UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM COMPUTAÇÃO

Diretrizes e Critérios de Cobertura de Teste a partir de Especificações UML

por

EMERSON AUGUSTO MIOTTO CORAZZA

Dissertação submetida à avaliação,
como requisito parcial, para a obtenção do grau de
Mestre em Ciência da Computação

Profa. Dra. Ana Maria de Alencar Price
Orientadora

Porto Alegre, maio de 2001

CIP – CATALOGAÇÃO NA PUBLICAÇÃO

Corazza, Emerson Augusto Miotto
Critérios de Cobertura de Teste a partir de Especificações UML / por Emerson Augusto Miotto Corazza. – Porto Alegre: PPGC da UFRGS, 2001.
138p.:il.

Dissertação (mestrado) – Universidade Federal do Rio Grande do Sul. Programa de Pós-Graduação em Computação, Porto Alegre, BR-RS, 2001. Orientadora: Price, Ana Maria de Alencar.

1. Teste de Software. 2. UML. 3.Critérios de teste. 4.Diretrizes de teste. I. Price, Ana Maria de Alencar. II. Título.

UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
Reitora: Profa. Wrana Maria Panizzi
Pró-Reitor de Ensino: Prof. José Carlos Ferraz Hennemann
Pró-Reitor Adjunto de Pós-Graduação: Prof. Philippe Olivier Alexandre Navaux
Diretor do Instituto de Informática: Prof. Philippe Olivier Alexandre Navaux
Coordenador do PPGC: Prof. Carlos Alberto Heuser
Bibliotecária-Chefe do Instituto de Informática: Beatriz Regina Bastos Haro

Dedicatória

Aos meus pais **Lourdes** e **Deolindo** que sempre acreditaram na minha capacidade.

A **Viviane** meu grande amor e ao **João Vitor** que com sua simplicidade e angelical harmonia apaziguou os momentos difíceis.

Agradecimentos

Ao DEUS Trino e MARIA, companheiros de todos os momentos, força, luz, inspiração e caminho.

À UNIDERP, pela oportunidade e apoio financeiro, sem os quais este trabalho tornar-se-ia inviável.

À Profa Ana Maria de Alencar Price, pelo apoio, compreensão, dedicação, paciência, incentivo e amizade.

Aos meus irmãos, Karine e Cleverton, que, com seus incentivos, serviram de apoio nesta caminhada.

A todos os professores da UFRGS que compartilharam seus conhecimentos e idéias.

Aos amigos do mestrado remoto com os quais compartilhamos bons e maus momentos, principalmente: Ana Fernanda, Cláudio, Edilene, Lincoln, Luiz Alberto (Betinho), Luiz Carlos, Melissa, Roberto e Terezinha.

À minha família na UFRGS – Adriana, Beatriz, Fabiana, Grace, Henrique, Ida, Jane e Zita – amizade, força, ajuda, abrigo e carinho.

Aos amigos e irmãos da Pensão "Tiazinha Eliane" – Eliane e Roberto

Aos meus irmãos de coração Ana Claudia de Oliveira Pedro Ândreo, Fernando Nery Sizilio, Glaucia Regina Medeiros Azambuja Sizilio, Paulo Henrique Cayres e Robson Soares Silva – amizade, incentivo, apoio, companheirismo e exemplo.

Aos Funcionários da UFRGS – pessoal da coordenação do PPGC, dos laboratórios, etc.

A todos os meus alunos que contribuíram com seu apoio e compreensão.

Demais amigos, amigas, colegas e familiares que de qualquer maneira, colaboraram com o desenvolvimento deste trabalho.

Sumário

Lista de Abreviaturas..... 8
Lista de Figuras..... 9
Lista de Tabelas..... 11
Resumo..... 13
Abstract..... 14
1 Introdução..... 15
1.1 Proposta de Trabalho..... 17
1.2 Organização do trabalho..... 17
2 Critérios de Cobertura e Técnicas de Teste de Software.... 19
2.1 Teste de Software Procedimental..... 19
2.1.1 Teste Estrutural..... 19
2.1.1.1 Critérios de Cobertura Baseados em Fluxo de Controle..... 20
2.1.1.2 Critérios de Cobertura Baseados em Fluxo de Dados..... 24
2.1.2 Teste Funcional..... 29
2.1.2.1 Método Particionamento por Equivalência..... 29
2.1.2.2 Método Análise do Valor Limite – BVA..... 30
2.2 Teste de Software Orientado a Objetos..... 31
2.2.1 Planejamento para o Teste de Software Orientado a Objetos..... 33
2.3 Níveis de Teste de Software OO..... 35
2.3.1 Teste Funcional de Classes..... 36
2.3.2 Teste Estrutural de Classes..... 36
2.3.3 Teste de Interação entre Classes..... 36
2.3.4 Teste Baseado em Estados..... 38
2.3.4.1 Critérios de Cobertura Baseados em Estados..... 41
2.3.5 Teste de Subclasses..... 42
2.3.6 Teste de Cluster..... 43
2.4 Teste Baseado em Especificações de Software..... 44
2.4.1 Proposta de Colanzi e Masieiro para o Desenvolvimento e Teste de Software para UML..... 44
2.4.2 Proposta de Binder baseada nos Cenários de um Diagrama de Seqüência.. 46
2.4.3 Proposta de Binder baseada na verificação da Associação entre Classes..... 48
3 Critérios de Cobertura e Diretrizes de Teste para Diagramas UML..... 52
3.1 Critérios de Cobertura e Diretrizes para o teste baseado no Diagrama de Classes..... 53
3.1.1 Derivando informações para a verificação dos Atributos da Classe..... 56
3.1.1.1 Cobertura de Teste de Condições e Combinação de Condições..... 58
3.1.2 Informações de Teste a partir dos Métodos da Classe..... 61
3.1.3 Informações para o Teste baseadas nos Relacionamentos da Classe..... 64

3.2.3.1 Analisando o Relacionamento do Tipo Generalização..... 64
3.1.3.2 Analisando o Relacionamento do tipo Associação..... 65
3.1.3.4 Analisando a Multiplicidade do Relacionamento..... 67
3.2 Critérios de Cobertura baseados em Diagrama de Seqüência..... 69
3.2.1 Implementando o teste baseado no diagrama de seqüência..... 74
3.3 Critérios de Cobertura baseados em Diagrama de Colaboração.. 77
3.3.1 Implementando o teste baseado no diagrama de colaboração..... 78
3.4 Critérios de Cobertura baseados em Diagrama de Estado..... 79
3.4.1 Implementando o teste baseado no diagrama de estados..... 82
4 Estudo de Caso: Módulo de controle para emissão de ordens de serviço em aeronaves..... 87
4.1 Analisando o Diagrama de Classes do Estudo de Caso..... 87
4.1.1 Cobertura de Restrições sobre Atributos..... 90
4.1.2 Cobertura de Restrições sobre Métodos..... 92
4.1.3 Cobertura de Teste para a Herança de classes..... 93
4.1.3 Cobertura de Restrição sobre Associações..... 95
4.1.4 Cobertura da Associação do Tipo Agregação Composta..... 96
4.1.5 Cobertura da Multiplicidade do Relacionamento..... 97
4.2 Analisando os Diagramas de Seqüência do Estudo de Caso..... 100
4.3 Analisando os Diagramas de Colaboração do Estudo de Caso..... 105
4.3.5 Analisando os Diagramas de Estados do Estudo de Caso..... 110
5 Conclusão..... 114
5.1 Trabalhos Futuros..... 116
Anexo 1 Proposta de Modelo Conceitual para uma Ferramenta de Geração de Caminhos de teste..... 117
Anexo 2 Diagrama de Classes Módulo Emissão de Ordens de Serviço..... 118
Anexo 3 Diagrama de Casos de Uso do Módulo Emissão de Ordens de Serviço..... 119
Anexo 4 Diagrama de Seqüência Manter Ordem de Serviço do Estudo de Caso..... 120
Anexo 5 Diagrama de Seqüência Fechar Ordem de Serviço do Estudo de Caso..... 121
Anexo 6 Diagrama de Seqüência Manter Produtos da OS do Estudo de Caso..... 122
Anexo 7 Diagrama de Seqüência Manter Aeronave do Estudo de Caso..... 123
Anexo 8 Diagrama de Seqüência Lançar Serviços Realizados na OS do Estudo de Caso..... 124
Anexo 9 Diagrama de Seqüência Alocar Mecânico nos Serviços da OS do Estudo de Caso..... 125

Anexo 10 Diagrama de Colaboração Manter Ordem de Serviço do Estudo de Caso..... 126
Anexo 11 Diagrama de Colaboração Fechar Ordem de Serviço do Estudo de Caso..... 127
Anexo 12 Diagrama de Colaboração Manter Produtos da OS do Estudo de Caso..... 128
Anexo 13 Diagrama de Colaboração Manter Aeronave do Estudo de Caso..... 129
Anexo 14 Diagrama de Colaboração Lançar Serviços Realizados na OS do Estudo de Caso..... 130
Anexo 15 Diagrama de Colaboração Alocar Mecânico nos Serviços da OS do Estudo de Caso..... 131
Anexo 16 Diagrama de Estados da Classe Ordem de Serviço do Estudo de Caso..... 132
Anexo 17 Diagrama de Estados da Classe Proprietário do Estudo de Caso..... 133
Bibliografia..... 134

Lista de Abreviaturas

UML	Linguagem de Modelagem Unificada (*Unified Modeling Language*)
GFC	Grafo de Fluxo de Controle
DC	Diagrama de Chamadas
BDR	Banco de Dados Relacional
SQL	Linguagem de Consulta Estruturada (*Structured Query Language*)
BVA	Análise de Valor Limite (*Boundary Value Analyze*)
OO	Orientação a objeto

Lista de Figuras

FIGURA 1.1 - O teste e as fases do desenvolvimento do software. 15
FIGURA 2.1 – Principais estruturas de um GFC 21
FIGURA 2.2 – Exemplo de conversão de um programa em GFC.....21
FIGURA 2.3 – Exemplo de um diagrama de chamadas (DC) 22
FIGURA 2.4 – Exemplo de um Grafo de Fluxo de Controle 27
FIGURA 2.5 – Exemplo de herança entre classes. 32
FIGURA 2.6 – Exemplo de um Grafo de Classe para o teste de interação intra-classe. 37
FIGURA 2.7 – Exemplo de um Grafo de Classe para o teste de interação inter-classe. 37
FIGURA 2.8 – Modelo de Conversão da Máquina de Estados para Árvore de Transições.....39
FIGURA 2.9 – Especificação da Máquina de Estados da classe Conta Telefônica.....40
FIGURA 2.10 – Árvore de transições da Máquina de Estados da Figura 2.9. 41
FIGURA 2.11 – Exemplo da Interface do Cluster 43
FIGURA 2.12 – Modelos gerados pela abordagem de teste de especificação 45
FIGURA 2.13 – Modelo de diagrama de seqüência. 46
FIGURA 2.14 – Modelo de gráfico de fluxo derivado do diagrama de seqüência.....47
FIGURA 2.15 – Multiplicidade da relação Pessoa-possui-Cachorro. 49
FIGURA 2.16 – Multiplicidade da relação Pessoa-possui-Cachorro na cidade de Kaynaine.....49
FIGURA 3.1 – Exemplo de Diagrama de Classes Representando a Locação de Veículos em uma Locadora.....55
FIGURA 3.2 – Elementos utilizados na verificação dos atributos da classe. 57
FIGURA 3.3 – Elementos envolvidos com a verificação dos métodos da classe.....63
FIGURA 3.4 – Exemplo de associação do tipo agregação por composição.....66
FIGURA 3.5 – Exemplo de restrição de associação. 66
FIGURA 3.6 – Exemplo de multiplicidade com valor máximo especificado 68
FIGURA 3.7 – Diagrama de Seqüência do Caso de Uso Manter Contrato de Locação. 70
FIGURA 3.8 – Representação do Critério Todas-C-Mensagens em um GFC. 71
FIGURA 3.9 – Representação do Critério Todas-SCS-Mensagens em um GFC.....72
FIGURA 3.10– Diagrama de Seqüência modificado após aplicação do critério Todas-SCS-Mensagens. 73
FIGURA 3.11 – Exemplo conversão do GFC em Matriz de Grafo 74
FIGURA 3.12 – Matriz de Caminho do diagrama de seqüência da Figura 3.10. 75
FIGURA 3.13 – Diagrama de Colaboração do Caso de Uso Manter Contrato de Locação. 77
FIGURA 3.14 – Diagrama de estados do caso de uso Locar Veículo. 79
FIGURA 3.15 – Diagrama de Estados para um objeto da classe Veículo do sistema de Locação de Veículos.....81
FIGURA 3.16 – Modelo conceitual para armazenamento dos dados do diagrama de estados. 83
FIGURA 3.17 – Árvore de transições do diagrama de estados da Figura 3.15. 84
FIGURA 3.18 – Caminhos de teste gerados pela árvore de transições da Figura 3.17.....84
FIGURA 4.1 – Diagrama de Classes do estudo de caso. 89

FIGURA 4.2 – Classe Aeronave com seus atributos e métodos.....90
FIGURA 4.3 – Classes Mecânico, Credenciado e Aprendiz com relacionamento do tipo generalização.....93
FIGURA 4.4 – Associação simples com restrição entre as classes Aeronave e Ordem de Serviço.....95
FIGURA 4.5 – Associação de agregação por composição entre as classes Aeronave, Hélice Aeronave e MotorAeronave.....96
FIGURA 4.6 – Multiplicidades do diagrama de classes do estudo de caso.....97
FIGURA 4.7 – Diagrama de casos de uso para o módulo de Controle de emissão de Ordens de Serviço. 100
FIGURA 4.8 – Diagrama de seqüência para o caso de uso “Manter Ordem de Serviço”.....101
FIGURA 4.9 – Diagrama de seqüência para o caso de uso “Fechar Ordem de Serviço”.....103
FIGURA 4.10 – Diagrama de colaboração para o caso de uso “Manter Aeronave”. .106
FIGURA 4.11 – Diagrama de colaboração para o caso de uso “Manter Ordem de Serviço”.....108
FIGURA 4.12 – Diagrama de estados da classe Proprietário. 111
FIGURA 4.13 – Árvore de transições do diagrama de estados da classe Proprietário.....111
FIGURA 4.14 – Diagrama de estados da classe Ordem de Serviço. 112
FIGURA 4.15 – Árvore de transições do Diagrama de estados da classe Ordem de Serviço.....112

Lista de Tabelas

TABELA 2.1 – Relação entre os critérios de Myers e Silva 22
TABELA 2.2 – Casos de teste para o caminhos derivados da árvore de transições da Figura 2.10.....41
TABELA 2.3 - Casos de teste para os caminhos do grafo da Figura 2.12.....48
TABELA 2.4 - Teste da Associação Pessoa-possui-Cachorro. 50
TABELA 2.5 - Teste da Associação Cachorro-pertence á-Pessoa. 50
TABELA 3.1 – Exemplo de ficha de avaliação da condição simples da restrição do atributo dtNascimento. 59
TABELA 3.2 – Exemplo de ficha de avaliação de condição composta para a restrição do atributo dtNascimento. 60
TABELA 3.3 – Exemplo de combinação para teste de condições utilizando operador lógico “e”. 60
TABELA 3.4 – Exemplo de combinação para teste de condições utilizando operador lógico “ou”. 61
TABELA 3.5 – Exemplo de casos de teste para objetos da classe Pessoa em relação à classe Locação 68
TABELA 3.6 – Exemplo de casos de teste para objetos da classe Locação em relação à classe Pessoa 68
TABELA 3.7 – Exemplo de casos de teste para objetos da classe Pessoa em relação à classe Locação 69
TABELA 3.8 – Caminhos gerados com base nos critérios de cobertura definidos para o diagrama de seqüência.....76
TABELA 3.9 – Informações armazenadas do diagrama de estados da Figura 3.15.....83
TABELA 3.10 – Relação de Critérios de Cobertura definidos para o teste baseado em especificações diagramáticas UML.....85
TABELA 3.11 – Relação das Diretrizes definidas para o teste baseado em especificações diagramáticas UML.....86
TABELA 4.1 – Casos de teste para o atributo dtÚltimaIAM da classe Aeronave. 90
TABELA 4.2 – Casos de teste para o atributo horasTotais da classe Aeronave. 91
TABELA 4.3 – Casos de teste com os valores válidos e inválidos para o atributo anoFabric em sua restrição {>=1940}.....91
TABELA 4.4 – Casos de teste com os valores válidos e inválidos para o atributo anoFabric em sua restrição {<=AnoAtual}.....91
TABELA 4.5 – Combinação de valores para o teste das condições do atributo anoFabric.....92
TABELA 4.6 – Combinação de valores para o teste das condições do método ExcluirAeronave().....93
TABELA 4.7 – Atributos e métodos das subclasses Credenciado e Aprendiz selecionados pelos critérios de cobertura para o teste de subclasses. 94
TABELA 4.8 – Casos de teste da restrição da associação entre as classes Aeronave e Ordem de Serviço. 95
TABELA 4.9 – Relação de multiplicidades para o critério Todas as Multiplicidades.....97
TABELA 4.10 – Relação de multiplicidades para o critério Todas-Multiplicidades-Obrigatórias. 98
TABELA 4.11 – Casos de teste para multiplicidade da associação Aeronave/OS.....99

TABELA 4.12 – Casos de teste para multiplicidade da associação OS/Aeronave.....99
TABELA 4.13 – Casos de teste para multiplicidade da associação Hélice/PásHélice 99
TABELA 4.14 – Caminhos derivados da Figura 4.8 de acordo com os critérios de cobertura Todas-S-Mensagens, Todas-C-Mensagens e Todas-Iterações. 101
TABELA 4.15 – Casos de teste para o caminhos dos Critérios Todas-Iterações da Tabela 4.14. 102
TABELA 4.16 – Caminhos derivados do Diagrama de seqüência para o caso de uso “Fechar Ordem de Serviço”. 103
TABELA 4.17 – Caminhos selecionados para o diagrama de seqüência da Figura 4.9. 104
TABELA 4.18 – Casos de teste para os caminhos selecionados da Tabela 4.17. 104
TABELA 4.19 – Relação das mensagens existentes na colaboração apresentada na Figura 4.10. 106
TABELA 4.20 – Relação das mensagens de acordo com os critérios de cobertura de teste definidos para o diagrama de colaboração da Figura 4.10. 107
TABELA 4.21 – Casos de teste para as mensagens selecionados da Tabela 4.20. 107
TABELA 4.22 – Relação das mensagens existentes na colaboração apresentada na Figura 4.11. 109
TABELA 4.23 – Relação das mensagens de acordo com os critérios de cobertura de teste definidos para o diagrama de colaboração da Figura 4.11. 109
TABELA 4.24 – Casos de teste para as mensagens selecionados da Tabela 4.23. 109
TABELA 4.25 – Casos de teste para os caminhos derivados da árvore de transições da Figura 4.13.....112
TABELA 4.26 – Casos de teste para o caminhos derivados da árvore de transições da Figura 4.15.....113

Resumo

As maiores dificuldades encontradas no teste de *software* estão relacionadas à definição dos dados de teste e a decisão de quando encerrar os testes. Uma das formas encontradas para minimizar tais dificuldades está centrada na utilização de critérios de cobertura. O principal objetivo dos critérios de cobertura é tornar o processo de testes mais rápido e preciso, fornecendo informações que determinem o que testar em um *software* para garantir sua qualidade.

