, "Cesar Steve Jobs", "Rua dos Marrecos,
2424", "24242424");
(29)         Cliente c = cc.search_clienteByCod(1010);
(30)         assertEquals(0, c.getNome( ),"Cesar Steve Jobs");
(31)   }

      public static Test suite(){
         TestSuite suite = new TestSuite();
              suite.addTest(new CatalogoClientesTest("testNew_cliente"));
              suite.addTest(new TestVenda("testSearch_clienteinteiro"));
              suite.addTest(new TestVenda("testDel_cliente"));
        return suite;
      }
(32)}
```

Figura 103 - *Driver* gerado para classe CatalogoCliente.

Como podemos observar pela Figura 103, além de todas as observações já descritas nos *driver*s gerados anteriormente, neste exemplo foi gerado o elemento *TestControl*, representado pelo código em negrito. Note que também foram gerados os pacotes próprios do JUnit referentes a classe TestSuite (import junit.framework.Test e import junit.framework.TestSuite), como

definido no template. Também foi gerada o código Java para o *testcomponent* referente à classe ConexaoBD (Figura 94).

Neste exemplo, foi gerado um *Stub* de teste (classe Cliente) a partir de diagramas de seqüência do modelo de projeto de teste, validando assim o algoritmo que gera esse elemento. O mesmo foi gerado como variável local para cada caso de teste. A Figura 104 ilustra a classe Java gerado para o *Stub* Cliente.

```
(1)package tmm.entities;
(2)public class Cliente{
(3)    private int codigo;
(4)    private String nome;
(5)    private String endereco;
(6)    private String telefone;

(7)    public Cliente(Integer c, String n, String e, String t){
(8)          this.codigo = c;
(9)          this.nome = n;
(10)         this.endereco = e;
(11)         this.telefone = t;
(12)   }

