UNIVERSIDADE DE PERNAMBUCO

ESCOLA POLITÉCNICA DE PERNAMBUCO

Programa de Pós-Graduação em Engenharia da Computação

UPE - POLI
eCOMP
Engenharia de Computação
MESTRADO

TESTIMONIUM: Um Método para Geração de Casos de Teste a partir de Regras de Negócio Expressas em OCL

Dissertação de Mestrado

Engenharia da Computação

Edilson Mendes Bizerra Junior
Orientador: Profª. Drª. Maria Lencastre
Co-orientador: Prof. Dr. Denis Silva da Silveira

Recife, julho de 2010

UNIVERSIDADE DE PERNAMBUCO

UNIVERSIDADE DE PERNAMBUCO

ESCOLA POLITÉCNICA DE PERNAMBUCO

TESTIMONIUM: Um Método para Geração de Casos de Teste a partir de Regras de Negócio Expressas em OCL

Dissertação de Mestrado

Engenharia da Computação

Esta Dissertação é apresentada como requisito parcial para obtenção do título de Mestre em Engenharia da Computação pela Escola Politécnica de Pernambuco – Universidade de Pernambuco.

Edilson Mendes Bizerra Junior
Orientador: Profª. Drª. Maria Lencastre
Co-orientador: Prof. Dr. Denis Silva da Silveira

Recife, julho de 2010

UNIVERSIDADE DE PERNAMBUCO

Edilson Mendes Bizerra Junior

TESTIMONIUM: Um Método para Geração de Casos de Teste a partir de Regras de Negócio Expressas em OCL

Agradecimentos

Agradeço acima de tudo a Deus.

À minha mãe, meu pai e minha avó pelo carinho, amor, dedicação e compreensão.

Ao meu irmão, que mesmo tão distante, tem me dado apoio.

À minha esposa, por todo carinho e compreensão ao longo do tempo em que estamos juntos.

Agradeço de maneira especial à minha orientadora Profª. Maria Lencastre e ao meu co-orientador Prof. Denis Silveira pela disponibilidade, receptividade e orientações constantes, que tornaram realização deste trabalho possível.

Aos professores Eber Assis Schmitz e Luis Meneses pela participação da banca examinadora desta dissertação.

Ao time do Centro de Estudos e Sistemas Avançados do Recife (C.E.S.A.R.) por ter me dado a oportunidade de atingir este objetivo.

A todos que compõem o Programa de Engenharia de Computação da POLI, em especial a Georgina pelo suporte e atenção.

Em fim, a todos que contribuíram diretamente ou indiretamente para a conclusão deste trabalho.

Resumo

Neste trabalho é apresentado um método, chamado *TESTIMONIUM*, para geração de casos de teste a partir de regras de negócio, expressas em OCL. Este método tem como objetivo dar um melhor suporte à atividade de testes, permitindo gerar testes em fases iniciais do ciclo de vida do desenvolvimento de *software*, podendo expor inconsistências e ambigüidades nas primeiras fases do projeto. Um diferencial deste método em relação aos outros com propósito semelhante é que, além da geração dos casos de teste, os testes gerados podem ser validados e acompanhados de forma automatizada através da ferramenta *ANIMARE*. Para auxiliar na execução das atividades proposta no método, foi desenvolvida a ferramenta *TESTIMONIUM Tool*.

Abstract

In this work a method called *TESTIMONIUM* is proposed for the generation of test cases from business rules, expressed using OCL. The aim of this method is to give one a better support for test activities like, for example, the generation of tests in initial phases of the software development lifecycle, showing the inconsistencies and ambiguities earlier in the project. A differential of this method, compared to others with similar intention, is that, beyond the generation of test cases, the generated tests could be validated and be followed in an automatized way through the *ANIMARE* tool. To help in the execution of the proposed method, the *TESTIMONIUM Tool* was implemented.

Sumário

Índice de Figuras

Índice de Tabelas

Tabela de Símbolos e Siglas

CASE	*Computer-Aided Software Engineering*
CSV	*Comma Separated Values*
DC	Diagrama de Classes
DA	Diagrama de Atividades
IDE	*Integrated Development Environment*
MCI	Modelo Conceitual de Informação
OMG	*Object Management Group*
OCL	*Object Constraint Language*
RCP	*Rich Client Platform*
RN	Regras de Negocio
SQA	*Software Quality Assurance*
TBM	Teste Baseado em Modelos
UML	*Unified Modeling Language*
VV&T	Validação e Verificação e Teste
XMI	*XML Metadata Interchange*
XML	*eXtensible Markup Language*

Capítulo 1

Introdução

1.1 Considerações Iniciais

Atualmente, os sistemas computacionais estão presentes em diversos setores, tais como: educação, saúde, segurança, transporte, e indústria. A evolução tecnológica dos últimos tempos demanda *software* e *hardware* maiores, com mais funcionalidades e cada vez mais complexos. Tais características fazem com que os sistemas fiquem mais suscetíveis a conter erros. A presença de erros pode acarretar diversos tipos de problemas, sejam financeiros, de prazo e, dependendo da aplicação, de segurança - podendo até por em risco vidas humanas. Com isto, é indispensável o uso de práticas que garantam a qualidade desde o início e durante todo o processo de desenvolvimento do sistema.

Para Inthurn (2001), um teste de *software* "tem como objetivo aprimorar a produtividade e fornecer evidências da confiabilidade e da qualidade do *software* em complemento a outras atividades de garantia de qualidade ao longo do processo de desenvolvimento de *software*".

Porém, realizar testes demanda muito esforço e tempo. Segundo Pressman (2006), cerca de 40% do esforço gasto em um projeto é dedicado à etapa de teste. A falta de testes pode trazer conseqüências ainda maiores, pois o custo da detecção de um erro após a entrega do produto é, pelo menos, 100 vezes maior do que se o erro tivesse sido detectado na fase de definição do sistema.

Os casos de testes auxiliam a realização dos testes de *software*. Um caso de teste consiste em um par ordenado $(d, S(d))$ tal que d é um elemento (dado de teste) do domínio de entrada de um programa P, denotado por $D(P)$, ou seja, $d \in D(P)$ e $S(d)$ é a saída esperada para uma dada função da especificação, quando d é utilizado como entrada; o domínio de entrada $D(P)$ é o conjunto de todos os possíveis valores que podem ser utilizados para executar P (Delamaro *et al*., 2007). Em princípio, o programa deveria ser executado com todas as combinações possíveis do

domínio de entrada, entretanto sabe-se que, geralmente, o teste exaustivo é impraticável, pois mesmo em programas simples o número de casos de testes é tão exorbitante, em alguns casos infinito, o que torna sua execução inviável.

Considere, por exemplo, um programa simples que calcula a soma de três números. O domínio de entrada é formado por todas as triplas de números inteiros (x, y, z). Isto produz um conjunto de cardinalidade $2^n * 2^n * 2^n$, onde n é o número de bits usado para representar um número inteiro, que varia de acordo com a arquitetura do computador. Em uma arquitetura de 32 bits teríamos o equivalente a 2^{96} = 79228162514264337593543950336 casos de teste. Supondo que cada caso de teste pudesse ser executado em um milissegundo, precisaríamos de 25.106.425.948.390.400 séculos para que todos pudessem ser executados. Fazendo uma analogia com uma idade da terra que é de aproximadamente 4,6 bilhões de anos, ou seja, algo em torno de 46.000.000 séculos, fica evidente o quão exorbitante esse tempo representa. Para testar todo domínio de entrada de um programa simples seria necessário um tempo 545.791.868 vezes maior que a idade da Terra, algo inviável. Devido a isso, bons casos de testes devem ser construídos: um bom caso de teste é aquele que tem alta probabilidade de achar erros, com uma quantidade mínima de tempo e esforço (Pressman, 2006).

Com objetivo de diminuir os custos, algumas técnicas e critérios que ajudam na condução e avaliação da atividade de teste têm sido propostas (Pressman, 2006; Delamaro *et al.*, 2007). O que diferencia essas técnicas é a fonte da informação usada para construir os casos de teste.

Uma das técnicas que tem sido muito incentivada na área de teste de *software* é o Teste Baseado em Modelos (TBM) (Bertolino, 2007). A especificação de modelos é uma técnica amplamente empregada em diversas modalidades de engenharia. Na Engenharia de *Software*, modelos podem ser elaborados em diversas atividades e em diferentes estágios do desenvolvimento de sistemas (Pressman, 2006). Um modelo de sistema é representado em uma linguagem de modelagem, que pode empregar uma combinação de representações gráficas e textuais (Silveira, 2009). Através dos modelos, a técnica de TBM permite a automação do processo de teste de *software*.

A aplicação da técnica TBM pode propiciar uma maior descoberta de erros, omissões, inconsistências e redundâncias ainda nas fases iniciais do processo de desenvolvimento de um *software*. Isto possibilita que os custos com a remoção de erros nessa fase sejam menores (Ryser & Glinz, 1999). Outro ganho possível, com uso de modelos, é a facilidade de atualização dos mesmos em caso de alterações ou adição de funcionalidades ao sistema, uma vez que é mais simples alterar um modelo do que o sistema propriamente dito.

A *Unified Modeling Language* (UML) (Booch *et al.*, 1998) é uma linguagem de modelagem padrão definida pelo *Object Management Group* (OMG) (OMG, 2005) para

documentar e visualizar os artefatos que são especificados e construídos durante as fases de análise e projeto de um sistema. A UML tem sido bastante difundida tanto nas empresas como no meio acadêmico, no contexto do desenvolvimento de Sistemas de Informação orientados a objeto ou baseados em componentes, tornando os seus modelos um padrão de fato para documentação.

Inicialmente, a UML não era suficientemente refinada para fornecer todos os aspectos relevantes de uma especificação. Havia uma necessidade de definir restrições adicionais sobre objetos nos modelos. Essas restrições eram descritas geralmente em linguagem natural, que é suscetível a apresentar ambigüidade quando da descrição dos objetos. Em 1997 a OMG minimizou este problema através da adoção da linguagem *Object Constraint Language* (OCL). A OCL é uma linguagem de expressões para especificar restrições sobre modelos orientados a objetos ou outros artefatos da linguagem UML, é uma forma de complementar a parte gráfica dos modelos, uma vez que permite descrever restrições que não conseguem ser diagramaticamente representadas. A OCL é uma linguagem precisa, textual e formal. Tal formalidade assegura a inexistência de interpretações ambíguas para as mesmas restrições. Uma das suas principais características é que seu uso não exige um forte conhecimento matemático, como ocorre nos modelos Z (Spivey, 1998) e VDM (Jones, 1990).

Uma abordagem baseada em modelos para geração de casos de testes tem potencial para dar um maior suporte à realização dos mesmos e, dessa forma, contribuir para a redução de custo e tempo gastos. A automação baseada em modelos é outra extensão desse potencial, já que a ausência de ferramentas para automatização da aplicação das técnicas e critérios torna a atividade de teste onerosa, propensa a erros e limitada a programas muito simples.

1.2 Objetivos

Apesar de na literatura existirem vários trabalhos que têm como objetivo facilitar o processo de testes, derivando os casos de testes a partir de modelos (Hartmann, 2000), (Cheon & Avila, 2010), (Kundu & Samanta, 2009), (Borba *et al.*, 2007), (Orozco *et al.*, 2009), a maioria não usa uma formalização para as regras de negócio.

Este trabalho tem como objetivo a geração de casos de testes a partir de modelos que sejam expressivos e incluam certo grau de formalização das regras de negócio. O intuito é a validação do próprio modelo, desde que expressos em OCL, uma vez que, o modelo é mais simples que a programação do sistema em teste, ou pelo menos mais fácil de verificar, modificar e manter.

Para isso, esta dissertação se propôs a definir um método para geração de casos de testes a partir de regras de negócio expressas na linguagem OCL, chamado *TESTIMONIUM*.

O método *TESTIMONIUM* está dividido nas seguintes etapas: a) Transformar as entradas (modelos) em uma estrutura interna de representação; b) Identificar todos os elementos presentes nas entradas (modelos); c) Gerar valores para cada elemento; e d) Combinar os valores para a geração dos cenários de testes.

Além disso, um diferencial do método *TESTIMONIUM* é que os testes gerados poderão ser integrados e validados de forma automatizada através de método/ferramenta de animação conhecido como *ANIMARE* (Silveira, 2009). O *ANIMARE* é um método para validação dos processos de negócio em relação às regras de negócio através da animação. A validação é obtida a partir da execução de cenários, cada um deles contendo instâncias dos objetos utilizados na animação do processo.

Este trabalho também apresenta a ferramenta *TESTIMONIUM Tool* que implementa e dá suporte ao método proposto, fornecendo apoio automatizado em todas as suas etapas.

1.3 Metodologia

Para a proposta e desenvolvimento de um método para dar suporte à geração de testes, algumas metas e estratégias foram estabelecidas:

i. Estudar os critérios de testes existentes na literatura especializa;
ii. Definir um método, chamado *TESTIMONIUM*, para geração de casos de teste a partir do modelo conceitual de informação (Diagrama de Classes – DC) e das regras de negócio expressas em OCL;
iii. Definir e a implementar uma ferramenta de suporte ao método proposto;
iv. Realizar um estudo de caso para analisar e avaliar o funcionamento do método *TESTIMONIUM*;
v. Integrar o método *TESTIMONIUM* ao método *ANIMARE*.

Como estratégia para viabilizar a realização do trabalho proposto, a seleção e a otimização dos casos de testes gerados não foi objetivo do método *TESTIMONIUM*. Isto permitiu reduzir a complexidade inerente a estas etapas. O método proposto se limitou apenas na geração de um subconjunto de casos de teste a partir de heurísticas baseadas nos critérios de testes referentes ao particionamento de equivalência e na análise do valor limite.

1.4 Organização do Trabalho

Este capítulo introduziu o trabalho, apresentando a motivação para realizá-lo e o objetivo a ser alcançado. O restante do trabalho está organizado da seguinte forma:

O Capítulo 2 apresenta um referencial teórico sobre os principais tópicos relacionados a este trabalho, especificamente, Teste de *software*, a linguagem OCL e o método *ANIMARE*.

O Capítulo 3 descreve o método *TESTIMONIUM* proposto, detalhando todas as suas etapas.

O Capítulo 4 fornece uma visão geral da ferramenta desenvolvida através da realização de um estudo de caso.

O Capítulo 5 apresenta os trabalhos relacionados.

O Capítulo 6 conclui este trabalho, apresentando suas contribuições, limitações e perspectivas de trabalhos futuros.

O Apêndice “A” possui o formato dos arquivos que representam o modelo conceitual de informação (DC).

No Apêndice “B” encontram-se os artefatos utilizados no estudo de caso apresentado no Capítulo 4.

Capítulo 2

Referencial Teórico

2.1 Considerações Iniciais

O mercado cada vez mais competitivo e exigente faz com que a preocupação com a qualidade dos produtos e serviços produzidos pela indústria seja cada vez maior, a fim de reduzir problemas com custos, atender prazos, aumentar produtividade, etc.

De acordo com Pressman (2006), um *software* possui qualidade se ele atende às necessidades dos clientes e não possui erros. Ainda, segundo o autor, a garantia da qualidade de um *software* está relacionada à aplicação de métodos e ferramentas de forma efetiva, realização de revisões, controle de mudanças, cumprimento de padrões, utilização de técnicas de medição e ainda o desenvolvimento e execução de estratégias e técnicas de testes.

Este capítulo apresenta um referencial teórico sobre os principais tópicos relacionados e necessários à compreensão deste trabalho. A Seção 2.2 descreve a área de Teste de *Software*, suas fases e seus critérios. A Seção 2.3 apresenta a linguagem OCL e suas características. A Seção 2.4 expõe o método *ANIMARE* e suas etapas. Finalmente, a Seção 2.5 apresenta considerações finais sobre este capítulo.

2.2 Teste de *Software*

A construção de *software*, assim como a maioria das atividades de engenharia, está propensa a diversos tipos de problemas que podem acarretar a criação de um produto diferente do esperado. O erro humano surge como a causa raiz na maioria destes problemas. Isto se deve ao fato que, mesmo com a utilização de métodos e ferramentas de engenharia de *software*, grande parte das atividades de engenharia, assim como na construção de *software*, depende da habilidade, da interpretação e da execução das pessoas que o constroem (Delamaro *et al*., 2007).

Existem vários métodos e técnicas para evitar, diminuir e anular a presença de tais erros. Dentre elas, revisões técnicas, inspeções, *walkthroughs* e teste de *software* (SWEBOK, 2001b; Peters & Pedrycz, 2001). Estas atividades são coletivamente chamadas de VV&T e têm como objetivo assegurar que tanto a forma pelo qual o *software* está sendo construído quanto o produto final em si estejam em conformidade com o que foi especificado (Delamaro *et al*., 2007; Pressman, 2006).

Enquanto a verificação se refere ao conjunto de atividades que garantem que o *software* implementa uma função específica, a validação se refere ao conjunto de atividades diferentes que garantem que o *software* construído corresponde aos requisitos do cliente (Pressman, 2006). Além disso, atividades de VV&T devem ser conduzidas durante todo o processo de desenvolvimento do *software*, não ficando restritas apenas ao produto final.

As atividades de VV&T são classificadas como estáticas e dinâmicas (Delamaro *et al.*, 2007). As estáticas caracterizam-se por não requerem a execução ou mesmo a existência de um programa ou modelo executável, já as dinâmicas se baseiam na execução de um programa ou modelo executável.

Teste de *software*, elemento central deste trabalho, se enquadra como uma atividade dinâmica de VV&T. Seu objetivo é executar o programa ou modelo utilizando entradas em particular e verificar se seu comportamento está de acordo com o esperado. Um erro ou defeito ocorre caso a execução apresente resultados não especificados. Os dados de tal execução podem auxiliar na identificação e correção de tais defeitos (Delamaro *et al.*, 2007).

A área de testes de *software* será descrita nas subseções seguintes, contemplando inicialmente conceitos e definições de teste de *software*, sendo seguida por fases da atividade de teste, etapas para a execução da atividade de testes, tipos de testes, finalizando com técnicas de testes.

2.2.1 Conceitos e Definições de Teste de *Software*

Teste de *software* é uma fase importante no desenvolvimento de sistemas, pois seu principal objetivo consiste em descobrir erros. Por este motivo, esta atividade torna-se indispensável para garantir a qualidade de um *software*. Embora a qualidade do *software* vá além dos testes realizados, ele é um processo importante que ajuda a garantir que o produto final atenderá aos requisitos existentes durante a sua concepção.

Para Hetzel (1987), "teste é qualquer atividade que vise a avaliar uma característica ou recurso de um programa ou sistema. Teste é a medida da qualidade do *software*". Esta definição deixa clara a relação entre teste e qualidade. Pressman (2006) complementa com a seguinte

afirmação, “a atividade de teste de *software* é um elemento crítico para a garantia da qualidade e representa a última revisão de especificação, projeto e codificação”.

O IEEE *Glossary of Software Engineering Terminology* (IEEE, 1990) define teste formalmente como sendo “o processo de operar um sistema ou um componente sobre as condições especificadas, observando ou gravando os resultados, e fazendo uma evolução de alguns aspectos do sistema ou componente”. As “condições especificadas”, citadas pela definição, fazem parte dos casos de teste que auxiliam a execução dos testes.

Um caso de teste consiste em um conjunto de dados de testes, condições de execução e resultados esperados para um objetivo de teste específico (Krutchen, 2003; Pressman, 2006). Os casos de testes serão bem sucedidos se os mesmos elevam a probabilidade de detecção de erros, ou seja, um teste bem sucedido é aquele que aponta falhas ainda não constatadas com o mínimo de tempo e esforço (Pressman, 2006).

Como se pode observar muitas são as definições de um teste, mas todas elas possuem em comum o teste como elemento fundamental para obtenção de *software* de boa qualidade.

2.2.2 Fases da Atividade de Teste

O teste de *software* pode ser realizado em diferentes fases com objetivos distintos durante todo o processo de desenvolvimento do *software*. A Figura 2.1 ilustra a relação entre estágio de teste e nível arquitetural do sistema. Cada fase é descrita a seguir.

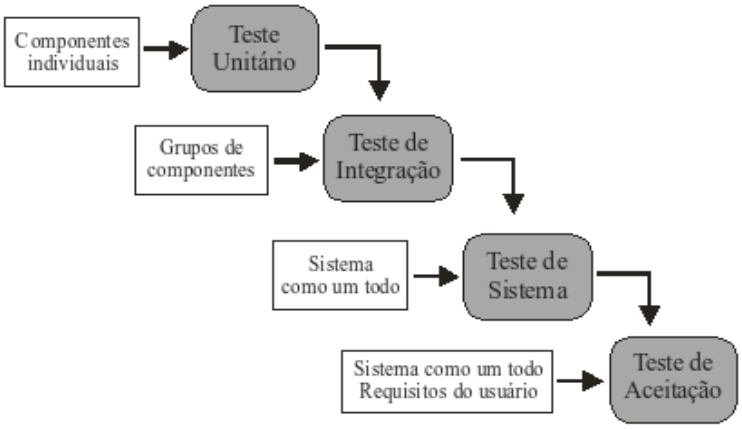

Figura 2.1. Relação entre Estágio de Teste e Nível Arquitetural do Sistema (Pressman, 2006).

2.2.2.1 Teste Unitário

Esta fase, também conhecida como teste de unidade, visa testar a menor unidade do projeto de *software* individualmente, por exemplo: função ou método no paradigma procedural, classe ou método no paradigma orientado a objetos. Com isso, espera-se que sejam identificados erros relacionados a algoritmos incorretos ou mal implementados, estruturas de dados incorretas,

ou simples erros de programação. O próprio desenvolvedor é responsável por realizar o teste unitário antes de passar o *software* para a equipe de testes.

2.2.2.2 Teste de Integração

O teste de integração tem como objetivo testar as unidades ou componentes, previamente testados em separado, de forma integrada. Com isso, é possível descobrir erros de interface entre os componentes do sistema. O teste de integração tende a ser executado pela própria equipe de desenvolvimento, pois ele exige um grande conhecimento das estruturas internas e das interações existentes entre as partes do sistema.