A modelagem é um dos elementos de maior importância nas atividades relacionadas ao desenvolvimento de *software*. Os modelos são construídos principalmente para melhor se entender o sistema, descrever a estrutura e comportamento desejados, visualizar a arquitetura e documentar as decisões tomadas durante o seu desenvolvimento. Atualmente, o sistema de notação mais utilizado para a modelagem de sistemas baseados nos conceitos de orientação a objetos é a Linguagem de Modelagem Unificada – UML [LAR 99]. Nesta notação, um sistema é descrito por um conjunto de diagramas que apresentam diferentes aspectos do sistema.

As informações disponibilizadas por estes diagramas propiciam, já nas fases iniciais do desenvolvimento da aplicação (análise e projeto), o planejamento dos casos de teste e a definição de critérios de cobertura. Observa-se que nestas fases a maioria das informações necessárias para o teste já estão disponíveis, como por exemplo, a definição das classes com seus atributos, métodos e relacionamentos, a representação da interação existente entre objetos para a realização de um cenário e a descrição dos possíveis estados e transições de um objeto em resposta a eventos externos e internos.

Este trabalho propõe um conjunto de diretrizes e critérios de cobertura de teste, tendo como base as especificações diagramáticas UML. As diretrizes estabelecem um conjunto de instruções para que o teste seja feito e os critérios de cobertura identificam os pontos principais e serem considerados durante o teste. Na definição das diretrizes e dos critérios foram avaliadas as informações disponibilizadas pelos diagramas de classes, seqüência, colaboração e estados.

Palavras-Chave: Engenharia de Software, teste de software, critérios de cobertura de teste, diretrizes de teste, UML.

TITLE: "TEST GUIDELINES AND TEST CRITERIA COVERAGE FROM UML SPECIFICATIONS"

Abstract

The greatest difficulties found in software testing are linked to the definition of test data and to the decision as to when to finish the test. One of the possibilities there is to lessen such difficulties is centered on the use of test criteria coverage. The main goal of the criteria coverage is to render the testing process quicker and more precise, providing information that determine what to test in a software so as to guarantee its quality.

Modeling is one of the elements of the highest importance to the activities related to the development of softwares. The models are built mainly so as to better understand the system, describe the structure and behavior desired, visualize its architecture and register decision taking during its development. Presently, the notation system most used in system modeling based on orientation concepts to objects is the Unified Modeling Language – UML [LAR 99]. Within this notation system, a system is described through a set of diagrams which present different aspects of the system.

The information made available through those diagrams enable, at the initial phases of the development of the application (analysis and project), planning the test cases and defining the criteria coverage. It is observed that at these phases most of the information needed for the test are already available, as for instance the definition of classes and their attributes, methods and relationships, the representation of the existing interaction between objects for the accomplishment of a scenery and the description of the possible states and transitions of an object in response to external and internal events.

This paper shows a set of guidelines and approaches of test covering, taking as support the UML design specifications. The guidelines establish a set of instructions so that testing may be carried out and the criteria coverage identify the main questions to be considered during the test. At the definition of the guidelines and of the criteria there was an evaluation of the information made available by the class diagrams, sequence, collaboration and states.

Keywords: Software Engineering, test software, test criteria coverage, test guidelines, UML.

1 Introdução

A atividade de teste é relativamente independente do método de desenvolvimento do *software*, ou seja, não é uma fase isolada, interage com todas as outras fases (análise de requisitos, projeto, codificação) [PRE 95]. Um exemplo desta interação pode ser observado na Figura 1.1,

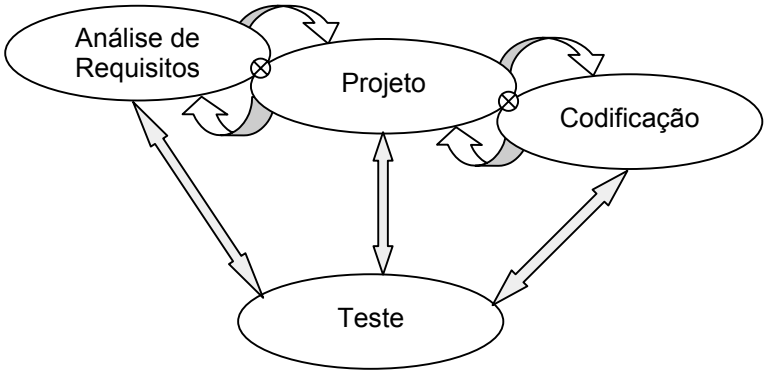


FIGURA 1.1 - O teste e as fases do desenvolvimento do *software.*

A maioria das técnicas de teste de *software* utilizadas atualmente, aplicam como referência os princípios estabelecidos por Myers [MYE 79], os quais apresentam o principal objetivo desta atividade como sendo o de encontrar erros. Este objetivo elimina a noção preconcebida de que um *software* em desenvolvimento esteja sempre correto.

Fowler em [FOW 2000], faz alguns comentários sobre a atividade de testes que confirmam a importância da iniciativa deste trabalho e de outros associados ao teste de aplicações OO ou procedimentais:

- nenhum código deveria ser escrito até que se saiba como testá-lo;
- quando escrever o código, escreva também os testes;
- enquanto os testes não estiverem funcionando não se pode anunciar o término da implementação do código;
- os testes de unidade devem ser escritos pelos projetistas de softwares;
- os testes funcionais devem ser desenvolvidos por uma pequena equipe, separada, encarregada somente de fazer os testes.

O teste de software envolve as seguintes ações: (1) determinar o conjunto de dados de teste e os resultados esperados com a aplicação destes dados; (2) a execução do programa com os dados de teste definidos; (3) a análise dos resultados computados para verificar se estão de acordo com os resultados esperados [MYE 79] e [MCG 96].

Segundo Bruce, as três principais metas da atividade de testes são: verificar que o *software* funciona de acordo com sua especificação; verificar que o *software* satisfaz todos os requisitos definidos na sua especificação de requisitos; e fornecer um *status* do progresso do projeto ao gerente de projeto [BRU 82].

Na fase de testes, diversos métodos, técnicas e estratégias de teste podem ser utilizadas, dentre elas: teste de unidade, teste de integração, teste de validação e teste de sistema. Um *software* pode ser testado utilizando-se alguns tipos de teste: (a) o teste de caixa-preta, quando as funções que o *software* deve executar são conhecidas, a fim de demonstrar a operacionalidade destas funções; (b) o teste de caixa-branca, quando a estrutura interna do *software* é conhecida, com isso, pode-se verificar se todos os componentes internos, quando exercitados, operam de maneira adequada e (c) o teste baseado em erros (análise de mutantes), quando deseja-se avaliar a qualidade dos dados de teste tendo como base o resultado da mutação do programa [PRE 95] e [VIN 97]. Cada uma destas técnicas revela tipos diferentes de erros e são vistas como complementares, sendo as duas primeiras as mais utilizadas.

A qualidade do teste está associada à eficiência dos seus dados de teste. Considera-se eficiente o conjunto de dados de teste que encontrar o maior número de erros em menor tempo [MYE 79], [WEY 80] e [MCG 94]. Pequenos programas podem possuir diversos caminhos lógicos e estes devem ser testados para garantir que os mesmos estejam livres de erros. Isto, na maioria das vezes, é impraticável pois nem todos os caminhos podem ser alcançados e à medida que estes aumentam, eleva-se o tempo e os custos do teste [PRE 95]. Culminando com o maior problema da área de testes, "determinar quando um *software* foi suficientemente testado" [WEY 80].

Uma estratégia de teste definida por vários autores como forma de minimizar o problema apresentado, são os critérios de cobertura [MYE 79],[WEY 80] e [NTA 84]. Trata-se de um conjunto de regras para determinar quais partes do programa devem ser testadas, reduzindo com isso, o número de dados de teste [KAN 99]. Weyuker definiu que um critério de cobertura será válido caso a execução de pelo menos um de seus casos de teste encontre erros no programa [WEY 80]. O principal objetivo de um critério de cobertura é facilitar a seleção dos dados de teste a serem utilizados no teste de software.

Um aspecto a ser considerado como facilitador na atividade de testes é o momento em que os dados de teste devem ser definidos. Em pesquisas realizadas observou-se que na fase de especificação do software, grande parte das informações sobre o que deve ser testado já estão disponíveis, dentre elas, as características do sistema e dos dados que o mesmo manipula. Sendo a inobservância destas características a causa da maioria dos erros encontrados na implementação do *software*, ou seja, o que foi implementado não está de acordo com o que foi especificado.

Uma das técnicas utilizadas na fase de especificação (projeto) do *software* é a modelagem lógica das informações, ou seja, a representação gráfica do que está sendo proposto. Esta representação identifica os aspectos importantes dos dados a serem manipulados pelo sistema, bem como as características da interface[1] com o usuário [FOW 2000]. A partir deste momento, os dados de teste já podem ser definidos.

[1] Trata-se da forma como o sistema será apresentado ao usuário.

Atualmente, a UML (*Unified Modeling Language*) está sendo a linguagem de modelagem[2] mais utilizada para expressar as características de *softwares* orientados a objetos. Um dos principais objetivos da UML é estabelecer um padrão para a visualização, especificação, construção e documentação de artefatos[3] [BOO 99], através dos diagramas que implementa. Com isso, procura promover a integração entre os membros da equipe de desenvolvimento, facilitando sua comunicação, confirmando assim o que Pressman apresenta como sendo um dos principais fatores para o sucesso de um projeto [PRE 95]. Os diagramas implementados pela UML, através de suas características, disponibilizam as informações necessárias para a definição do que deve ser posteriormente testado no *software* em desenvolvimento.

1.1 Proposta de Trabalho

O objetivo principal deste trabalho é a definição de um conjunto de diretrizes de teste e critérios de cobertura de teste tendo como base as especificações diagramáticas UML. Ou seja, quais informações podem ser extraídas dos diagramas UML para auxiliar o desenvolvedor no momento em que for testar o *software*.

Os critérios de cobertura propostos baseiam-se na especificação diagramática do software que está sendo desenvolvido. Como neste momento, na maioria das vezes a implementação ainda não existe, a presente proposta pretende gerar um conjunto de informações para que o desenvolvedor[4] verifique, posteriormente, a equivalência da implementação com a especificação.

1.2 Organização do Texto

Neste capítulo, fez-se uma breve introdução sobre o tema e proposta deste trabalho. O capítulo 2 apresenta uma revisão bibliográfica relacionada aos principais conceitos sobre testes de software, enfatizando o uso de critérios de cobertura na seleção de dados de teste. Para isso, apresentam-se as características do teste de software orientado a objetos e, como alguns critérios de cobertura definidos para aplicações procedimentais também aplicam-se às OO, apresentam-se os conceitos sobre este tipo de teste, suas técnicas e critérios de cobertura.

O capítulo 3 apresenta as diretrizes e os critérios de cobertura de teste definidos para especificações diagramáticas UML. Relacionam-se para os diagramas de classe, seqüência, colaboração e estado implementados pela UML, as informações que podem ser obtidas e utilizadas no teste do *software* que está sendo desenvolvido. Apresenta-se para os diagramas de seqüência, colaboração e estado, uma proposta para a implementação de uma ferramenta que faça a geração automática dos caminhos de teste.

---

[2] Segundo [BOO 99] e [FOW 2000] é a notação utilizada para expressar projetos.

[3] A UML utiliza o termo artefato, para referenciar qualquer tipo de produto que possa ser modelado, dentre eles, *software*, *hardware* e regras de negócio.

[4] Responsável pelo projeto e/ou desenvolvimento da aplicação que está sendo modelada.

O capítulo 4 apresenta um estudo de caso com o objetivo de auxiliar o entendimento sobre a aplicação dos critérios de cobertura de teste definidos no capítulo 3 e avaliar suas contribuições no processo de teste.

No capítulo 5 são apresentadas as conclusões e considerações finais a respeito dos temas tratados neste trabalho e são relacionadas as sugestões para trabalhos futuros.

2 Critérios de Cobertura e Técnicas de Teste de Software

Um dos problemas da atividade de teste de *software* é decidir quando parar de testar. Como forma de determinar o que testar, pode-se utilizar um critério de adequação que possui, como uma de suas características, a capacidade de determinar se um programa foi suficientemente testado [WEY 80]. Isto é altamente necessário pois, em algumas situações, para testar por completo uma aplicação pode-se gerar um número exorbitante de dados de teste. Uma das soluções para este problema seria a seleção de um subconjunto destes dados, tendo como base algum tipo de critério. Considera-se importante o conjunto de dados que satisfaça o critério selecionado [HUA 75].

A estes critérios dá-se o nome de critérios de cobertura ou critérios de seleção. Trata-se de condições que devem ser preenchidas pelo teste, as quais selecionam determinados caminhos que visam cobrir o código implementado ou representação gráfica deste. O critério de cobertura será válido se a execução de pelo menos um dos casos de teste detectar erros no programa. Espera-se que todos os erros do programa que o critério de cobertura propõe detectar sejam encontrados pelos casos de teste [WEY 80] e [HER 97].

Na literatura, pode-se encontrar diversas técnicas de teste para *software* procedimental e orientado a objetos. Neste capítulo serão apresentadas somente aquelas que serviram de base para a consecução deste trabalho. Apresenta-se, também, os critérios de cobertura que fundamentam estas técnicas.

2.1 Teste de *Software* Procedimental

Esta seção tem por objetivo apresentar os tipos de teste para *software* procedimental, aqui divididos em estrutural e funcional. Segundo McGregor [MCG 96], pode-se aplicar estes testes em *softwares* construídos segundo o paradigma da orientação a objeto, desde que sejam feitas as adaptações necessárias. Estas adaptações estão relacionadas às características da orientação a objetos, não tratadas pelas técnicas clássicas de teste, dentre elas, a herança, o encapsulamento, o polimorfismo e a ligação dinâmica.

2.1.1 Teste Estrutural

O teste estrutural (caixa-branca) analisa a estrutura interna dos programas e, através do exame minucioso dos detalhes procedimentais, deriva casos de teste. Tem por objetivo causar a execução de caminhos identificados no programa, baseados no fluxo de controle e no fluxo de dados. É normalmente utilizado para encontrar erros na lógica dos programas [PRE 95].

Segundo Myers, os casos de teste derivados devem possuir as seguintes características [MYE 79] e [PRE 95]:

- garantir que todos os caminhos independentes dentro de um módulo tenham sido exercitados pelo menos uma vez;
- exercitar todas as decisões lógicas para valores falsos ou verdadeiros;
- executar todos os laços (*loops*) em suas fronteiras e dentro de seus limites operacionais; e
- exercitar as estruturas de dados internas para garantir a sua validade .

Não se pode desprezar o teste de caixa branca, pois conforme declarou Beizer [BEI 90], na maioria das vezes os erros escondem-se pelos cantos da aplicação e congregam-se nas fronteiras. Logo, os testes de caixa branca têm maior probabilidade de descobri-los.

Alguns problemas desta abordagem foram apresentados por Myers e Pressman [MYE 79] e [PRE 95], dentre eles:

- número infinito de caminhos em programas com laços torna necessária a aplicação de testes exaustivos no programa, o que é considerado impraticável;
- desperdício de tempo com caminhos, possivelmente, não executáveis (*infeasible paths*) no programa.

Não existe garantia de que um programa esteja correto, simplesmente pela execução bem sucedida de um caminho, pois com outro caso de teste, um erro pode ser descoberto. Uma das soluções encontradas para o controle da eficácia na atividade de teste, está centrada na utilização de critérios de seleção de caminhos ou como são mais conhecidos, critérios de cobertura.

As seções seguintes apresentam os aspectos dos critérios de cobertura para o teste estrutural baseados no fluxo de controle e no fluxo de dados. Ambos utilizam como base o Grafo de Fluxo de Controle ou como McCabe [MCC 76] chamou de Grafo de Programa, sendo este utilizado para representar o fluxo de controle de um programa. Trata-se de um grafo dirigido com um único nó de entrada e um único nó de saída, onde cada nó representa uma seqüência de comandos que são sempre executados como um bloco de comandos e cada arco representa uma transferência de controle entre esses blocos.

2.1.1.1 Critérios de Cobertura Baseados em Fluxo de Controle

Este tipo de critério fundamenta-se na seleção de um conjunto *S* de elementos (arcos, laços, nós ou subcaminhos) no grafo de fluxo de controle(GFC) do programa em teste. Apresenta-se, na Figura 2.1, a notação básica para a representação das estruturas lógicas (bloco seqüencial, *if-then-else*, *while-do*, *repeat-until* e *case*) de um programa através de um GFC.

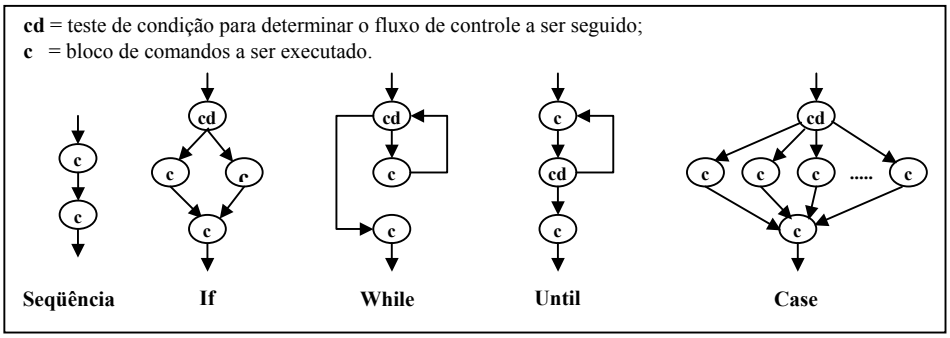
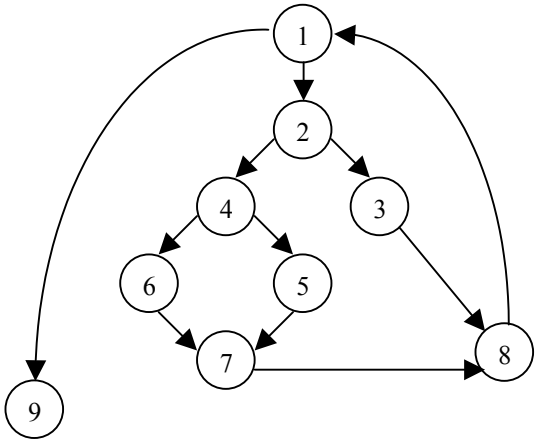


FIGURA 2.1 – Principais estruturas de um GFC [HET 87].