(13)   public int getCodigo(){
(14)          return codigo ;
(15)   }

(16)   public void setCodigo(int c){
(17)          this.codigo =   c ;
(18)   }

(19)   public String getNome(){
(20)          return nome ;
(21)   }

(22)   public void setNome(String n){
(23)          this.nome =   n ;
(24)   }

(25)   public String getEndereco(){
(26)          return endereco ;
(27)   }

(28)   public void setEndereco(String e){
(29)          this.endereco =   e ;
(30)   }
(31)}
```

Figura 104 – Classe Java gerada para o *Stub* Cliente.

8.4 Análise Quantitativa

Os três experimentos descritos na seção 8.3 permitiram também uma análise quantitativa sobre a abordagem proposta neste trabalho.

Para isto, foram comparados os tempos envolvidos para produção dos *driver*s e *stub*s pelo testador do CPTS, bem como envolvidos na geração automática, incluindo neste o tempo

necessário para as respectivas especificações em U2TP. As Tabelas 7, 8 e 9 apresentam os resultados.

Tabela 7 - Análise do experimento para classe Categoria.

***Driver* para classe Categoria**	**Casos de teste**	**Tempo (min)**	**Linhas de Código**
Geração das especificações U2TP e geração automática de *drivers* e *stubs*	4	13	23
Codificação pelo testador	4	15	27

Tabela 8 - Análise do experimento para classe CatalogoCategoria

***Driver* para classe CatalogoCategoria**	**Casos de teste**	**Tempo(min)**	**Linhas de Código**
Geração das especificações U2TP e geração automática de *drivers* e *stubs*	4	25	44
Codificação pelo testador	4	30	51

Tabela 9 - Análise do experimento para classe CatalogoCliente.

***Driver* para classe CatalogoCliente**	**Caso de teste**	**Tempo (min)**	**Linhas de Código**
Geração das especificações U2TP e geração automática de *drivers* e *stubs*	3	23	32
Codificação pelo testador	3	25	36

A Tabela 10 mostra o tempo total e o número de linhas de código total entre a geração do código a partir de especificações U2TP com a codificação do teste feita pelo testador.

Tabela 10 - Análise dos três drivers gerados

Resumo total para as três classes	**Tempo Total (min)**	**Linhas de Código Total**
Geração das especificações U2TP e geração automática de *drivers* e *stubs*	61	99
Codificação pelo testador	70	114

Como podemos observar pela Tabela 10, o tempo total gasto na geração das especificações e geração automática de *drivers* e *stubs* foi nove minutos menor que o tempo gasto na codificação pelo testador, tendo sido geradas 15 linhas de código a menos. Desta

forma, pode-se verificar que uma especificação de teste feita com o U2TP pode ser muito bem aplicada no contexto de teste unitário, sendo que a qualidade do código gerado é melhor, pois o mesmo é gerado em um nível mais alto de abstração e por uma pessoa especialista no assunto, um engenheiro de teste.

8.5 Considerações Finais

Com base nos *driver*s gerados e nos resultados obtidos na seção 8.4 pode-se concluir que o estudo de caso foi bastante satisfatório. A Tabela 11 mostra que foi possível gerar todos os elementos e que os experimentos contemplaram todos cenários previstos neste trabalho.

Tabela 11 - Geração correta de todos elementos.

Elemento/Classe	Categoria	CatalogoCategoria	CatalogoCliente
TestContext	1	1	1
Casos de teste	4	4	3
TestComponent	0	2	1
Stub	0	0	1
Setup	0	1	1
Teardown	0	1	1
TestControl	0	0	1

Outra questão importante em relação ao estudo de caso é quanto à documentação dos artefatos de teste. Pôde-se verificar a facilidade do engenheiro de teste manter e documentar os testes a partir do projeto de teste especificado, o que possibilita facilmente a realização de testes de regressão toda vez que o projeto de teste for alterado.

9 CONSIDERAÇÕES FINAIS

Esta dissertação de mestrado apresentou uma proposta para geração automatizada de *drivers* e *stubs* de teste para ferramenta JUnit a partir de especificações de testes modeladas com o U2TP. Foram especificados elementos pertencentes ao grupo de arquitetura e comportamento de teste. Para prova de conceito, foi implementado um protótipo que permitiu gerar todo código de teste.

Como visto, o teste unitário é o principal responsável pela detecção de defeitos dentro da fase de codificação do software. Entretando, como na maioria das vezes o teste unitário é feito pelo próprio programador, ele ainda é feito de maneira *ad-hoc*, ou seja, sem nenhum planejamento. Dentre os principais problemas e dificuldades encontradas no processo de teste unitário, foi ressaltado principalmente a dificuldade de documentar e especificar casos de teste em nível de unidade, os quais dão origem aos *drivers* de teste.