A integração pode ser incremental ou não-incremental. Enquanto na integração não-incremental o sistema é integrado por completo, na integração incremental o sistema é integrado em etapas, facilitando assim o isolamento do erro.

A integração incremental possui duas abordagens: *Top-Down* e *Bottom-Up*. Na abordagem *Top-Down* os módulos de alto nível são testados e integrados primeiro, permitindo encontrar primeiro os erros de lógica e fluxo de dado de alto nível. Por outro lado, a abordagem *Bottom-Up* requer o teste e integração dos módulos de baixo nível primeiro.

2.2.2.3 Teste de Sistema

A essência do teste de sistema é verificar, sob o ponto de vista do usuário final, se o sistema se comporta da maneira requisitada pelo cliente, ou seja, se ele contém as funcionalidades requeridas e as executa corretamente (Pfleeger, 2004). Aspectos de correção, completude e coerência devem ser explorados, bem como requisitos não funcionais como segurança, desempenho e robustez.

O teste de sistema é executado geralmente por uma equipe de testes. Para assegurar que os requisitos propostos serão realizados, os testes são executados em condições similares de ambiente, interfaces sistêmicas e massas de dados, àquelas que um usuário utilizará no seu dia-a-dia de manipulação do sistema.

2.2.2.4 Teste de Aceitação

O teste de aceitação é conduzido, geralmente, por um grupo restrito de usuários finais do sistema. Estes usuários verificam a conformidade do sistema de acordo com o especificado através da realização de operações de rotina do sistema. O teste de aceitação divide-se em dois tipos: teste alfa e teste beta. O teste alfa é feito pelos usuários nas instalações do desenvolvedor. O desenvolvedor, por meio de observações, registra os erros e problemas. Já teste beta é realizado

sem a supervisão do desenvolvedor nas instalações dos usuários, que relata os problemas detectados ao desenvolvedor.

Além das quatro fases testes, vale destacar o que se costuma chamar de teste de regressão, que é realizando durante a manutenção do *software*, considerando se que a cada modificação realizada no sistema, após sua liberação, corre-se o risco de que novos defeitos sejam introduzidos. Devido a isso, após a manutenção é necessário realizar testes que mostrem que as modificações efetuadas estão corretas, ou seja, garantir que tanto os novos requisitos implementados (se aplicável) funcionam como o esperado quanto que os requisitos anteriormente testados continuam válidos (Delamaro *et al*., 2007).

2.2.3 Etapas para Execução da Atividade de Testes

Independente da fase de teste, existem algumas etapas bem definidas para a execução da atividade de teste. São elas: planejamento, projeto, execução e avaliação dos resultados de teste (Beizer, 1990; Myers, 2004, Pressman, 2006; Delamaro *et al*., 2007).

2.2.3.1 Planejamento

No planejamento são levantadas informações como: requisitos a serem testados; regras e responsabilidades; estratégias, ferramentas e técnicas a serem utilizadas; os recursos necessários; e um cronograma para as atividades de teste. Estas informações são registradas em um plano de testes.

O planejamento precisa ser iniciado o quanto antes possível, pois o mesmo não é estático e deve ser evoluído à medida que o produto e o projeto se desenvolvem (Bandeira, 2008).

2.2.3.2 Projeto

Tem como objetivo a documentação e especificação dos casos de testes, constando os passos a serem seguidos e os resultados esperados. É baseado no planejamento previamente realizado para reunir informações como: objetivos a serem alcançados e técnica a ser utilizada.

Para Pressman (2006), projetar testes para *software* ou para outros produtos de engenharia pode ser tão desafiador quanto o projeto inicial do produto de *software*. Apesar disso, engenheiros de *software* muitas vezes tratam a atividade de teste sem pensar nela previamente; desenvolvem casos de teste que acham corretos, mas com pouca garantia de estarem completos. Os testes devem ser projetados de forma que se tenha uma alta probabilidade de achar a maior quantidade de erros, com uma quantidade mínima de tempo e esforço. Outro ponto relevante é que, as mudanças nos requisitos devem ser refletidas nos casos de testes impactados por elas (Bandeira, 2008).

2.2.3.3 Execução

Nesta etapa, o teste deve ser executado de acordo com procedimentos documentados usando uma versão claramente definida do *software* sob o teste. Os passos seguidos durante a execução do teste devem estar claramente documentados com intuito de que outra pessoa possa reproduzir os resultados (SWEBOK, 2001a).

2.2.3.4 Avaliação dos Resultados

A etapa de avaliação dos resultados se baseia nas informações provenientes da etapa de execução dos testes. Os resultados da execução de cada teste determinam se ele executou de forma esperada ou não. Os erros encontrados durante a execução dos testes precisam ser analisados e corrigidos. Bandeira (2008) apresenta um processo de acompanhamento de teste, que poder ser visualizado na Figura 2.2.

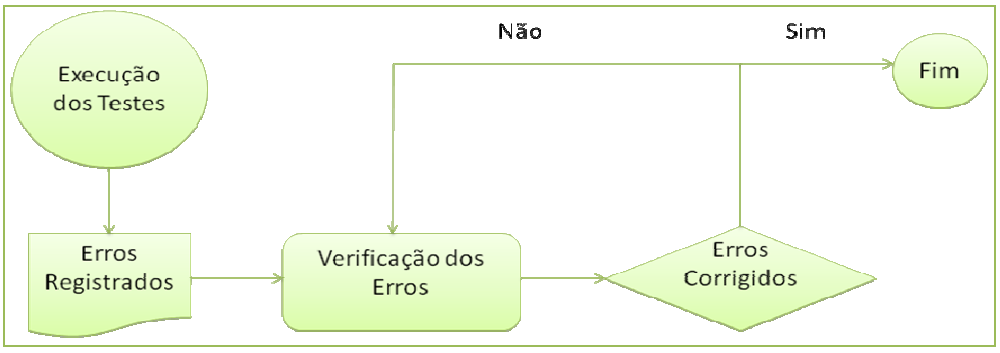

Figura 2.2. Processo de Acompanhamento de Teste. Adaptado de (Bandeira, 2008).

Neste processo, os erros originados da etapa de execução dos testes são documentados e, logo após, verificados. Após a correção, os erros são verificados com o objetivo de analisar se a correção foi devidamente efetuada. Caso o erro tenha sido corrigido ele é considerado resolvido, caso contrário, ele retorna para ser efetivamente corrigido. O acompanhamento do resultado dos testes, além de avaliar a qualidade do produto, avalia a qualidade dos casos de testes projetados.

2.2.4 Tipos de Teste

Os tipos de teste referem-se às características do *software* que podem ser testadas. Eles são divididos de forma que cada um deles foca a detecção de um problema. Assim, a abordagem utilizada para isso varia de acordo com o propósito. Dentre os tipos de teste (Peters & Pedrycz, 2001):

- **Teste de Funcionalidade**: testa as funcionalidades básicas do programa, testa tanto com condições válidas quanto inválidas para verificar a consistência do programa;

- **Teste de Estresse:** verifica as funcionalidades do programa em situações limite, exige recursos em quantidade, freqüência e volume diferentes do normal;
- **Teste de Volume:** está mais voltado para transações de banco de dados, testa o sistema com altos volumes de dados numa transação, verifica quantidade de terminais, se são suficientes, dentre outros testes que são realizados;
- **Teste de Segurança:** verifica se todos os mecanismos de proteção de acesso estão funcionando corretamente. É muito importante hoje em dia diante do grande uso de programas *web*;
- **Teste de Desempenho:** exige configuração de *hardware* e *software* para testar diferentes configurações, número de usuários, tamanho do banco de dados, e pode verificar o tempo de resposta e processamento do sistema.

2.2.5 Técnicas de Teste

Idealmente um programa deveria ser testado com todas as possíveis entrada, o que é conhecido como teste exaustivo (Myers, 2004). Isso garantiria, em teoria, detectar todos os defeitos do programa. Entretanto, mesmo para programa simples, conforme já foi demonstrado no capítulo anterior, isso não é possível, porque o domínio de entrada por ser muito grande ou até infinito, fazendo com essa possibilidade se torne inviável.

Isso fica claro ao observar o exemplo fornecido por (Peters & Pedrycz, 2001), onde ele descreve o esforço para testar todas as possibilidades de uma aplicação simples. O programa em questão calcula a equação do segundo grau $ax^2 + bx + c = 0$, onde x é a incógnita. A equação possui os parâmetros a, b, c. O tamanho do domínio de entrada muitas vezes depende da representação interna dos parâmetros. Neste exemplo foi adotada a representação numérica de 16 *bits*. Assim, cada entrada produz 2^{16} valores diferentes, cada um implicando 2^{16} casos de teste. Isso resulta no total de $2^{16} * 2^{16} * 2^{16} = 2^{48}$ casos de teste que precisam ser exercitados, o que é praticamente inviável. Tal limitação da atividade de teste contribuiu para a definição das técnicas de testes e os diversos critérios pertencentes a cada uma delas, permitindo que a atividade de teste seja conduzida de forma mais sistemática.

As técnicas de testes servem como base para gerar casos de testes. As técnicas de testes mais conhecidas são a técnica estrutural (ou caixa branca) e a técnica funcional (ou caixa preta). A diferença entre essas técnicas está na fonte de informação usada para avaliar ou construir os conjuntos de casos de teste. Além disso, cada critério de teste procura explorar determinados tipos de defeitos, estabelecendo requisitos de teste para quais valores específicos do domínio de entrada do programa devem ser definidos com o intuído de exercitá-los (Delamaro *et al*., 2007). Nenhuma dessas técnicas é completa o suficiente para garantir a qualidade dos testes. Pressman

(2006) defende que, para obter testes de melhor qualidade, essas diferentes técnicas devem ser usadas em conjunto, pois segundo o autor as mesmas se complementam. A seguir detalhamos cada uma delas.

2.2.5.1 Técnica Funcional

O teste de caixa preta, ou teste funcional, utiliza a especificação funcional do programa, ou seja, os requisitos do sistema, como fonte para geração dos casos de teste, sem se importar com a estrutura interna ou forma de implementação do sistema. O sistema é visto como uma caixa onde apenas a interface, ou seja, lado externo está visível e disponível (Beizer, 1990).

O teste funcional tenta encontrar erros das seguintes categorias (Pressman, 2006):

1. Funções incorretas ou omitidas;
2. Erros de interface;
3. Erros de estrutura de dados ou de acesso a base de dados externa;
4. Erros de comportamento ou desempenho;
5. Erros de iniciação e término.

Dessa forma, nesta técnica de teste, as entradas são fornecidas, e suas saídas são verificadas de acordo com o comportamento especificado. Como o teste funcional ignora a estrutura interna, a atenção volta-se para o domínio da informação. Os testes são projetados para responder às seguintes questões (Pressman, 2006):

- Como a validade funcional é testada?
- Como o comportamento e o desempenho do sistema são testados?
- Que classes de entrada vão construir bons casos de teste?
- O sistema é particularmente sensível a certos valores de entrada?
- Como são isolados os limites de uma classe de dados?
- Que taxas e volumes de dados o sistema pode tolerar?
- Que efeito as combinações específicas de dados vão ter na operação do sistema?

Os critérios mais conhecidos da técnica de teste funcional são: particionamento de equivalência e análise do valor limite (Pressman, 2006). Delamaro *et al*. (2007) ressalta que a qualidade de tais critérios depende da existência de uma boa especificação de requisitos, já que os critérios baseados nesta técnica baseiam-se apenas na especificação do produto testado. Especificações ausentes ou incompletas tornarão difícil a aplicação dos critérios funcionais. Ainda segundo o autor, os critérios funcionais podem ser aplicados em todas as fases de testes e em produtos desenvolvidos com qualquer paradigma de programação, já que não levam em consideração os detalhes de implementação. Esses critérios são descritos a seguir:

- **Particionamento de Equivalência:** este critério (Figura 2.3) divide o domínio de entrada do sistema em classes de equivalência válidas e inválidas. Ele supõe que se um elemento de uma classe provocar a falha de um teste, todos os demais membros da classe também provocarão falhas. O contrário também é válido, se um elemento de entrada resulta num teste bem sucedido, então os outros elementos da mesma classe também o serão. O uso deste critério permite restringir o número de casos de teste necessários.

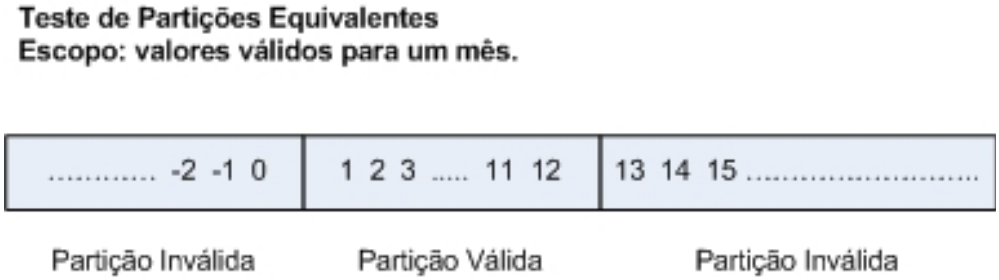

Figura 2.3. Exemplo de Teste de Particionamento de Equivalência (Leal, 2008).

As classes de equivalência são determinadas de acordo com as seguintes diretrizes (Delamaro *et al.*, 2007):

1. Se as condições de entrada especificam um intervalo de valores, definir uma classe com entradas válidas e duas classes com entradas inválidas.
2. Se as condições de entrada especificam um valor específico (ou um determinado número de entradas), definir uma classe válida e duas inválidas.
3. Se uma condição de entradas especifica um conjunto de valores, duas situações são possíveis:
    a. Se cada elemento do conjunto é tratado de maneira diferente pelo *software*, definir uma classe válida para cada elemento, e uma classe inválida.
    b. Se cada elemento do conjunto é tratado da mesma maneira pelo *software*, definir uma classe válida e uma inválida.
4. Se uma condição de entrada especifica um valor lógico, definir uma classe válida e uma inválida.

A Tabela 2.1 ilustra exemplos de acordo com as diretrizes acima (mostradas em parênteses):

Tabela 2.1. Condições de Entrada para Classes de Equivalência – Exemplo 1 (Corso, 2008).

Condições de Entrada	Classes Válidas	Classes Inválidas
"O valor do desconto não pode ser inferior a R$ 1,00 e nem superior a R$ 50,00" (1)	C1: valor do desconto deve ser entre [1,50]	C2: valor do desconto < 1 C3: valor do desconto > 50
"O programa deve ler 3 valores..." (2)	C1: 3 valores são fornecidos	C2: fornecer menos de 3 valores C3: fornecer mais de 3 valores

"Para efeito de cálculo do salário um empregado pode ser horista, diarista ou mensalista" (3-a)	C1: empregado = horista C2: empregado = diarista C3: empregado = mensalista	C4: empregado não pertence a {horista, diarista, mensalista}
"Um identificador deve iniciar por uma letra" (3-b)	C1: 1º letra do identificador [A,Z]	C2: 1º letra do identificador não pertence a [A,Z]
"A senha pode ser ou não fornecida" (4)	C1: senha é fornecida	C2: senha não é fornecida

- **Análise do Valor Limite:** o critério de análise do valor limite (Figura 2.4) complementa o particionamento de equivalência. Ele é usado, pois uma grande quantidade de erros geralmente ocorre nas fronteiras do domínio de entrada (Pressman, 2006). Com isso, a técnica de análise de valor limite, ao invés de selecionar um elemento qualquer de uma classe de equivalência, seleciona elementos que atuam nas extremidades da classe (Myers, 2004). Myers (2004) ainda sugere as seguintes diretrizes para este critério:
    1. Se a condição de entrada especifica um intervalo de valores, devem ser definidos dados de teste para os limites desse intervalo e dados de teste imediatamente subseqüentes, que explorem as classes inválidas vizinhas desse intervalo. Por exemplo, se uma classe válida estiver no intervalo -1,0 e +1,0, devem ser definidos os seguintes dados de teste: -1,0; +1,0; -1, 001 e +1, 001;
    2. Se a condição de entrada especifica uma quantidade de valores, por exemplo, de 1 a 255 valores, devem ser definidos dados de teste com nenhum valor de entrada, somente um valor, 255 valores e 256 valores de entrada;
    3. Usar a diretriz 1 para as condições de saída;
    4. Usar a diretriz 2 para as condições de saída;
    5. Se a entrada ou saída for um conjunto ordenado, deve ser dada maior atenção ao primeiro e último elementos desse conjunto;
    6. Usar a intuição para definir outras condições limites.

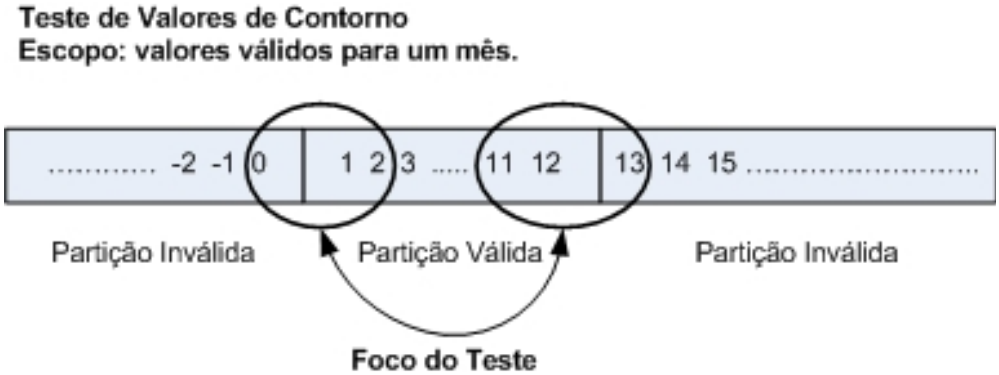

Figura 2.4. Exemplo de Teste de Análise do Valor Limite (Leal, 2008).

2.2.5.2 Técnica Estrutural

O teste de caixa branca, também conhecido por teste estrutural, deriva os testes através do conhecimento da estrutura interna e da implementação do *software*. Dessa forma, os dados de testes são obtidos por meio da lógica do programa, sem se preocupar com os requisitos do sistema.

Através do teste de caixa branca, por exemplo, pode-se: garantir que todos os caminhos dentro de um módulo sejam exercitados pelo menos uma vez, exercitar todas as decisões lógicas para valores falsos ou verdadeiros, executar todos os laços em suas fronteiras e dentro de seus limites operacionais e exercitar as estruturas de dados internas para garantir a sua validade (Pressman, 2006).

Uma vantagem do teste de caixa branca é que, através do exame do código pode-se identificar erros que não seriam descobertos pelo teste de caixa preta. Entretanto, a complexidade do pode inviabilizar a execução de todos os caminhos lógicos possíveis pelo programa, devido ao alto esforço e a elevada quantidade de casos de teste (Lewis, 2000).

2.3 OCL

Embora a UML ofereça um grande número de diagramas que possibilitam a construção de visões estáticas e dinâmicas de um sistema, eles não são suficientes para descrever todas as restrições que compõem um *software*. Algumas regras de negócio e definições contratuais de operações são exemplos de informações que não são cobertas por esses diagramas, e que demandam uma especificação mais precisa (Warmer & Kleppe, 2003).

Freqüentemente, as regras de negócio são descritas em linguagem natural. Entretanto, especificações produzidas em linguagem natural estão intrinsecamente ligadas a problemas de ambigüidade (Berry & Kamsties, 2004). O emprego de uma linguagem mais formal na especificação dessas regras é uma alternativa natural para lidar com a questão da ambigüidade, que abre várias possibilidades de apoio automatizado ao longo do processo de desenvolvimento (Pfleeger, 2004).

Aproximadamente em 1995, a linguagem OCL começou a ser desenvolvida por Jos Warmer e Steve Cook, na IBM, como uma linguagem de modelagem de negócio. Em 1997 ela foi formalmente definida como parte do padrão UML pelo OMG. O seu objetivo foi o de acrescentar à definição de UML a possibilidade de especificar restrições aos modelos UML (Warmer & Kleppe, 1999).

A OCL é uma linguagem de expressões textuais e precisas, cujo estilo da notação é similar às linguagens orientadas a objetos mais comuns, que visa complementar a parte gráfica dos modelos, para descrever restrições que não conseguem ser diagramaticamente representadas.

As *restrições* são definições de limites em um ou mais valores, de parte ou do todo de um objeto, em um modelo orientado a objeto ou de um sistema. Na prática são detalhes e informações extras, que não conseguem ser expressos pelos modelos gráficos. A utilização da idéia de restrições acrescenta algumas vantagens, dentre elas (Alencar, 1999):

- Melhor documentação, por acrescentar aos modelos, informações sobre os elementos e seus relacionamentos. Um modelo gráfico pode conter algumas restrições, como é o caso da multiplicidade nos relacionamentos; no entanto, alguns detalhes não são possíveis de serem representados;
- Precisão aumentada, dado que as restrições não podem ser interpretadas de forma diferente por várias pessoas, são não ambíguas, tornando o modelo ou sistema, sobre o qual se aplicam, mais preciso; e
- Comunicação sem enganos: em função da redução da ambigüidade na descrição das informações, os desenvolvedores passam a ser capazes de comunicar seus intentos de forma mais precisa, reduzindo desde cedo, no processo de desenvolvimento, possíveis defeitos, evitando desperdício de dinheiro e evitando-se frustrações.

As expressões OCL são declarativas e sem efeitos colaterais. Declarativa significa que descreve o que um sistema deveria fazer, e não como deveria ser feito, separando efetivamente a especificação da implementação. Sem efeitos colaterais significa que elas não alteram o estado do sistema (Warmer & Kleppe, 2003).

Nesta dissertação, a OCL foi escolhida como forma de representação das regras de negócio. Embora seja classificada como uma linguagem formal baseada em modelos, ela possui uma sintaxe simples, não simbólica, que se utiliza de símbolos matemáticos mais simples da teoria de conjuntos e da lógica, numa proposta de ser precisa, porém de fácil compreensão, tanto no tocante da escrita quanto da leitura. Dessa forma, ela pode ser usada mais facilmente por quem não possui um profundo conhecimento matemático, diferentemente de outras linguagens formais matemáticas como Z (Spivey, 1992) e VDM (Jones, 1990).