Qualquer procedimento pode ser traduzido num grafo de fluxo. Segundo Rapps [RAP 85], para a representação de um módulo do sistema através de um GFC, o código do programa (módulo) deve ser decomposto em um conjunto de blocos desconexos, com uma característica que permita a execução, em ordem, de todos os comandos do bloco, assim que o primeiro for executado. Um GFC estabelece correspondência entre nós e bloco, e entre arcos e fluxo de controle. Um caminho em um GFC é uma seqüência finita de nós ($n_1, n_2, ..., n_k$), onde $k \geq 2$, tal que exista um arco (ramo) de n_i para n_{i+1} sendo $i = 1,2,...,k-1$. Logo, cada círculo (nó) representa uma ou mais instruções procedimentais e os arcos (ramos) o fluxo de controle. No exemplo da Figura 2.2, pode-se observar a conversão de um procedimento hipotético (escrito em uma determinada linguagem) em um grafo de fluxo de controle (GFC).




```
   procedure: teste
1:  do while valor<>0
      read valor;
2:   if valor > 2
3:   then soma=soma+1;
           valor:=1;
           contador=cont+1;
4:       elseif valor < 2
5:           then contador=0;
6:           else soma=soma+1;
           valor:=0;
7:       endif
     endif
8:   end do
9: end
```

FIGURA 2.2 – Exemplo de conversão de um programa em GFC.

Myers [MYE 79] apresentou os critérios de cobertura baseados no fluxo de controle e como estes seriam satisfeitos.

- **Cobertura por nós (ou comandos):** o critério é satisfeito se todos os nós/comandos forem executados pelo menos uma vez.
- **Cobertura por ramos (ou decisões):** o critério é satisfeito se todo arco/decisão for executado pelo menos uma vez.
- **Cobertura por todos os caminhos:** o critério será satisfeito se todos os caminhos possíveis forem executados pelo menos uma vez. Observa-se que problemas

podem ocorrer quando o programa possuir laços (*loops*), pois, o número de caminhos completos (possíveis) pode ser infinito ou muito elevado [RAP 85].

- **Cobertura por condições:** o critério é satisfeito se todas as condições em uma decisão forem avaliadas para verdadeiro e falso pelo menos uma vez.
- **Cobertura por condições múltiplas:** o critério será satisfeito se todas as combinações possíveis em cada decisão forem executadas pelo menos uma vez.

Um conjunto de critérios para a cobertura do fluxo de controle em nível de sistema foi proposto por Silva em [SIL 95]. Trata-se de uma adaptação dos critérios de cobertura apresentados por Myers [MYE 79]. Na Tabela 2.1 pode-se observar a relação entre estes critérios.

TABELA 2.1 – Relação entre os critérios de Myers e Silva [MYE 79] e [SIL 95].

Critérios Fluxo de Controle por Módulo (Unidade) Myers [MYE 79]	**Critérios Fluxo de Controle do Sistema Silva [SIL 95]**
todos-caminhos *(Cobertura por todos os caminhos)*	todos-caminhos-interprocedimentais
todos-arcos *(Cobertura por ramos)*	todos-arcos-chamada
todos-nós *(Cobertura por nós)*	todos-módulos

Diferente do fluxo de controle por módulo, que utiliza um GFC para a representação dos nódos e arcos, o fluxo de controle de sistema é representado por um diagrama de chamadas. Um exemplo deste diagrama pode ser observado na Figura 2.3. Este tipo de diagrama representa as relações interprocedurais do sistema, mostrando as possíveis ordens de execução entre seus módulos.

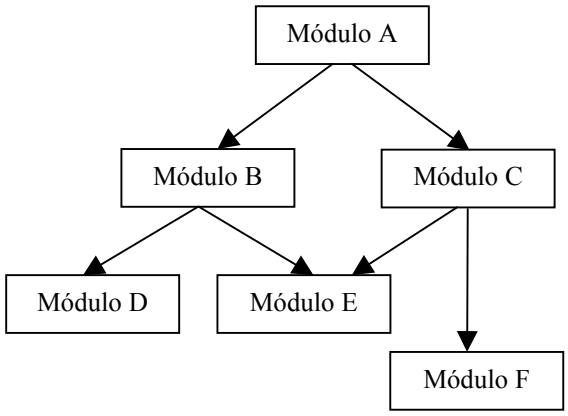


FIGURA 2.3 – Exemplo de um diagrama de chamadas (DC) [SIL 95].

Seguem algumas definições utilizadas nos critérios de cobertura baseados no fluxo de controle do sistema:

- caminho possível de execução – trata-se de uma lista com nomes de módulos, ordenada de acordo com a ordem de chamadas realizadas, sendo que o primeiro elemento da lista é o módulo principal;
- caminho de execução – é similar ao caminho possível de execução, entretanto, neste são consideradas apenas as chamadas realizadas em uma determinada execução, ou seja, realiza-se a análise dinâmica, a partir da execução do sistema instrumentado.

Os critérios que analisam o fluxo de controle do sistema são apresentados a seguir:

- **Todos-caminhos-interprocedurais** – O caminho possível de execução é convertido em uma expressão regular, ou seja, não se tem como resultado subcaminhos e sim a expressão resultante. A partir desta expressão, é construída uma máquina de autômato finito não-determinístico[5], para a avaliação do caminho de execução.
- **Todos-arcos-chamadas** – Os arcos existentes no diagrama de chamadas são comparados com caminhos de execução, procurando por arcos que não tenham, ainda, sido exercitados. O conjunto de caminhos satisfaz o critério, quando todos os arcos estiverem incluídos no mesmo.
- **Todos-módulos** – Os módulos existentes no diagrama de chamada estão interligados por arcos. Se o módulo não for chamado por nenhum outro módulo, verifica-se explicitamente, sua não executabilidade. Para satisfação deste critério cada módulo deve ser executado pelo menos uma vez.

Vergilio [VER 97] introduziu os critérios restritos com o objetivo de aumentar a eficácia dos critérios estruturais. Uma maneira de aumentar a capacidade desses critérios de revelar erros é combinar a técnica estrutural com aspectos da técnica funcional e/ou baseada em erros e associar a cada elemento requerido uma restrição a ser satisfeita durante a execução do caminho que exercitará esse elemento. Mais de um caminho candidato a cobrir cada elemento requerido poderá ser exercitado, ou ainda, um caminho poderá ser executado com um ou mais dados de teste que satisfarão restrições que descrevem erros e, portanto, têm alta probabilidade de revelar um erro ainda não descoberto. Quando associa-se uma restrição a um elemento requerido por um critério, restringe-se o número de caminhos candidatos a cobrir o elemento e, consequentemente, o domínio a ele associado, gerando assim um elemento restrito. Os critérios restritos podem ser aplicados a qualquer critério de teste estrutural. Para os critérios baseados em fluxo de controle foram definidos:

- **Critério todos-nós restritos:** deve-se associar a cada nó uma ou mais restrições (i, C_1),...., (i, C_n). A restrição é dada pelo par (i,C), sendo i o elemento e "C" a restrição e será coberto pelo caminho que executa o nó i que satisfaz C.
- **Critério todos-ramos restritos:** deve-se associar a cada ramo uma ou mais restrições. A restrição é dada pelo par ((i,j),C) e será coberto pelo caminho que executa o ramo (i,j) que satisfaz C.
- **Critério todos-caminhos restritos:** deve-se associar a cada caminho uma ou mais restrições. A restrição é dada pelo par (P,C) e será coberto pelo caminho (P) que satisfaz C.

Segundo Rapps [RAP 85], um dos problemas que ocorre com a utilização de critérios orientados ao fluxo de controle, é a falta de confiabilidade do teste

[5]*Em uma máquina composta por fita, unidade de controle e uma função de transição, pode-se afirmar que um Autômato não-determinístico assume um conjunto de estados alternativos, como se houvesse uma multiplicação da unidade de controle, uma para cada alternativa, processando independentemente, sem compartilhar recursos com as demais. Assim, o processamento de um caminho não influi no estado, símbolo lido e posição da cabeça dos demais caminhos alternativos*. (Trecho extraído na integra de [MEN 98])

realizado. Estes critérios particionam o Domínio D de dados de entrada em subdomínios, de forma que $D = \cup D_{[j]}$, tal que para todo $d \in D$, $d \in D_{[j]}$, isto ocorrerá, se e somente se a execução de um programa com entrada d faça com que o caminho d_j seja executado. Um conjunto de testes $T = \{d_1,,d_2,, ..., d_j\}$, onde $d_j \in D_{[j]}$, poderia ser considerado um teste adequado para o programa, caso não existisse a possibilidade de haver um $D_{[j]}$ para o qual algum dado de entrada $d_1 \in D_{[j]}$ o programa computasse um valor correto, enquanto que para outro $d_2 \in D_{[j]}$ o programa computasse um valor incorreto. Se em algum momento somente o dado d_1 for selecionado para o teste, o erro não será detectado.

Na próxima seção são apresentados os critérios de cobertura baseados em fluxos de dados. Estes critérios foram desenvolvidos para avaliar os aspectos não cobertos pelos critérios baseados no fluxo de controle. Este novo grupo de critérios tem como um de seus objetivos, fornecer uma hierarquia de critérios (baseada em fluxo de dados) entre os critérios todos os ramos e todos os caminhos (baseados no fluxo de controle).

2.1.1.2 Critérios de Cobertura Baseados em Fluxo de Dados

Laski define a análise do fluxo de dados como sendo uma análise estática do programa para o fornecimento de informações sobre as ações sofridas pelas variáveis utilizadas no programa [LAS 83]. Através dessas ações pode-se encontrar anomalias que podem surgir em diversos pontos do programa e que caracterizam suspeitas de possíveis presenças de defeitos[6] nesses pontos.

Os critérios de cobertura baseados em fluxo de dados baseiam-se na análise de fluxo de dados do programa a ser testado. Uma característica comum nesta classe de critérios é a de que eles requerem que sejam testadas as interações que envolvam definições de variáveis de programas e subsequentes referências a estas variáveis. Essa cobertura(seleção) fornece um número de caminhos sempre finito e resolve algumas falhas encontradas nos critérios baseados no fluxo de controle [NTA 84], [RAP 85], [NTA 88] e [VIL 97a].

A idéia básica destes critérios consiste em focalizar a atribuição de valores às variáveis e o uso posterior destes valores, estabelecendo que a ocorrência de uma variável pode ser de dois tipos: definição e uso. Portanto estes critérios estão fundamentados nas associações entre a definição de uma variável e o seu uso para a derivação de casos de teste.

Segundo Myers [MYE 79], um caminho que nunca foi executado não pode ser considerado confiável. Rapps [RAP 85] complementa dizendo que um programa também não deve ser considerado suficientemente testado até que todos os resultados computacionais sejam utilizados pelo menos uma vez.

Inicialmente os trabalhos envolvendo teste baseado em análise de fluxo de dados eram aplicados somente ao teste de unidade [VIL 97a]. Para representação do teste de unidade utiliza-se o GFC estendido, acrescentando-se aos nós e arcos as

[6] É detectada sempre que o resultado obtido não for igual ao resultado esperado.

relações de definições e referências à variáveis, gerando um *grafo def/uso* (definições/usos).

De acordo com Rapps [RAP 85] um *c-uso* (*computation-use*) ocorre quando a variável esta sendo referenciada em uma computação. Um *p-uso* (*predicate-use*) ocorre quando a variável é utilizada, afeta diretamente o fluxo de controle do programa. Definições *c-uso* são associadas a nós, enquanto *p-usos* são associados a arcos.

Um caminho é uma seqüência finita de nós ($n_1,n_2...,n_k$), sendo $k \geq 2$, tal que exista um arco de n_1 para n_{i+1} para i = 1,2,...,k-1. Um caminho é um caminho simples se todos os nós que compõem esse caminho, exceto possivelmente o primeiro e o último, são distintos. Um caminho é livre de laço (*loop-free*) se todos os nós forem distintos. Um caminho ($i,n_1,...,n_m,j$), m ≥ 0, do GFC, que não contenha uma definição de uma variável x nos nós $n_1,...,n_m$ é chamado de caminho livre de definição (*def-clear path*) com respeito a (c.r.a) [7] x do nó i ao nó j e do nó i ao arco (n_m , j). Um nó i possui uma definição global (*global def*) de uma variável x se ocorre uma definição de x no nó i e existe um caminho livre de definição de i para algum nó ou para algum arco que contenha um c-uso ou um p-uso, respectivamente, da variável x. Um c-uso da variável x em um nó j é um c-uso global se não existe uma definição de x no nó j precedendo este c-uso. Caso contrário, este é um c-uso local.

Os critérios de fluxo de dados utilizam o seguinte grupo de definições:

- *def(i)* é o conjunto de definições globais associadas a um nó *i*;
- *c-usos(i)* é o conjunto de *c-usos* globais;
- *p-usos(i,j)* é o conjunto de *p-usos* associado a um arco *(i,j)*.

Seja i um nó e x uma variável, tal que $x \in def(i)$ então:

- *dcu(x,i)* é o conjunto de todos os nós *j* tais que $x \in c\text{-}usos(j)$ e há um caminho livre de definições c.r.a *x* de *i* para *j*;
- *dpu(x,i)* é o conjunto de todos os arcos *(j,k)* tais que $x \in p\text{-}uso(j,k)$ e há um caminho livre de definições c.r.a *x* de *i* para *j*.

Maldonado, apresentou o conceito de *potencial-uso*, que procura investigar todos os possíveis efeitos a partir de uma mudança de estado no programa em teste, decorrente da definição de variáveis em determinado nó [MAL 91]. A diferença entre este novo conceito e os demais está na relação definição-uso necessária nos outros critérios baseados em fluxo de dados ser caracterizada sem a necessidade de ocorrência de um uso. Para o conceito de *potencial-uso*, deve-se considerar as definições anteriores e as seguintes:

- *pdcu(x,i)* é o conjunto de todos os nós *j*, tais que existe um caminho livre de definição c.r.a *x* do nó *i* para o nó *j*;
- *pdpu(x,i)* é o conjunto de todos os arcos *(j,k)* tais que existe um caminho livre de definição c.r.a *x* do nó *i* para o arco *(j,k)*;

[7] Do ingles *with respect to (w.r.t.) = com respeito a (c.r.a)*

- *potencial-du-caminho* com relação à (c.r.a) variável x é um caminho livre de definição c.r.a x do nó n_1 para o nó n_k e para o arco (n_j, n_k), onde o caminho $(n_1,n_2,...,n_j)$ é um caminho livre de laço e no nó n_1 ocorre uma definição de x.

Vários critérios foram propostos por Rapps e Weyuker [RAP 82] e [RAP 85], Maldonado [MAL 91], Ntafos [NTA 84] e Laski e Korel [LAS 83]. A seguir serão apresentados alguns destes critérios.

Para os critérios definidos abaixo, G representa um grafo definição/uso e P um conjunto de caminhos completos de G.

- **Critério todas-definições:** P satisfaz este critério se para todo nó i de G e para todo $x \in def(i)$, P inclui um caminho livre de definições c.r.a x de i para algum elemento de $dcu(x,i)$ e $dpu(x,i)$.
- **Critério todos-P-usos:** P satisfaz este critério se para todo nó i e para todo $x \in def(i)$, P inclui um caminho livre de definições c.r.a x de i para todos os elementos de $dpu(x,i)$.
- **Critério todos-usos:** P satisfaz este critério se para todo nó i e para todo $x \in def(i)$, P inclui um caminho livre de definição c.r.a x de i para todos os elementos de $dcu(x,i)$ e para todos os elementos de $dpu(x,i)$.
- **Critério todos-potenciais-usos:** P satisfaz este critério se, para todo nó i e para toda variável x para a qual existe uma definição em i, P inclui pelo menos um caminho livre de definição c.r.a x do nó i para todo nó e para todo arco possível de ser alcançado a partir de i.
- **Critério todos-potenciais-du-caminhos:** P satisfaz este critério se, para todo nó i, tal que $def(i) \neq \varnothing$, P inclui todos os potenciais-du-caminhos c.r.a todas as variáveis $x \in def(i)$ a partir do nó i para todo nó $j \in pdcu(x,i)$ e para todo arco $(j,k) \in pdpu(x,i)$.

Vergilio [VER 97] também definiu um conjunto de critérios restritos para os critérios baseados em fluxo de dados:

➢ Na família de critérios de Rapps e Weyuker [RAP 85]:

- **Critério todos-usos restritos:** deve-se associar a cada associação requerida uma ou mais restrições. A restrição é dada por $((i,j,x),C)$ ou $((i,(j,k),x),C)$ e será coberta por um caminho livre de definição c.r.a x, ou seja, um caminho que cubra a associação e satisfaça C.
- **Critério todos-du-caminhos restritos:** deve-se associar a cada du-caminho uma ou mais restrições. A restrição é dada pelo par (DP,C) e será coberto por um caminho que inclua DP e satisfaça C.

➢ Na família de critérios potenciais usos de Maldonado [MAL 91]:

- **Critério todos-potenciais-usos restritos:** deve-se associar a cada potencial-associação requerida uma ou mais restrições. A restrição é dada por $((i,(j,k),x),C)$ e será coberta por um caminho livre de definição c.r.a x, ou seja, um caminho que cubra a associação e satisfaça C.

- **Critério todos-potenciais-usos/du restritos:** deve-se associar a cada potencial-associação requerida uma ou mais restrições. A restrição é dada por ((*i*,(*j*,*k*),*x*),C) e será coberta por um potencial-du-caminho livre de definição c.r.a *x*, ou seja, um potencial-du-caminho que cubra a associação e satisfaça C.
- **Critério todos-potenciais-du-caminhos restritos:** deve-se associar a cada potencial-du-caminho uma ou mais restrições. A restrição é dada pelo par (PDP,C) e será coberto por um caminho que inclua PDP e satisfaça C.

Spoto [SPO 97] apresenta uma extensão à abordagem de teste estrutural, visando as aplicações de Banco de Dados Relacional (BDR) com SQL embutido. Para isso, definiu uma estratégia de teste estrutural para BDR e dois grupos de critérios de cobertura para atender este teste: (1) critérios intra-modular[8] e (2) critérios inter-modular[9], representando o escopo em que estão inseridos.

Segundo o autor, um sistema gerenciador de banco de dados relacional apresenta dois níveis diferentes de fluxo de controle: (1) linguagem de programa e (2) linguagem de manipulação da base de dados (SQL), sendo respectivamente o fluxo de controle da linguagem hospedeira e o fluxo de controle da linguagem SQL padrão. Foi adotado um grafo dirigido para representar o fluxo de controle dos dois tipos de linguagens utilizadas na definição dos critérios, sendo G_{BD}=(N* , E) onde N* representa o conjunto de nós N_h provenientes da linguagem hospedeira (C, Pascal ou PL/I) e de nós N_S provenientes da linguagem SQL. Na Figura 2.4, é apresentado um exemplo de grafo de fluxo de controle, onde os nós N_h são representados por nós arredondados enquanto que os nós N_S por nós retangulares e os arcos por setas que mostram as direções do fluxo de controle de um nó ao outro.