Neste trabalho, foi assumido que um engenheiro de teste especifica e projeta os casos de teste usando o U2TP. Já o programador, a quem freqüentemente é atribuído o papel de testador no teste unitário, é responsável por executar estes testes. Além de minimizar problemas como dificuldade de documentação e especificação dos testes, outras vantagens podem ser ressaltadas, entre elas (i) a qualidade do código gerado é melhor, pois os testes são especificados em um nível mais alto de abstração e (ii) a qualidade dos testes também é melhor, pois estes são especificados por um engenheiro de teste, que detém maior conhecimento que o programador. O tempo aparenta ser no mínimo equivalente, como visto no estudo de caso.

Em relação à ferramenta JUnit, foi destacado que, além de ser uma das ferramentas mais usadas, é possível gerar a estrutura do *driver* de teste automaticamente através de IDEs, por exemplo. Porém, também foi destacado que o tempo, custo e esforço necessário para desenvolver o código de teste para esta ferramenta ainda é muito grande, o que muitas vezes inviabilizam o seu uso.

Através do resultado do estudo de caso, pode-se verificar que as convenções adotadas e os algoritmos propostos foram suficientes para geração de todo código de teste, pois os *drivers* foram gerados corretamente. Observa-se que o código foi gerado com 15 linhas de código a menos que na codificação dos *drivers* feita pelo testador, embora o código seja dependente da habilidade do programador do *driver*. Também vale ressaltar que embora o tempo gasto para especificação dos testes com o U2TP e o tempo gasto para codificação dos testes pelo testador

tenham sido parecidos, na especificação dos testes foi consumido um tempo menor e res ultando em uma melhor qualidade do código gerado.

A Tabela 12 descreve as principais diferenças deste trabalho em relação aos trabalhos relacionados apresentados no Capítulo 5.

Tabela 12 - Diferença deste trabalho em relação aos trabalhos relacionados.

Abordagens	Usa diagramas UML?	Interação com usuário?	Geração total do código?	Utiliza conceitos da U2TP?	Geração para JUnit?	Geração de *Stubs*?
IDE	Não	Sim	Não, apenas a estrutura	Não	Sim	Não
TestExpert	Sim, diagramas de classe	Sim	Não, apenas a estrutura	Não	Sim	Não
Especificações JML	Não	Sim	Não, é necessário especificar os valores de entrada	Não	Sim	Não
Especificações AOTDL	Não	Sim	Não, é necessário especificar os valores de entrada	Não	Sim	Não
SCENTOR	Sim, diagramas de seqüência	Sim	Parcial, é necessário especificar os valores de entrada	Não	Sim	Não
SEDITEC	Sim, diagramas de seqüência	Não	Sim	Não	Não	Sim
Ferramenta Proposta	Sim, Diagrama de classes, atividades seqüência	Não	Sim	Sim	Sim	Sim

Como podemos observar pela última linha da Tabela 12, este trabalho é o único que usa três tipos de diagramas da UML para geração do código (Diagrama de classes, atividades e

seqüência). Apenas SEDITEC é comparável a este trabalho, ao não requerer nenhuma interação com o usuário e gera todo o código de teste. Embora SEDITEC não tenha JUnit como ferramenta alvo. Este trabalho é o único de utiliza os conceitos do U2TP para geração do código de teste. Outro diferencial está relacionado com a geração de *Stub*s de Teste, pois é o único que gera esse elemento a partir de outros diagramas de seqüência do modelo de projeto de Software. No SEDITEC, os *stubs* são gerados como pré-condições.

Através deste trabalho, é possível introduzir e incentivar o teste já nas primeiras fases de desenvolvimento do software, e esse é um dos principais desafios dentro do processo de teste, ou seja, conseguir identificar e remover os defeitos o quanto antes, diminuindo assim o custo e esforço gasto nas atividades de teste.

A geração dos *Drivers* e *Stubs* de teste para ferramenta JUnit ficou limitada ao tipo de documento XMI gerado pela ferramenta *Rationale Rose*.

Entretanto, uma limitação deste trabalho, é que os algoritmos desenvolvidos são dependentes do documento XMI gerado pela ferramenta *Rationale Rose* e também do parser DOM. Dessa forma, não foi possível propor algoritmos genéricos, que gerasse todo código a partir de documentos XMI gerados por outras ferramentas, pois seria necessário converter o XMI gerado por cada uma das ferramentas para este modelo.

Em relação ao grupo Dados de Teste, não foi possível gerar esses elementos automaticamente, devido a complexidade de mapeamento para o código equivalente a ferramenta JUnit.

Outra limitação deste trabalho é que a geração de *Drivers* e *Stubs* ficou limitada a ferramenta JUnit, embora também seja possível estender para outras ferramentas, como: CppUnit e NUnit.

9.1 Trabalhos Futuros

Como trabalhos futuros, são propostos:

- Especificar e implementar os requisitos e conversões necessárias para gerar de forma genérica os elementos *DataPool* e *DataPartition*.