Nesta seção adotamos o seguinte DC (Figura 2.5) de uma locadora de veículos, extraído do trabalho de Silveira (2009), como base para alguns exemplos a serem apresentados.

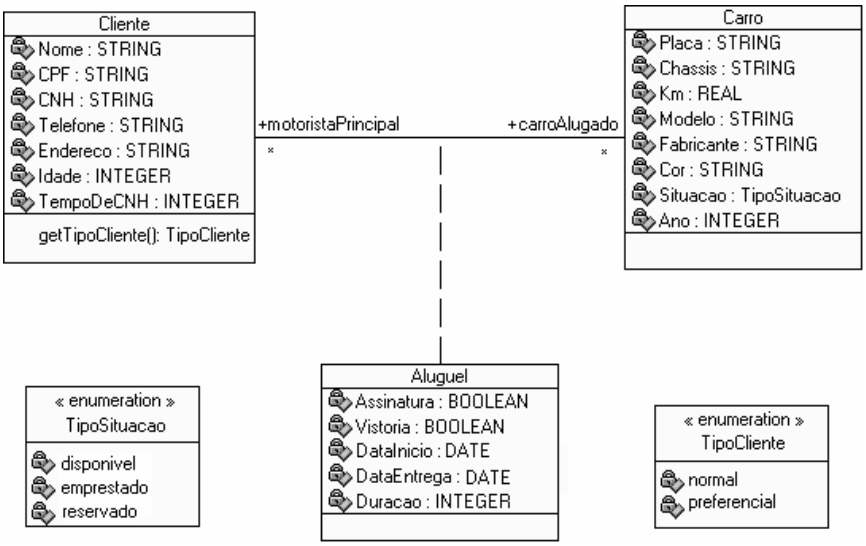

Figura 2.5. Trecho de um DC de um Sistema de Locadora de Veículos.

O modelo conceitual de informação é a base de todas as restrições e expressões em OCL. Segundo Larman (2004), o modelo conceitual de informação descreve as entidades ou classes, suas características e seus relacionamentos. Schmitz & Silveira (2000) definem entidade como sendo uma categoria atribuída ao conjunto de objetos existentes em um ambiente, agrupados em função de suas semelhanças. Os objetos representados passam a ser denominados instâncias destas entidades e definem os termos de negócio que serão manipulados pelas expressões OCL.

Dentre os diferentes tipos de restrições que podem ser especificadas em OCL, destacam-se (Silveira, 2009): Invariantes (`inv`), Derivação de Atributos e Associações (`derive`), Corpo de operação de consulta (`body`), Valor inicial de atributos (`init`), Pré e Pós-condições (`pre` e `post`). Como o método proposto gera casos de teste a partir das invariantes, pré e pós-condições, estas serão detalhas a seguir:

- Invariantes (`inv`): são restrições que estabelecem uma condição que sempre deve ser verdadeira, por todas as instâncias da classe, tipo ou interface. São descritas usando-se uma expressão que será avaliada como verdadeira, se o invariante é satisfeito durante todo o tempo. Uma restrição do tipo invariante é definida no contexto de um classificador por uma expressão através da palavra reservada `inv`. A restrição da Figura 2.6 se aplica para todas as instâncias de `Cliente`.

```
001. context Cliente
002.  inv idadeMinima: idade >= 21
```

Figura 2.6. Exemplo de uma Restrição do Tipo Invariante.

Neste exemplo, a invariante denominada `idadeMinima` foi definida. Esta invariante restringe a idade de todas as instâncias de `Cliente` a um valor maior ou igual a 21.

- Pré e pós-condições (`pre` e `post`): são restrições utilizadas para definir, de forma declarativa, a semântica das operações de um modelo. Cada operação do modelo pode estar associada a pré-condição e a pós-condição. A Figura 2.7 ilustra um exemplo contendo pré e pós-condição.

```
001. context Aluguel::Alugar(umData : Date)
002.  pre duracaoMinima: Date.difference(umData, Date.curDate) > 0
003.  pre duracaoMaxima: Date.difference(umData, Date.curDate) <= 20
004.  post contratoAssinado:(Assinatura@pre=false) and (Assinatura=true)
```

Figura 2.7. Exemplo de Especificação de Pré e Pós-condição.

Neste exemplo, duas pré-condições foram definidas para a operação `Alugar` da classe `Aluguel`. Um nome pode ser associado a cada pré ou pós-condição (por exemplo, `duracaoMinima`, `duracaoMaxima` e `contratoAssinado`).

2.3.1 Tipos

A OCL é uma linguagem tipada (OMG, 2003). Expressões OCL podem resultar em um valor primitivo, um objeto, uma tupla, ou uma coleção desses elementos. Os tipos podem ser classificados como (Alencar, 1999):

i. Pré-definidos, onde se enquadram:
   a. os tipos ditos básicos – *Integer* são os números inteiros tal qual na matemática; *Real* são os números reais tal qual na matemática; *String* são as sequências de caracteres, por exemplo 'OCL'; e *Boolean* com apenas os valores '*true*' e '*false*' -, que independem do modelo UML. Segundo (Richters & Gogolla, 1998) o tipo enumerado é uma extensão dos tipos básicos através de uma enumeração de valores literais, tal como, $enum\{valore_1, ..., valor_n\}$; e
   b. os tipos coleção, cujos construtores são: *Set*, *Bag*, *Sequence* e *OrderedSet*. Um *Set* é uma coleção não ordenada que contem instâncias de um tipo OCL válido, não contendo elementos duplicados. Um *Bag* é uma coleção que contém elementos duplicados. Uma *Sequence* é uma coleção de elementos duplicados, mas ordenados. Enquanto que um *OrderedSet* é uma coleção de elementos não duplicados, mas ordenados.

ii. Tipos do modelo – todos os elementos presentes num modelo UML definem tipos em OCL. Esses tipos podem ser classes, interfaces, associações de classes, atores, casos de uso e tipos de dados definidos no modelo UML.

Além dos tipos, há um conjunto de operações definidos na OCL que permite a manipulação de valores dos tipos primitivos e também das coleções como, por exemplo, operações lógicas (`and`, `or`, `xor`, `implies`, `if-then-else-endif`), aritiméticas (`+`, `-`, `*`, `/`), manipulação de strings (`size`, `concat`, `substring`), manipulação de coleções (`size`, `includes`, `isEmpty`). Na especificação da linguagem (OMG, 2003) é possível encontrar a relação completa dos tipos e das suas respectivas operações.

2.3.2 Navegação

O núcleo central de OCL está na navegação, ou seja, no relacionamento entre os objetos. As expressões OCL permitem escrever restrições no comportamento dos objetos identificados, navegando-se entre estes objetos. Um objeto ou uma coleção de objetos podem representar um elemento inicial de uma navegação. Com isso, o resultado de cada navegação consiste em uma coleção contendo os objetos associados ao elemento inicial. O classificador destino da navegação e a cardinalidade definem o tipo da coleção resultante. A navegação por uma associação é definida por uma expressão com a estrutura `<origem>.<papel>`, onde origem corresponde a um objeto ou a uma coleção de objetos de uma classe `A`, e papel corresponde ao nome do papel (`rolename`) de uma classe associada à classe `A` (Silveira, 2009). Para acessar um atributo ou operação, navega-se pelas associações utilizando o operador "."; para acessar propriedades de uma coleção (que pode ser definida pelas instâncias de uma classe, ou pelas instâncias ligadas a uma dada instância de uma classe), utiliza-se o operador "→". Além disso, o operador → sempre precede as operações de manipulação.

Por exemplo, seja o diagrama de classe de uma aplicação fictícia representado na Figura 2.8, extraído do trabalho de Kleppe & Warmer (2003).

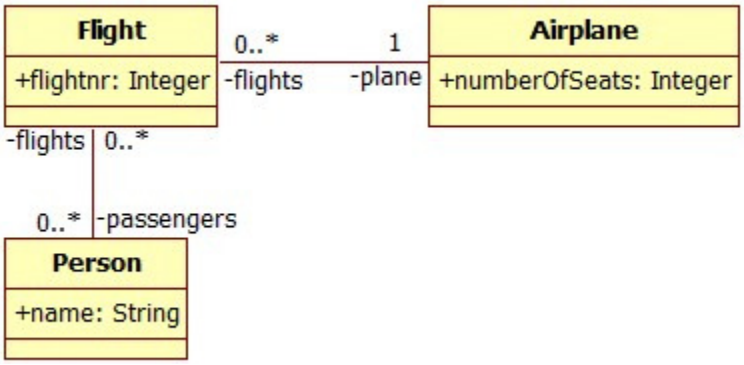

Figura 2.8. Diagrama-exemplo do Uso de OCL: Máximo de Passageiros por Vôo.

A Figura 2.9 ilustra uma expressão OCL que demonstra o uso dos operadores "." e "→". Esta expressão faz com que o número máximo de passageiros (papel `passengers`) em um vôo

(classe `Flight`) seja limitado pela capacidade do avião respectivo (classe `Airplane`, atributo `numberOfSeats`).

```
001. context Flight
002.        inv passageirosPorVoo: self.passengers→size() <=
             self.plane.numberOfSeats
```

Figura 2.9. Expressão OCL que Restringe o Número Máximo de Passageiros por Vôo.

O contexto de onde parte a navegação é fornecido (classe `Flight`); a restrição é um invariante (palavra-chave `inv`) nomeado (`passageirosPorVoo`). Para a avaliação do lado esquerdo da inequação, navega-se do contexto (`self` - a palavra reservada `self` é usada para referenciar uma instância do contexto) até o alvo da associação (papel `passengers`); como sua multiplicidade é '*', esse sentido da associação aponta para uma coleção de elementos (neste caso, um conjunto); para acessar uma das propriedades dessa coleção utiliza-se o operador "→". A operação executada (`size()`) retorna o número de elementos dentro da coleção, na forma de um inteiro (tipo OCL `Integer`). Para a avaliação do lado direito, navega-se do contexto (`self`) em direção ao alvo da outra associação (papel `plane`); como sua multiplicidade é '1', esse sentido da associação aponta para um único elemento (operador "."), cujo atributo pode ser recuperado (`numberOfSeats`) retornando-se um inteiro. Por fim, efetua-se a comparação entre os inteiros obtidos em ambos os lados da inequação. A OCL não especifica uma ordem para avaliação; apenas para fins de explicação do exemplo, avaliou-se o lado esquerdo primeiro.

Vale destacar que a avaliação de algumas expressões pode resultar em um valor indefinido. Isso ocorre principalmente em caso excepcionais tais como divisão de um número por zero, o resultado da operação `first` aplicada a uma coleção sem elementos, dentre outros. Além disso, como um atributo de uma classe pode ser definido com multiplicidade [0..1], indicando que nem toda instância dessa classe precisa ter um valor definido para esse atributo, expressões que envolvam um atributo com essa multiplicidade também podem resultar em um valor indefinido (Silveira, 2009).

2.4 O Método *ANIMARE*

O *ANIMARE* é um método para validação dos processos de negócio de um sistema, tendo como foco as regras de negócio. O método utiliza o Diagrama de Atividades (DA) da UML para a modelagem dos processos de negócio, expressando em OCL as regras de negócio. A partir destes dados, é possível gerar um grafo orientado (grafo de fluxo) que representa o processo com as suas regras de negócio. A validação é obtida a partir da execução de cenários. Um cenário representa o conjunto de instâncias selecionadas para teste, geradas a partir dos objetos definidos

no DC e manipulados pelo DA. Os cenários são utilizados na animação do processo. O acompanhamento da animação ao longo do grafo de fluxo permite a detecção de situações que violam a execução correta do processo. A ocorrência destas situações deve ser tratada tanto pela mudança do processo de negócio, como pela alteração e/ou introdução de novas regras (Silveira, 2009).

O método é dividido nas seguintes etapas: (a) especificar o modelo conceitual de informação na forma de um DC; (b) especificar o modelo do processo de negócio na forma de um DA; (c) especificar as regras de negócio comportamentais na forma de expressões OCL; (d) definir o cenário utilizado na animação; e (e) executar o algoritmo de animação. A Figura 2.10 ilustra as etapas do método *ANIMARE*, conforme descritas a seguir.

a) Especificar o Modelo Conceitual de Informação (MCI): definir os termos do negócio e as restrições impostas pelo negócio (Regras de Negócio Estruturais) na forma de um DC/UML.
b) Especificar o modelo do processo de negócio: definir as atividades, insumos e produtos do processo na forma de um DA/UML. As operações envolvendo objetos do negócio, executadas tanto pelas atividades como pelas decisões devem ser formalmente especificados usando *Action Semantics* (AS) da UML.
c) Especificar as Regras de Negócio Comportamentais: anotar nos elementos gráficos do DA/UML as regras de negócio especificadas na forma de expressões OCL (RN/OCL).
d) Definir o cenário a ser utilizado na animação do DA/UML: anotar no elemento que representa a atividade inicial (`InitialNode`) os valores dos objetos utilizados no processo de animação.
e) Executar o algoritmo de animação: o algoritmo de animação recebe como parâmetros para sua execução os modelos anotados com as regras de negócio e o cenário. A execução prossegue até que uma inconsistência seja encontrada ou a atividade final (`FinalNode`) seja encontrada.
f) Caso uma inconsistência seja encontrada, o projetista do processo e o especialista do negócio investigam a causa da inconsistência. O modelo do processo é redefinido e os modelos são alterados de forma a corrigir a inconsistência.

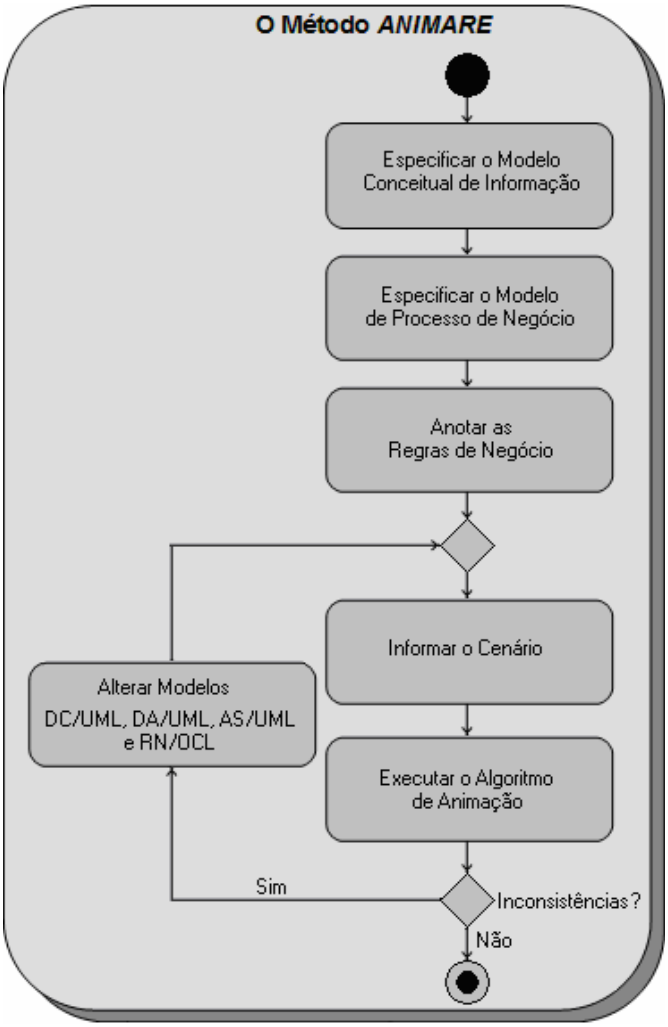

Figura 2.10. Um Esquema com as Etapas do Método *ANIMARE*.

O método *ANIMARE* ainda conta com uma ferramenta, de igual nome, que apóia as atividades de modelagem do DC/UML, DA/UML, RN/OCL e AS/UML, bem como a animação do DA/UML que possibilitará a verificação da conformidade dele com as RN/OCL. É através desta ferramenta que há a integração entre o *ANIMARE* e o método *TESTIMUNIUM*.

2.5 Considerações Finais

Este capítulo apresentou os conceitos relacionados a teste de *software*, linguagem OCL e método *ANIMARE.* Sobre o teste de *software* foram abordados conceitos básicos, etapas, fases, tipos e técnica de teste. Dentre as técnicas de teste destacou-se a técnica funcional (Seção 2.2.5) e seus critérios, pois são diretamente usados no método *TESTIMONIUM*. Abordou-se também a linguagem OCL, apresentando suas características, tipos e um pouco de sua sintaxe. O método *ANINAMRE* e suas etapas também foram apresentados.

Capítulo 3

O Método *TESTIMONIUM*

3.1 Considerações Iniciais

A principal proposta deste trabalho é dar suporte as atividades dinâmicas de testes funcionais de software, nas fases iniciais do ciclo de vida do seu desenvolvimento. Para isso é proposto o Método *TESTIMONIUM*, detalhado neste capítulo. Este método visa possibilitar a identificação de inconsistências em modelos, nas etapas iniciais do processo de desenvolvimento. Um diferencial deste método é que ele, além de suporte a testes gerando os casos de testes, também inclui algum suporte à fase de validação, uma vez que foi projetado para ser facilmente integrado ao método *ANIMARE* (onde os testes gerados podem ser validados e acompanhados de forma automatizada através da sua ferramenta). No entanto, o método aqui apresentado pode interoperar com outras ferramentas através das suas entradas e saídas expressas em linguagem XML[1] (*eXtensible Markup Language*). Dessa forma, o método é capaz de gerar casos de teste para qualquer modelo, por exemplo, modelo de caso de uso, desde que o mesmo seja expresso através de expressões OCL. Entretanto, nesta dissertação, para validação do método, foi adotado o *ANIMARE*, que utiliza o modelo DA anotados com expressões OCL para expressar processos de negócio.

Este capítulo está organizado da seguinte forma: a Seção 3.2 descreve o método proposto para geração de casos de teste; a Seção 3.3 ilustra a integração deste método com o método *ANIMARE*; por fim a Seção 3.4 apresenta brevemente algumas considerações finais.

[1] XML: É uma linguagem padronizada com formato simples, textual, estruturado, flexível a mudanças e portável em diversas plataformas tecnológicas (W3C, 2010).

3.2 O Método Proposto

O método *TESTIMONIUM* tem como objetivo a geração e execução de casos de teste para teste de modelos, chamados de cenários de teste, utilizando os critérios de particionamento de equivalência e análise do valor limite da técnica de teste funcional. Com este método de geração de casos de teste deseja-se identificar as falhas em fases iniciais de projeto, podendo implicar em uma redução de custos e tempo para a conclusão projeto.

A Figura 3.1 ilustra as quatro principais etapas do método *TESTIMONIUM*, suas entradas e saídas. A saber: a) Transformar DC e regras de negócio – elementos de entrada para o método – em uma estrutura interna; b) Identificar todos os parâmetros das expressões OCL; c) Gerar valores para cada parâmetro; d) Combinar todos os valores possíveis para a geração dos cenários de testes.

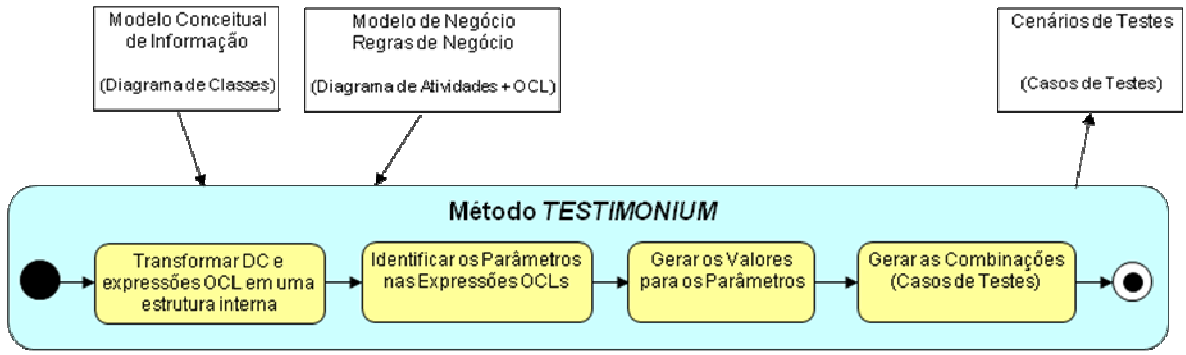

Figura 3.1. O Diagrama de Atividades do Método Proposto.

Nas próximas subseções serão detalhados todos os passos do método *TESTIMONIUM*.

3.2.1 Transformação do DC e das Expressões OCLs em uma Estrutura Interna

Os modelos fornecidos são transformados em uma estrutura interna (lista de objetos), que representa uma versão simplificada do metamodelo do DC/UML (OMG, 2005), a partir do qual serão aplicadas as regras de que darão origem aos cenários de teste, como ilustra a Figura 3.2. É interessante destacar que a simplificação do metamodelo foi baseada no trabalho de (Warmer & Kleppe, 2003) e tal mudança não apresenta influência na expressividade do metamodelo original. Em contrapartida, a alteração permitiu a remoção de elementos inutilizados, e, com isso, facilitando a compreensão do metamododelo.

Como mostra a Figura 3.2, na raiz do modelo, que foi implementado na ferramenta que dá suporte ao método *TESTIMONIUM*, encontra-se a classe `ModelElement`. Deste modo, todos os elementos de entrada, utilizados no método, são subtipos da classe `ModelElement`. No modelo encontra-se a classe `Classifier`, que representa uma classe abstrata para os tipos `Class` e

`DataType` que deverão ser tratados na leitura do DC. Além disso, todo classificador possui `Features`, seja `Attributes` ou `Operations`. E toda `Feature` tem um tipo, no caso dos atributos é simplesmente o tipo do atributo. Já para os métodos ela representa o tipo de retorno. As expressões OCL são representadas pela `OCLExpression`. As pré e pós-condições são instâncias de `OCLExpression` e são ligadas a uma instância de `Operation`. Este relacionamento representa a relação entre uma expressão OCL e seu contexto. É interessante ressaltar que o modelo que foi utilizado para implementar a ferramenta que dá suporte ao método é um subconjunto do metamodelo da UML, que pode ser encontrado em (OMG, 2005).