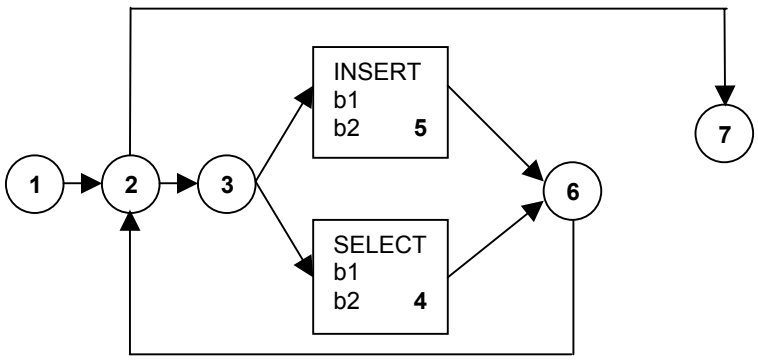


FIGURA 2.4 – Exemplo de um Grafo de Fluxo de Controle [SPO 97].

Para o entendimento dos critérios de Spoto, deve-se considerar a seguinte definição:

- Π é um conjunto de caminhos executados por um grupo de dados de teste dentro de uma aplicação de BDR, que satisfaz a propriedade de aplicabilidade.

➢ Critérios baseados em fluxo de dados Intra-modular:

[8] Para o teste das variáveis *host* e variáveis de tabela.
[9] Utilizado para o teste de integração entre os programas da aplicação de BDR até atingir toda a aplicação.

- **Todas as Definições-h**: Π satisfaz este critério se, para todo nó $n_i \in N^*$ em que existir uma definição de uma variável *host*[10], Π cobrir[11] uma associação definição-uso dessa variável, declarada no módulo da aplicação.
- **Todos os Usos-h:** Π satisfaz este critério se Π cobrir todas as associações definição-uso, possíveis para cada nó $n_i \in N^*$ e para cada variável *host* definida no nó n_i.
- **Todos os Potenciais Usos-h:** Π satisfaz este critério se, para todo nó *i* e para toda variável *h* para a qual existe uma definição em *i*, Π inclui pelo menos um caminho livre de definição c.r.a *h* do nó *i* para todo nó e para todo arco possível de ser alcançado a partir de *i*.
- **Todas as Associações-du-t-1:** Π satisfaz este critério se Π cobrir todas as associações definição-uso de variáveis de tabela, entre as unidades dentro de um mesmo módulo da aplicação (inter-unidades).
- **Todas Interações k-du-t-1:** Π satisfaz este critério se Π cobrir todas as interações definição-uso envolvendo uma ou mais variáveis de tabela, para um mesmo módulo da aplicação.

➢ Critérios baseados em fluxo de dados Inter-modulares:

- **Todas Associações-du-t-2:** Π satisfaz este critério se Π cobrir todas combinações possíveis que levam de uma definição de uma variável de tabela t_i em uma execução do módulo m_j para todos os módulos diferentes de m_j que usam a variável de tabela t_i.
- **Todas Interações k-du-t+:** Π satisfaz este critério se Π cobrir todas as interações definição-uso envolvendo uma ou mais variáveis de tabela, em diferentes módulos da aplicação.

Nos critérios de fluxo de dados para requerer um determinado elemento (caminho, associação, etc.) é necessário que exista a ocorrência explicita de um uso de variável. Segundo Vilela este aspecto está associado as limitações deste tipo de critério [VIL 97].

Este problema pôde ser amenizado com a criação dos critérios potenciais usos de Maldonado que introduziram pequenas, mas, fundamentais modificações nos conceitos relacionados aos critérios de Rapps e Weyuker. Os elementos a serem requeridos são caracterizados independentemente da ocorrência explicita de uma referência a uma determinada definição; se um uso dessa definição pode existir – um potencial-uso – a "potencial associação" entre a definição e o potencial uso é caracterizada, e eventualmente requerida [MAL 91] e [VIL 97].

Os princípios dos critérios baseados no fluxo de controle definidos por Myers, foram utilizados como base na definição dos critérios apresentados no Capítulo 3. As técnicas de teste estrutural não puderam ser aplicadas nos critérios definidos pois a proposta deste trabalho não preve a avaliação da estrutura da aplicação, uma vez que na fase de especificação a implementação, na maioria das vezes, ainda não existe.

---

[10] variável de ligação.

[11] Π cobre uma associação definição-uso de uma variável *x*, se ele incluir um caminho livre de definição c.r.a *x*, do nó em que ocorre a definição até o nó ou arco em que ocorre o uso.

2.1.2 Teste Funcional

O teste funcional (caixa preta) deriva os casos de teste a partir da análise da funcionalidade do *software*, sem levar em consideração a estrutura interna do mesmo [MYE 79]. Na maioria das vezes é aplicado nas fases finais da atividade de teste.

O teste de caixa preta serve como complemento para o teste de caixa branca (estrutural), pois permite que sejam encontradas outras categorias de erros, dentre elas: (i) funções incorretas ou ausentes; (ii) erros de interface; e (iii) erros nas estruturas de dados ou no acesso a bancos de dados externos.

Na literatura pode-se encontrar alguns métodos do teste funcional para o software procedimental, dentre eles: o particionamento por equivalência e a análise do valor limite. Nas seções seguintes, são apresentadas as principais características destes métodos, que serviram de base para a definição dos critérios apresentados no Capítulo 3.

2.1.2.1 Método Particionamento por Equivalência

O particionamento de equivalência é um método de teste de caixa preta que divide o domínio de entrada de um programa em classes de dados a partir das quais os casos de teste podem ser derivados [PRE 95].

Esta técnica procura definir um caso de teste que descubra classes de erros, reduzindo o número total de casos de teste que devem ser desenvolvidos. Myers apresenta os passos para sua execução [MYE 79]:

- identificar as classes de equivalência necessárias;
- definir os casos de teste a serem aplicados.

Podem ser definidos dois tipos de classes de equivalência, as válidas que representam um conjunto de estados válidos e as inválidas para as condições de entrada errôneas, podendo ser um valor numérico, um intervalo de valores, ou uma condição *booleana*. Algumas diretrizes para definição de classes de equivalência foram apresentadas por Myers [MYE 79] e [PRE 95]:

- se a condição de entrada especificar um intervalo, são definidas uma classe de equivalência válida e duas inválidas;
- se uma condição de entrada exigir um valor específico, são definidas uma classe de equivalência válida e duas inválidas;
- se uma condição de entrada especificar um membro de um conjunto, são definidas uma classe de equivalência válida e uma inválida;
- se uma condição de entrada for *booleana*, são definidas uma classe de equivalência válida e uma inválida.

Myers apresentou as etapas a serem seguidas no processo de seleção de casos de teste [MYE 79]:

- atribuir um número único de casos de teste para cada uma das classes de equivalência;
- até que todas as classes de equivalência válidas, tenham sido cobertas pelos casos de teste, escreva um novo caso de teste para cobertura, se possível, de outras classes de equivalência válidas;
- até que todas as classes de equivalência inválidas, tenham sido cobertas pelos casos de teste, escreva um caso de teste que cubra uma e somente uma das classes de equivalência invalidas.

A seleção dos casos de teste, deve ser feita de forma que o maior número de atributos de uma classe de equivalência seja exercitado de uma só vez.

2.1.2.2 Método Análise do Valor Limite[12] - BVA

Trata-se de uma técnica que leva a escolha de casos de teste para análise e teste dos valores fronteiriços, pois um número maior de erros tende a ocorrer nas fronteiras do domínio de entrada, principalmente quando estão sendo utilizados valores comparativos, operadores lógicos e/ou relacionais [MYE 79] e [PRE 95].

Esta técnica complementa o particionamento de equivalência, descrito anteriormente. Em vez de selecionar qualquer elemento de uma classe de equivalência, a BVA leva à seleção de casos de teste nas extremidades da classe. Em vez de concentrar somente nas condições de entrada, a BVA deriva os casos de teste também do domínio de saída [MYE 79].

Diretrizes para a BVA, foram propostas por Myers, as quais são muito semelhantes às propostas para o particionamento de equivalência. A aplicação destas diretrizes proporciona testes mais completos, com uma probabilidade maior de revelar erros.

- Se uma condição de entrada especificar um intervalo delimitado pelos valores a e b, os casos de teste devem ser projetados com valores a e b, logo acima e logo abaixo de a e b, respectivamente;
- se uma condição de entrada especificar uma série de valores, os casos de teste que ponham à prova números máximos e mínimos devem ser desenvolvidos. Valores logo acima e logo abaixo de mínimo e do máximo também são testados;
- aplique as diretrizes 1 e 2 às condições de saída, para analisar que tipos de informações de saída devem ser geradas, para se testar as entradas;
- se as estruturas internas de dados do programa tiverem fronteiras preestabelecidas, certifique-se de projetar um caso de teste para exercitar a estrutura de dados em sua fronteira.

[12] Originalmente chamada de *Boundary Value Analysis - BVA*

O número de aplicações desenvolvidas segundo o paradigma procedimental, ainda é muito grande, mas, a orientação a objetos vem sendo cada vez mais utilizada, despontando como uma das soluções para que se consiga resolver de forma adequada os problemas originados pela crescente complexidade dos sistemas atuais. Mesmo sendo possível utilizar alguns dos conceitos do teste baseado em sistemas procedimentais, o teste em aplicações OO é mais complexo, principalmente, devido as características de herança, encapsulamento, polimorfismo e ligação dinâmica. Daí a importância em definir-se técnicas de teste que considerem estes aspectos. Na próxima seção são abordadas algumas das técnicas de teste para *software* orientado a objetos, principalmente aquelas utilizadas na fundamentação deste trabalho.

2.2 Teste de Software Orientado a Objetos

Nesta seção, apresentar-se-ão as principais características do teste de *software* orientado a objetos (OO), segundo a opinião de diversos autores.

Com o advento e aceitação do paradigma baseado na Orientação a Objetos, suas características introduziram um novo conjunto de problemas de teste e manutenção de *software*. Segundo Fayad e Tsai [FAY 95] testes extensivos, verificação e validação durante o desenvolvimento do software são essenciais para a construção de *softwares* confiáveis. A prática do teste de programas OO tem melhorado significativamente nos últimos anos, devido ao surgimento de novas técnicas e ferramentas. Porém, alguns problemas primários precisam ser resolvidos: (1) definição de padrões amplamente aceitos para serem embutidos no teste; (2) estratégias práticas de processos para aumentar a qualidade dos testes; e (3) desenvolvimento da automação do teste. McGregor em [MCG 96] apresenta alguns dos objetivos para estudos e trabalhos relacionados à validação de software OO, destacando-se dentre eles:

- definição de técnicas específicas para a seleção, construção e execução de casos de teste;
- especificação de um modelo de processo de teste que inclua diretivas para o teste de *software* desde a sua fase de projeto e para a escolha de abordagens organizacionais;
- desenvolvimento de técnicas para a construção de casos de teste para componentes;
- desenvolvimento de técnicas para a construção de casos de teste para sistemas;
- elaboração de diretrizes para a construção de classes testáveis.

Testes são necessários de sistemas OO como em sistemas estruturados, pois o erro pode sempre ocorrer por meio de: (a) falha na especificação de requisitos; e (b) falha na programação ou em outros estágios do desenvolvimento [MAR 95].Os principais fatores que diferenciam testes de software OO são apresentados a seguir:

- **Ocultamento de informação:** Objetos são caixas pretas. Mesmo que um projeto possa reduzir uma mudança de estado em outro objeto, o primeiro não consegue visualizar diretamente esta mudança de estado. Isto ocorre porque todo

conhecimento relativo ao estado de um objeto é obtido por uma interface bem definida, dificultando a verificação do resultado dos testes.

- **A existência de classes não instanciáveis:** As classes abstratas e genéricas, não permitem a criação de instâncias, por não conterem informações suficientes, sendo assim, impossível testá-las exatamente como são.
- **Herança:** objetos podem herdar características de outros objetos. A maioria dos aspectos que já foram testados na classe que fornece as características devem ser testados novamente nas classes que os herda, mesmo que não tenham sido modificados. Para esta operação podem ser utilizados diferentes casos de teste. O conceito de herança deve ser analisado com atenção, pois cada subclasse tem contexto diferente da superclasse. A Figura 2.5 apresenta um exemplo de herança, onde apesar dos métodos *A* e *B* trabalharem perfeitamente na superclasse *Teste* a ação e a interação de *A*, *B* e *C* na subclasse *Teste1* devem ser testadas, verificando se *C* usa *A* e *B*. Entretanto, se o projeto for feito corretamente, tendo uma excelente estrutura de herança, alguns dos testes usados na superclasse poderão ser utilizados nas subclasses.

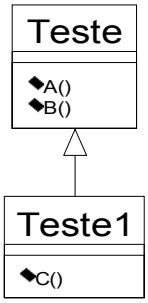


FIGURA 2.5 – Exemplo de herança entre classes.

- **A combinação de polimorfismo, herança e encapsulamento:** é única em linguagens OO, podendo ocasionar erros que não existem nas linguagens convencionais. A estratégia utilizada para o teste deve ser capaz de identificar estes erros e oferecer critérios para decidir quando os testes já foram executados satisfatoriamente.
- **A reutilização:** um código já testado, quando for reutilizado, deve ser novamente testado para que seja verificada a sua adequação ao novo contexto. Sendo assim, a reutilização não substitui o teste, e sim fornece uma classe ou um *cluster* (*framework*) confiável, o que leva a pensar em um "teste de inserção" quando um *framework* é integrado a um sistema, verificando-se então que aspectos podem ser afetados.

Algumas vantagens e desvantagens do teste de *software* OO em relação ao teste de *software* procedimental foram apresentadas por McGregor [MCG 96]:

Vantagens:

- interfaces de classes e métodos são bem definidas e explícitas;
- número reduzido de parâmetros implica um número menor de casos de teste para sua cobertura;
- a herança sugere uma maneira natural de reutilizar casos de teste.

Desvantagens:

- encapsulamento de informações dificulta a avaliação da integridade da classe;
- os múltiplos pontos de entrada de uma classe dificultam o gerenciamento de teste;
- polimorfismo e a ligação dinâmica expandem as possíveis interações entre objetos.

É comum o surgimento de algumas implicações, durante a fase de testes em aplicações orientadas a objetos, principalmente, devido a suas características. McGregor em [MCG 96], destacou aquelas com maior relevância e que ocorrem com maior freqüência:

- devido à explicita separação entre especificação e implementação da classe, casos de teste funcional podem ser gerados separadamente de casos de teste que requerem conhecimento do código da aplicação;
- pode-se testar todas as classes, mas não todos os objetos;
- gerenciamento de mensagens entre objetos é similar ao de chamadas de rotinas, entretanto a troca de mensagens ocorre mais freqüentemente;
- mudanças feitas na definição de uma classe, podem ser automaticamente propagadas para outras;
- algumas definições e declarações são reutilizadas em vários níveis da árvore de herança, interagindo com novas definições e declarações.

2.2.1 Planejamento para o Teste de Software Orientado a Objetos

Esta seção aborda os conceitos sobre planejamento de testes, apresentados por McGregor em [MCG 96].Como em qualquer atividade, o teste de software deve possuir um planejamento para determinar como o mesmo deve ser realizado. Os elementos fundamentais para um plano de testes são:

- **a extensão do teste** – o que será testado (abrangência);
- **os critérios de teste** – o que será utilizado para determinar o fim do teste e quais seriam os dados necessários para este teste;
- **a descrição dos casos de teste** – como será realizado o teste;
- **os dados de teste** – quais são as informações para o teste.
- **as ferramentas a serem utilizadas** – quais são os recursos necessários à realização do teste;
- **uma estimativa de tempo** – qual o tempo necessário para que o teste seja realizado;

McGregor apresentou, ainda, um modelo genérico para o plano de teste contendo: (a) nome do Componente; (b) nome do responsável pelo teste; (c) os objetivos para a classe; (c) os requisitos de inspeção; (d) a construção e armazenamento dos casos de teste; (e) os casos de teste funcionais; (f) os casos de teste estruturais; (g) os casos de testes baseados em estados; e (h) os casos de teste de interações. Para a execução do plano de teste, faz-se necessária a definição das

estratégias para o teste dos componentes, que tem como meta a redução do número de casos de teste a serem gerados.

Um software orientado a objetos é formado por componentes, que representam uma classe individual ou um *cluster* de classes, com complexidade suficiente para serem testadas individualmente. Uma especificação completa destes componentes, deve abordar os seguintes aspectos:

- **objetos** – a especificação de todos os objetos: (a) que enviam ou recebam mensagens, (b) que são utilizados como parâmetros e (c) que retornam informações;
- **métodos** – suas pré-condições (situações que devem ser verdadeiras para que ele seja executado), suas pós-condições (situações que devem ser verdadeiras após sua execução) e suas invariantes (são as condições que assume-se verdadeiras todo o tempo);
- **classes** – a especificação de todos os métodos e o modelo de estados.

O teste de *software*, na maioria das vezes, faz-se pelo uso dos casos de teste, que representam um conjunto de informações que permitem ao testador simular e conhecer os vários aspectos sobre o funcionamento dos componentes do sistema em teste. Existem três tipos de casos de teste:

- **casos de teste funcional** – informações para o teste das funcionalidades da aplicação;
- **casos de teste estrutural** – informações para o teste dos caminhos do código da aplicação;
- **casos de teste interação** – informações para o teste da interação entre os objetos.

Um dos aspectos importantes na utilização de casos de teste é determinar a abrangência destes casos. Pensando nisto, McGregor apresentou uma seqüência básica para a seleção de casos de teste:

- desenvolver um conjunto de testes funcionais que cubram completamente as especificações da classe;
- desenvolver casos de teste adicionais, baseados em estados, para que todas as transições no modelo dinâmico sejam cobertas;
- utilizar ferramentas para cobertura do teste a ser realizado;
- desenvolver casos de teste estruturais para a cobertura das várias linhas de código;
- desenvolver casos de teste para o teste de interações entre métodos e classes;
- desenvolver casos de teste que cubram a interação entre os objetos das classes em teste com outras classes da aplicação.

Deve-se associar a definição de casos de teste com as características do teste que pretende-se realizar. Na seção seguinte, apresentam-se os níveis de teste de software OO com o objetivo de determinar o que deve ser testado em uma aplicação.

2.3 Níveis de Teste de *Software* OO

Os autores Smith [SMI 92] e McGregor em [MCG 94] e [MCG 96] apresentam os níveis de teste de código, determinado como aplicações orientadas a objeto devem ser testadas:

- **Teste de Classe:** é a menor unidade a ser testada. É formada por um grupo de atributos e comportamentos altamente coesos. Este é o primeiro nível de teste, combina teste tradicional de unidade[13] com alguns aspectos de teste de integração[14];
- **Teste de *Cluster*[15]:** Neste tipo de teste considera-se como aspecto principal, as interações entre as instancias da classe no *cluster*;
- **Teste de Sistema:** O sistema é testado funcionalmente, utilizando-se para isso casos de teste derivados de casos de uso e de outros requisitos do sistema. Com isso, verifica-se a correta execução das tarefas para os quais o sistema foi implementado.