- Explorar as conversões necessárias para usar a U2TP na geração de *Driver*s e *Stub*s de teste para outras ferramentas de teste unitário, especialmente cppUnit e NUnit;
- Explorar e estender os conceitos da U2TP para outros níveis de teste, com a geração de código correspondente;

- Desenvolver um framework que possibilite a geração do código considerando diferentes documentos XMI.
- Integrar o protótipo em um ambiente de desenvolvimento de software, como, por exemplo, Eclipse.

BIBLIOGRAFIA

[BEC98] Beck, K.; Gama, E. “Test infected: Programmers love writing tests”. Java Report vol. 3-7, July 1998, pp. 51-56.

[BEC03] Beck, K. “Test-Driven Development”. Addison-Wesley, 2003, 220p.

[BUR03] Burnstein, I. “Pratical software testing: a process-oriented approach”. Springer-Verlag, 2003, 709p.

[CAG04] Cagnin, M.; Maldonado, J.; Chain, A. “Reuso na Atividade de Teste para Reduzir Custo e Esforço d e VV&T no Desenvolvimento e na Reengenharia de Software”**.** In: XVIII Simpósio Brasileiro de Engenharia de Software (SBES), 2004, pp.71-84.

[CHE02] Cheon, Y.; Leavens, G. T. “A Simple and pratical approach to unit testing: the JML and JUnit way”. In: 16th European Conference Object-Oriented Programming (ECOOP), 2002, pp. 231-255.

[CPP05] CppUnit. “Project: CppUnit - C++”. Capturado em: http://sourceforge.net/projects/cppunit, July 2005.

[CRE04] Crespo, A. N.; Silva, O. J.; Borges, C. A.; Salviano, C. F.; Jino, M. “Uma Metodologia para Teste de Software no Contexto da Melhoria de Processo”. In: III Simpósio Brasileiro de Qualidade de Software (SBQS), 2004, 15p.

[DAI03] Dai, J.; Grabowski, A. R. “The UML 2.0 Testing Profile and its Relation to TTCN-3”. In: 15th IFIP International Conference on Testing of Communicating Systems (TestCom), 2003, pp. 79-94.

[DAI04A] Dai, Z. R.; Grabowski, J.; Neukirchen, H.; Pals, H. “From Design to Test with UML”. In: 16th IFIP International Conference on Testing of Communicating Systems (TestCom), 2004, pp. 33-49.

[DAI04B] Dai, Z. R. “Model-Driven Testing with UML 2.0”. Technical Report TR-CTIT-04-12, Fraunhofer, 2004. Capturado em: www.fokus.gmd.de/web-dokumente/Flyer_deutsch/Publikationen_2004.pdf, May 2005, 9p.

[DOE05] Doederlein, O. P. "Byte Code XML Turbinado com Java". Javamagazine, vol. 22 - 3, Março 2005, pp. 36-48.

[DUN05] DUnit. "DUnit". Capturado em: http://dunit.sourceforge.net/, July 2005.

[EBE05] Ebert, C. "JUnit: Unit Testing and Coding in Tandem". IEEE Software, vol. 22-4, July 2005, pp. 12-15.

[ECL05] Eclipse. "Eclipse". Capturado em: http://www.eclipse.org, July 2005.

[FOW00] Fowler, M; Scott. K. "UML essencial: um breve guia para a linguagem padrão de modelagem de objetos". Bookman, 2000, 169p.

[FRA02] Fraikin, F.; Leonhardt, T. "SeDiTeC – Testing Base don Sequence Diagrams." In: 17th IEEE International Conference on Automated Software Engineering (ASE), 2002, pp. 262-266.

[GRO03] Gross, H. "Testing and the UML – a perfect fit". Technical Report TR-110.03 Fraunhofer, 2003. Capturado em: www.iese.fraunhofer.de/pdf_files/iese-110_03.pdf, Apr 2005, 53p.

[HEI01] Heineman, G. T. "Component-based software engineering: putting the pieces together". Addison-Wesley, 2001, 818p.

[IBM05] IBM. "IBM Developerworks – Test Expert". Capturado em: http://www-128.ibm.com/developerworks/rational/library/content/03July/2500/2834/Rose/rational_rose.html, May 2005.

[IEE98] IEEE. "IEEE Standard for Software Test Documentation". Technical Norm, IEEE, 1998, The Institute of Electrical and Electronics Engineers, 1998. Capturado em: http://standards.ieee.org/reading/ieee/std_public/description/se/8291983desc.html, June 2005, Std 829-1998 59p.

[JUN04] JUnit. "JUnit". Capturado em: http://www.junit.org/index.htm, Dec 2004.

[LAR00] Larmam, G. "Utilizando UML e padrões: uma introdução à análise e ao projeto orientado a objetos". Bookman, 2000, 492p.

[LEU98] Leung, H. K. N. “Test Tools for the Year 2000 Challenges”. In: 24 th. EUROMICRO Conference (EUROMICRO) , 1998, pp. 830-837.

[MOL03] Molinari, L. “Testes de Software: Produzindo Sistemas Melhores e Mais Confiáveis”. Érica, 2003, 228p.

[NET05] NetBeans. “NetBeans”. Capturado em: http://www.netbeans.org, May 2005.

[NUN05] NUnit. “NUnit”. Capturado em: http://www.nunit.org/, July 2005.

[OLA03] Olan, M. “Unit Testing: Test Early, Test Often”. Journal of Computing Sciences in Colleges , vol. 19-2, Dec 2003, pp. 319-328.

[OMG03] OMG. “Model Driver Architecture”. Technical Report PTC/03-06-01, OMG, 2003. Capturado em: http://www.omg.org/docs/omg/03-06-01.pdf, July 2003, 62p.

[OMG04A] OMG. “UML 2.0 Testing Profile Specification”. Technical Report PTC/04-04-02, OMG, 2004. Capturado em: http://www.omg.org/docs/ptc/04-04-02.pdf, June 2004, 114p.