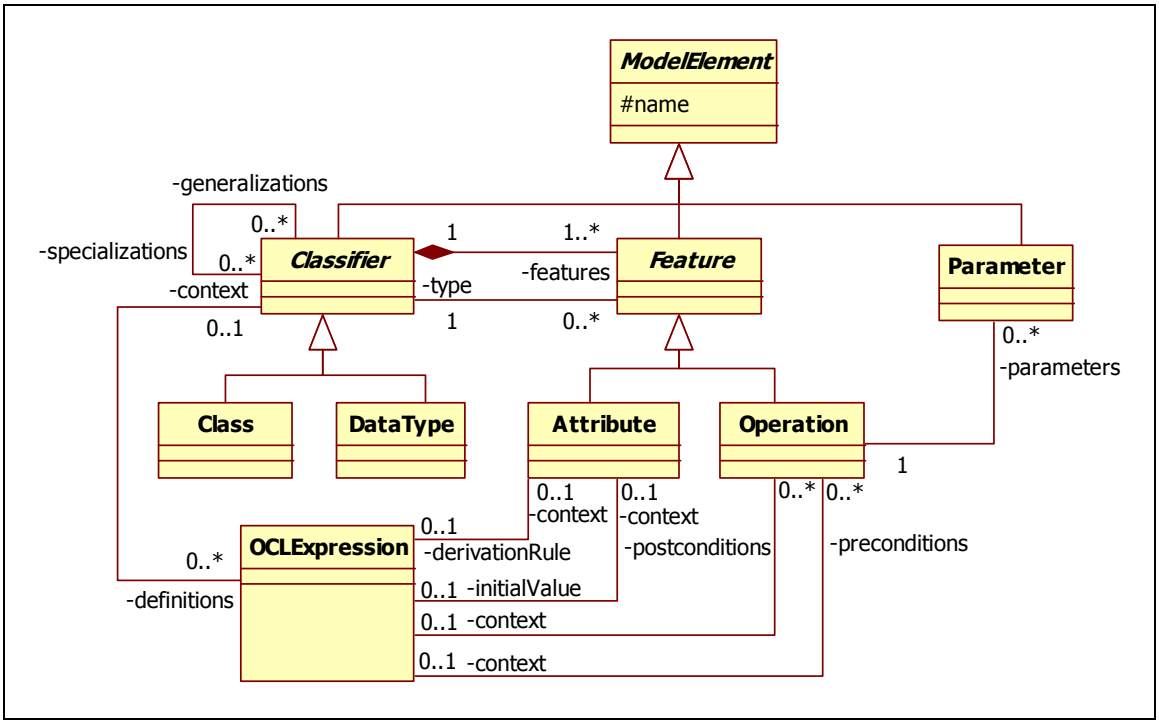

Figura 3.2. O Diagrama de Classe Implementado na Ferramenta *TESTIMONIUM*.

Para realizar a transformação, os modelos fornecidos para o método proposto, no caso o DC e as expressões OCLs, são analisados e à medida que os elementos vão sendo identificados eles são mapeados em suas classes correspondentes no diagrama apresentado na Figura 3.2. A Tabela 3.1 apresenta os elementos de entrada e suas classes correspondentes no modelo proposto.

Tabela 3.1. Elementos de Entrada e suas Classes Correspondentes no Modelo Proposto.

Elemento	Formato	Classe

Classe	`<element type="TObjClasse">`
	`  <name>`**`Carro`**`</name>`
	`</element>`
		Class

Atributo	`<attribute visibility="1" type="STRING">`
	`  <name>`**`placa`**`</name>`
	`</attribute>`
		Attribute

Tipo do	`<attribute visibility="1" type="`**`STRING`**`">`	*DataType*

Atributo	<name>placa</name> </attribute>	
Pré-condição	pre: **paramDataAluguel >= Data::now**	*OCLExpression*
Pós-condição	post: **grupo->select(g | g.nome = paramGrupo)->size() = 1**	*OCLExpression*

No final desta etapa, teremos todos os elementos transformados na sua respectiva classe do modelo, que será utilizado pela ferramenta *TESTIMONIUM* para a geração dos cenários de teste.

3.2.2 Identificação dos Parâmetros nas Expressões OCLs e seus Tipos

As expressões em OCL devem ser especificadas na forma de gabaritos (*templates*) definidos de acordo com a sintaxe da OCL. Um gabarito é um modelo parametrizado responsável por especificar os elementos participantes de um modelo (Miller & Strooper, 2001). Na estratégia utilizando gabaritos, um conjunto de marcas é associado a um gabarito para indicar como as instâncias de um modelo devem ser transformadas. A Figura 3.3 ilustra três gabaritos, onde é possível encontrar a marca **<***param***>** que foi usada para identificar os parâmetros de entrada nas expressões OCL.

O método proposto gera os valores para esses parâmetros de entrada. Por exemplo, na expressão OCL *pre: paramNome→size() > 0* é identificado o parâmetro de entrada *paramNome*. Após identificar o parâmetro, o nome do atributo é extraído desconsiderando a marca **<***param***>** do parâmetro. Neste caso, desconsiderando o *param* do parâmetro *paramNome*, temos o atributo *Nome*. Depois disso, o parâmetro *Nome* deve ser associado a algum atributo das classes do modelo conceitual de informação. Essa associação é necessária para identificar o tipo primitivo desse parâmetro *(Boolean, Integer, Real* e *String*).

```
001. pre: param<AttributeName1> = Data::now
002. pre: param<AttributeName1> > param<AttributeName2>
003. pre: param<AttributeName3>->size() > 0
```

Figura 3.3. Exemplos de Gabaritos.

3.2.3 Geração de Valores para os Parâmetros

Após a identificação dos tipos para os parâmetros de entrada das expressões OCLs, ocorre à geração dos valores para os mesmos de acordo com os tipos, considerando os seguintes critérios:

- Tipo *String*: Será gerado um valor inválido, ou seja, um valor nulo (*null*). Os dados válidos não serão gerados. Estes ficam a cargo do projetista, que deve fornecer os valores válidos para o tipo *String*. Como os cenários de testes são gerados pela combinação dos dados de todos os parâmetros (Seção 3.2.4), quanto mais valores ele fornecer, maior a quantidade de casos de testes que serão gerados. Na Seção 4.5 apresenta a forma como o projetista deve informar os valores válidos.
- Tipo numérico: Para este tipo, quando encontrado algum tipo de restrição nas expressões OCLs, foram adotados os critérios de particionamento de equivalência e análise de valor limite. A existência de uma restrição com um parâmetro é caracterizada pela presença de operadores relacionais (<, >, <=, >=, =) nas expressões OCLs em que o parâmetro esteja sendo usado. Neste caso, é preciso identificar o valor que o parâmetro esta sendo comparado para ser gerado nos cenários de testes os valores de acordo com os critérios utilizados. É interessante ressaltar que alguns valores são gerados de acordo com funcionalidades pré-definidas das expressões OCL, como por exemplo, a função *Data::now*, que equivale ao valor da data corrente. No caso de não existir nenhuma restrição serão gerados três valores aleatórios: um positivo, um negativo e um nulo (zero).
- Tipo *Date*: Assim como o tipo numérico, caso seja encontrado alguma restrição os critérios de particionamento de equivalência e o de análise do valor limite serão empregados. No caso de não ocorrer nenhuma restrição uma data válida e uma inválida serão geradas.
- Tipo *Booleano*: Será gerado um valor verdadeiro (*true*) e um falso (*false*).

Por fim, ocorre a geração dos cenários, que é realizada a partir da combinação com todos os valores gerados nesta ação anterior.

3.2.4 Geração de Cenários

Para a geração dos cenários foi implementado um algoritmo (Figura 3.4) que faz o produto cartesiano dos conjuntos dos valores gerados na etapa anterior. A noção de conjunto é um conceito primitivo da Matemática Moderna; isto é, um dos conceitos adotados como ponto de partida e que servem de base para a definição dos outros conceitos. Intuitivamente, um conjunto é encarado como uma coleção de valores de natureza qualquer, os quais se dizem elementos do conjunto. Representa-se simbolicamente por $x \in X$ a proposição "x é um elemento do conjunto X" que também se lê "x pertence a X". A negação desta proposição escreve-se $x \notin X$ (Lévy, 1979).

```
1. Combinação_Valores (M[n,m])
2. Entrada
3.    M[n,m] : matriz de parâmetros por valores;
4. Variáveis Locais
5.    numComb, ind, i, j, k : inteiro;
6.    result[n,m] : matriz de parâmetros por valores;
7. Início
8.   numComb ← 1;
9.   Para i de 1 até M.count Faça
10.    numComb ← numComb * M[i].count;
11.  Próximo i
12.  Para i de 1 até numComb Faça
13.    j ← 1;
14.    Para k de 1 até M.count Faça
15.      ind ← (i / j) % M[k].count; // % equivale ao resto da divisão.
16.      result[i][k] ← M[k][ind];
17.      j ← j * M[k].count;
18.    Próximo k
19.  Próximo i
20. Fim.
```

Figura 3.4. Algoritmo de Combinação.

Seja A_1, A_2, . . . ,A_n conjuntos quaisquer, o produto cartesiano de A_1, A_2, ..., A_n é o conjunto $A_1 \times A_2 \times ... \times A_n$, formado por todas as sequências $(x_1, x_2, ..., x_n)$, nesta ordem, tais que $x_1 \in A_1$, $x_2 \in A_2$, ..., $x_n \in A_n$. Logo, o produto cartesiano é formado pelo conjunto $A_1 \times A_2 \times ... \times A_n = \{ (x_1, x_2, ..., x_n) : x_1 \in A_1, x_2 \in A_2, ..., x_n \in A_n) \}$ (Lévy, 1979).

O algoritmo ilustrado na Figura 3.4 é responsável pela combinação de valores e utiliza o conceito de produto cartesiano de conjunto. O algoritmo recebe como entrada uma matriz de parâmetros por valores e retorna outra matriz de parâmetros por valores contendo o resultado da combinação. Primeiro é calculado o número total de combinações (linhas 8 a 10). O número de combinações corresponde ao produtório da quantidade de valores de cada parâmetro. A matriz de entrada é percorrida e então é obtido o número de valores de cada parâmetro através do trecho `M[i].count` na linha 10 . De posse do número de combinações, faz-se uma iteração, controlada pela variável `i`, de 1 até o número total de combinações (linha 12). A cada iteração o valor 1 é atribuído a variável `j` e dentro dessa iteração há outro laço (linha 14), controlado pela variável `k`, que vai de 1 até a quantidade de parâmetros. Dentro deste último laço (linhas 14 a 18) é onde ocorre, de fato, a combinação. Primeiro calcula-se a divisão da variável `i` pela variável `j` e atribui à variável `ind`, que serve como índice para os valores dos parâmetros na matriz de entrada, o resto da divisão do resultado desta operação pela quantidade de valores do parâmetro na posição de indíce `k` da matriz de entrada. Em seguida, a matriz que armazena o resultado da combinação, recebe na posição i e k o valor da matriz de entrada na posição k e ind. Após isso, a variável j é atualizada com o resultado da multiplicação dela mesma pelo número de valores do parâmetro da

posição k da matriz de entrada. Ao final de todas as iterações dos dois laços, o resultado da combinação de todos os valores dos parâmetros será armazenado na matriz de saída.

A Tabela 3.2 ilustra três conjuntos (variáveis) com os seus respectivos valores já gerados pelas etapas anteriores. É interessante ressaltar que para este exemplo a quantidade de cenários gerados foi igual a 30 (trina). A Tabela 3.3 apresenta um subconjunto com seis cenários.

Tabela 3.2. Relação dos Parâmetros e seus Respectivos Valores.

***paramA* (*String*)**	***paramB* (*Integer*)**	***paramC* (*Date*)**
João da Silva	21	12/01/2010
Maria José	22	23/15/2010
NULL	20	
	-3	
	0	

Tabela 3.3. Subconjunto dos Casos de Teste Gerados.

Cenários	***paramA* (*String*)**	***paramB* (*Integer*)**	***paramC* (*Date*)**
1	João da Silva	21	12/01/2010
2	João da Silva	21	23/15/2010
3	João da Silva	22	12/01/2010
4	João da Silva	22	23/15/2010
5	João da Silva	20	12/01/2010
6	João da Silva	20	23/15/2010

Por meio da combinação de valores, cada cenário de teste gerado contém um valor de cada variável. Isto significa que todos os cenários de teste contêm valores de todas as variáveis. Fica a cargo do projetista selecionar quais valores dos parâmetros do modelo serão usados em cada cenário. Além disso, sobre a quantidade de cenários de casos de teste, é possível observar que dependendo da quantidade de variáveis a ser avaliada, poderá ocorrer uma "*explosão combinatorial*" de possibilidades. Essa situação foi minimizada com a inserção de algumas possibilidades para restringir os cenários gerados.

Essas possibilidades são oferecidas pela a ferramenta que dá suporte ao método *TESTIMONIUM*. Para isso, foram disponibilizadas, para os tipos numéricos e data, pela ferramenta *TESTIMONIUM* as opções de definir valores apenas fora de um intervalo, gerar o valor zero e gerar valor negativo. Na primeira opção, o usuário define um intervalo de valores, e o método gera valores fora do intervalo informado, procurando explorar as extremidades do intervalo seguindo o critério funcional de análise do valor limite. Com isso, ao invés de três ou mais valores serem gerados (um fora do limite inferior, um ou mais dentro do limite e um fora do

limite superior), apenas dois valores são gerados, um fora do limite inferior e outro um fora do limite superior.

A opção de gerar o valor zero, como diz nome, gera o valor zero para parâmetros de tipo numérico e para paramêtros do tipo data, é gerado o valor 00-00-000, simulando uma data nula. Já a opção para gerar número negativo, gera um número negativo no caso de parâmetros do tipo numérico e para os parâmetros do tipo data, gera uma data inválida. Por exemplo, 33/12/2010.

As alternativas de gerar valores fora de um intervalo, gerar o valor zero e gerar valor negativo de são oferecidas de forma opcional e o responsável pela geração dos testes, analisando o contexto do negócio em teste, pode optar por uma, por outra ou por todas. Isso é importante porque dependendo do tipo do negócio para o qual os cenários de teste serão gerados, não faz sentido ultilizar uma opção, por exemplo, para gerar o número zero. Vale destacar que quanto mais opções forem ultilizadas, maior o número de cenários de teste gerados.

Reduzir um ou dois valores de um parâmetro pode parecer pouco. Entretanto, é interessante ressaltar que o número de cenários gerados, considerando o algoritmo de combinação deste trabalho, é igual ao produto da quantidade de valores de todos os parâmetros. Portanto, remover um valor de um parâmetro tem um impacto maior na redução no número de total de cenários do que simplesmente remover um cenário. Por exemplo, considero os parâmetros *paramA*, *paramB* e *paramC* da Tabela 3.2, cada um com 3, 5 e 2 valores, respectivamente. Com isso serão gerados um total de 3 * 5 * 2 = 30 cenários. Agora vamos supor que o parâmetro *paramB* possua 4 valores ao invés de 5. O que resulta em 3 * 4 * 2 = 24 cenários de teste. Neste caso, houve uma redução de 6 cenários, o que equivale a 20%, considerando a redução de apenas um valor e um parâmetro. Contudo, essa taxa de redução não é fixa e depende do número de parâmetros e valores de cada parâmetro. À medida que a quantidade de parâmetros e seus valores aumenta a taxa de redução tende a diminuir. Ainda assim, a taxa de redução pode ser melhora se aplicada a um maior número de parâmetros e seus valores.

3.3 Integração com o método *ANIMARE*

Como visto na seção 2.4, o *ANIMARE* é um método para validação dos processos de negócio de um sistema através da animação, tendo como foco as regras de negócio, expressas em OCL. Conforme a Figura 2.10, o método é dividido nas seguintes etapas: (a) especificar o modelo conceitual de informação na forma de um DC; (b) especificar o modelo do processo de negócio na forma de um DA; (c) especificar as regras de negócio comportamentais na forma de expressões OCLs; (d) definir o cenário utilizado na animação; e (e) executar o algoritmo de animação.

A integração do método *TESTIMONIUM* com o método *ANIMARE* ocorre em dois momentos: i) o modelo conceitual de informação e o modelo de negócio anotado com as regras de negócio expressas em OCL, que foram modelados no *ANIMARE*, formam os insumos do método *TESTIMONIUM*; ii) de posse dos modelos fornecidos, o método *TESTIMONIUM*, vai então gerar como resultado os cenários de testes, que na ferramenta *ANIMARE* serão usados na animação, permitindo ao projetista verificar a ocorrência de erros em seu modelo de negócio. A Figura 3.5 ilustra a integração dos métodos *TESTIMONIUM* na infra-estrutura do *ANIMARE*.

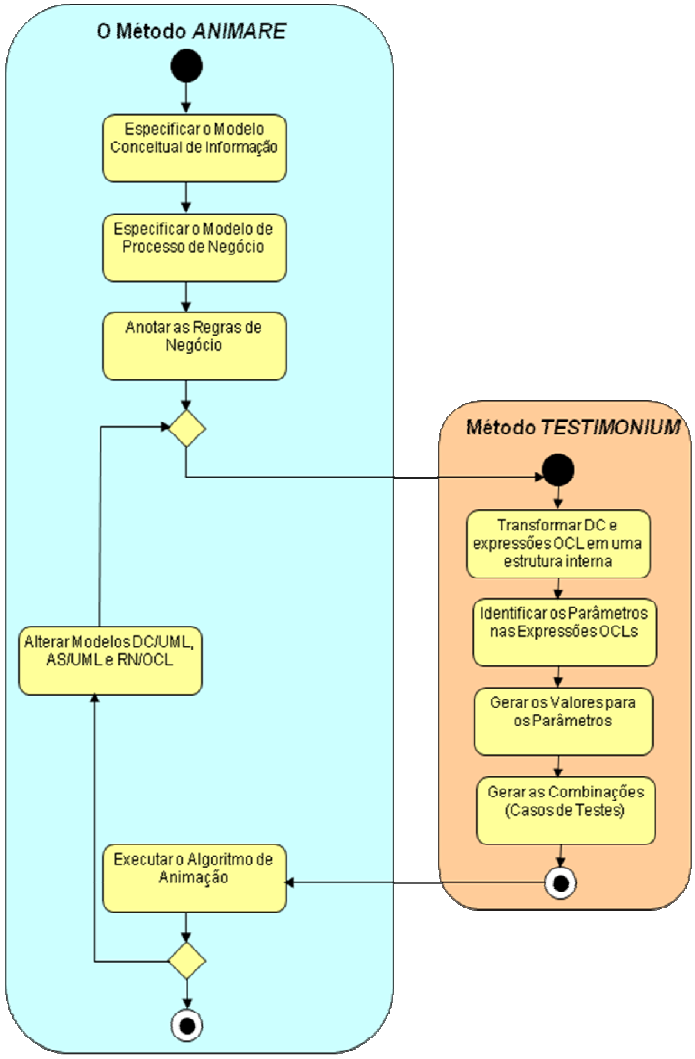

Figura 3.5. Integração do Método *TESTIMONIUM* com o *ANIMARE*.

Como pode ser visto na Figura 3.5 em comparação com a Figura 2.10, a etapa de definir *Cenário de Animação* foi substituída pelo método *TESTIMONIUM*. Dessa forma, ao invés dos cenários serem informados manualmente pelo projetista, eles serão gerados de forma automática pelo método *TESTIMONIUM* por meio da aplicação de critérios da técnica de teste funcional.

3.4 Considerações Finais

Este capítulo apresentou o método *TESTIMONIUM*, detalhando as suas etapas. O próximo capítulo descreve a ferramenta, implementada para dar suporte a este método na geração dos casos de testes, juntamente com a ilustração da aplicação do método através de um estudo de caso.

Capítulo 4

A Ferramenta Desenvolvida

4.1 Considerações Iniciais

A *TESTIMONIUM Tool* é uma aplicação *desktop*, de código aberto, em processo de desenvolvimento, cujo objetivo é fornecer suporte às principais atividades do método *TESTIMONIUM*, descrito no capítulo anterior. A ferramenta ao automatizar os passos do método passa pelas seguintes fases através de um *wizard*[2]: i) o projetista aponta os arquivos XML que descrevem o DC/UML e o arquivo com as expressões OCL; ii) transforma os insumos (DC e OCL) em uma estrutura interna, instanciado os objetos conforme o modelo apresentado na Figura 3.2; iii) associa os parâmetros identificados nas expressões OCL com os seus respectivos tipos; iv) associa um arquivo para cada parâmetro com um conjunto de valores (válidos ou não); v) gera a combinação dos valores; e vi) salva um arquivo no com a combinação gerada.

Visando demonstrar a ferramenta *TESTIMONIUM Tool* é apresentado um estudo de caso de um empreendimento muito comum em diversos países: o de locação de veículos. A locadora ilustrada apresenta as características básicas encontradas na grande maioria dos empreendimentos deste tipo de negócio.

Este capítulo está organizado da seguinte forma: a Seção 4.2 descreve o estudo de caso; a Seção 4.3 ilustra a informação do modelo conceitual de informação e das regras de negócios; a Seção 4.4 descreve a associação dos parâmetros aos seus respectivos atributos; a Seção 4.5 apresenta como informar os valores iniciais dos atributos; a Seção 4.6 ilustra a visualização dos valores gerados e informados de todos os parâmetros; a Seção 4.7 mostra a arquitetura da ferramenta; e a Seção 4.8 apresenta as considerações finais deste capítulo.

[2] *Wizard* é um passo a passo que guia o usuário para realizar uma determinada tarefa.

4.2 Descrição do Estudo de Caso

O estudo de caso, extraído do trabalho de Silveira (2009), refere-se a uma locadora fictícia de automóveis – denominada *Tu-Aluga* – que apresenta um subconjunto de regras de negócio do caso *EU-Rent* idealizado pelo *Business Rules Group* (BRG - http://www.businessrulesgroup.org/egsbrg.shtml).

A Figura 4.1 apresenta o fragmento do DC/UML do estudo de caso, onde se pode observar as principais classes de domínio.