Normalmente, testam-se as classes pela execução de seus objetos. Os testes a serem realizados devem incluir casos de teste que atendam a todas as possibilidades de instanciação de uma classe, incluindo nisto o teste de objetos em todos os possíveis estados iniciais.

Pode-se encontrar alguns erros em classes, mensagens e métodos. Estes, na maioria das vezes, impedindo ou invalidando o processo de teste dos componentes:

a) Erros nas <u>classes</u>:

- alguns estados requeridos na especificação não estão implementados;
- alguns comportamentos requeridos na especificação não estão implementados;
- a especificação da classe não reflete corretamente os conceitos descritos nos requerimentos;
- falhas para a preservação das classes invariantes;
- violação de projeto e padrões de codificação;
- documentação inconsistente com a implementação.

b) Erros nas <u>mensagens</u>:

- parâmetros impróprios para a mensagem;
- recipiente de mensagens não retorna valores apropriados ao remetente ;
- recipiente de mensagens não possui um tratamento apropriado para as exceções do remetente;
- associações indicadas no projeto não fornecidas como uma mensagem na implementação.

[13] Tem por objetivo analisar uma parte do código.
[14] Para o teste de integração entre métodos da classe.
[15] O *cluster* é formado por um conjunto de classes cooperativas.

c) Erros nos métodos:

- erros de sintaxe na linguagem de implementação do método;
- falhas na realização das pós-condições;
- falhas no fornecimento de valores de retorno apropriados;
- tempo de resposta insuficiente para aplicações em tempo real;
- código não executável.

2.3.1 Teste Funcional de Classes

Os casos de teste funcionais são construídos para análise da especificação da classe. A cobertura pode ser expressada em termos de porcentagem de pós-condições ou porcentagem de transições na representação de um estado, sendo cobertos por casos de teste selecionados. A especificação de uma classe agrega as especificações dos métodos desta classe [MCG 96].

Durante a construção de casos para o teste funcional, as pré-condições são utilizadas para estabelecer o tipo de teste apropriado para o objeto. Cada pós-condição representa uma declaração lógica construída como uma seqüência de cláusulas unidas por "*and*" e "*or*" ou uma seqüência de cláusulas "*if* ". Para cada cláusula "*or*", deve-se criar um caso de teste e verificar se o resultado determinou que as cláusulas "*and*" dentro de cláusulas "*or*" foram suficientemente testadas. Casos de teste para tratamento de exceções e condições de limite óbvias, também devem ser criados.

Os critérios de cobertura definidos na Seção 3.1 abordam as características do teste funcional de classes, apresentados nesta seção.

2.3.2 Teste Estrutural de Classes

Neste tipo de teste, constroem-se casos de teste para a análise da implementação interna da classe. A cobertura é expressa em termos da porcentagem de caminhos executados no código testado. A grande dificuldade deste tipo de teste é determinar o número de caminhos necessários para testar-se uma classe de forma eficiente, portanto, utilizam-se os critérios de cobertura do teste. Em algumas situações, dependendo da característica do teste, pode-se utilizar os critérios definidos para o teste procedimental, desde que sejam feitas as adaptações necessárias.

2.3.3 Teste de Interação entre Classes

O conceito de classes introduz um novo escopo, chamado local global. Os múltiplos métodos de uma classe devem ser testados apropriadamente. Um gráfo da classe identifica todos os métodos que enviam mensagens para seus atributos. A interação pode identificar antecipadamente as transações em múltiplos seguimentos de uma representação do estado. O teste de interações desse, também, ocorre durante os testes de *cluster* e *framework* [MCG 96].

Casos de teste para analisar a interação intra_classe são identificados considerando-se os métodos que acessam atributos em comum. Isto pode ser observado utilizando-se um grafo de classe representando a interação entre os métodos e atributos da classe. A cobertura é expressada pela porcentagem de interações testadas. A Figura 2.6, apresenta a relação entre o atributo *largura* e os métodos *set_largura()* e *get_largura()* da classe *A*

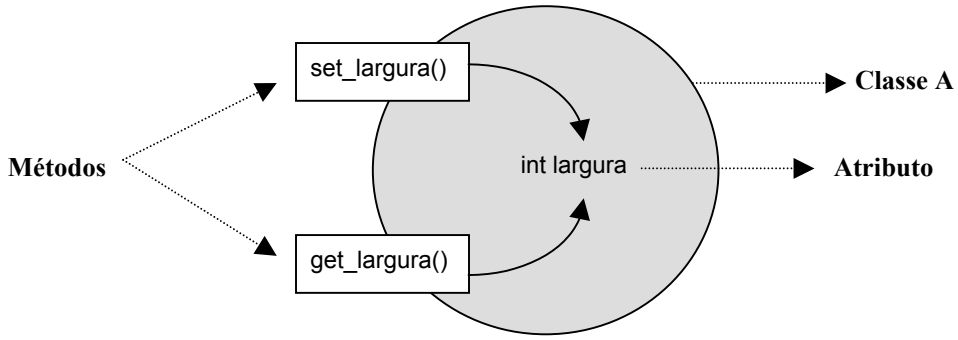


FIGURA 2.6 – Exemplo de um Grafo de Classe para o teste de interação intra-classe.

Os casos de teste para a análise da interação inter-classe são identificados pela troca de mensagens entre classes. O mecanismo de ligação dinâmica[16] torna este nível de teste particularmente importante. Este teste faz parte do teste de integração que enfatiza o mecanismo pelo qual duas ou mais classes interagem através de seqüências em comum ou informações compartilhadas. Um exemplo desta interação pode ser observado na Figura 2.7. Nesta figura, o método *md1()* da classe *A* contém a mensagem *b.md2()*. O método *md2()* pertence à classe *B*, portanto, as classes *A* e *B* devem ser integradas.

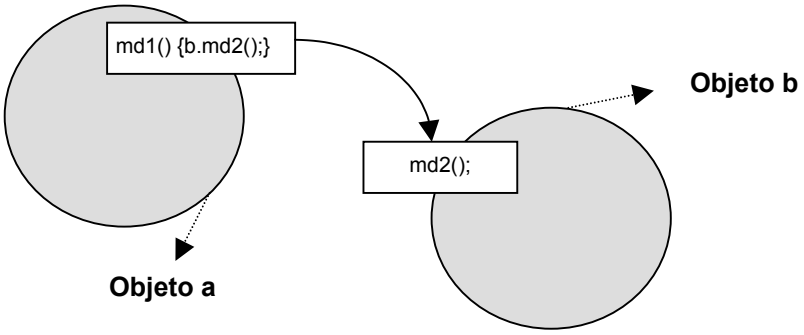


FIGURA 2.7 – Exemplo de um Grafo de Classe para o teste de interação inter-classe.

Os conceitos apresentados para o nível de teste de interação inter-classes foram aplicados na definição dos critérios de cobertura baseados em diagramas de seqüência e colaboração, que estão descritos na Seção 3.3 e 3.4.

[16] Ocorre quando duas ou mais classes interagem pelo compartilhamento de informações.

2.3.4 Teste Baseado em Estados

Durante o tempo em que um objeto permanece instanciado, assume-se que este objeto está em algum estado que satisfaz uma determinada condição, e esta executando alguma atividade[17] ou aguardando a ocorrência de algum evento[18]. Esta condição, reflete o resultado dos eventos caracterizados pela reação do sistema [BOO 99] e [LAR 99].

Um estado é definido como sendo um subconjunto de todas as combinações possíveis dos valores de atributos da classe. As pré-condições, pós-condições e invariantes definem o contrato da classe[19]. As classes cliente[20] definem pré-condições para que uma mensagem seja aceita, enquanto que para as classes servidor[21] são definidas pós-condições para expressar os resultados de uma operação. Uma pós-condição também define o estado resultante da ativação do método. Existe, pelo menos, um estado para cada pós-condição de uma classe. Caso existam *n* operadores lógicos "*ou*" na expressão da pós-condição, o método pode computar *n*+1 estados [BIN 95a].

Segundo Binder, o objetivo do teste de estados é verificar o sistema OO utilizando um pequeno número de combinação de dados sem afetar a confiabilidade, reduzindo, pois, tempo e custo de teste. Como resultado, desenvolveu uma técnica que baseia-se em uma máquina de estados[22] para gerar uma árvore de transição de estados a partir da qual derivam-se os casos de teste [BIN 95a].

Para derivar seqüências de teste de transições, Binder utilizou o procedimento descrito por Chow [CHO 78] que necessita de:

- um modelo completo de máquina de estados finitos;
- um modelo de estados mínimo (sem estados redundantes ou desnecessários);
- um estado inicial;
- a não utilização de estados inatingíveis.

O estado inicial da máquina de estados é transformado na raiz da árvore de transições. Para cada transição partindo do estado raiz, um arco é ligado a um nó que representa o estado resultante[23] na máquina de estados. Isto é repetido para cada nó de estado resultante a menos que: (1) o estado resultante já apareça em um nó anterior; ou (2) o estado resultante seja o final. A Figura 2.8 apresenta o modelo de conversão da máquina de estados para a árvore de transições.

O próximo passo consiste em transcrever seqüências de teste de transições a partir da árvore. Cada caminho a partir da raiz deve dar origem a pelo menos um

[17] Em resposta a um evento.
[18] É a ocorrência de um estímulo capaz de ativar uma transição de estado
[19] Regras (direitos e obrigações) definidas para interação entre a classe e seus clientes [MYE 97] e [FOW 2000].
[20] Que recebem as mensagens.
[21] Que enviam as mensagens.
[22] É um comportamento que especifica as seqüências de estados pelas quais um objeto passa durante seu tempo de vida em resposta a eventos, juntamente com suas respostas a esses eventos [BOO 99].
[23] Estado do objeto depois da transição, ou seja, o estado de destino.

caso de teste. Completa-se o plano de teste com a identificação dos valores de parâmetros dos métodos, estados esperados e exceções de cada transição. O teste consiste em inicializar o objeto a ser testado com o estado inicial e a partir deste, aplicam-se as seqüências de casos de teste para comparar o estado resultante com o estado esperado. Os seguintes tipos de erros podem ser encontrados com a utilização deste tipo de teste: (a) transições omitidas; (b) transições incorretas; (c) ações de saída incorretas; e (d) estados incorretos.

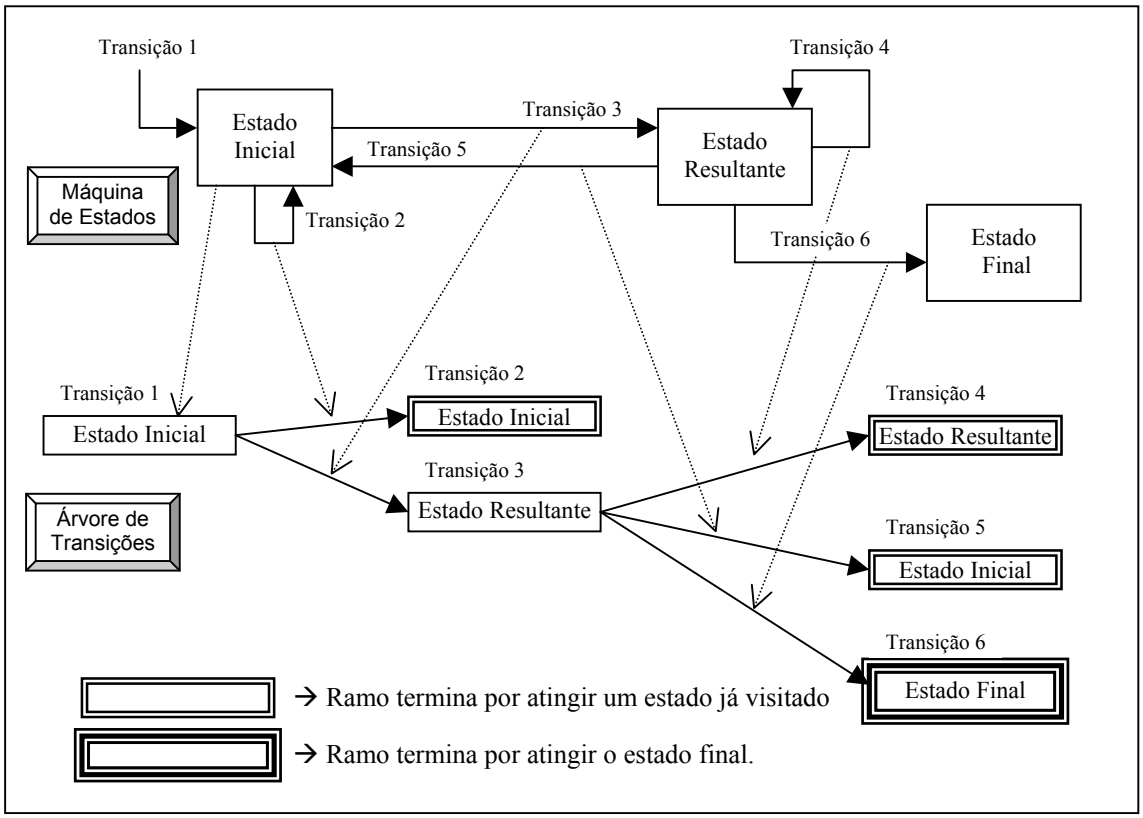


FIGURA 2.8 – Modelo de Conversão da Máquina de Estados para Árvore de Transições.

Demonstrando o que foi apresentado na Figura 2.8, em uma situação real, especificou-se a máquina de estados para representar os possíveis estados de um objeto da classe *Linha Telefônica*. Esta classe possui como atributos: *número*, *cliente*, *débito*, *consumo*, *status* e *dtVencto* e como operações (eventos):

(1) *cadastra* – envia como argumento o *número* e cria a linha telefônica com *cliente*=Telefônica, *status*=disponível, *débito*=0, *dtVencto*=nula e *consumo*=0;
(2) *apropria* – envia como argumento o nomeCliente, será aplicável se *status*=disponível. Modifica o *status* para ligada, *cliente*=nomeCliente e *dtVencto*=dataHoje;
(3) *liga* – aplicável se *status*=desligada. Altera o *status* para ligada e *dtVencto*=dataHoje;
(4) *desliga* – aplicável se *status*=ligado e se *débito*=0 e *consumo*=0. Modifica o *status* para desligada e *dtVencto*=nula;
(5) *paga* – se *status*=ligada, zera o *débito*, atualiza *dtVencto*=dataHoje+30 dias;

(6) *paga* – se *status*=bloqueada, zera o *débito,* atualiza *status*=ligada e *dtVencto*=dataHoje+30 dias;
(7) *bloqueia* - caso dataHoje-*dtVencto*>15 dias, alterar o *status* para bloqueada;
(8) *registraChamada* – envia como argumento o tempo e soma este ao *consumo* existente;
(9) *calculaDébito* – caso a dataHoje=*dtVencto*–10 dias, calcula *consumo* em reais (*débito*=*consumo**preço do segundo) e zera o *consumo*.

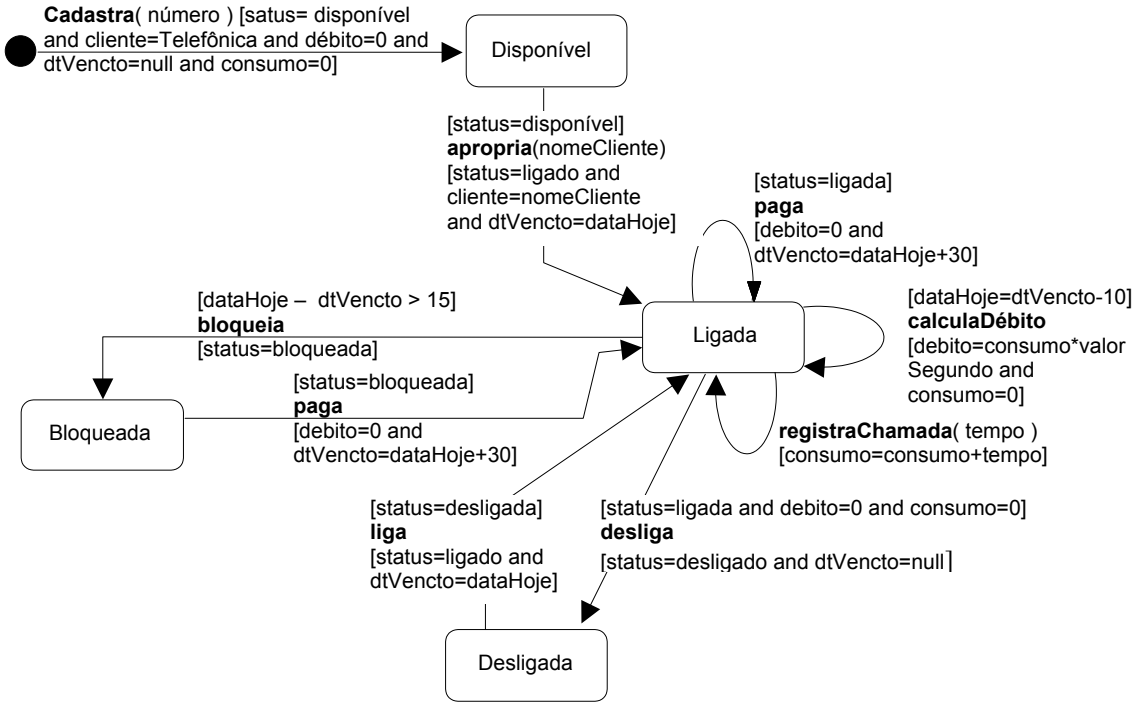


FIGURA 2.9 – Especificação da Máquina de Estados da classe Conta Telefônica.

A árvore de transições gerada à partir desta máquina de estados é apresentada na Figura 2.10. Com isso, pôde-se definir os caminhos para o testa das transições de estado do objeto linha telefônica.

Os casos de teste para os caminhos derivados da árvore de transições da Figura 2.10, são apresentados na Tabela 2.2. Com esta seqüência pode-se testar as transições de estado do objeto linha telefônica.