[OMG04B] OMG. “UML 2.0 Superstructure Specification”. Technical Report PTC/05-07-04, OMG, 2004. Capturado em: http://www.omg.org/cgi-bin/doc?formal/05-07-04, Nov. 2004, 804p.

[OMG05] OMG. “MOF 2.0/XMI Mapping Specification”. Technical Report PTC/05-09-01, OMG, 2005. Capturado em: http://www.omg.org/cgi-bin/doc?formal/2005-09-01, Jul. 2005, 120p.

[OMG06] OMG. “Meta Object Facility (MOF) Specification”. Technical Report PTC/06-06-01, OMG, 2006. Capturado em: http://www.omg.org/cgi-bin/doc?formal/2006-01-01, Oct. 2004, 90p.

[PET02] Peters, J. F. “Engenharia de Software: teoria e prática”. Campus, 2002, 602p.

[PRE95] Pressman, R. S. “Engenharia de software”. Makron, 1995, 1056p.

[PRE01] Pressman, R. S. “Software engineering: a practitioner's approach”. McGraw-Hill, 2001, 888p.

[SCH00] Schneider, A. “JUnit best practices”**.** Capturado em: http://www.javaworld.com/javaworld/jw-12-2000/jw-1221-junit.html, Dec. 2000.

[VBU05] VBUnit. “VBUnit”. Capturado em: http://www.vbunit.org/, Jun 2005.

[ZON04] Zongyuan, Y.; Guoqing, X; Haitao, H; Qian, C; Ling, C; Fengbin, X. “JAOUT: Automated Generation of Aspect-Oriented Unit Test”. In: 11th Asia-Pacific Software Engineering Conference (APSEC), 2004, pp. 374-381.

[WIT01] Wittevrongel, J.; Maurer, F. “SCENTOR: Scenario-Based Testing of E-Business Applications”. In: 10th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2001, pp. 41-48.

[W3C05] W3C. World Wide Web. Capturado em: http://www.w3.org/, Jul, 2005.

ANEXOS

Anexo I – Descrição dos métodos públicos e privados da Classe Algoritmo.

Métodos públicos:

Nome	Retorno	Descrição
getTestContext()	*TestContext*	Método que retorna um objeto "*testcontext*".
getSuts()	ArrayList	Método que retorna uma lista de objetos "*sut*".
getTestCases()	ArrayList	Método que retorna uma lista de objetos "*testcase*" com os respectivos comportamentos.
getSetup()	*setup*	Método que retorna um objeto "*setup*" com o respectivo comportamento.
getTeardown()	*teardown*	Método que retorna um objeto "*teardown*" com o respectivo comportamento.
getTestComponents()	ArrayList	Método que retorna uma lista de objetos "*TestComponent*".
getStubs()	ArrayList	Método que retorna uma lista de objetos "Sutb".

Métodos privados:

Nome	Retorno	Descrição
getExtendedElementsByStereotype (String stereotype)	ArrayList	retorna uma lista com os "xmi.id" das classes e/ou operações que o estereótipo estende.
getSutById(String idClass)	*Sut*	retorna um objeto "*sut*" através do "xmi.id" da classe.
getClass(String idClass)	String	retorna o nome de uma classe a partir do "xmi.id" da classe.
getObjeto(String idClass)	String	retorna o nome da instância do elemento "*testComponent*" ou "*stub*".
getObjetosSu(String IdSut)	ArrayList	retorna uma lista de nomes das instâncias do "*sut*"
getTestCase(String idOperation)	*testcase*	retorna um objeto "*testcase*" através do "xmi.id" da operação.
getAttribute(String idClass)	ArrayList	retorna uma lista de atributos a partir do "xmi.id" da classe.
getOperation(String idClass)	ArrayList	retorna uma lista de operações a partir do "xmi.id" da classe.
getParameter(String IdOperation)	ArrayList	retorna uma lista de parâmetros a partir do "xmi.id" da operação.
getTypeData(String IdType)	String	retorna o nome do tipo de dados a partir do "xmi.id" do tipo de dados.
getConstrutor(String IdClass)	ArrayList	retorna uma lista de métodos construtores a partir do "xmi.id" da classe.
getIdOperation(String nameOperation)	String	retorna o "xmi.id" de uma operação a partir do "name" da operação.
getComportamentoSetup(String nameOperation)	comportamento	retorna um objeto "comportamento" a partir do nome da operação com o estereótipo "*setup*".
getComportamentoTeardown(String nameOperation)	Comportamento	retorna um objeto "comportamento" a partir do nome da operação com o estereótipo "teadow".
getAssertiva(String DataType)	String	retorna a assertiva usada no caso de teste a partir do nome do tipo de dados.
verificaSut(String IdClass)	boolean	retorna "true" caso o "xmi.id" passado por parâmetro for um *sut*, caso contrário, retorna "false".
getTestControl(String nameTestCase)	*TestControl*	retorna um objeto do tipo "*testcontrol*" a partir do nome do caso de teste.
getComportamentoTestCase(String nameTestCase)	comportamento	retorna um objeto do tipo "comportamento" a partir do nome do caso de teste.
getSetUp(String IdOperation)	*setup*	retorna um objeto "*setup*" a partir do "xmi.id" da operação que possui o estereótipo "*setup*".
getTeardown(String IdOperation)	*teardown*	retorna um objeto "*teardown*" a partir do "xmi.id" da operação que possui o estereótipo "*teardown*".

Anexo II - Algoritmos

Algoritmo 1: getClassById(String IdClass)