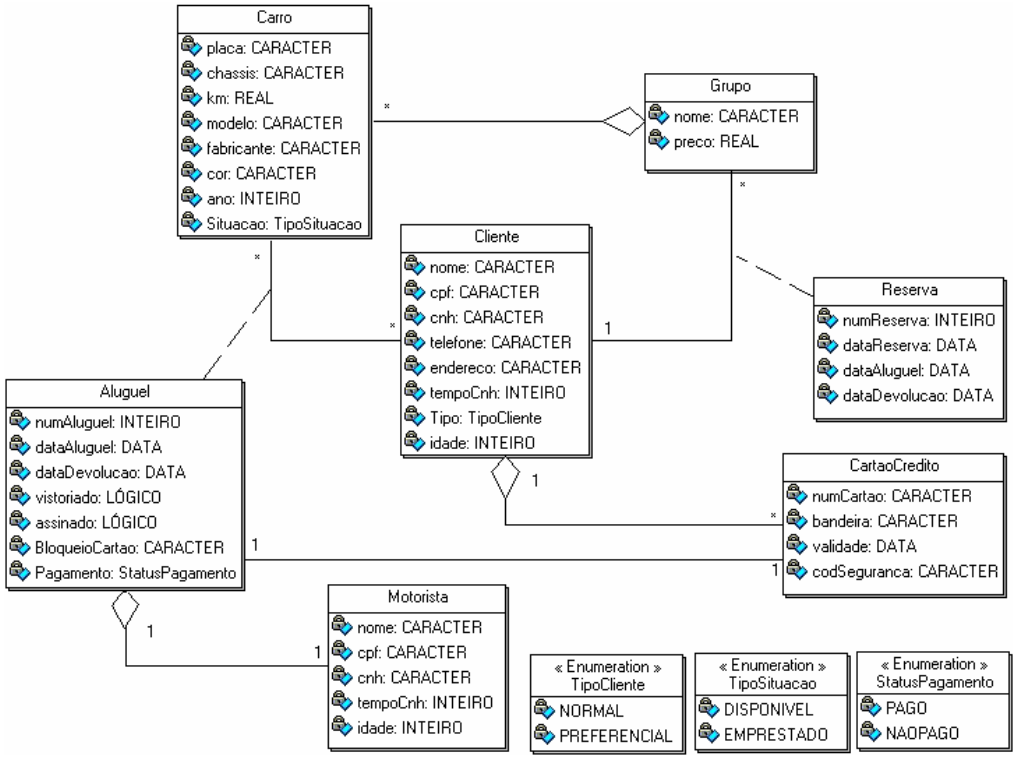

Figura 4.1. Um Fragmento do DC/UML usado no Estudo de Caso (Silveira, 2009).

Os objetos das classes `Reserva` e `Aluguel` são instanciados a partir das respectivas associações: `Grupo` com `Cliente` e `Carro` com `Cliente`. As demais associações são do tipo agregação. `Grupo` agrega `Carro`, `Cliente` agrega `CartaoCredito` e `Aluguel` agrega `Motorista`.

Neste trabalho, optamos por detalhar um dos principais processos de negócio do estudo de caso escolhido: aluguel (com ou sem reserva). A Figura 4.2 apresenta o DA modelado com os objetos que transitam pelo processo de aluguel. As seções que se seguem apresentam as etapas sendo aplicadas passo a passo a este processo.

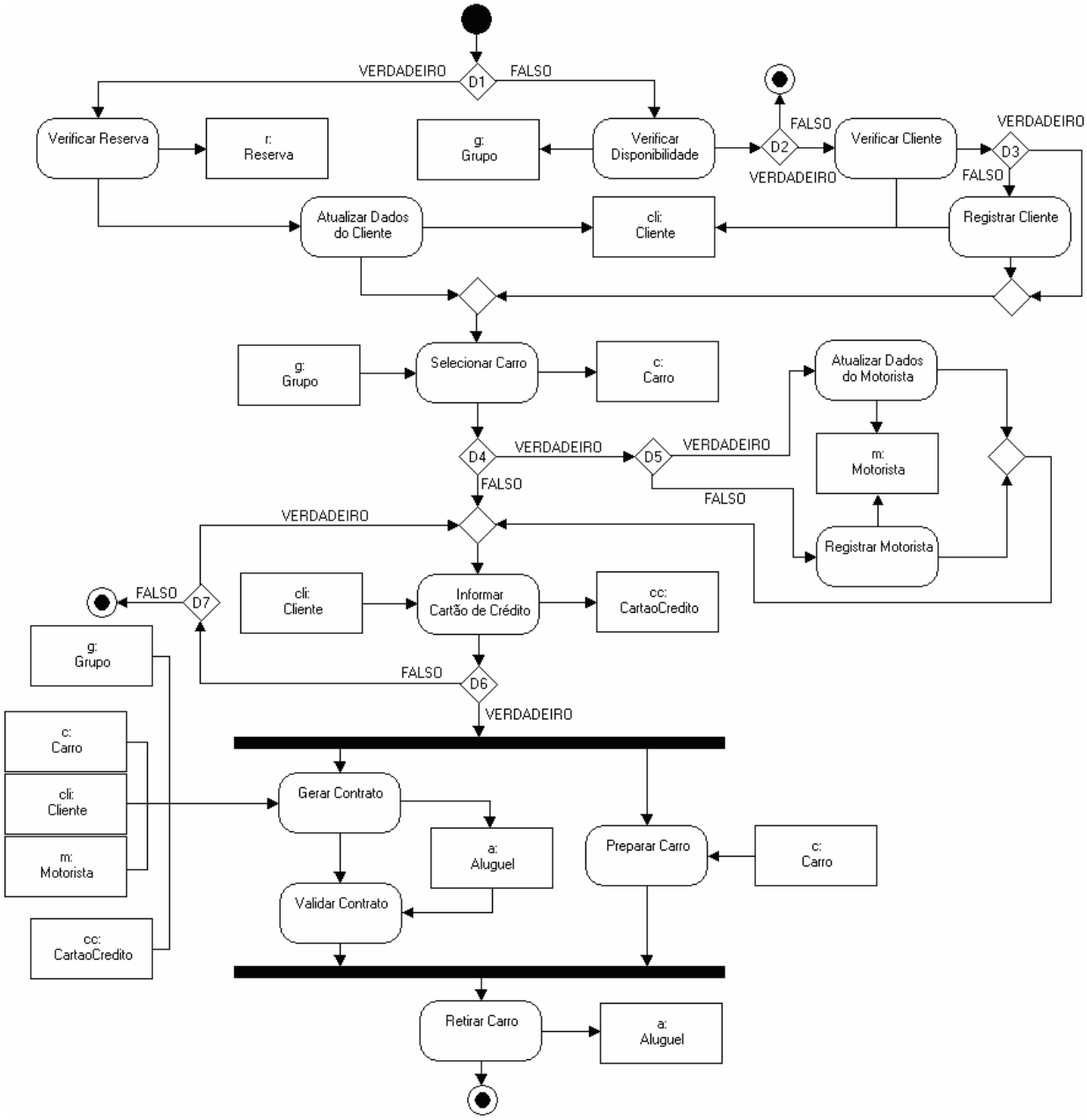

Figura 4.2. DA/UML para o Processo de Negócio de *Alugar Carro*.

4.3 Informar o DC e as regras de negócio

Conforme já mencionando, a *TESTIMONIUM Tool* possui um *wizard*. O primeiro passo desse *wizard* consiste na informação dos insumos (DC e expressões OCL) que irão auxiliar na geração dos casos de testes. A Figura 4.3 apresenta a interface que solicita essas informações.

Para informar o DC foi adotada a linguagem XML. Atualmente, existem diversas ferramentas CASE (*Computer-Aided Software Engineering)* que utilizam esse formato XML para trocar e armazenar dados. Contudo, não existe um padrão de estrutura para os documentos XML gerados por estas ferramentas. Com isso, a *TESTIMONIUM Tool* adotou o padrão usado pela ferramenta *RAPDIS* que originou o *ANIMARE*. Estas ferramentas usam uma combinação de dois

arquivos para armazenar informações sobre o DC, são eles: *Classes.XML* e *Definitions.XML* (Morgado, 2007). O arquivo *Classes.XML* armazena os informações das classes do sistema, como nome e identificador da classe. Já o arquivo *Definitions.XML* armazena todas as definições dos elementos das classes, dentre elas, nome, tipo e identificador de cada atributo. Embora tenha sido adotada a estrutura do *RAPDIS*, qualquer ferramenta que gere as informações do DC neste formato pode ser usada pela *TESTIMONIUM Tool*. O apêndice "A" descreve o formato utilizado através da linguagem *XML Schema*.

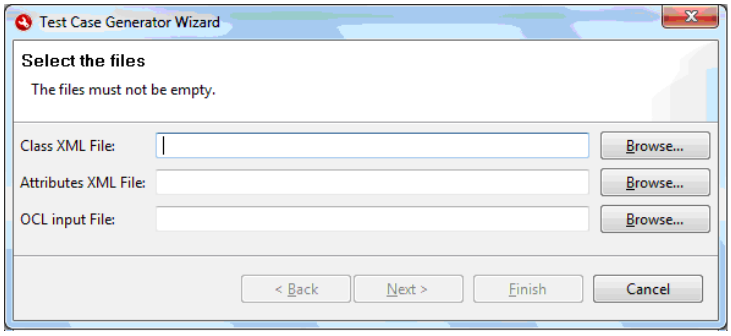

Figura 4.3. Interface para Informar os Arquivos Contendo o DC e as Regras de Negócio.

Através dos campos *Class XML File* e *Attributes XML File* (Figura 4.3) o usuário pode informar os arquivos contendo informações sobre as classes (*Classes.XML*) e seus elementos (*Definitions.XML*), respectivamente. Esses arquivos podem possuir qualquer nome, desde que estejam no formato informado e com a extensão "*.XML*".

Já o campo *OCL Input File* é usado para informar o arquivo com as expressões OCLs que devem expressar as regras de negócio do modelo de negócio (DA). Nele o usuário informa um único arquivo de extensão "*.OCL*" contendo todas as regras de negócio expressas em OCL. Vale destacar que não é objetivo da ferramenta validar se os arquivos informados estão sintaticamente corretos.

O apêndice "B" apresenta o conteúdo dos arquivos XML que representam o modelo conceitual de informação, bem como o arquivo OCL com as regras de negócio que foram utilizadas no exemplo que segue.

É interessante ressaltar que embora não seja visível para o projetista, o usuário da ferramenta, após a informação desses insumos ocorre o *parser* dos mesmos para a estrutura interna – uma lista encadeada com todos os elementos – que será manipulada pela ferramenta nas outras etapas do processo de geração dos casos de teste.

4.4 Associar os Parâmetros aos seus Respectivos Atributos

Neste passo ocorre a associação dos parâmetros com os seus respectivos atributos do modelo conceitual de informação. Como o método proposto gera cenários de teste de acordo com o tipo de dado, a associação entre atributos e parâmetros é necessária para se obter os tipos dos parâmetros.

Os atributos foram obtidos pelo *parser* dos arquivos XML fornecidos no primeiro passo e que juntos representam o DC. Os parâmetros são identificados através da compilação das expressões OCL. A Seção 3.2.2 demonstrou como ocorre essa identificação. A interface ilustrada pela Figura 4.4 possui uma relação com todas as classes do modelo que foi informado, bem como seus atributos e os tipos dos atributos. Nesta interface, também existe uma tabela que descreve os parâmetros e os atributos nos quais eles estão associados.

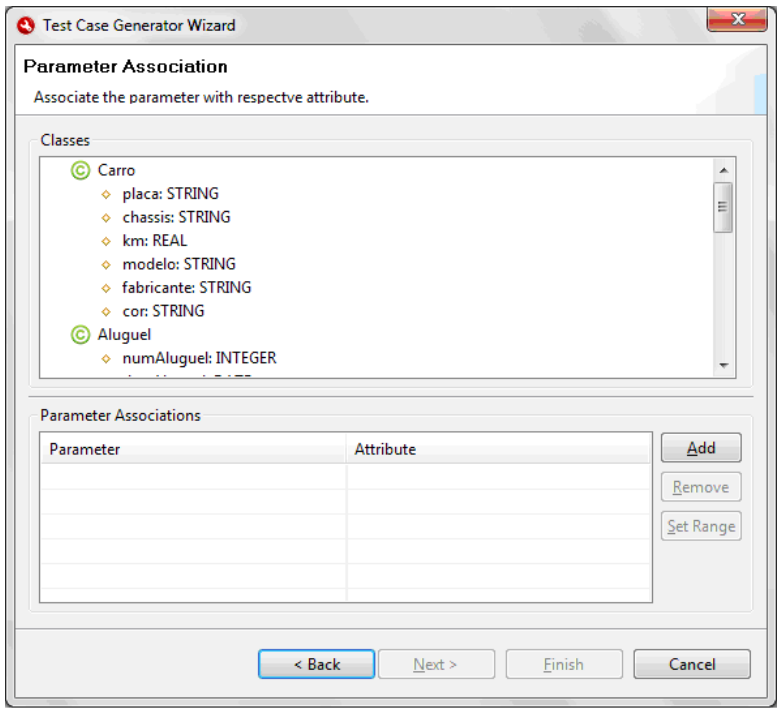

Figura 4.4. Interface de Associação de Parâmetros e Atributos.

Ainda na interface da Figura 4.4 existem os botões: *Add*, *Remove* e *Set Range*. Ao acionar o botão *Add*, a interface ilustra na Figura 4.5 é chamada. Nela o projetista poderá para associar o parâmetro ao seu respectivo atributo.

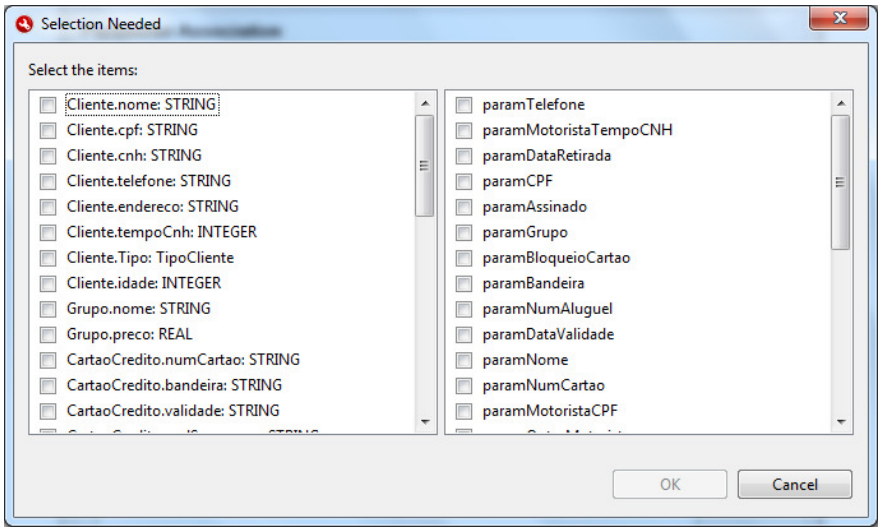

Figura 4.5. Interface para Criar Associação entre Parâmetros e Atributos.

Conforme pode ser observado na Figura 4.5, do lado esquerdo da interface é possível observar relação de todos os atributos com suas respectivas classes e tipos, no seguinte formato: <ClassName>.<AttributeName>: <AttributeType>. Já no lado direito encontra-se os parâmetros identificados a partir das expressões em OCL.

Porém, para criar uma associação, o usuário deve selecionar um atributo e um parâmetro e pressiona o botão *OK*, com isso, a associação é criada e adicionada na lista de associações. A Figura 4.6 mostra o resultado da associação dos parâmetros aos atributos para esse estudo de caso.

O botão *Remove* (Figura 4.4) serve para remover uma associação da lista de associações disponíveis. Neste caso, o usuário deve selecionar a associação que deseja remover, o que automaticamente habilita o botão *Remove* e acionar o botão. Com isso a associação é removida e não constará mais na lista de associações.

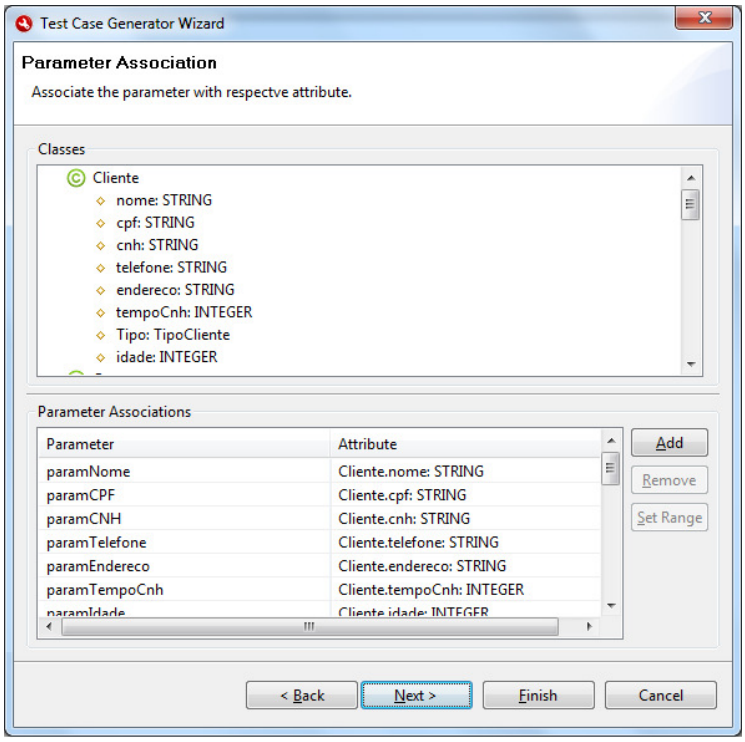

Figura 4.6. Resultado da Associação dos Parâmetros aos seus Respectivos Atributos.

A ferramenta *TESTIMONIUM Tool* disponibiliza, para os tipos numéricos e o tipo *Date*, uma opção para definir faixas de valores. Para tal, o usuário deve selecionar o botão *Set Range* que consta na Figura 4.4. Após selecionar esse botão, a ferramenta apresenta a interface apresentada na Figura 4.7, que possibilita ao usuário restringir os valores que deverão ser criados. O objetivo disto é tentar reduzir o número de cenários de teste gerados, já que a método proposto, assim como qualquer método automático, pode gerar um grande número de casos de teste, incluindo redundância (Binder, 1999).

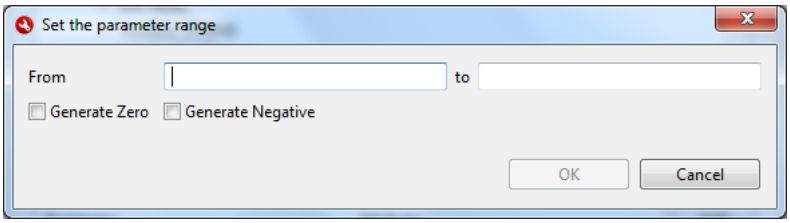

Figura 4.7. Interface para Definir Intervalos de Valores.

Na Figura 4.7 é possível observar os campos de edição: *From* e *To*. O campo *From* representa o início do intervalo e o campo *to* o final. Neste caso, os casos de teste são gerados definindo valores fora desse intervalo. Além disso, encontram-se as opções: *Generate Zero* e *Generate Negative*. A opção *Generate Zero*, quando habilitada, indica que o valor zero deve ser gerado. Já no caso da opção *Generate Negative*, se habilitada, indica que um valor negativo deve ser gerado.

4.5 Informar Valores Iniciais dos Atributos

Além dos valores gerados, a ferramenta permite ao usuário informar valores para qualquer parâmetro. Isto é útil principalmente para parâmetros do tipo *String*, já que a ferramenta não gera valores para este tipo, porque tais valores são muitos subjetivos e depende do contexto onde estão inseridos. Portanto, como já mencionado, fica a cargo do usuário informar valores para este tipo.

A Figura 4.8 e a Figura 4.9 apresentam as interfaces da ferramenta que permitem ao usuário informar os valores para esse tipo de atributo. A Figura 4.8 ilustra a relação das classes e seus respectivos arquivos contendo os valores de seus parâmetros. Nela, ao acionar o botão *Add*, a interface que permite adicionar a combinação classe e arquivo é exibida, como ilustra a Figura 4.9.

Na interface representada pela Figura 4.9, o campo *Class CSV File* permite que o arquivo contendo os valores dos parâmetros seja informado. Os valores devem ser informados através de um formato de arquivo conhecido como *Comma Separated Values* (CSV). Este formato de arquivo é uma implementação particular de arquivos de texto separados por um delimitador, que usa a vírgula e a quebra de linhas para separar valores. Deve ser informado um arquivo por classe. Se houver necessidade de informar valores para mais de um parâmetro, estes devem ser separados por vírgula. A Figura 4.11 mostra o formato do arquivo CSV usado neste passo.

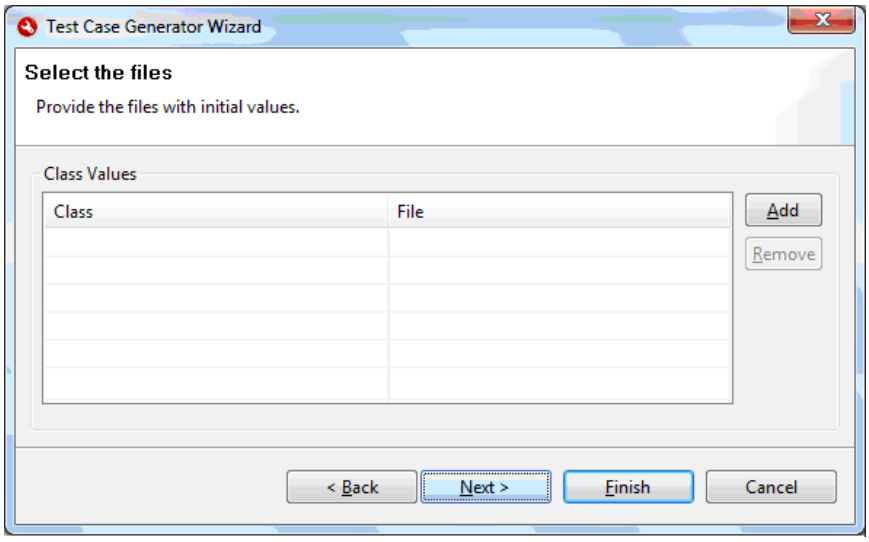

Figura 4.8. Interface Contendo a Relação de Classes e seus Arquivos.