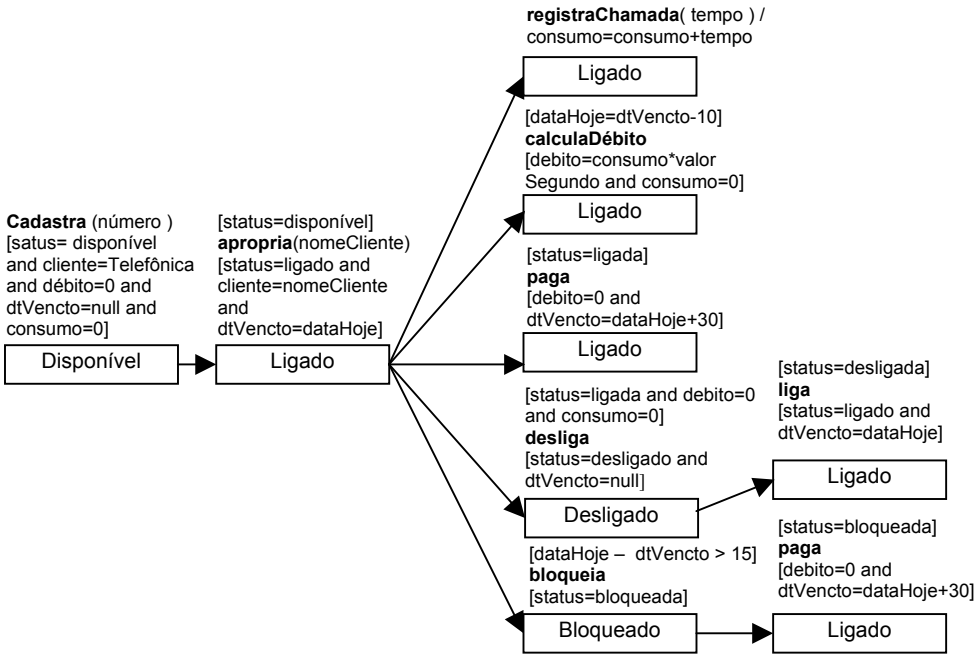


FIGURA 2.10 – Árvore de transições da Máquina de Estados da Figura 2.9.

TABELA 2.2 – Casos de teste para o caminhos derivados da árvore de transições da Figura 2.10.

	1	2	3	4	5
Operação	**Cadastra(1)**	**cadastra(2)**	**cadastra(3)**	**cadastra(4)**	**cadastra(5)**
Dados	(“D”,”Tel”,0,null,0)	(“D”,”Tel”,0,null,0)	(“D”,”Tel”,0,null,0)	(“D”,”Tel”,0,null,0)	(“D”,”Tel”,0,null,0)
Operação	**Apropria(**“L1”**)**	**apropria(**“L2”**)**	**apropria(**“L3”**)**	**apropria(**“L4”**)**	**apropria(**“L5”**)**
Dados	(1,“L”,”L1”,02/10/00)	(2,“L”,”L2”,02/10/00)	(3,“L”,”L3”,02/10/00)	(4,“L”,”L4”,02/10/00)	(5,“L”,”L5”,02/10/00)
Operação	**RegistraChamada(**10**)**	**calculaDébito()**	**paga()**	**desliga()**	**bloqueia()**
Dados	(1)	(02/10/00) (2,0.28, 0)	(“L”) (3,0,02/11/00)	(“L”,0,0) (4,“D”,null)	(20/10/00) (5,“B”)
Operação				**liga()**	**paga()**
Dados				(“D”) (4,“L”, 02/10/00)	(“B”) (5,0,02/11/00)

2.3.4.1 Critérios de Cobertura Baseados em Estados

McGregor [MCG 96] apresentou algumas técnicas de teste de componentes e, dentre elas, o teste baseado em estados. Como forma de efetivar os testes, definiu o seguinte conjunto de critérios de cobertura:

- **Todos-Métodos(*All methods exercised*)**: Este critério assegura que nenhum método necessário fique faltando, mas não garante que todos os estados necessários sejam exercitados. Excute todos os métodos da classe;
- **Todos-Estados (*All states visited*)**: Este critério assegura que nenhum estado fique faltando, mas não garante que todos os métodos sejam exercitados;

- **Todas-Transições (*All transitions exercised*)**: Este critério identifica a ausência de estados e métodos e a presença de estados extras;
- **Todas-N-Transições (*N-way switch cover*)**: Este critério testa as combinações de transições, encontrando estados extras e interrupções desnecessárias;
- **Todos-Caminhos (*All paths followed*)**: Este critério identifica todos os estados e métodos extras. Na prática, devido a sua complexidade, este critério é raramente utilizado.

Juntamente com os critérios de cobertura, McGregor definiu uma lista de verificação ("*checklist*"), para certificar a abrangência de todos os critérios utilizados:

- testar a criação de objetos usando todos os métodos construtores;
- testar pares de métodos auxiliares, dentre eles "*get*" e "*set*";
- testar cada pós-condição de cada método modificador;
- testar cada transição representada no diagrama de estados;
- testar os caminhos através do código;
- testar combinações de métodos que utilizem os mesmos atributos;
- testar se cada objeto libera completamente toda a memória por ele reservada;

O teste baseado em estados, além de servir para detecção de mudanças no estado dos objetos, serve também para detectar a construção correta de uma estrutura de dados complexa e dinâmica, sendo essencial para determinar não só as mudanças específicas na estrutura como também a ocasião em que podem ocorrer.

Utilizando a lista de verificação como referência para o teste, associada a correta aplicação dos critérios de cobertura, McGregor garante que a aplicação estará, na maioria das vezes, isenta de erros.

A técnica de teste e critérios de cobertura abordados nesta seção, foram utilizados como base para a definição das diretrizes e critérios de cobertura de teste baseados nas informações derivadas do diagrama de estados apresentado na Seção 3.4.

2.3.5 Teste de Subclasses

A construção dos casos de teste deve começar do topo da árvore de herança e continuar às folhas, ou seja, da raiz da árvore às subclasses. Este procedimento expande o número de casos de teste quando a interface da classe aumenta, gerando novas oportunidades para o reuso dos casos de teste. McGregor apresenta algumas considerações para teste de subclasses [MCG 96]:

- As subclasses, são especializações das superclasses? – Isto deve ser sempre verdadeiro.
- Os casos de teste utilizados em uma superclasse podem ser reutilizados em suas subclasses? – Inclusive nas novas.
- Quais partes da subclasse devem ser testadas? – o universo de teste na subclasse.

Harrold et al. definiu uma técnica que testa as classes isoladamente reutilizando os casos de teste sempre que possível, de forma incremental na hierarquia

de herança, utilizando uma abordagem *top-down* [HAR 92]. Esta técnica consiste em reutilizar incrementalmente informações de teste de uma superclasse em suas subclasses, atualizando estas informações se necessário. Deve-se iniciar os testes pelas "classes-base" (superclasses) e, para isso, cria-se uma "história de teste" que faz a associação entre os casos de teste os atributos e os métodos que serão testados.

As subclasses herdam atributos, métodos e a história de teste, sendo esta incrementalmente atualizada para mostrar as diferenças existentes em relação à superclasse. Esta técnica indica os atributos e métodos novos da subclasse que devem ser testados, bem como aqueles que devem ser retestados. Alguns atributos herdados devem ser retestados devido ao seu novo contexto na subclasse. Para isso, faz-se o teste de suas interações com os novos atributos da subclasse.

Com a aplicação desta técnica não é necessário o teste completo de cada subclasse, pois a história de teste permite determinar que atributos e métodos devem ser testados e quais casos de teste da superclasse podem ser reutilizados. Isso possibilita a redução dos recursos gastos com o teste de *software*.

2.3.6 Teste de *Cluster*

Para McGregor, um *cluster* é um conjunto de objetos intimamente relacionados que interagem mais entre si do que com outros objetos de uma dada aplicação [MCG 96]. A especificação de um *cluster* deve ser vista com o mesmo formato da classe. Existe um número limitado de mensagens, vindas do ambiente externo, que são enviadas para o *cluster*. Estas mensagens representam os métodos especificados no *cluster*.

Pré e pós-condições podem ser derivadas a partir de cada método da interface, utilizando condições especificadas para cada método em cada classe sendo integrada. A interface do *cluster* é a coleção de mensagens que são enviadas de fora para dentro do *cluster*. A especificação do *cluster* é um subconjunto das especificações dos métodos individuais, já que as mensagens de entrada podem ser enviadas apenas para alguns métodos do *cluster*. Uma visão desta interface é apresentada na Figura 2.11.

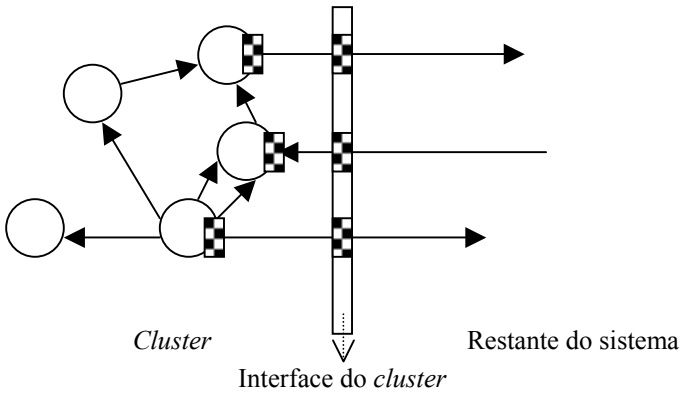


FIGURA 2.11 – Exemplo da Interface do *Cluster* .

Duas importantes características a serem consideradas no teste de *cluster*, são o polimorfismo e a ligação dinâmica. Estas, permitem que uma variável com determinado tipo, ligada a uma classe, assuma outros tipos, durante a execução. Para o teste de interações polimórficas, devem ser seguidas as seguintes etapas:

- identificar o conjunto de possíveis substitutos;
- criar uma classe que possa oferecer a infra-estrutura necessária para o teste;
- testar todos os substitutos caso o conjunto seja suficientemente pequeno;
- utilizar amostragem estática para atingir uma cobertura satisfatória, caso o conjunto seja muito grande.

Segundo os autores McGregor [MCG 94] e Smith [SMI 92], neste nível de teste deve-se considerar que as classes que compõem o *cluster* já tenham sido individualmente testadas. Em resumo, este tipo de teste deve verificar os métodos inter-objetos (de classes diferentes) e as saídas corretas na interação de grupos de objetos cooperantes.

2.4 Teste Baseado em especificações de Software

Com o advento da orientação a objetos, tornaram-se necessários não somente os testes de estrutura e funcionalidade mas, também, técnicas de teste que apoiem todas as fases do desenvolvimento de software, desde a fase de especificação até a implementação. A maioria das técnicas de teste propostas na literatura tratam dos aspectos relativos à implementação de software, desconsiderando sua especificação. Durante o estudo sobre critérios de cobertura, pode-se observar a apresentação de algumas propostas que tratam da realização do teste a partir de sua especificação.

As seções seguintes apresentam propostas de teste baseados em especificações de *software* orientado a objetos.

2.4.1 Proposta de Colanzi e Masieiro para o Desenvolvimento e Teste de Software para UML

Uma abordagem que apoia a atividade de teste durante todas as fases do ciclo de vida do software, apresentada por Colanzi e Masiero [COL 98], [COR 99] e [COL 2000], tem como objetivo estabelecer as regras para a criação de modelos de teste a partir dos modelos gerados nas fases de desenvolvimento (análise de requisitos, análise e projeto, implementação e testes), adequando-se para isto, uma ou mais técnicas de teste baseadas em implementação.

Esta abordagem está baseada em especificações UML, a partir das quais são gerados: (1) modelo de teste dos requisitos; (2) modelo de teste de análise; (3) modelo de teste do projeto; (4) modelo de teste da implementação; e (5) modelo de avaliação do teste (em todas as fases). As características do teste em cada uma das fases são:

- **Engenharia de Requisitos** – gera-se o modelo de teste dos requisitos do sistema em termos de casos de uso tendo como base as informações obtidas dos diagramas de casos de uso e de seqüência.
- **Análise e Projeto** – cria-se o modelo de teste do projeto com base nas informações do modelo de teste dos requisitos e as obtidas dos diagramas de classes, atividade, estado, colaboração e seqüência.
- **Implementação** – cria-se o modelo de teste da implementação, observando-se os aspectos relativos à transformação do projeto em código, permitindo a geração do teste de integração e a criação de um conjunto de testes reutilizáveis nos casos em que exista herança entre classes. Para isso utiliza-se a técnica proposta por Harrold [HAR 92].
- **Testes** – permeia todas as outras fases onde, para cada uma gera-se o modelo de avaliação do teste que utiliza as informações do modelo de teste da fase correspondente.

Na Figura 2.12, é mostrada a interação existente entre essas fases, os modelos de teste e a origem das informações para a geração destes modelos.

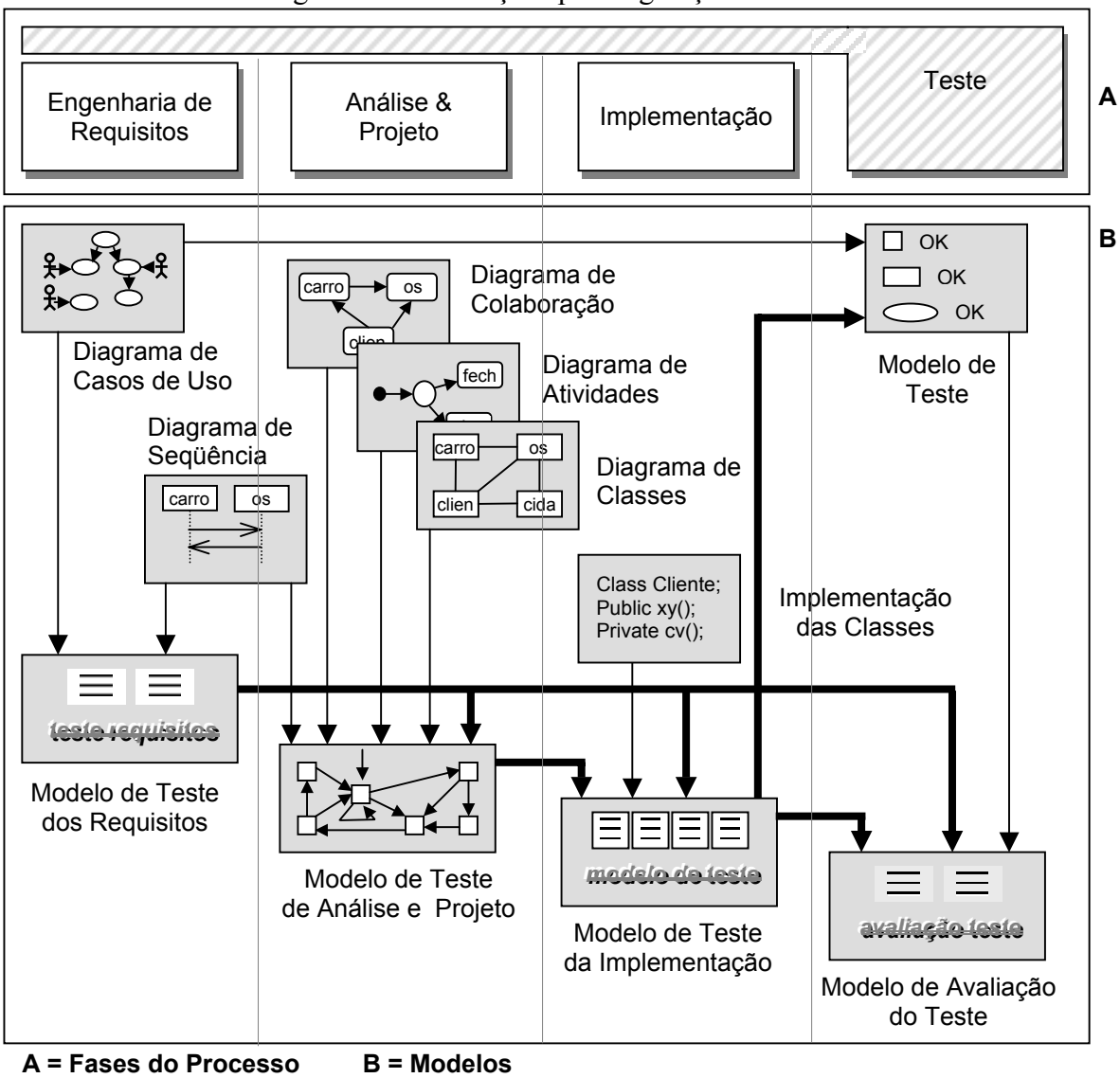


FIGURA 2.12 – Modelos gerados pela abordagem de teste de especificação [COL 98].

Os autores afirmam que para efetivar este tipo de teste deve-se: (a) adequar algumas técnicas e critérios de teste para realizar o teste em todas as fases do ciclo de vida do *software*; e (b) adaptar as técnicas de teste baseadas em implementação para o teste baseado em especificação. Uma das vantagens de incluir a atividade de teste ao longo do processo de desenvolvimento é a possibilidade de testar a especificação e corrigir os erros detectados antes de concluir a implementação do *software*, economizando tempo e custo [COL 2000].

2.4.2 Proposta de Binder baseada nos Cenários de um Diagrama de Seqüência

Binder apresentou uma proposta de teste que visa a geração de dados de teste tendo como base os cenários de diagramas de seqüência [BIN 98]. Para aplicação desta técnica, o autor definiu quatro passos estratégicos: (1) converter o diagrama de seqüência em um grafo de fluxo; (2) identificar os caminhos para o teste a partir do grafo de fluxo; (3) identificar entradas e estados necessários para causar um caminho particular a ser feito; e (4) identificar os resultados esperados para cada teste.

A Figura 2.13 apresenta um diagrama de seqüência que representa o procedimento de inserção de um cartão de crédito em uma hipotética máquina de caixa eletrônico. O banco cobra uma taxa especial para que usuários de outros bancos não filiados, utilizem seus serviços. Caso o cliente não aceite, o cartão é devolvido e a operação é encerrada. Somente depois que todos os dados forem validados, as demais transações serão executadas.

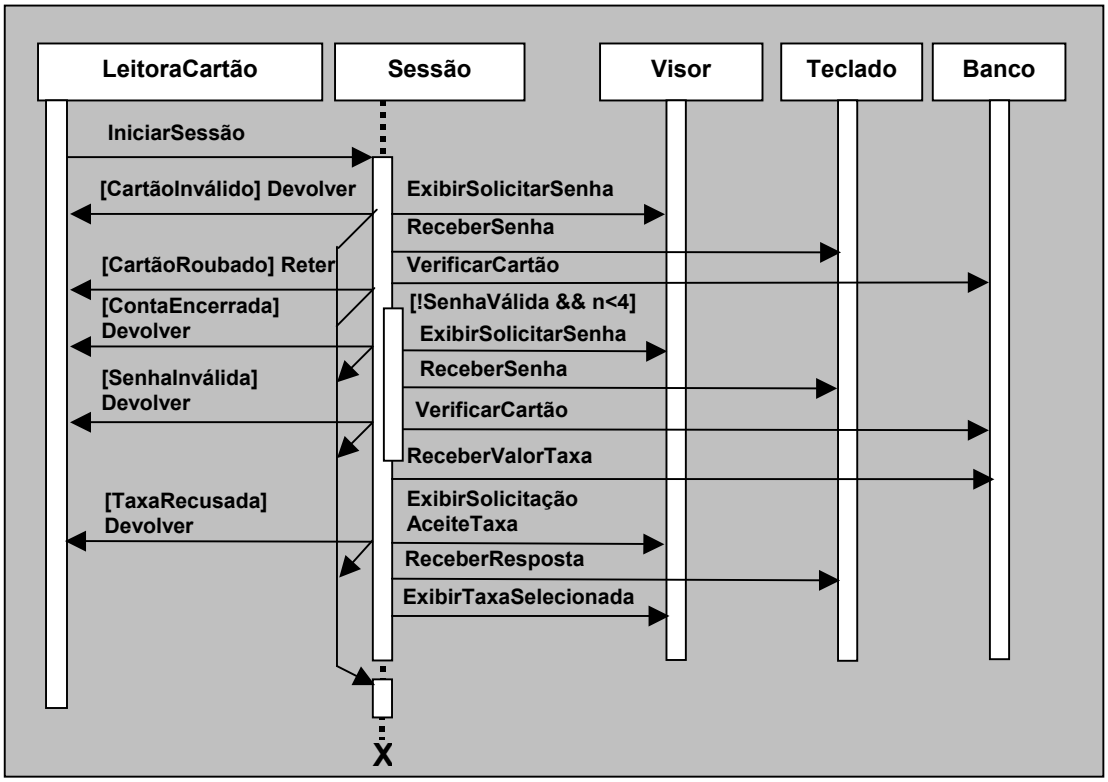


FIGURA 2.13 – Modelo de diagrama de seqüência.