```
Algoritmo: getClassById(String idClass);
Entrada: nameMessage;
Saída: boolean;
1 Iterator NodeListClass = parser.getNodeListClass().iterator;
2 enquanto NodeListClass.hasNext()faça
3    Node node := NodeListClass.next();
4   se node.xmiId == idClass então
5      return node.name;
```

Este algoritmo retorna o nome de uma classe a partir de seu identificador.

Algoritmo 2: verificaConstrutor(String nameMessage)

```
Algoritmo: verificaConstrutor(String nameMessage);
Entrada: nameMessage;
Saída: boolean;
1 Iterator NodeListClass = parser.getNodeListClass().iterator;
2 enquanto NodeListClass.hasNext()faça
3    Node node := NodeListClass.next();
4   se node.name == nameMessage então
5      return true;
6   senão return false;
```

Este algoritmo retorna “true” caso a mensagem do diagrama de seqüência for igual ao construtor de uma classe, caso contrário retorna “false”.

Algoritmo 3: getIdOperation(String nomeMessage, String IdClass)

```
Algoritmo: getIdOperation(String nomeMessage, String IdClass);
Entrada: nomeMessage, IdClass;
Saída: type;
1 NodeListClass = parser.getNodeListClass();
2 para i de 0 até tamanho de NodeListClass faça
3  Node nodeClass := NodeListClass[i];
4  se nodeClass.xmiId == IdClass então
5   //busca filhos da classe, as operações
6    NodeListOperation := NodeListClass.getChildNodes();
7    para j de 0 até tamanho de NodeListOperation faça
8      Node nodeOperation := NodeListOperation[j];
9     se nodeOperation.name == nomeMessage então
10       //busca filhos da operação, os parâmetros
11       NodeListParameter := NodelistOperation.getChildNodes();
12       para k de 0 até tamanho de NodeListParameter faça
13         Node nodeParameter := NodeListParameter[k];
14         return nodeParameter.type;
```

Este algoritmo busca o “xmi.id” de uma operação (método) através do nome da mensagem do diagrama de seqüência.

Algoritmo 4: getDataType(String type)

```
Algoritmo: getDataType(String type);
Entrada: Type;
Saída: nome;
1 Iterator NodeListDataType := parser.getNodeListDataType().iterator;
2 Iterator NodeListClass := parser.getNodeListClass().iterator;
3 enquanto NodelistDataType.hasNext() faça
4   Node nodeType := NodelistDataType.next();
5    se nodeType.xmiId == type então
6      return nodeType.name;
7   else
8      enquanto NodeListClass.hasNext() faça
9        Node nodeClass := NodeListClass.next();
10         se nodeClass.xmiId == yype então
11           return nodeClass.name;
```

Este algoritmo busca o nome do tipo de dados a partir do identificador do tipo de dados. O nome pode ser tanto um tipo primitivo (linhas 2 a 6) como um tipo de classe (linhas 8 a 11).

Algoritmo 5: getAssertiva(String nomeType)

```
Algoritmo: getAssertiva(String nomeType);
Entrada: nomeType;
Saída: assertiva;
1 se nomeType == "double" ou nomeType == "int" ou nomeType == "boolean" ou nomeType == "byte" então
2    return assertTrue;
3 else return assertEquals;
```

Este algoritmo retorna a assertiva usada no caso de teste a partir do nome do tipo de dados passado por parâmetro.

Algoritmo 6: getConstrutores(String IdClass)