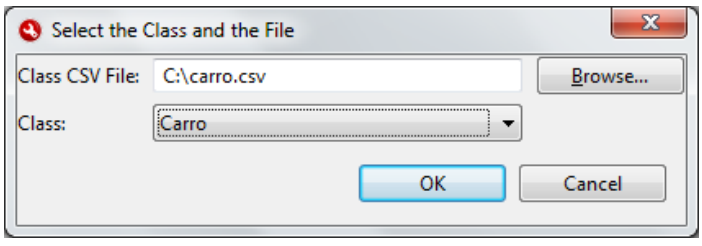

Figura 4.9. Interface para Adicionar a Combinação Classe e Arquivo CSV.

O campo *Class* contém a lista de todas as classes do DC, informado no primeiro passo deste *wizard*. Após selecionar uma classe e acionar o botão *OK* uma nova entrada com a combinação classe e arquivo de valores será mostrada na tabela que contem a relação de classes e seus arquivos (Figura 4.10).

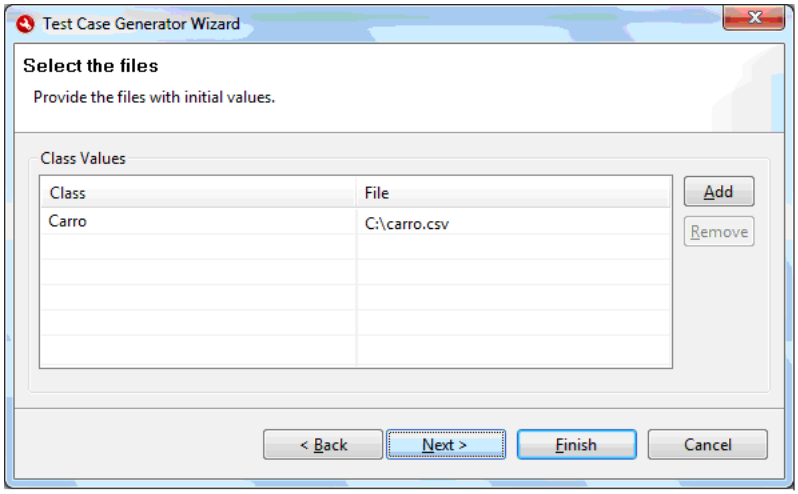

Figura 4.10. Interface Contendo a Relação de Classes e seus Arquivos.

```
001. <AttributeName1>,...,<AttributeNameN>
002. <AttributeName1Value1>,...,<AttributeNameNValue1>
003. <AttributeName1Value2>,...,<AttributeNameNValue2>
...
00m. <AttributeName1ValueM>,...,<AttributeNameNValueM>
```

Figura 4.11. Formato do Arquivo CSV Usado no passo de Informar Valores.

As Figuras que seguem (4.12 até 4.16) ilustram os arquivos CSV que foram usados no estudo de caso para as classes `Cliente`, `Grupo`, `CartaoCredito`, `Carro`, `Motorista` e `Aluguel`, respectivamente.

```
001. nome, telefone, cpf, cnh, endereco
002. "João da Silva", "3231-3455", "343.123.555-01", "327.657.098", "Rua
Amapa"
003. "Beatriz Castro", "9843-0908", "222.930.745-34", "109.827.940", "Av Rio
Branco"
```

Figura 4.12. Arquivo CSV da Entidade Cliente.

```
001. nome
002. "Popular com ar"
003. "Popular sem ar"
```

Figura 4.13. Arquivo CSV da Entidade Grupo.

```
001. numCartao, bandeira, codSeguranca
002. "5123 3425 6780 8732", "Visa", "054"
003. "9633 6252 0972 6252", "MasterCard", "180"
```

Figura 4.14. Arquivo CSV da Entidade CartaoCredito.

```
001. nome, cpf, cnh
002. "Paula Souza", "987.098.165-06", "521.093.452"
003. "Leonardo Pereira", "432.652.891-87", "915.243.674"
```

Figura 4.15. Arquivo CSV da Entidade Motorista.

```
001. BloqueioCartao
002. "3213"
003. "8764"
```

Figura 4.16. Arquivo CSV da Entidade Aluguel.

4.6 Visualização dos Valores Gerados e Informados

A visualização dos valores gerados é o último passo do *wizard* e tem por objetivo a pré-visualização tanto dos valores gerados pelo método *TESTIMONIUM* através das regras definidas na Seção 3.2.3 do capítulo anterior, quanto dos valores informados pelo usuário através do arquivo CSV, como mostrado na seção anterior. A Figura 4.17 apresenta a interface que representa este passo com os valores gerados para o estudo de caso em questão.

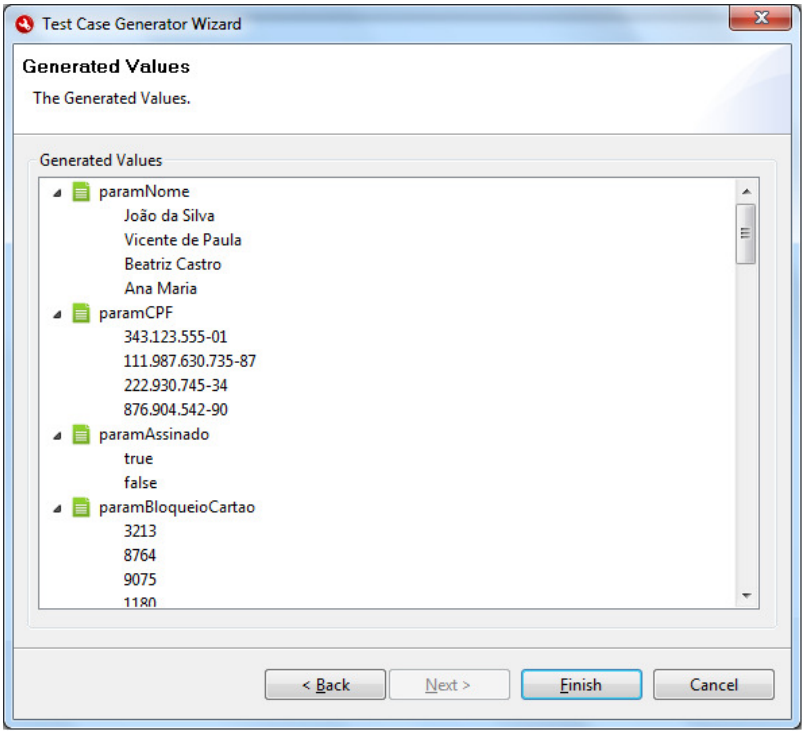

Figura 4.17. Interface que Exibe os Valores e Gerados de Todos os Parâmetros.

Ao pressionar o botão *Finish* os valores são combinados e um arquivo CSV será gerado com todas as combinações de cenários de teste. O arquivo, de nome "*generatedValues.csv*", é gerado no diretório raiz da ferramenta. A Figura 4.18 exibe o formato do arquivo gerado.

```
001. <NameParameter1>,...,<NameParameterN>
002. <Parameter1Value1>,...,<ParameterNValue1>
003. <Parameter1Value2>,...,<ParameterNValue2>
...
00n. <Parameter1ValueN>,...,<ParameterNameNValueN>
```

Figura 4.18. Formato do Arquivo CSV Gerado Contendo os Cenários de Teste.

Para o exemplo do estudo de caso foram gerados 75.497.472 cenários de teste a partir dos 24 parâmetros que seguem: *paramNome*, *paramCPF*, *paramCNH*, *paramTelefone*, *paramEndereco*, *paramTempoCnh*, *paramIdade*, *paramGrupo*, *paramMotoristaNome*, *paramMotoristaCPF*, *paramMotoristaTempoCNH*, *paramMotoristaCNH*, *paramMotoristaIdade*, *paramNumCartao*, *paramDataValidade*, *paramCodSeguranca*, *paramBandeira*, *paramNumAluguel*, *paramVistoriado*, *paramAssinado*, *paramBloqueioCartao*, *paramDataRetirada*, *paramDataDevolucao* e *paramNumReserva*. A geração de todos os cenários levou cerca de quinze minutos.

Como pôde ser observado, a elevada quantidade de cenários que foi gerado deixa claro a necessidade de adotar alguma estratégia para redução destes cenários. Entretanto, o objetivo desta pesquisa é a geração dos cenários, ficando como um trabalho futuro a adoção de novas estratégias para a redução do número de cenários de teste. Porém, inicialmente, estamos dando a oportunidade do usuário do *TESTIMONIUM* gerar os em cenários vários arquivos. Cada um desses arquivos contém 5% do total de cenários. Esta possibilidade facilita a execução de testes por amostragem, pois considera-se apenas um subconjunto dos cenários gerados diminuindo o esforço e o tempo necessário para a execução dos mesmos. Além disso, a execução de testes por amostragem é uma característica já consolidada da indústria (Kalton, 1990; Henry, 1990; Jerome, 2002).

Neste exemplo, o método gerou 20 outros arquivos, contendo cada um deles um total de 3.774.873 cenários. Supondo que todos os 3.774.873 cenários fossem executados no *ANIMARE* de forma automática e que o mesmo levasse seis milissegundos para executar cada cenário de teste, a execução de todos os cenários levaria em torno de 6 horas.

A Figura 4.19 ilustra o conteúdo de um dos arquivos gerado contendo um subconjunto dos cenários de teste gerados para o estudo de caso.

```
001. Nome, TempoCnh, DataDevolucao, DataAluguel, Grupo, Vistoriado
002. "João da Silva","0","32/01/2009","31/12/2008","Popular com ar","true"
003. "João da Silva","0","32/01/2009","31/12/2008","Popular com ar","false"
004. "João da Silva","0","32/01/2009","31/12/2008","Popular sem ar","true"
005. "João da Silva","0","32/01/2009","31/12/2008","Popular sem ar","false"
006. "João da Silva","0","32/01/2009","32/01/2009","Popular com ar","true"
007. "João da Silva","0","32/01/2009","32/01/2009","Popular com ar","false"
008. "João da Silva","0","32/01/2009","32/01/2009","Popular sem ar","true"
009. "João da Silva","0","32/01/2009","32/01/2009","Popular sem ar","false"
010. "João da Silva","-576","00/00/0000","32/01/2009","Popular sem ar","true"
011. "João da Silva","-576","00/00/0000","32/01/2009","Popular sem ar","false"
012. "João da Silva","-576","00/00/0000","32/01/2009","Popular com ar","true"
013. "João da Silva","-576","00/00/0000","32/01/2009","Popular com ar","false"
014. "Beatriz Castro","1","32/01/2009","00/00/0000","Popular sem ar","false"
015. "Beatriz Castro","1","32/01/2009","00/00/0000","Popular sem ar","true"
```

```
016. "Beatriz Castro","1","32/01/2009","00/00/0000","Popular com ar","true"
017. "Beatriz Castro","1","32/01/2009","00/00/0000","Popular com ar","false"
018. "Beatriz Castro","1","32/01/2009","32/01/2009","Popular sem ar","false"
019. "Beatriz Castro","1","32/01/2009","32/01/2009","Popular sem ar","true"
020. "Beatriz Castro","1","32/01/2009","32/01/2009","Popular com ar","false"
021. "Beatriz Castro","1","32/01/2009","32/01/2009","Popular com ar","true"
022. "Beatriz Castro","81","31/01/2009","00/00/0000","Popular com ar","false"
023. "Beatriz Castro","81","31/01/2009","00/00/0000","Popular com ar","true"
024. "Beatriz Castro","81","31/01/2009","00/00/0000","Popular sem ar","false"
025. "Beatriz Castro","81","31/01/2009","00/00/0000","Popular sem ar","true"
```

Figura 4.19. Subconjunto dos Cenários de Teste Gerados.

4.7 Arquitetura da Ferramenta

Com relação à arquitetura, a ferramenta foi baseada no padrão arquitetural em Camadas, estando dividida em duas, como mostra a Figura 4.20: camada de apresentação e camada de negócio.

Para modelagem da camada de apresentação foi utilizado o ambiente de desenvolvimento *Eclipse Rich Client Platform* (RCP). O *Eclipse* RCP permite aos desenvolvedores usar à arquitetura do *Eclipse* para projetar aplicações *desktop* extensíveis reusando funcionalidades e padrões de código existentes na IDE (*Integrated Development Environment*) *Eclipse*.

O *Eclipse* possui uma arquitetura baseada em *plug-in* (Clayberg & Rubel, 2006; Gamma & Back, 2004; McAffer & Lemieux, 2005) que provê extensibilidade e reusabilidade das principais funcionalidades de uma IDE.

Todas as funcionalidades do *Eclipse* são descritas como *plug-ins* e *features* (conjunto de *plug-ins*). Um *plug-in* é a estrutura básica da arquitetura do *Eclipse*. A comunicação e relacionamentos de dependência de plug-ins são descritos através de seus pontos de extensão (*extension points*). Cada *plug-in* declara a lista de *plug-ins* que ele estende (depende) e os pontos que ele dispõe para que outros *plug-ins* possam estender (Clayberg & Rubel, 2006).

Dentre os principais benefícios do RCP, estão (McAffer & Lemieux, 2005): i) Interface gráfica consistente e atrativa, pois é baseada no Sistema Operacional em que a aplicação está sendo executada; ii) Serviços comuns para aplicações, permitindo ao desenvolvedor focar apenas no núcleo da aplicação. Entre alguns serviços estão: gerenciador de conteúdo de Ajuda; suporte a instalação e atualização; editores de textos; manipulação de dados, etc.; iii) Executa em múltiplas plataformas, dispositivos e configurações.

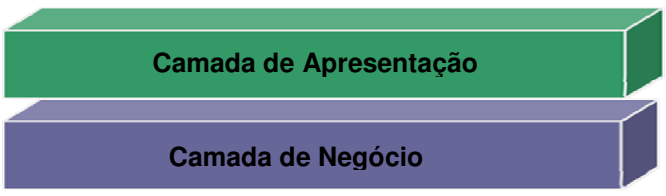

Figura 4.20. Arquitetura da *TESTIMONIUM Tool.*

Para a camada de negócio foi utiliza uma versão simplificada do metamodelo da UML, que foi apresentada na Figura 3.2.

4.8 Considerações Finais

O estudo de caso apresentado neste capítulo teve como objetivo mostrar uma aplicação do método *TESTIMONIUM* através da ferramenta *TESTIMONIUM Tool* no intuito de apresentar indícios de sua aplicabilidade e de mostrar o potencial dos casos de teste gerados em revelar falhas. Como resultado, o estudo de caso apontou a viabilidade do método proposto. Contudo, é de conhecimento que uma validação mais abrangente da utilidade do método deve ser realizada. Nesta validação exige um experimento controlado envolvendo diversos projetos com uma população maior de usuários. Estes experimentos permitiriam a obtenção de resultados estatisticamente significativos para aceitar a hipótese central desta dissertação sobre a utilidade do método.

Capítulo 5

Trabalhos Relacionados

A realização de testes exaustivos ainda é um grande desafio, e o sucesso da atividade de teste depende da qualidade dos casos de teste a serem aplicados no sistema. Portanto, muitos pesquisadores têm investido em melhorar a qualidade dos casos de teste. Um estudo realizado em (Prasanna *et al*., 2005) sobre pesquisas sendo desenvolvidas em geração automática de casos de teste na última década, mostrou que a maior parte das investigações tem se concentrado em geração de casos de teste baseadas em modelos. Na literatura especializada também são encontrados vários trabalhos que visam facilitar o processo de testes derivando os casos de testes a partir de modelos (Hartmann, 2000), (Cheon & Avila, 2010), (Kundu & Samanta, 2009), (Borba *et al.*, 2007), (Orozco *et al.*, 2009).

Cheon & Ávila (2010) apresentam uma forma automatizar testes de programas escritos na linguagem Java através de aspectos do DC da UML e de restrições OCL. As restrições OCL são traduzidas em verificações em tempo de execução. Para isso as restrições OCL são traduzidas em código do paradigma orientado a aspectos, que será compilado junto com o código Java do sistema. O objetivo dos autores é detectar violações nas restrições em tempo de execução. Esta abordagem é dinâmica, pois necessita que o código que será testado seja executado. Primeiramente os dados são gerados de forma randômica e em seguida realiza-se a execução dos testes através da invocação dos métodos. Assim como a abordagem que foi proposta nesta dissertação, a geração dos dados de teste no trabalho desses autores depende do tipo de dado. Para tipos primitivos, um valor arbitrário para o tipo é selecionado randomicamente. Porém, é interessante ressaltar que a desvantagem desta abordagem em relação ao método proposto é que ela é dependente da linguagem *Java*, e só pode ser aplicada na fase de codificação.

Borba *et al.* (2007) propõe em seu trabalho um a ferramenta com o nome de *TaRGeT*, que automatiza uma abordagem sistemática para tratar artefatos de requisitos e testes de forma integrada. Nesta abordagem, os casos de teste podem ser gerados automaticamente a partir de

cenários de casos de uso escritos em linguagem natural. Os casos de uso devem ser escritos seguindo um *template* do editor de texto *Microsoft Word*, que foi criado contendo informações necessárias para gerar procedimentos de teste, descrições e requisitos relacionados. Além disso, a ferramenta também pode gerar uma matriz de rastreabilidade entre casos de teste, casos de uso e requisitos. O autor ainda destaca que os três principais aspectos que diferenciam a sua ferramenta das outras ferramentas simulares são:

- o uso de seleção de casos de teste baseada em propósitos, para restringir o número de casos de teste gerados focando os casos de teste que são mais críticos ou relevantes para uma determinada tarefa. Os propósitos de teste especificam que uma funcionalidade particular do sistema deve ser testada. Assim, dado o propósito de teste e o modelo da aplicação, é realizado um casamento de padrões entre eles, e somente casos que atendam ao propósito de teste são gerados (Andrade, 2007);
- uso de algoritmos para eliminação de casos de teste similares, reduzindo o tamanho das suítes de teste sem impactos significantes. Porém, é interessante ressaltar que o trabalho não esclarece como os casos de teste similares são eliminados;
- uso da linguagem natural para descrever os casos de uso, mais fácil para os engenheiros.

Entretanto, como já foi dito anteriormente, o uso da linguagem natural pode levar a problemas de inconsistências e ambigüidades. Em geral, especificações não rigorosas deixam grande margem a opiniões e especulações (Delamaro *et al*., 2007). Considere um exemplo apresentado por (Apfelbaum & Doyle, 1997), no qual a frase é apresentada em um documento de requisitos: "*Se um dígito inválido é fornecido, ele deve ser tratado de maneira adequada*". Porém, este requisito deixa uma dúvida em relação ao que vem a ser uma '*maneira adequada*'. Um desenvolvedor pode julgar que a '*maneira adequada*' é permitir que o usuário tente digitar novamente. Outro desenvolvedor pode julgar que o mais adequado é abortar o comando. Qualquer uma das duas implementações é plausível, mas apenas uma delas deve ser utilizada. O testador fica em uma posição na qual recusa tudo que não está de acordo com o próprio julgamento, ou aceita tudo o que pode estar correto no julgamento de alguém.

No trabalho proposto por Hartmann *et al.* (2000) são usados modelos UML. Nele os autores propõem a descrição detalhada dos Casos de Uso através de DA. Dessa forma, cada caso de uso é representado por um DA e os casos de teste para cada caso de uso são criados a partir de seus respectivos DAs. Entretanto, o trabalho de Hartmann *et al.* (2000) considera apenas a criação de testes para cada caso de uso isoladamente. Isso pode onerar na execução dos casos de teste, pois em vários sistemas há muitas dependências entre os casos de uso, ou seja, um caso de teste pode depender da execução de outros casos de uso, mas contém somente o fluxo interno do

caso de uso para o qual foi projetado. Outro problema é que a ferramenta por eles proposta depende da IBM *Rational Rose*, que é proprietária.

Kundu & Samanta (2009) propõem uma abordagem para geração de casos de teste a partir de DA. Porém, para a geração dos casos de testes o DA deve ser acrescido das seguintes informações:

1. As atividades que alteram o estado de algum objeto devem explicitá-lo como pino de entrada e de saída. Juntamente como os pinos o projetista deve informar o estado inicial no pino de entrada e o estado final no pino de saída.
2. As atividades que criam objetos devem explicitá-lo apenas como pino de saída.
3. Os elementos de decisão, separação, junção e *loops* devem ser substituídos por uma atividade de nível de abstração maior.

Depois disso, este diagrama é transformado em um grafo de atividades. Neste passo, cada elemento do DA é mapeado em um tipo diferente de nó do grafo. Por fim, os testes são obtidos através de critério que visam percorrer de várias formas os caminhos do grafo de atividades. A abordagem não dispõe de uma ferramenta para geração automática e considera apenas DA para um caso de uso por vez, assim como no trabalho de Hartmann *et al.* (2000).

Em Orozco *et al.* (2009), é apresentada uma abordagem de teste de *software* baseada em modelos que se concentra na identificação, automatização e derivação completa, ou parcial, de casos de teste funcional, a partir do DA/UML. Características relacionadas a testes funcionais tais como a ação do usuário e o resultado esperado são inseridos no DA. Nesta abordagem, para conseguir redução de esforço na geração de testes, os autores propõem a composição dos diferentes DA do sistema a ser testado em um único diagrama, evitando assim atividades redundantes e, conseqüentemente, casos de teste redundantes.

A Tabela 5.1 sumariza as características dos trabalhos apresentados, considerando: o tipo de técnica de teste usada, se é funcional ou estrutural, os diagramas usados, se considera a eliminação de casos de teste semelhantes, se existe a execução de casos de testes, em qual fase do processo é aplicada, e se dependente de tecnologia.

Tabela 5.1. Resumo das Características das Abordagens.