Neste procedimento, algumas condições, são verificadas, a saber: (1) o cartão é do banco proprietário do caixa eletrônico; (2) o cartão foi roubado; (3) a conta está com problemas, ou seja, a conta do cartão não está ativa; e (4) a senha está correta (o cliente possui três chances para acertá-la). Estas condições apresentam diferentes cenários, e estes devem ser testados.

Um cenário corresponde a diversos caminhos em um diagrama de seqüência. Como analisar estes caminhos considerando-se que o diagrama de seqüência não enfatiza fluxos de controle? A solução apresentada por Binder está centrada na conversão do diagrama em um grafo de fluxo, onde os retângulos representam ações, os hexágonos as decisões e as setas o fluxo de execução.

A conversão do diagrama de seqüência para um grafo pode revelar ambigüidades e omissões no diagrama, as quais podem levar o programador a improvisar algum comportamento não apresentado. Isto pode ser observado antes que qualquer caso de teste seja executado. A Figura 2.14 apresenta o grafo de fluxo derivado do diagrama de seqüência apresentado na Figura 2.13.

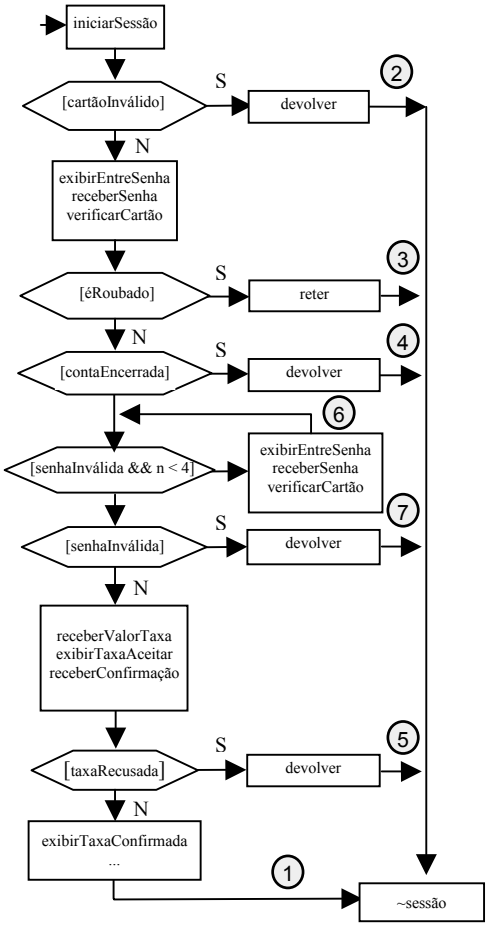


FIGURA 2.14 – Modelo de gráfico de fluxo derivado do diagrama de seqüência.

Na identificação dos caminhos a serem executados no grafo de fluxo, deve-se observar a existência de laços ou ramificações, pois estes, podem levar à geração de inúmeros caminhos de teste que tornariam o teste muito caro. A estratégia

adotada neste teste foi a baseada na cobertura por decisões, onde cada ramo deve ser exercitado pelo menos uma vez. No caso dos laços (*loops*) isto pode ser feito de três formas: (1) evitando o laço; (2) ou passando uma vez pelo laço (número mínimo de repetições); (3) ou passando várias vezes pelo laço (número máximo de repetições).

O próximo passo é a elaboração de casos de teste que ativem a execução de cada caminho particular. Para isso definem-se entradas específicas que atendam à todas as condições estabelecidas para o caminho. A Tabela 2.3 apresenta os casos de teste e as condições dos caminhos do grafo de fluxo.

Para que o conjunto de casos de teste esteja completo, deve-se determinar os resultados esperados com os casos de teste. Isto pode ser feito simulando-se a ação de cada caminho. Tendo essas informações, os caminhos do grafo de fluxo já podem ser testados. O autor salienta que este processo de teste pode ser automatizado, desde que seja implementado em algum ambiente que suporte suas características.

TABELA 2.3 - Casos de teste para os caminhos do grafo da Figura 2.12.

Caminho	Casos de Teste	Comentário
1	CartãoNãoVálido, NãoRoubado, ContaNãoEncerrada, SenhaVálida, TaxaNãoRecusada	Caminho mais longo, todo falso. Laço não realizado.
2	CartãoInválido	Laço não realizado.
3	CartãoNãoVálido, Éroubado	
4	CartãoNãoVálido, NãoRoubado, ContaEncerrada	
5	CartãoNãoVálido, NãoRoubado, ContaNãoEncerrada, SenhaVálida, TaxaRecusada	
6	CartãoNãoVálido, NãoRoubado, ContaNãoEncerrada, SenhaInválida && n == 1, SenhaInválida && n == 2, TaxaNãoRecusada	Mesmo caminho 1 até o laço. Passa uma vez pelo laço.
7	CartãoNãoVálido, NãoRoubado, ContaNãoEncerrada, SenhaInválida && n == 1, SenhaInválida && n == 2, SenhaInválida && n == 3, SenhaInválida	Mesmo caminho 1 até o laço. Passa o número máximo de vezes pelo laço.

Os conceitos apresentados nesta seção, foram utilizados na definição da proposta de implementação do teste baseado no diagrama de seqüência, descrito na Seção 3.3.1. Onde, associando estes conceitos, à proposta de Pressman [PRE 95] para a conversão de um GFC em uma matriz de grafo [BEI 90], pode-se automatizar a geração dos caminhos a serem exercitados durante o teste de cenários.

2.4.3 Proposta de Binder baseada na verificação da Associação entre Classes

A proposta apresentada nesta seção baseia-se no artigo [BIN 97], no qual o autor demonstra a importância em testar-se a associação existente entre classes. Para isso, utiliza a multiplicidade existente como fonte de informações para a geração de dados de teste.

A associação entre duas classes deve implementar os limites mútuos ou dependências existentes. O exemplo utilizado para demostrar o teste foi o problema apresentado por Meiler Page-Jones [PAG 95], que modela um sistema para o cadastro dos cachorros atendidos em uma clínica veterinária, representando a relação existente entre cachorros e seus donos. Este relacionamento pode ser observado na Figura 2.15 onde uma pessoa pode não possuir cachorro ou ter vários e, em contra partida, um cachorro pode não ter dono ou pertencer somente a um dono.

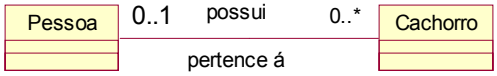


FIGURA 2.15 – Multiplicidade da relação Pessoa-*possui*-Cachorro.

O autor apresenta, também, uma variação na relação Pessoa-*possui*-Cachorro para demonstrar as possibilidades de mudança em uma mesma associação. Para tal, utilizou como exemplo uma hipotética cidade chamada *Kaynine,* na qual toda pessoa deveria possuir pelo menos dois e não mais que quatorze cachorros. Para completar, em uma assembléia municipal, o governo decidiu que todo cachorro seria possuído por somente uma pessoa. Este exemplo pode ser observado na Figura 2.16.

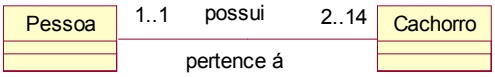


FIGURA 2.16 – Multiplicidade da relação Pessoa-*possui*-Cachorro na cidade de *Kaynaine.*

a) Erros de Associação

Como em qualquer tipo de teste, inicialmente, deve-se considerar como provável a incorreta implementação das associações, simplificando com isso, a busca por erros. O autor salienta que mesmo nos casos em que a associação seja simples, a implementação pode ser feita de diversas maneiras. Isto apresenta oportunidades diferentes para a ocorrência de erros, requerendo testes estruturais diferenciados para atender às características de cada uma delas. Pode-se planejar os testes de associações de duas formas: (1) utilizando-se cenários prováveis e improváveis; e (2) examinando-se os casos de uso da aplicação. Utiliza-se a multiplicidade da associação como fonte de informações para a geração de casos de teste que verifiquem a correta implementação de suas características. Binder apresentou alguns dos prováveis erros em associações:

- **Multiplicidade Incorreta**: a implementação da associação permite a rejeição ou aceitação de uma combinação de instâncias ilegal.
- **Anomalias na Alteração**: a ligação que associa diversas instâncias de uma classe com uma única instância de outra classe possui alguma informação redundante.

Quando alguma mudança for feita em uma instância de classe, as instâncias associadas também devem ser modificadas.

- **Anomalias na Eliminação**: O que acontece aos objetos associados à uma classe caso esta seja eliminada? Deve-se eliminá-los automaticamente ? Dependendo da forma com que o sistema foi projetado, a implementação deverá encontrar, remover ou alterar cada uma das associações.

b) Teste de Multiplicidade

Utiliza-se nos casos de teste, baseados em multiplicidade, os valores mínimo e máximo de cada associação. Assim, pode-se verificar se os valores encontram-se: (a) dentro do limite; (b) fora do limite – mas próximo deste; e (c) fora do limite – por uma grande diferença.

Para demonstrar a aplicação dos casos de teste, Binder [BIN 97] utilizou a associação Pessoa-*possui*-Cachorro da Figura 2.15. Como as associações são tipicamente bidirecionais, deve-se fazer os teste nas duas direções. Para tal, utilizando o mesmo padrão de testes, foram criados casos de teste para a associação Cachorro-*pertence á*-Pessoa, apresentando-os nas Tabelas 2.4 e 2.5.

TABELA 2.4 - Teste da Associação Pessoa-*possui*-Cachorro.

Valor Teste Multiplicidade	**N°de Pessoas**	**N°de Cachorros**
Mínimo	1	0
Mínimo + 1	1	1
Média do limite	1	127
Máximo –1	1	255
Máximo	1	256
Mínimo – n	1	-3
Mínimo – 1	1	-1
Máximo +1	1	257
Máximo + n	1	32767

TABELA 2.5 - Teste da Associação Cachorro-*pertence á*-Pessoa.

Valor Teste Multiplicidade	**N°de Cachorros**	**N°de Pessoas**
Mínimo	1	0
Mínimo + 1	1	1
Média do limite	1	1
Máximo –1	1	1
Máximo	1	1
Mínimo – n	1	-3
Mínimo – 1	1	-1
Máximo +1	1	2
Máximo + n	1	79

Torna-se difícil a geração de condições, caso estas estejam fora dos limites especificados na multiplicidade, mesmo assim deve-se testá-las. Quando o valor do parâmetro máximo não é especificado, deve-se utilizar como referência o valor máximo suportado pelo tipo de dado *"int"* da linguagem que está sendo utilizada para a construção da aplicação.

O padrão para a geração de casos de teste apresentado nesta seção, faz uso da técnica BVA descrita na Seção 2.1.2.2 e utilizada no teste funcional de *software*.

c) Teste para a Anomalia de Alteração e/ou Eliminação

Uma vez que as restrições de multiplicidade estejam corretamente implementadas, verifica-se a existência de anomalias de alteração ou de eliminação. Para isso, deve-se gerar casos de teste que, por hipótese, eliminem e alterem a associação entre instâncias da classe como forma de observar-se o comportamento da aplicação. Neste caso, as modificações em uma instância, devem ser automaticamente repassadas às demais associadas.

A técnica de teste apresentada nesta seção, bem como suas características, foram utilizadas na definição das diretrizes e critérios de cobertura de teste baseados nas informações derivadas da especificação de multiplicidades em um diagrama de classes.

No próximo capítulo serão apresentados os critérios de cobertura e as diretrizes de teste definidas neste trabalho. Citando-se as características da Linguagem de Modelagem Unificada (UML); as particularidades dos diagramas implementados pela UML e que serviram de base para a definição destes critérios e diretrizes de teste; as peculiaridades das informações disponibilizadas pelos diagramas e seus componentes; a forma como os conceitos apresentados nos capítulos anteriores aplicam-se a este tipo de teste e sugere-se ainda uma proposta de implementação e automação da geração de caminhos de teste tendo como base os critérios definidos para cada um dos diagramas.

3 Critérios de Cobertura e Diretrizes de Teste para Diagramas UML

Este capítulo tem por objetivo descrever as diretrizes e os critérios de cobertura de teste propostos nesta dissertação. Utilizou-se para isso, os conceitos apresentados nos capítulos anteriores e alguns dos diagramas definidos na Linguagem de Modelagem Unificada – UML.

A UML tornou-se popular entre os projetistas de sistemas de informação e vem sendo utilizada em um grande número de empresas de desenvolvimento de *software* orientado a objetos e no meio acadêmico [BOO 99], [LAR 99] e [ERI 98]. UML surgiu a partir da fusão dos conceitos definidos por Grady Booch, Ivar Jacobson e James Rumbaugh. Trata-se de uma linguagem para a especificação, construção, visualização e documentação de artefatos, gerados a partir de algum processo de desenvolvimento de *software*. Um dos principais objetivos da UML é fazer com que equipes de desenvolvimento de *software* utilizem a mesma linguagem como instrumento de comunicação e modelagem de coisas diferentes. Segundo Larman [LAR 99], a UML é um sistema de notação (que inclui semântica também) dirigida à modelagem de *software*, usando como base os conceitos de orientação a objetos.

Fowler [FOW 2000] apresenta algumas justificativas para a utilização da UML:

- permite comunicar certos conceitos mais claramente do que as linguagens alternativas – a linguagem natural é imprecisa, complicada no tratamento de conceitos mais complexos;
- ajuda a obter uma visão geral do sistema com a utilização de um pequeno grupo de diagramas – utilizando o diagrama de classes podem ser visualizadas as abstrações envolvidas no sistema e pelos diagramas de interação verifica-se a colaboração entre estas abstrações (classes);
- disponibiliza ferramentas para o mapeamento de grandes sistemas – através do uso do conceito de pacotes (*packages*) para apresentar as principais partes do sistema e suas interdependências;
- não define um processo-padrão – disponibiliza uma notação básica, sem definir os passos a serem seguidos na elaboração de um projeto.

Levando-se em consideração o que foi apresentado por Fowler [FOW 2000] e as colocações de Booch [BOO 99] da OMG [OMG 99] e de Larman [LAR 99], conclui-se que a UML pode ser utilizada em todas as fases do ciclo de desenvolvimento de sistemas, desde a especificação de requisitos até os testes finais do sistema. Para isso, implementa até nove tipos de diagramas[24], que representam a estrutura estática[25], dinâmica[26] e de implementação[27], dos produtos para os quais tenha sido utilizada como ferramenta de modelagem.

---

[24] Diagramas de: classes, objetos, casos de uso, seqüências, colaboração, atividades, estados, componentes e implantação.

[25] Representada pelos diagramas de classes e objetos.

[26] Representada pelos diagramas de caso de uso, seqüência, colaboração, atividade e estado.

[27] Representada pelos diagramas de implantação e componentes.

Dentre os diagramas definidos pela UML selecionou-se os diagramas de classes, de estado, de seqüência e de colaboração, de acordo com os seguintes critérios:

- **facilidades para a geração de dados de teste** – as características e os elementos que compõem esses diagramas possuem a maioria dos dados para a análise de uma aplicação, destacando-se as classes, os atributos, os métodos, os relacionamentos e as mensagens;
- **adequação às técnicas de teste** – esses diagramas permitem a aplicação das técnicas teste encontradas na literatura;

O diagrama de casos de uso é uma importante ferramenta para a verificação da compatibilidade do que foi especificado em relação aos requisitos exigidos pelo sistema e seus usuários. Utilizam-se os casos de uso para auxiliar na validação da arquitetura e verificação do sistema à medida que o mesmo evolui durante seu desenvolvimento. Seu principal objetivo é a representação da interação entre os agentes externos (usuários) e o sistema, permitindo a visualização de todo o sistema como sendo uma caixa preta, ou seja, é possível visualizar o que está fora do sistema e como este reage a algo externo, mas não se pode visualizar o seu funcionamento interno.

Neste trabalho, o diagrama de casos de uso será utilizado somente para especificar o comportamento desejado do sistema, fornecendo as interações (cenários) necessárias para a construção dos diagramas de seqüência e colaboração.

Considerando que na fase de especificação a implementação ainda não existe, desenvolveu-se um conjunto de diretrizes para a realização da verificação entre as informações extraídas dos diagramas e a posterior implementação do sistema. Estas diretrizes serão apresentadas nas seções seguintes, onde forem abordadas as características mais comumente utilizadas em cada um dos diagramas selecionados para este trabalho.

Em sua maioria, os critérios de cobertura definidos neste capítulo, fazem referência a critérios já definidos por outros autores, dentre eles os de McGregor [MCG 96], Myers [MYE 79] e Rapps [RAP 85] descritos nas seções 2.3.4.1 e 2.1.1.1, respectivamente. Nas seções seguintes, são apresentadas as diretrizes de verificação e os critérios de cobertura para o teste de software com base nas especificações diagramáticas UML.

3.1 Critérios de Cobertura e Diretrizes para o teste baseado no Diagrama de Classes

O diagrama de classes permite a visualização da estrutura estática do sistema, apresentando o conjunto de classes e seus relacionamentos. Segundo Booch [BOO 99], a perspectiva estática oferece, principalmente, suporte para especificar os requisitos funcionais de um sistema, ou seja, os serviços que o mesmo deverá fornecer aos usuários finais. Neste sentido, utiliza-se o diagrama de classes para:

- **modelar o vocabulário de um sistema**, envolvendo a decisão de quais abstrações irão compor o sistema, especificando suas características e responsabilidades;
- **modelar colaborações simples**, visualizando e especificando o conjunto de classes que funcionam de forma cooperada para a realização de uma semântica maior do que cada uma delas individualmente e;
- **modelar o esquema lógico de um banco de dados**, definindo a estrutura na qual pretende-se armazenar informações persistentes em um banco de dados relacional ou orientado a objetos.

Os usos do diagrama de classes estão ligados às perspectivas de projeto direcionadas por Cook e Daniels [COO 94] e apresentadas por Fowler [FOW 2000]. Segundo o que foi apresentado por estes autores, existem três perspectivas, conceitual, especificação e implementação, as quais podem ser utilizadas em diagrama de classes, ou qualquer outro modelo. Descreve-se a seguir alguns aspectos destas perspectivas:

- **Conceitual** - nesta perspectiva, representam-se os conceitos no domínio que está sendo estudado. Sendo estes conceitos naturalmente relacionados a concepção das classes que irão executá-los. Na verdade, um modelo conceitual é projetado com pouca ou nenhuma preocupação com a linguagem a ser utilizada na implementação do *software*.