```
Algoritmo: getConstrutores(String IdClass);
Entrada: IdClass;
Saída: ListConstrutores;
1 NodeListClass := parser.getNodeListClass();
2 para i de 0 até tamanho de NodeListClass faça
3    Node nodeClass := NodeListClass[i];
4    se nodeClass.xmiId == IdClass então
5    //busca filhos da classe, as operações
6    NodeListOperation := NodeListClass.getChildNodes();
7    para j de 0 até tamanho de NodeListOperation faça
8      Node nodeOperation := NodeListOperation[j];
9      se verificaConstrutor(nodeOperation.name) == true então
10       //busca filhos da operação, os parâmetros
11       NodeListParameter := NodeListOperation.getChildNodes();
12       para k de 0 até tamanho de NodeListParameter faça
13       Node nodeParameter := NodeListParameter[k];
14       ListParametrosConstrutor.add(getParametros(nodeParameter.type));
15       ListConstrutores.add(nodeOperation.name, ListParametrosConstrutor);
16  return ListConstrutores;
```

Esse algoritmo busca os construtores de uma classe qualquer, exceto *sut*, a partir do "xmi.id" da classe passado por parâmetro. Primeiramente a lista com os nodos <UML:Class> é percorrida até encontrar o atributo "xmi.id" igual ao IdClass. Caso encontrado, são identificados os filhos da classe, isto é, as operações. Cada nodo <UML:Operation> é percorrido e para cada nodo é verificado se o "name" da operação é igual ao construtor da classe, através do método verificaConstrutor. Se verdadeiro, são identificados os filhos da operação, isto é, os parâmetros. Assim, todos os parâmetros do construtor são armazenados na lista ListParametrosConstrutor (linha 11). Finalizando o algoritmo, o "name" e os parâmetros do construtor são adicionados na lista ListConstrutores (linha 12).

Algoritmo 7: getAttribute(String IdClass)

Algoritmo: getAtributos(String IdClass);
Entrada: IdClass;
Saída: ListAtributos;
1 NodeListClass := parser.getNodeListClass(); 2 **para** i **de** 0 **até** tamanho de NodeListClass **faça** **3** Node nodeClass **:=** NodeListClass[i]; 4 **se** nodeClass.xmiId == IdClass **então** 5 //busca filhos da classe, neste caso os atributos 6 NodeListAtributos := NodeListClass.getChildNodes(); 7 **para** j **de** 0 **até** tamanho de NodeListAtributos **faça** **8** Node nodeListAtributos := NodeListClass[j]; 9 nome := nodeListAtributos.name; 10 visibilidade := nodeListAtributos.visibility; 11 tipo := **getDataType**(nodeListAtributos.type); 12 ListAtributos.add(visibilidade + tipo + nome); 13 **return** ListAtributos;

Esse algoritmo busca os atributos de uma classe qualquer, exceto *sut*, a partir do "xmi.id" da classe. A lista com os nodes <UML:Class> é percorrida até encontrar o atributo "xmi.id" igual ao IdClass. Caso encontrado, são identificados os filhos da classe, isto é, os atributos. Assim, os nodos <UML: Attribute> são percorridos e para cada node são adicionados na lista ListAtributos, o nome, a visibilidade e o tipo do atributo, sendo este último buscado através do método getDataType(ver algoritmo 4).

Algoritmo 8: getOperation(String IdClass)

```
Algoritmo: getOperacoes(String IdClass);
Entrada: IdClass;
Saída: ListOperacoes;
1 NodeListClass := parser.getNodeListClass();
2 para i de 0 até tamanho de NodeListClass faça
3   Node nodeClass := NodeListClass[i];
4   se nodeClass.xmiId == IdClass então
5     //busca filhos da classe, neste caso as operações
6     Iterator NodeListOperation := NodeListOperation.getChildNodes();
7     para j de 0 até tamanho de NodeListOperation faça
8       Node nodeOperation := NodeListOperation[j];
9       Id := nodeParameter.xmiId;
10      //buca filhos de operação, nesse caso os parâmetros
11      NodeListParameter := NodeListOperation.getChildNodes();
12      para k de 0 até tamanho de NodeListParameter faça
13        Node nodeParameter := NodeListParameter[k];
14        se nodeParameter.kind == "inout" então
15          nome := nodeOperation.name;
16          visibilidde := nodeOperation. visibility;
17          ListOperacoes.add(visibilidde + nome + getParametros(Id));
18        se nodeParameter.kind == "return" então
19          nome := nodeParameter.name;
20          visibilidade := nodeParameter.next.visibility;
21          tipo := getDataType(nodeParameter.type);
22          ListOperacoes.add(visibilidade + tipo  + nome);
23 return ListOperacoes;
```

Esse algoritmo busca as operações de uma classe qualquer a partir do "xmi.id" da mesma passado por parâmetro. Primeiramente são percorridos todos os nodes <UML:Class> até encontrar o atributo "xmi.id" igual ao IdClass. Caso encontrado, são identificados os filhos da classe, isto é, as operações. Assim, os nodos <UML:Operation> são percorridos e para cada operação encontrada são identificados os filhos da operação, isto é, os parâmetros. Cada nodo <UML:Parameter> é percorrido, e se o atributo "kind" for igual a inout (a operação não retornada nada), então são adicionados na lista ListOperacoes, o nome, a visibilidade e os parâmetros da operação. Caso contrário, se o atributo "kind" for igual a "return", são adicionados na lista ListOperacoes, o nome, a visibilidade e o tipo da operação.

Anexo III - Drivers

Driver da Classe Categoria

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

(4)public class CategoriaTest extends TestCase {

(5)    public void testGetCategoria() {
(6)        String e = "Ouro";
(7)        Categoria c = new Categoria("Ouro", 5.50);
(8)        assertEquals("getCategoria() com problema", e, c.getCategoria());
(9)    }

(10)   public void testSetCategoria() {
(11)       String p = "Prata";
(12)       Categoria c = new Categoria(null, 5.50);
(13)       c.setCategoria(p);
(14)       assertEquals("setCatgoria() com problema","Prata", c.getCategoria());
(15)   }

(16)   public void testGetPreco() {
(17)       double e = 3.00;
(18)       Categoria c = new Categoria("Bronze", 3.00);
(19)       assertEquals("getPreco() com problema", e, c.getPreco(), 0.0);
(20)   }

(21)   public void testSetPreco() {
(22)      double p = 5.50;
(23)       Categoria c = new Categoria(“Prata”, 0.00);
(24)       c.setCategoria(p);
(25)       assertEquals("setPreco() com problema",p, c.getCategoria());
(26)   }
(27)}
```