Trabalho	Técnica de Teste	Modelo(s)	Eliminação dos Casos de Teste Similares?	Execução dos Casos de Teste	Fase do Processo	Depen-dente de Tecnologia
Cheon & Avila, 2010	Estrutural	DC + OCL	Não	Sim	Implemen-tação	Sim
Borba *et al.*, 2007	Funcional	Diagrama de Casos de Uso	Sim	Não	Requisitos	Não
Hartmann *et al*., 2000	Funcional	Diagrama de Casos de Uso e DA	Não	Não	Análise	Sim
Kundu & Samanta, 2009	Funcional	DA	Não	Não	Análise	Não
Orozco *et al.*, 2009	Funcional	DA	Sim	Não	Análise	Não
TESTIMONIUM	Funcional	DC e DA + OCL	Não	Sim	Análise	Não

Como ilustra a Tabela 5.1, a maioria dos trabalhos utiliza a técnica funcional. Com isso eles podem ser utilizados antes da fase de implementação fazendo com que o os erros sejam descobertos o quanto antes implicando em redução de tempo e custo. O método *TESTIMONIUM*, através da OCL, garante que as regras sejam expressas de forma clara e precisa, minimizando problemas de imprecisões e inconsistências. Outro diferencial do método aqui proposto, em relação ao outros trabalhos, é que, além da geração dos casos de teste, os testes gerados poderão ser executados e acompanhados de forma automatizada através de uma implementação do método de animação *ANIMARE*.

Capítulo 6

Conclusões e Trabalhos Futuros

6.1 Considerações

O desenvolvimento de *software* deve passar por uma série de atividades de garantia de qualidade, chamadas coletivamente de VV&T. Conforme já apresentado, a atividade de testes é crucial no processo de desenvolvimento de *software*, porém realizar testes com qualidade demanda muito esforço e tempo (Pressman, 2006; Delamaro *et al.*, 2007). Idealmente, a atividade de testes deve estar presente nas fases iniciais do ciclo de desenvolvimento do *software*, pois quanto mais cedo um erro é descoberto, antes de sua propagação em fases subseqüentes, menos dispendiosa é a sua eliminação.

Atualmente, testes baseados em modelos têm sido cada vez mais utilizados. No contexto desta dissertação a vantagem oferecida pelo uso de modelos é possibilitar a geração de casos de teste de forma automática, a partir da especificação nas fases iniciais do ciclo de desenvolvimento de *software*, servindo com uma estrutura de teste com baixo impacto de manutenção.

Grande parte do esforço de pesquisa desta dissertação está na concepção e elaboração de um método para o auxílio à geração de casos de testes funcionais a partir de modelos de negócio, anotados com regas de negócio. A estratégia adotada foi o uso de especificações de regras de negócio mais formais, para geração dos casos de teste, uma vez que as mesmas vão proporcionar uma geração mais precisa do que as expressas em linguagem natural. A geração de casos de testes proposta aqui foi apoiada por critérios de testes funcionais como a análise do valor limite e particionamento de equivalência.

Um aspecto interessante a ressaltar é que os próprios modelos servem como fonte dos testes. As mudanças nos modelos refletem mudanças nos testes. Através do método proposto, este processo exige pouco esforço, pois a geração de testes a partir de modelos pode ser feita de forma automática.

Como apontado ao longo desta dissertação, existem trabalhos relacionados que propõem a construção de casos de teste a partir de linguagem em texto natural, ou de modelos como DC com OCL incluindo certo grau de formalização das regras de negócio. Contudo, as metodologias resultantes destas pesquisas ou levam a problemas de inconsistências e ambigüidades, decorrente do uso de linguagem natural, ou usam OCL mas são limitadas a uma tecnologia específica.

Em contrapartida, o método e a ferramenta desenvolvidos nesta dissertação são independentes de linguagem e promovem uma maior automação das atividades de teste, considerando que usam modelos com maior formalização das regras de negócio (através do uso de expressões em OCL, o que permite que sejam mais precisos e mais automatizáveis). Com isto, o testador do sistema terá mais tempo para se dedicar à definição do critério de teste e à análise dos resultados. Além disso, os critérios funcionais, utilizados no método proposto, podem ser aplicados em todas as fases de testes e em produtos desenvolvidos com qualquer paradigma de programação, já que não levam em consideração os detalhes de implementação.

Um diferencial deste método é que ele foi projetado para poder ser facilmente integrável a qualquer ferramenta CASE que exporte os seus modelos em XML. No entanto, inicialmente ela foi integrada a ferramenta *ANIMARE*, que possibilitou verificar a potencialidade do método aqui proposto. A possibilidade do uso da animação, provida pelo *ANIMARE*, para execução dos casos de teste, permitiu levantar questões sobre a especificação do método, onde foi possível obter respostas de forma rápida e automática, através de um ambiente interativo de execução que a ferramenta *ANIMARE* propicia.

6.2 Contribuições

Dentre as contribuições deste trabalho de pesquisa, pode-se destacar resumidamente:

- O levantamento do estado da arte sobre as técnicas de testes baseadas em modelo;
- A definição do método, chamado *TESTIMONIUM*, para geração de instâncias de testes a partir das regras de negócio expressas em OCL e do modelo conceitual de informação no formado de um DC;
- A implementação do protótipo de uma ferramenta de suporte ao método;
- Definição de algumas heurísticas para a geração dos casos de teste mais importantes, em outras palavras, casos de teste que possuam alta probabilidade de achar a maior quantidade de erros com uma quantidade mínima de tempo e esforço, para um dado contexto, que, avaliam propriedades específicas;
- Definição de uma proposta de integração com o método/ferramenta *ANIMARE*.

6.3 Trabalhos Futuros

No método *TESTIMONIUM* foram definidas algumas heurísticas para a geração dos casos de teste mais importantes. Entretanto, mesmo com essas heurísticas, dependendo da complexidade das características de um sistema em teste, pode ocorrer a explosão do espaço de teste (explosão combinatorial na geração de casos de teste) (Binder, 1999). Milhares de testes gerados acabam apresentando redundância e longo tempo de processamento. Isto pode ser futuramente minimizado através da adoção de estratégias para a seleção e priorização dos casos de teste mais importantes.

Em trabalhos futuros, considera-se ainda relevante o planejamento e execução de um experimento em outros domínios de aplicação com o objetivo de avaliar, de forma mais abrangente, os impactos relativos ao uso do método proposto nesta dissertação por um número maior de participantes, em diferentes ambientes. Outro aspecto que deve ser trabalhado, para que o método seja aceito mais facilmente pela academia e indústria, é a geração de instâncias de teste para tipos abstratos de dados e não apenas para os tipos primitivos. Por fim, à medida que método for sendo mais usado em diferentes contextos, muito provavelmente, sofrerá adaptações que originarão outros trabalhos.

Referências Bibliográficas

[Alencar, 1999] Alencar, F. M. R. (1999). **Mapeando a Modelagem Organizacional em Especificações Precisas**. Tese de Doutorado. Centro de Informática – Universidade Federal de Pernambuco.

[Andrade, 2007] Andrade, W. L. (2007). **Geração de casos de teste de interação para aplicações de celulares**. Dissertação de Mestrado. Universidade Federal de Campina Grande.

[Apfelbaum & Doyle, 1997] Apfelbaum, L.; Doyle, J. (1997). **Model-based testing**. Proceedings of the software Quality Week Conference.

[Bandeira, 2008] Bandeira, L. R. P. (2008). **Metodologia Baseada em Métricas de Teste para Indicação de Testes a Serem Melhorados**. Dissertação de Mestrado. Centro de Informática – Universidade Federal de Pernambuco.

[Beizer, 1990] Beizer, B. (1990). **Software Testing Techniques**. 2nd Ed., Van Nostrand Reinhold Company. New York.

[Berry & Kamsties, 2004] Berry, D. M.; Kamsties, E. (2004). **Ambiguity in Requirements Specification**. In Leite, J. C. S. P.; Doom, J. H. (eds), Perspectives On Software Requirements, 1st Ed., chapter 2, Kluwer Academic Publishers.

[Bertolino, 2007] Bertolino, A. (2007). **Software testing research: Achievements, challenges, dreams**. In FOSE '07: 2007 Future of Software Engineering, pages 85–103, Washington, DC, USA, 2007. IEEE Computer Society.

[Binder, 1999] Binder, R. (1999). **Testing Object Oriented Systems: Models, Patterns and Tools**. AddisonWesley.

[Borba *et al.*, 2007] Borba, P.; Torres, D.; Marques, R; Wetzel, L. (2007). **TaRGeT: Test and Requirements Generation Tool**. In Motorola's 2007 Innovation Conference (IC'2007), Software Expo Session, Lombard, Illinois, USA, October 5.

[Cheon & Avila, 2010] Cheon Y.; Avila, C. (2010). **Automating Java program testing using OCL and AspectJ**. In: ITNG 2010: 7th International Conference on Information Technology: New Generations, April 12-14, 2010, Las Vegas, NV. IEEE Computer Society.

[Clayberg & Rubel, 2006] Clayberg, E.; Rubel, D. (2006). **Eclipse: Building Commercial-Quality Plug-ins**. Addison-Wesley Professional, 2nd Ed.

[Corso, 2008] Corso, A. L. (2008). **Avaliação da Aplicabilidade e Eficácia de um Modelo de Maturidade de Teste**. Dissertação de Mestrado. Universidade Metodista de Piracicaba.

[Delamaro *et al.*, 2007] Delamaro, M. E.; Maldonado, J. C.; Jino, M. (2007). **Introdução ao Teste de Software**. Campus, Rio de Janeiro.

[Gamma & Back, 2004] Gamma, E.; Back, K. (2004). **Contributing to Eclipse: Principles, Patterns, and Plug-Ins**. Addison-Wesley Professional.

[Hartmann *et al*., 2000] Hartmann, J.; Imoberdorf, C.; Meisinger, M. (2000). **UML-Based integration testing**. ACM SIGSOFT Software Engineering Notes. Vol 25, 5th Ed.

[Henry, 1990] Henry, G. T. (1990). **Practical sampling**. Sage Publications, Newbury Park, CA.

[Hetzel, 1987] Hetzel, W. (1987). **Guia Completo ao Teste de Software**. Editora Campus. São Paulo.

[Inthrun, 2001] Inthurn, C. (2001). **Qualidade e teste de software**. Editora Visual Books, 1ª Edição.

[IEEE, 1990] **IEEE Standart Glossary of Software Engineering Terminology: IEEE Standart 610.12-1990**. ISBN 1-55937-067-X (1990).

[Jerome, 2002] Jerome, P. R. (2002). **Topics in Survey Sampling/Finite Population Sampling and inference: A Prediction Approach**. Journal of the American Statical Association, 97 (457) (March 2002): 357-358.

[Jones, 1990] Jones, C. B. (1990). **Systematic Software Development Using VDM**. Prentice-Hall, Inc., 2nd Ed.

[Kalton, 1990] Kalton G. (1983). **Introduction to Survey Sampling**. Thousand Oaks, CA: Sage.

[Kundu & Samanta, 2009] Kundu, D.; Samanta, D. (2009). **A novel approach to generate test cases from UML activity diagram**. In: Journal of Object Technologies, pp. 65-83.

[Larman, 2004] Larman, C. (2004). **Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development**. Prentice-Hall, 3rd Ed. New York.

[Leal, 2008] Leal, R. A. B. M. (2008). **Teste Funcional baseado em Modelos Gramaticais.** Dissertação de Mestrado. Pontifícia Universidade Católica do Rio de Janeiro - PUC-Rio.

[Lewis, 2000] Lewis, E. W. (2000). **Software Testing and Continuous Quality Improvement**. Auerbach Publications.

[Lévy, 1979] Lévy, A. (1979). **Basic Set Theory**. Springer-Verlag**.** New York.

[McAffer & Lemieux, 2005] McAffer, J.; Lemieux, J. M. (2005). **Eclipse Rich Client Platform: Design, Coding and Packaging Java Applications**. Addison-Wesley Professional.

[Miller & Strooper, 2001] Miller, T.; Strooper, P. (2001). **Animation Can Show Only the Presence of Errors, Never Their absence**. In: Proceedings of the Australian Software Engineering Conference - ASWEC, pp. 76-88, Camberra.

[Morgado, 2007] Morgado, G. (2007). **RAPDIS: Um Processo e um Ambiente MDA para o Desenvolvimento de Sistemas de Informação**. Dissertação de Mestrado. Instituto de Matemática – Núcleo de Computação Eletrônica – Universidade Federal do Rio de Janeiro.

[Myers, 2004] Myers, G. J. (2004). **The art of Software Testing**. John Wiley & Sons, 2nd Ed.

[OMG, 2003] OMG - Object Management Group (2003). **UML 2.0 OCL Specification**. Disponível em: <http://www.omg.org/cgi-bin/doc?ptc/2003-10-14>. Acesso em: 02 de Abril de 2010.

[OMG, 2005] OMG - Object Management Group (2005). **Unified Modeling Language (UML) Superstructure Specification, version 2.0**. Disponível em: <http://www.omg.org/cgi-bin/doc?formal/05-07-04>. Acesso em: 02 de Abril de 2010.

[Orozco *et al.*, 2009] Orozco, A. M.; Oliveira, K.; Oliveira, F. M.; Zorzo, A. F. (2009). **Derivação de Casos de Testes Funcionais: uma Abordagem baseada em Modelos UML**. In: Simpósio Brasileiro de Sistemas de Informação, Brasilia, DF, Brasil. SBSI 2009.

[Peters & Pedrycz, 2001] Peters, J. F.; Pedrycz, Wifold. (2001). **Engenharia de Software: Teoria e Prática**. Editora Campus.

[Pfleeger, 2004] Pfleeger, S. L. (2004). **Engenharia de Software – Teoria e Prática**. Editora Pearson, 2ª Edição.

[Prasanna *et al*., 2005] Prasanna, M.; Sivanadam, S. N.; Venkatesan, R.; Sundarrajan, R. (2005). **A survey on automatic test case generation**. Disponível em: <www.acadjournal.com/2005/v15/part6/p4/>. Acesso em: 01 de Junho de 2010.

[Pressman, 2006] Pressman, R. S. (2006). **Engenharia de Software**. McGraw-Hill, 6ª Edição.

[Richters & Gogolla, 1998] Richters, M.; Gogolla, M. (1998). **On Formalizing the UML Object Constraint Language OCL**. In Tok Wang Ling, Sudha

Ram, and Mong Li Lee, editors, Pro. 17th Int. Conf. Conceptual Modeling (ER'98), pp. 449-464. Springer, Berlin, LNCS 1507.

[Ryser & Glinz, 1999] Ryser, J.; Glinz, M. (1999). **A practical approach to validating and testing software systems using scenarios**. In: International Software Quality Week Europe (QWE'99), 3., 1999, Brussels. Proceedings. San Francisco: Software Research Institute.

[Schmitz & Silveira, 2009] Schmitz, E. A.; Silveira, D. S. (2000). **Desenvolvimento de Software Orientado a Objetos**. Editora Brasport, Rio de Janeiro, pp. 211.

[Silveira, 2009] Silveira, D. S. (2009). **Animare: Um Método de Validação dos Processos de Negócio Através da Animação**. Tese de Doutorado. COPPE - Universidade Federal do Rio de Janeiro.

[Spivey, 1998] Spivey, J. M. (1998). **The Z Notation: A Reference Manual**. Prentice Hall International, 2nd Ed.

[SWEBOK, 2001a] SWEBOK – Guide to the Software Engineering Body of Knowledge. (2001). **Software Testing Knowledge Area**. IEEE – Trial Version 1.00.

[SWEBOK, 2001b] SWEBOK – Guide to the Software Engineering Body of Knowledge. (2001). **Software Quality Knowledge Area**. IEEE – Trial Version 1.00.

[W3C, 2010] W3 Consortium. (2010). **Extensible Markup Language (XML)**. Disponível em: <http://www.w3c.org/XML.>. Acesso em: 25 abril.

[Warmer & Kleppe, 1999] Warmer, J. B.; Kleppe, A. G. (1999). **The Object Constraint Language: Precise Modeling with UML**. Addison-Wesley.

[Warmer & Kleppe, 2003] Warmer, J. B.; Kleppe, A. G. (2003). **The Object Constraint Language: Getting Your Models Ready for MDA**. The Addison-Wesley Object Technology Series. Addison-Wesley, 2nd Ed.

Apêndice A

Formato dos Arquivos Classes.XML e Definitions.XML em XML *Schema*

A.1 Classes.XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
<xs:element name="classes">
<xs:complexType>
<xs:sequence>
<xs:element ref="domainclasses.head"/>
<xs:element ref="classes.content"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="domainclasses.head">
<xs:complexType>
<xs:sequence>
<xs:element ref="project"/>
<xs:element ref="tool"/>
<xs:element ref="version"/>
<xs:element ref="date"/>
<xs:element ref="time"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="project" type="xs:string"/>
<xs:element name="tool" type="xs:NCName"/>
<xs:element name="version" type="xs:decimal"/>
<xs:element name="date" type="xs:string"/>
<xs:element name="time" type="xs:NMTOKEN"/>
<xs:element name="classes.content">
<xs:complexType>
<xs:sequence>
<xs:element ref="diagrams"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="diagrams">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" ref="diagram"/>
```

```
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="diagram">
<xs:complexType>
<xs:complexContent>
<xs:extension base="name">
<xs:sequence>
<xs:element ref="elements"/>
<xs:element ref="links"/>
</xs:sequence>
<xs:attribute name="id" use="required" type="xs:integer"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="elements">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" ref="element"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="element">
<xs:complexType>
<xs:complexContent>
<xs:extension base="name">
<xs:attribute name="MostraAtributo" use="required"
type="xs:integer"/>
<xs:attribute name="MostraOperacao" use="required"
type="xs:integer"/>
<xs:attribute name="idExternal" use="required"
type="xs:integer"/>
<xs:attribute name="idInternal" use="required"
type="xs:integer"/>
<xs:attribute name="removed" use="required" type="xs:boolean"/>
<xs:attribute name="type" use="required" type="xs:NCName"/>
<xs:attribute name="xBeginning" use="required"
type="xs:integer"/>
<xs:attribute name="xEnd" use="required" type="xs:integer"/>
<xs:attribute name="yBeginning" use="required"
type="xs:integer"/>
<xs:attribute name="yEnd" use="required" type="xs:integer"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="links">
<xs:complexType>
<xs:sequence>
<xs:element minOccurs="0" maxOccurs="unbounded" ref="link"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="link">
<xs:complexType>
<xs:complexContent>
<xs:extension base="name">
<xs:sequence>
<xs:element ref="breaks"/>
</xs:sequence>
<xs:attribute name="cardDestination" use="required"/>
<xs:attribute name="cardOrigin" use="required"/>
```

```
<xs:attribute name="id" use="required" type="xs:integer"/>
<xs:attribute name="idSource" use="required"
type="xs:integer"/>
<xs:attribute name="idTarget" use="required"
type="xs:integer"/>
<xs:attribute name="qtyBreaks" use="required"
type="xs:integer"/>
<xs:attribute name="removed" use="required" type="xs:boolean"/>
<xs:attribute name="type" use="required" type="xs:NCName"/>
<xs:attribute name="xCardDestination" use="required"
type="xs:integer"/>
<xs:attribute name="xCardOrigin" use="required"
type="xs:integer"/>
<xs:attribute name="xText" use="required" type="xs:integer"/>
<xs:attribute name="yCardDestination" use="required"
type="xs:integer"/>
<xs:attribute name="yCardOrigin" use="required"
type="xs:integer"/>
<xs:attribute name="yText" use="required" type="xs:integer"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="breaks">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" ref="break"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="break">
<xs:complexType>
<xs:attribute name="x" use="required" type="xs:integer"/>
<xs:attribute name="y" use="required" type="xs:integer"/>
</xs:complexType>
</xs:element>
<xs:complexType name="name">
<xs:sequence>
<xs:element ref="name"/>
</xs:sequence>
</xs:complexType>
<xs:element name="name" type="xs:string"/>
</xs:schema>
```

A.2 Definitions.XSD

```
<?xml version="1.0" encoding="UTF-8"?>
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
elementFormDefault="qualified">
<xs:element name="definitions">
<xs:complexType>
<xs:sequence>
<xs:element ref="definitions.head"/>
<xs:element ref="definitions.content"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="definitions.head">
<xs:complexType>
<xs:sequence>
<xs:element ref="project"/>
```

```
<xs:element ref="tool"/>
<xs:element ref="version"/>
<xs:element ref="date"/>
<xs:element ref="time"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="project" type="xs:string"/>
<xs:element name="tool" type="xs:NCName"/>
<xs:element name="version" type="xs:decimal"/>
<xs:element name="date" type="xs:string"/>
<xs:element name="time" type="xs:NMTOKEN"/>
<xs:element name="definitions.content">
<xs:complexType>
<xs:sequence>
<xs:element ref="activities"/>
<xs:element ref="signals"/>
<xs:element ref="objects"/>
<xs:element ref="actors"/>
<xs:element ref="usecases"/>
<xs:element ref="classes"/>
<xs:element ref="associations"/>
<xs:element ref="states"/>
<xs:element ref="transitions"/>
<xs:element ref="swimlanes"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="activities">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" ref="activity"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="activity">
<xs:complexType>
<xs:sequence>
<xs:element ref="description"/>
<xs:sequence minOccurs="0">
<xs:element ref="precondition"/>
<xs:element ref="postcondition"/>
</xs:sequence>
</xs:sequence>
<xs:attribute name="idElement" use="required" type="xs:integer"/>
</xs:complexType>
</xs:element>
<xs:element name="postcondition">
<xs:complexType/>
</xs:element>
<xs:element name="signals">
<xs:complexType/>
</xs:element>
<xs:element name="objects">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" ref="object"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="object">
<xs:complexType>
<xs:sequence>
```

```
<xs:element ref="description"/>
</xs:sequence>
<xs:attribute name="idElement" use="required" type="xs:integer"/>
</xs:complexType>
</xs:element>
<xs:element name="usecases">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" ref="usecase"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="usecase">
<xs:complexType>
<xs:sequence>
<xs:element ref="actors"/>
<xs:element ref="scenario"/>
<xs:element ref="precondition"/>
<xs:element ref="poscondition"/>
<xs:element ref="exception"/>
<xs:element ref="extension"/>
</xs:sequence>
<xs:attribute name="idElement" use="required" type="xs:integer"/>
</xs:complexType>
</xs:element>
<xs:element name="scenario" type="xs:string"/>
<xs:element name="poscondition" type="xs:string"/>
<xs:element name="exception" type="xs:string"/>
<xs:element name="extension">
<xs:complexType/>
</xs:element>
<xs:element name="classes">
<xs:complexType>
<xs:sequence>
<xs:element maxOccurs="unbounded" ref="class"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="class">
<xs:complexType>
<xs:sequence>
<xs:element ref="heranca"/>
<xs:element ref="classtype"/>
<xs:element ref="description"/>
<xs:element ref="atributes"/>
<xs:element ref="methods"/>
</xs:sequence>
<xs:attribute name="idElement" use="required" type="xs:integer"/>
</xs:complexType>
</xs:element>
<xs:element name="heranca" type="xs:NCName"/>
<xs:element name="classtype" type="xs:integer"/>
<xs:element name="atributes">
<xs:complexType>
<xs:choice>
<xs:element maxOccurs="unbounded" ref="atribute"/>
<xs:element maxOccurs="unbounded" ref="attribute"/>
</xs:choice>
</xs:complexType>
</xs:element>
<xs:element name="atribute">
<xs:complexType>
<xs:complexContent>
```

```
<xs:extension base="name">
<xs:attribute name="type" use="required" type="xs:NCName"/>
<xs:attribute name="visibility" use="required"
type="xs:integer"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="attribute">
<xs:complexType>
<xs:complexContent>
<xs:extension base="name">
<xs:attribute name="type" use="required" type="xs:NCName"/>
<xs:attribute name="visibility" use="required"
type="xs:integer"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="methods">
<xs:complexType>
<xs:sequence>
<xs:element minOccurs="0" maxOccurs="unbounded" ref="method"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="method">
<xs:complexType>
<xs:complexContent>
<xs:extension base="name">
<xs:sequence>
<xs:element ref="description"/>
<xs:element ref="parameters"/>
</xs:sequence>
<xs:attribute name="visibility" use="required"
type="xs:integer"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="parameters">
<xs:complexType>
<xs:sequence>
<xs:element minOccurs="0" maxOccurs="unbounded" ref="parameter"/>
</xs:sequence>
</xs:complexType>
</xs:element>
<xs:element name="parameter">
<xs:complexType>
<xs:complexContent>
<xs:extension base="name">
<xs:attribute name="mode" use="required" type="xs:NCName"/>
<xs:attribute name="type" use="required" type="xs:NCName"/>
</xs:extension>
</xs:complexContent>
</xs:complexType>
</xs:element>
<xs:element name="associations">
<xs:complexType>
<xs:sequence>
<xs:element ref="association"/>
</xs:sequence>
</xs:complexType>
```

```
</xs:element>
<xs:element name="association">
<xs:complexType>
<xs:sequence>
<xs:element ref="role"/>
<xs:element ref="roleClass1"/>
<xs:element ref="cardinalityClass1"/>
<xs:element ref="roleClass2"/>
<xs:element ref="cardinalityClass2"/>
<xs:element ref="notes"/>
</xs:sequence>
<xs:attribute name="idLink" use="required" type="xs:integer"/>
</xs:complexType>
</xs:element>
<xs:element name="role" type="xs:NCName"/>
<xs:element name="roleClass1">
<xs:complexType/>
</xs:element>
<xs:element name="cardinalityClass1" type="xs:NMTOKEN"/>
<xs:element name="roleClass2">
<xs:complexType/>
</xs:element>
<xs:element name="cardinalityClass2" type="xs:NMTOKEN"/>
<xs:element name="notes">
<xs:complexType/>
</xs:element>
<xs:element name="states">
<xs:complexType/>
</xs:element>
<xs:element name="transitions">
<xs:complexType/>
</xs:element>
<xs:element name="swimlanes">
<xs:complexType/>
</xs:element>
<xs:element name="description" type="xs:string"/>
<xs:element name="precondition" type="xs:string"/>
<xs:element name="actors" type="xs:string"/>
<xs:complexType name="name">
<xs:sequence>
<xs:element ref="name"/>
</xs:sequence>
</xs:complexType>
<xs:element name="name" type="xs:string"/>
</xs:schema>
```

Apêndice B

Artefatos usados no Estudo de Caso

B.1 Conteúdo dos Arquivos XML do DC

B.1.1 Classes.xml

```
<?xml version="1.0" encoding="utf-8"?>
<classes>
   <domainclasses.head>
      <project>Project Name</project>
      <tool>RAPDIS</tool>
      <version>1.0</version>
      <date>08/06/2010</date>
      <time>21:07</time>
   </domainclasses.head>
   <classes.content>
      <diagrams>
         <diagram id="70">
            <name>TuAlugaDC</name>
            <elements>
               <element type="TObjClasse" idExternal="71" idInternal="72"
xBeginning="83" yBeginning="40" xEnd="216" yEnd="202" removed="false"
MostraAtributo="1" MostraOperacao="1">
                  <name>Carro</name>
               </element>
               <element type="TObjClasse" idExternal="73" idInternal="74"
xBeginning="402" yBeginning="90" xEnd="510" yEnd="154" removed="false"
MostraAtributo="1" MostraOperacao="1">
                  <name>Grupo</name>
               </element>
               <element type="TObjClasse" idExternal="75" idInternal="76"
xBeginning="242" yBeginning="200" xEnd="380" yEnd="362" removed="false"
MostraAtributo="1" MostraOperacao="1">
                  <name>Cliente</name>
               </element>
               <element type="TObjClasse" idExternal="77" idInternal="78"
xBeginning="531" yBeginning="222" xEnd="681" yEnd="322" removed="false"
MostraAtributo="1" MostraOperacao="1">
                  <name>Reserva</name>
               </element>
               <element type="TObjClasse" idExternal="79" idInternal="80"
xBeginning="19" yBeginning="322" xEnd="180" yEnd="467" removed="false"
MostraAtributo="1" MostraOperacao="1">
                  <name>Aluguel</name>
               </element>
```

```
               <element type="TObjClasse" idExternal="81" idInternal="82"
xBeginning="265" yBeginning="454" xEnd="398" yEnd="552" removed="false"
MostraAtributo="1" MostraOperacao="1">
                  <name>Motorista</name>
               </element>
               <element type="TObjClasse" idExternal="83" idInternal="84"
xBeginning="486" yBeginning="343" xEnd="633" yEnd="443" removed="false"
MostraAtributo="1" MostraOperacao="1">
                  <name>CartaoCredito</name>
               </element>
            </elements>
            <links>
               <link type="TObjAgregacao" id="85" idSource="72" idTarget="74"
qtyBreaks="2" xText="285" yText="109" removed="false" cardOrigin="*"
cardDestination="" xCardOrigin="246" yCardOrigin="121" xCardDestination="0"
yCardDestination="0">
                  <name>
                  </name>
                  <breaks>
                     <break x="215" y="121"/>
                     <break x="402" y="121"/>
                  </breaks>
               </link>
               <link type="TObjAssociacao" id="86" idSource="72" idTarget="76"
qtyBreaks="3" xText="195" yText="278" removed="false" cardOrigin="*"
cardDestination="*" xCardOrigin="149" yCardOrigin="221" xCardDestination="219"
yCardDestination="281">
                  <name>
                  </name>
                  <breaks>
                     <break x="149" y="202"/>
                     <break x="149" y="281"/>
                     <break x="242" y="281"/>
                  </breaks>
               </link>
               <link type="TObjAssociacao" id="87" idSource="76" idTarget="74"
qtyBreaks="3" xText="447" yText="217" removed="false" cardOrigin="*"
cardDestination="*" xCardOrigin="398" yCardOrigin="281" xCardDestination="455"
yCardDestination="185">
                  <name>
                  </name>
                  <breaks>
                     <break x="380" y="281"/>
                     <break x="455" y="281"/>
                     <break x="455" y="154"/>
                  </breaks>
               </link>
               <link type="TObjLinkAssociativa" id="88" idSource="78"
idTarget="87" qtyBreaks="2" xText="492" yText="230" removed="false"
cardOrigin="" cardDestination="" xCardOrigin="0" yCardOrigin="0"
xCardDestination="0" yCardDestination="0">
                  <name>
                  </name>
                  <breaks>
                     <break x="531" y="244"/>
                     <break x="454" y="216"/>
                  </breaks>
               </link>
               <link type="TObjLinkAssociativa" id="89" idSource="80"
idTarget="86" qtyBreaks="2" xText="136" yText="275" removed="false"
cardOrigin="" cardDestination="" xCardOrigin="0" yCardOrigin="0"
xCardDestination="0" yCardDestination="0">
                  <name>
```

```
                  </name>
                  <breaks>
                     <break x="122" y="322"/>
                     <break x="150" y="240"/>
                  </breaks>
               </link>
               <link type="TObjAgregacao" id="91" idSource="84" idTarget="76"
qtyBreaks="3" xText="305" yText="380" removed="false" cardOrigin="1..*"
cardDestination="" xCardOrigin="445" yCardOrigin="398" xCardDestination="0"
yCardDestination="0">
                  <name>
                  </name>
                  <breaks>
                     <break x="486" y="400"/>
                     <break x="310" y="419"/>
                     <break x="310" y="362"/>
                  </breaks>
               </link>
               <link type="TObjAssociacao" id="94" idSource="84" idTarget="80"
qtyBreaks="2" xText="335" yText="396" removed="false" cardOrigin=""
cardDestination="" xCardOrigin="0" yCardOrigin="0" xCardDestination="0"
yCardDestination="0">
                  <name>
                  </name>
                  <breaks>
                     <break x="486" y="393"/>
                     <break x="179" y="393"/>
                  </breaks>
               </link>
               <link type="TObjAgregacao" id="95" idSource="82" idTarget="80"
qtyBreaks="3" xText="94" yText="488" removed="false" cardOrigin="1..1"
cardDestination="" xCardOrigin="223" yCardOrigin="502" xCardDestination="0"
yCardDestination="0">
                  <name>
                  </name>
                  <breaks>
                     <break x="265" y="502"/>
                     <break x="97" y="502"/>
                     <break x="97" y="466"/>
                  </breaks>
               </link>
            </links>
         </diagram>
      </diagrams>
   </classes.content>
</classes>
```

B.1.2 Definitions.xml

```
<?xml version="1.0" encoding="utf-8"?>
<definitions>
   <definitions.head>
      <project>Project Name</project>
      <tool>RAPDIS</tool>
      <version>1.0</version>
      <date>08/06/2010</date>
      <time>21:07</time>
   </definitions.head>
   <definitions.content>
      <activities/>
      <signals/>
      <objects/>
      <actors/>
```

```
<usecases/>
<classes>
   <class idElement="71">
      <heranca>TObject</heranca>
      <classtype>0</classtype>
      <description>
      </description>
      <atributes>
         <attribute visibility="1" type="STRING">
            <name>placa</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>chassis</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>km</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>modelo</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>fabricante</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>cor</name>
         </attribute>
         <attribute visibility="1" type="INTEGER">
            <name>ano</name>
         </attribute>
         <attribute visibility="1" type="BYTE">
            <name>Situacao</name>
         </attribute>
      </atributes>
      <methods/>
   </class>
   <class idElement="73">
      <heranca>TObject</heranca>
      <classtype>0</classtype>
      <description>
      </description>
      <atributes>
         <attribute visibility="1" type="STRING">
            <name>nome</name>
         </attribute>
         <attribute visibility="1" type="REAL">
            <name>preco</name>
         </attribute>
      </atributes>
      <methods/>
   </class>
   <class idElement="75">
      <heranca>TObject</heranca>
      <classtype>0</classtype>
      <description>
      </description>
      <atributes>
         <attribute visibility="1" type="STRING">
            <name>nome</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>cpf</name>
         </attribute>
         <attribute visibility="1" type="STRING">
```

```
            <name>cnh</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>telefone</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>endereco</name>
         </attribute>
         <attribute visibility="1" type="INTEGER">
            <name>tempoCnh</name>
         </attribute>
         <attribute visibility="1" type="BYTE">
            <name>Tipo</name>
         </attribute>
         <attribute visibility="1" type="INTEGER">
            <name>idade</name>
         </attribute>
      </atributes>
      <methods/>
   </class>
   <class idElement="77">
      <heranca>TObject</heranca>
      <classtype>0</classtype>
      <description>
      </description>
      <atributes>
         <attribute visibility="1" type="INTEGER">
            <name>numReserva</name>
         </attribute>
         <attribute visibility="1" type="DATE">
            <name>dataReserva</name>
         </attribute>
         <attribute visibility="1" type="DATE">
            <name>dataAluguel</name>
         </attribute>
         <attribute visibility="1" type="DATE">
            <name>dataDevolucao</name>
         </attribute>
      </atributes>
      <methods/>
   </class>
   <class idElement="79">
      <heranca>TObject</heranca>
      <classtype>0</classtype>
      <description>
      </description>
      <atributes>
         <attribute visibility="1" type="INTEGER">
            <name>numAluguel</name>
         </attribute>
         <attribute visibility="1" type="DATE">
            <name>dataAluguel</name>
         </attribute>
         <attribute visibility="1" type="DATE">
            <name>dataDevolucao</name>
         </attribute>
         <attribute visibility="1" type="BOOLEAN">
            <name>vistoriado</name>
         </attribute>
         <attribute visibility="1" type="BOOLEAN">
            <name>assinado</name>
         </attribute>
         <attribute visibility="1" type="STRING">
```

```
            <name>BloqueioCartao</name>
         </attribute>
         <attribute visibility="1" type="BYTE">
            <name>Pagamento</name>
         </attribute>
      </atributes>
      <methods/>
   </class>
   <class idElement="81">
      <heranca>TObject</heranca>
      <classtype>0</classtype>
      <description>
      </description>
      <atributes>
         <attribute visibility="1" type="STRING">
            <name>nome</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>cpf</name>
         </attribute>
         <attribute visibility="1" type="INTEGER">
            <name>tempoCnh</name>
         </attribute>
         <attribute visibility="1" type="INTEGER">
            <name>idade</name>
         </attribute>
      </atributes>
      <methods/>
   </class>
   <class idElement="83">
      <heranca>TObject</heranca>
      <classtype>0</classtype>
      <description>
      </description>
      <atributes>
         <attribute visibility="1" type="STRING">
            <name>numCartao</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>bandeira</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>validade</name>
         </attribute>
         <attribute visibility="1" type="STRING">
            <name>codSeguranca</name>
         </attribute>
      </atributes>
      <methods/>
   </class>
</classes>
<associations>
   <association idLink="85">
      <role>
      </role>
      <roleClass1>
      </roleClass1>
      <cardinalityClass1>*</cardinalityClass1>
      <roleClass2>
      </roleClass2>
      <cardinalityClass2>
      </cardinalityClass2>
      <notes>
```

```
            </notes>
         </association>
         <association idLink="86">
            <role>
            </role>
            <roleClass1>
            </roleClass1>
            <cardinalityClass1>*</cardinalityClass1>
            <roleClass2>
            </roleClass2>
            <cardinalityClass2>*</cardinalityClass2>
            <notes>
            </notes>
         </association>
         <association idLink="87">
            <role>
            </role>
            <roleClass1>
            </roleClass1>
            <cardinalityClass1>*</cardinalityClass1>
            <roleClass2>
            </roleClass2>
            <cardinalityClass2>*</cardinalityClass2>
            <notes>
            </notes>
         </association>
         <association idLink="91">
            <role>
            </role>
            <roleClass1>
            </roleClass1>
            <cardinalityClass1>1..*</cardinalityClass1>
            <roleClass2>
            </roleClass2>
            <cardinalityClass2>
            </cardinalityClass2>
            <notes>
            </notes>
         </association>
         <association idLink="95">
            <role>
            </role>
            <roleClass1>
            </roleClass1>
            <cardinalityClass1>1..1</cardinalityClass1>
            <roleClass2>
            </roleClass2>
            <cardinalityClass2>
            </cardinalityClass2>
            <notes>
            </notes>
         </association>
      </associations>
      <states/>
      <transitions/>
      <swimlanes/>
   </definitions.content>
</definitions>
</definitions>
```

B.2 Regras de Negócio Expressas em OCL

```
001. context AlugarCarro::D1(
002. paramReserva = true
003. )
004.
005. context AlugarCarro::VerificarReserva(
006. pre: reserva->select(r | r.NumReserva = paramNumReserva)->size() = 1
007. post: r.dataAluguel = Data::now
008. )
009.
010. context AlugarCarro::AtualizarDadosdoCliente(
011. pre: cliente->select(cli | cli.cpf = paramCPF)->size() = 1
012. action:
013. cli.telefone := paramTelefone;
014. cli.endereco := paramEndereco;
015. cli.tempoCnh := paramTempoCnh;
016. cli.idade := paramIdade;
017. )
018.
019. context AlugarCarro::VerificarDisponibilidade(
020. pre: paramDataDevolucao > Data::now
021. pre: paramGrupo.size() > 0
022. post: grupo->select(g | g.nome = paramGrupo)->size()=1
023. )
024.
025. context AlugarCarro::D2(
026. carro->select(c | c.grupo.nome = paramGrupo)->size() >
( aluguel->select(a | a.dataAluguel >= Data::now and
a.dataDevolucao <= paramDataDevolucao and
a.carro.grupo.nome = paramGrupo)->size() +
aluguel->select(a | a.dataAluguel <= paramDataDevolucao and
a.dataDevolucao >= paramDataDevolucao and
a.carro.grupo.nome = paramGrupo)->size() +
reserva->select(r | r.dataAluguel >= Data::now and
r.dataDevolucao <= paramDataDevolucao and
r.grupo.nome = paramGrupo)->size() +
reserva->select(r | r.dataAluguel <= paramDataDevolucao and
r.dataDevolucao >= paramDataDevolucao and
r.grupo.nome = paramGrupo)->size() )
027. )
028.
029. context AlugarCarro::VerificarCliente(
030. pre: paramCPF.size() > 0
031. pre: paramNome.size() > 0
032. )
033.
034. context AlugarCarro::D3(
035. cliente->select(c | c.cpf = paramCPF)->size() = 1
036. )
037.
038. context AlugarCarro::RegistrarCliente(
039. pre: paramNome.size() > 0
040. pre: paramCPF.size() > 0
041. action:
042. var cli:= cliente.create(paramNome, paramCPF, paramCNH,
paramTelefone, paramEndereco,
paramTempoCNH, TipoCliente::NORMAL,
paramIdade);
043. post: cliente->one(cli | cli.oclisnew())
044. )
045.
```

```
046. context AlugarCarro::SelecionarCarro(
047. pre: paramGrupo.size() > 0
048. post: carro->select(c | c.situacao = TipoSituacao::DISPONIVEL and
c.grupo.nome = paramGrupo)->size() > 0
049. )
050.
051. context AlugarCarro::D4(
052. paramOutroMotorista = true
053. )
054.
055. context AlugarCarro::D5(
056. motorista->select(m | m.cpf = paramMotoristaCPF)->size() = 1
057. )
058.
059. context AlugarCarro::AtualizarDadosdoMotorista(
060. action:
061. m.Idade := paramMotoristaIdade;
062. m.TempoCNH := paramMotoristaTempoCNH;
063. )
064.
065. context AlugarCarro::RegistrarMotorista(
066. pre: paramMotoristaNome.size() > 0
067. pre: paramMotoristaCPF.size() > 0
068. pre: paramMotoristaCNH.size() > 0
069. pre: paramMotoristaIdade > 21
070. pre: paramMotoristaTempoCNH > 2
071. action:
072. var m:= motorista.create(paramMotoristaNome, paramMotoristaCPF,
paramMotoristaCNH, paramMotoristaTempoCNH,
paramMotoristaIdade);
073. post: motorista->one(m | m.oclisnew())
074. )
075.
076. context AlugarCarro::InformarCartaodeCredito(
077. pre: paramNumCartao.size() >0
078. pre: paramBandeira.size() > 0
079. pre: paramCodSeguranca.size() > 0
080. pre: paramDataValidade > paramDataDevolucao
081. action:
082. var cc:= cartaocredito.create(paramNumCartao, paramBandeira,
paramDataValidade, paramCodSeguranca);
083. cc.cliente := cli;
084. post: cartaocredito->one(cc | cc.oclisnew())
085. )
086.
087. context AlugarCarro::GerarContrato(
088. pre: paramBloqueioCartao.size() > 0
089. pre: (m->notEmpty()) or
(m->isEmpty() and cli.idade > 21 and cli.tempoCnh > 2)
090. action:
091. var a := aluguel.create(paramNumAluguel, paramDataRetirada,
paramDataDevolucao, Boolean::FALSE,
Boolean::FALSE, paramBloqueioCartao,
Boolean::FALSE);
092. a.carro := c;
093. a.cliente := cli;
094. a.motorista := m;
095. post: aluguel->one(a | a.oclisnew())
096. )
097.
098. context AlugarCarro::ValidarContrato(
099. action:
100. a.assinado := paramAssinado;
```

```
101. c.situacao := TipoSituacao::EMPRESTADO;
102. delete (r);
103. )
104.
105. context AlugarCarro::PrepararCarro(
106. pre: paramCarroPreparado = true
108. )
109.
110. context AlugarCarro::RetirarCarro(
111. pre: paramAssinado = true
112. action:
113. a.vistoriado := paramVistoriado;
114. )
```

Livros Grátis

(http://www.livrosgratis.com.br)

Milhares de Livros para Download:

Baixar livros de Literatura
Baixar livros de Literatura de Cordel
Baixar livros de Literatura Infantil
Baixar livros de Matemática
Baixar livros de Medicina
Baixar livros de Medicina Veterinária
Baixar livros de Meio Ambiente
Baixar livros de Meteorologia
Baixar Monografias e TCC
Baixar livros Multidisciplinar
Baixar livros de Música
Baixar livros de Psicologia
Baixar livros de Química
Baixar livros de Saúde Coletiva
Baixar livros de Serviço Social
Baixar livros de Sociologia
Baixar livros de Teologia
Baixar livros de Trabalho
Baixar livros de Turismo