***Driver* da Classe CatalogoCategoria**

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

(6)public class CatalogoCategoriaTest extends TestCase {
(7)    private CatalogoCategoria cat;

(8)    protected void setUp() throws Exception {
(9)           super.setUp();
(10)          cat = new CatalogoCategoria();
(11)          new ConexaoBD().conecta();
(12)   }

(13)   protected void tearDown() throws Exception {
(14)          super.tearDown();
(15)            cat = null;
(16)   }

(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)          cat.del_config(c);
(24)   }

(25)   public void testDel_config() {//funcionoiu
(26)          Categoria c = new Categoria("CategoriaTest1", 1.99);
(27)          cat.new_config(c);
(28)          cat.del_config(c);
(29)          Categoria cRes = cat.searchCategoria("CategoriaTest1");
(30)          assertNull("É null??", cRes);
(31)   }

(32)   public void testEdit_config() {
(33)          Categoria c = new Categoria("CategoriaTest0", 2.99);
(34)          cat.new_config(c);
(35)          c.setPreco(1.0);
(36)          double res = 1.0
(37)          cat.edit_config(c);
(38)          Categoria cRes = cat.searchCategoria("CategoriaTest0");
(39)          assertEquals(cRes.getPreco(), res, 0);
(40)          cat.del_config(c);
(41)          }

(42)   public void testSearchCategoriaString() {
(43)          Categoria c = new Categoria("CategoriaTest0", 1.99);
(44)          cat.new_config(c);
(45)          Categoria cRes = cat.searchCategoria("CategoriaTest0");
(46)          assertEquals(cRes.getPreco(), c.getPreco(), 0);
(47)          assertEquals("", cRes.getCategoria(), c.getCategoria());
(49)          cat.del_config(c);
(50)   }
(51)}
```

***Driver* da Classe CatalogoCliente**

```
(1)package tmm.test;
(2)import tmm.bdi.CatalogoClientes;
(3)import tmm.bdi.ConexaoBD;
(4)import tmm.entities.Cliente;
(5)import junit.framework.TestCase;

(6)public class CatalogoClientesTest extends TestCase {

(7)    CatalogoClientes cc;

(8)    protected void setUp() throws Exception {
(9)           cc = new CatalogoClientes();
(10)          new ConexaoBD().conecta();
(11)          super.setUp();
(12)          }

(13)   protected void tearDown() throws Exception {
(14)          cc = null;
(15)          super.tearDown();
(16)   }

(17)   public void testNew_cliente() {
(18)          String e = "Cliente teste";
(19)          Cliente c = new Cliente(100011, "Cliente teste", "Rua dos Transviados,
240", "33336624");
(20)          cc.new_cliente(c);
(21)          assertEquals("O método new_cliente() não está funcionando.", e,
cc.search_cliente(100011).getNome());
(22)          cc.del_cliente(c);
(23)   }

(24)   public void testDel_cliente() {
(25)          String e = "";
(26)          Cliente c = new Cliente(0, "Cliente teste", "Rua dos Transviados,
240", "33336624");
(27)          cc.new_cliente(c);
(28)          cc.del_cliente(c);
(29)          assertEquals("Método com problema", e,
cc.search_cliente(1011).getNome());
(30)   }

(31)   public void testSearch_clienteinteiro() {
(32)        String e = "Cesar Steve Jobs";
(33)        Cliente c = cc.search_cliente(1010);
(34)        assertEquals("O método search_cliente(int)", e, c.getNome());
(35)   }
(36)}
```