- **Especificação** - examina-se o *software* analisando-se suas interfaces, não sua implementação, ou seja, como as classes devem ser visualizadas, seus relacionamentos e particularidades. Neste nível de modelagem já existe uma pequena preocupação com a implementação, mas não com a linguagem ou banco de dados (no caso de classes persistentes) a ser utilizado.

- **Implementação** - nesta visão, considera-se a existência real das classes, com isso, a implementação é a grande preocupação. Neste sentido, procura-se adequar o modelo à linguagem de programação e banco de dados utilizados na construção do *software*.

As linhas que separam estas perspectivas não são muito rígidas e a maioria dos projetista de *softwares* não se preocupa em diferenciá-las. Isto não afeta as perspectivas conceitual e de especificação mas, deve-se distinguir principalmente a de especificação em relação à de implementação, pois acredita-se que uma especificação não deve ser direcionada a uma linguagem. Ou seja, o modelo desenvolvido deve adequar-se a qualquer tipo de ambiente, daí a necessidade da perspectiva de implementação.

Neste trabalho, foram consideradas as perspectivas conceitual e de especificação, permitindo que na visão de implementação o desenvolvedor possa adequar as características do teste de acordo com a linguagem e com o banco de dados que forem utilizados na implementação do *software*.

O diagrama de classes possui diversas notações dentro da UML. Para este trabalho, foram considerados os elementos mais utilizados na modelagem de sistemas e estes fazem parte da notação básica e alguns elementos da notação avançada. Uma breve descrição de cada um destes elementos será apresentada nas seções seguintes,

onde os mesmos forem tratados. Nestas descrições, utilizou-se os conceitos apresentados por Fowler [FOW 2000], por Booch [BOO 99] e pela OMG [OMG 99].

Alguns aspectos relativos ao diagrama de classes são indispensáveis na construção de um sistema, dentre eles, a multiplicidade, as particularidades de cada tipo de relacionamento e a classe com a definição de métodos e atributos. Tendo isto como base, elegeu-se os elementos a serem avaliados e para os quais foram definidos critérios e/ou diretrizes: atributos, métodos, relacionamentos(associação, generalização e agregação), multiplicidades, condições e restrições.

No diagrama de classes, como o próprio nome indica, a classe é o elemento mais importante. Por isso, suas características devem ser bem definidas, principalmente métodos, atributos e relacionamentos. A Figura 3.1 apresenta um exemplo de diagrama de classes construído para representar os principais conceitos considerados nesta modalidade de teste.

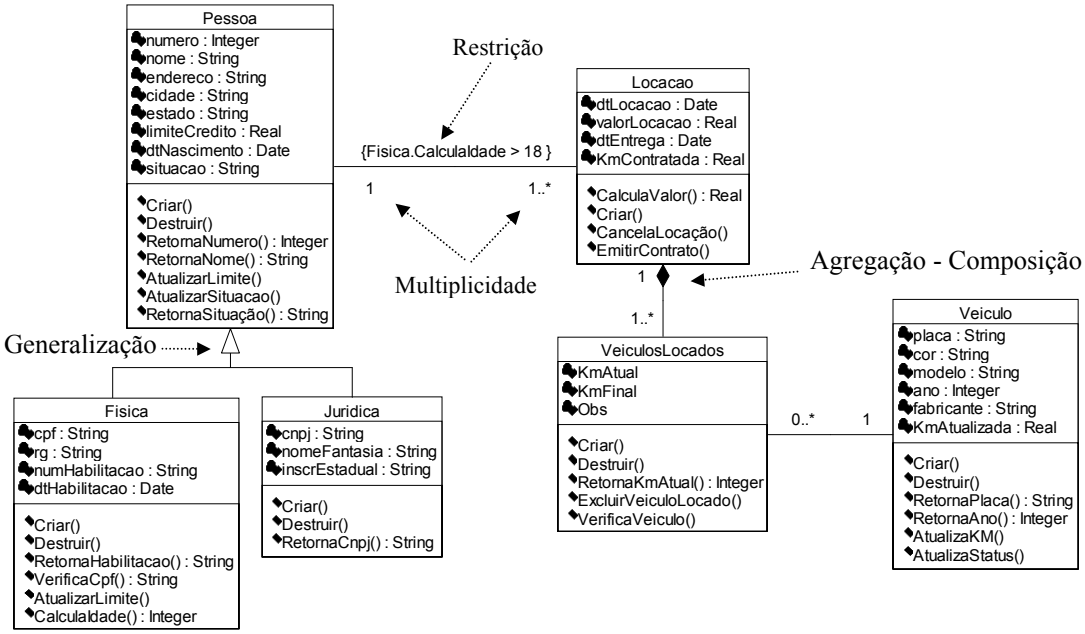


FIGURA 3.1 – Exemplo de Diagrama de Classes Representando a Locação de Veículos em uma Locadora.

Alguns projetistas de *software*, quando constróem a especificação de seus sistemas utilizam todos os recursos disponíveis na ferramenta de modelagem. Outros utilizam apenas o necessário para a compreensão do modelo. Logo, quando utiliza-se os recursos mínimos, algumas informações importantes podem ser suprimidas, como por exemplo: (1) as restrições de associação[28]; (2) as restrições dos atributos[29]; (3) a lista de parâmetros dos métodos e; (4) as pré e pós condições dos métodos.

[28] Especifica condições que devem ser mantidas como verdadeiras para que o modelo seja bem construído [BOO 99] e [OMG 99].

[29] Pertence as propriedades do atributo, especifica o domínio de valores válidos para o atributo, normalmente são expressas por condições [BOO 99].

Ao testar classes deve-se analisar: (a) a declaração de todos os atributos inseridos; (b) a implementação de todos os métodos concretos; (c) a declaração de todos os métodos abstratos; e (d) seus inter e intra-relacionamentos [MCG 94] e [MCG 96].

3.1.1 Derivando informações para a verificação dos Atributos da Classe

Nesta seção, serão apresentadas as informações que podem ser extraídas do diagrama de classes para a análise dos atributos de cada classe, com o objetivo de auxiliar o desenvolvedor no momento em que for testar o *software* que está sendo implementado. Estas informações dizem respeito às características dos atributos (visibilidade, tipo, valor inicial e propriedades) que devem ser verificadas.

Atributo é uma propriedade nomeada de uma classe, representa alguma propriedade do item que está sendo modelado, o qual é compartilhado por todos os objetos dessa classe. Em sua forma plena, a sintaxe de um atributo na UML é :

visibilidade nome [multiplicidade] :tipo =valor-inicial {propriedade}

a) **Visibilidade**: indica a forma como o mesmo poderá ser acessado podendo ser:

- pública (+) - significa que todos têm acesso ao atributo, podendo ser referenciado por métodos de outras classes - esta é a visibilidade padrão (*default*);
- protegida (#) - significa que o atributo pode ser acessado somente por métodos dentro da mesma classe e por métodos que pertençam a classes do pacote no qual a classe foi definida;
- privada (-) - significa que o atributo somente será acessado por métodos definidos na mesma classe.

As características da visibilidade, com exceção da pública, podem variar de acordo com a linguagem utilizada para implementar o *software*.

b) **Nome**: conjunto de caracteres que representa o nome do atributo.

c) **Multiplicidade**: deve ser colocada entre colchetes, identifica a possibilidade (mínima e máxima) de valores para o atributo, podendo ser suprimida quando for exatamente um ([1..1]).

d) **Tipo**: Indica o tipo do atributo, depende da linguagem de programação que será utilizada para implementar o *software*.

e) **Valor-Inicial**: é opcional, representa o valor inicial para a criação de objetos, também depende da linguagem de programação utilizada.

f) **Propriedade**: quando utilizada deve ser expressa entre chaves "{}", indica as propriedades a serem aplicadas ao atributo. Nas propriedades podem ser especificadas a descrição do atributo, tipo de atributo (*changeable, addOnly, frozen*), domínio de valores e restrições. Quanto ao tipo de atributo: (1) {*changeable*} - indica que não

existem restrições para modificar o valor do atributo; (2) {*addOnly*} - utilizado quando a multiplicidade do atributo for maior que um, neste caso valores poderão ser adicionados, não podendo ser removido ou alterado; (3) {*frozen*} - depois que o objeto for criado, seu valor não poderá ser modificado. Quando não especificado o tipo o valor padrão será {*changeable*}.

Com exceção da visibilidade e do nome, todos os demais elementos da sintaxe de um atributo podem ser suprimidos. Logo, algumas informações importantes para o teste podem ser suprimidas. Daí a importância em construir-se diagramas completos, o processo pode ser longo mas facilita o uso posterior das informações. Com base nisto, a maioria dos autores salienta que mesmo com o grade número de recursos disponíveis para a modelagem de *software*, o sucesso está associado a mudança na forma de agir das equipes de desenvolvimento de *software*.

As seguintes diretrizes foram definidas e devem ser seguidas quando o desenvolvedor for verificar os atributos especificados em uma classe em relação ao que foi implementado:

- verificar se os atributos especificados no modelo foram devidamente declarados na implementação da classe;
  - observar se a visibilidade dos atributos obedece à definida na especificação da classe;
  - verificar se o tipo do atributo declarado é igual ao especificado na classe;
- verificar se as condições da restrição do atributo, quando declaradas, foram devidamente implementadas nos métodos que alteram o valor deste atributo;
- nos casos em que for especificado valor inicial para o atributo, verificar se esta característica foi corretamente implementada;
- verificar se o atributo foi implementado de acordo com sua propriedade (*changeable, addOnly, frozen*). Caso esta seja declarada.

Na Figura 3.2 são apresentados os elementos a serem verificados, onde de um lado tem-se a especificação dos atributos da classe e do outro, parte do código em *ObjectPascal* referenciando a implementação da mesma. Com isso, pode-se comparar o que foi especificado com o que foi implementado, utilizando-se as diretrizes para a verificação de atributos.

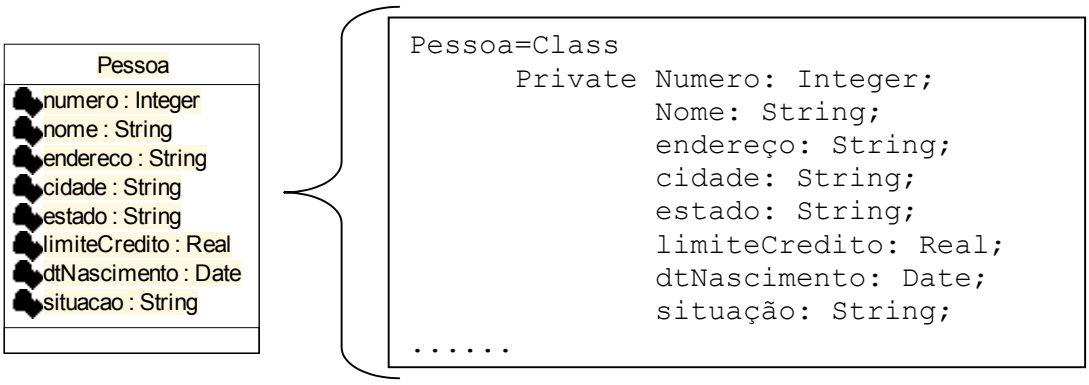


FIGURA 3.2 – Elementos utilizados na verificação dos atributos da classe.

A verificação da restrição de um atributo deve ser feita, primeiramente, na forma funcional, ou seja, desconsiderando-se a implementação, depois analisando-se o código dos métodos implementados, que manipulem o atributo. Neste caso, para facilitar o teste, deve-se utilizar a cobertura por condições e combinação de condições, definidos na próxima seção. Nos casos em que não seja especificada nenhuma restrição para o atributo, deve-se testá-lo, pela atribuição de valores, de acordo com as características de tipo do atributo (*integer, string, data, boolean*) e limitações impostas pela linguagem a ser utilizada. Na Seção 3.1.3.1 são definidos os critérios de cobertura para o teste dos atributos das subclasses no relacionamento do tipo generalização (herança).

3.1.1.1 Cobertura de Teste de Condições e Combinação de Condições

As condições envolvendo dados ou variáveis são propriedades fundamentais para a análise de informações. No atributo são chamadas de restrições e quando declaradas, expressam o valor aceitável para que este exista. Nos métodos, as condições são apresentadas como pré-condições (valores válidos para que o método seja executado) e pós-condições (valores esperados com a execução do método). Nas mensagens as condições expressam os valores válidos para que estas sejam enviadas. Já para as transições de estado dos objetos, indicam o valor válido para que a mesma ocorra [FOW 2000] e [BOO 99].

Como apresentado por Tai [TAI 89], uma condição pode ser simples ou composta. Sendo simples, a mesma pode ser uma variável lógica ou uma expressão relacional. Uma expressão relacional é apresentada como *E1* <*operador-relacional*> *E2* onde, *E1* e *E2* são expressões aritméticas e o <*operador-relacional*> pode ser "<", "<=", "=", "<>", ">" e ">=". Uma condição composta ou combinação de condições é composta por duas ou mais condições simples, operadores lógicos e parênteses. Operadores lógicos são: "ou" (*OR*), "e" (*AND*) e "não" (*NOT*).

O teste de condições, segundo Tai [TAI 89], pode encontrar os seguintes tipos de erro:

(a) erro de operador lógico (*boolean operators*) – o operador lógico especificado não é o correto;
(b) erro de variável lógica (*boolean variable*) – a variável não foi corretamente especificada;
(c) erro de parênteses lógicos[30] (*boolean parentheses*) – os parênteses foram colocados na ordem incorreta;
(d) erro de operador relacional (*relational operators*) – o operador relacional utilizado não é o correto;
(e) erro de expressão aritmética (*arithmetic expression*) – a expressão aritmética não corretamente declarada.

Uma das formas existentes para direcionar o teste é a utilização de critérios que disponibilizam ao desenvolvedor meios de reduzir o número de casos de teste a serem exercitados. Considerando o que foi apresentado, desenvolveu-se um conjunto de novos critérios sobre condições e/ou combinações de condições. Partiu-se do

[30] Parênteses colocados ao redor de uma condição simples ou composta.

critério "**Cobertura por Condições**" baseado no fluxo de controle, aplicado às técnicas do teste particionamento por equivalência e análise do valor limite (BVA)[31], definidas por Myers [MYE 79].

A verificação das condições deve ser feita com base em um conjunto de dados de teste previamente definidos e em conformidade com o contexto em que as condições estiverem inseridas. Na Tabela 3.1, é apresentado um exemplo de ficha definido para a verificação de condições simples (sem operadores lógicos). Neste caso, avalia-se a restrição do atributo ***dtNascimento*** de uma pessoa em um sistema de locação de veículos. Em conformidade com as regras pré-estabelecidas, uma pessoa somente será cadastrada se possuir no mínimo dezoito anos de idade.

A ficha de avaliação de condições é utilizada para verificar os limites da expressão aritmética (valor da condição) de acordo com o operador relacional utilizado. Verifica-se o limite, limite+1 e limite-1. Estes valores de teste podem ser utilizados na elaboração da combinação para o teste de condições múltiplas.

Deve-se verificar funcionalmente a corretitude da condição comparando o resultado esperado com o resultado obtido. Com isso pode-se determinar a presença de erros.

TABELA 3.1 – Exemplo de ficha de avaliação da condição simples da restrição do atributo ***dtNascimento***.

Atributo: ***dtNascimento***		*Data Atual:* ***29/03/2000***		
Restrição(ões): (1) ***dtNascimento >= 18 anos***				
*Parâmetro**		*Valores de Teste*	*Resultado Esperado*	*Resultado Obtido*
1	*Limite*	***29/03/1982***	***V***	
2	*Limite +1*	***29/03/1983***	***V***	
3	*Limite - 1*	***29/03/1981***	***F***	

* Em relação ao atributo sendo verificado.

Quando em uma condição não forem utilizados valores numéricos, testá-la com um valor verdadeiro e outro falso. Como exemplo a condição *situação=”Locado”* testar com "*Locado*" e "*Disponível*".

Os resultados em uma condição múltipla podem variar em função do operador lógico utilizado. O número de casos de teste para a combinação é calculado em função do número de condições na combinação. Para que seja feito o teste, primeiramente devem ser verificados individualmente, pela ficha de avaliação de condições simples, os limites de cada condição e depois estes valores devem ser combinados e submetidos a teste. Na Tabela 3.2 pode-se observar a ficha para avaliação da restrição com condições múltiplas do atributo ***dtNascimento***. Considerou-se que a empresa de locação de veículos, resolveu cadastrar somente clientes que possuíssem idade mínima de dezoito anos e máxima de setenta anos, e que fossem habilitados a pelo menos 4 anos.

[31] Vide Seção 2.1.2.2 Método análise do valor limite.

TABELA 3.2 – Exemplo de ficha de avaliação de condição composta para a restrição do atributo ***dtNascimento***.

Atributo(s): ***dtNascimento e tempoHabilitação***	*Data Atual:* ***29/03/2000***

Restrição(ões): ***(dtNascimento >= 18) e (dtNascimento <= 70) e (tempoHabilitação >= 4)***

	Combinação			**Resultado Esperado**	**Resultado Obtido**
	>= 18 anos	*<= 70 anos*	*>= 4 anos*		
(a)	***29/03/1982***	***29/03/1930***	**4**	**V**	
(b)	***29/03/1983***	***29/03/1929***	**3**	**F**	
(c)	***29/03/1982***	***29/03/1930***	**3**	**F**	
(d)	***29/03/1982***	***29/03/1929***	**3**	**F**	
(e)	***29/03/1983***	***29/03/1930***	**4**	**F**	
(f)	***29/03/1983***	***29/03/1929***	**4**	**F**	
(g)	***29/03/1983***	***29/03/1930***	**3**	**F**	
(h)	***29/03/1982***	***29/03/1929***	**4**	**F**	

Levando-se em consideração o que foi apresentado anteriormente e tentando minimizar o número de casos de teste necessários para verificar-se as combinações de condições, definiu-se um conjunto de critérios de cobertura para cada um dos operadores lógicos "*e*" e "*ou*". A estes critérios atribuiu-se o nome de "Critérios de Cobertura por Combinação de Condições".

Os critérios de cobertura definidos para o operador lógico **"e"** foram:

- **Todas-Combinações-E** – Todas as condições de uma combinação devem ser testadas pelo menos uma vez. A Tabela 3.3, apresenta um exemplo da aplicação deste critério, onde avalia-se uma combinação com duas condições (C1=1 e C2=1).

TABELA 3.3 – Exemplo de combinação para teste de condições utilizando operador lógico **"e"**.

	C1	**C2**	**Resultado**
(a)	**1**	**1**	**V**
(b)	**0**	**0**	**F**
(c)	**1**	**0**	**F**
(d)	**0**	**1**	**F**

- **Combinação-VF** – Testar uma combinação que resulte verdadeiro e uma que resulte falso. Utilizando-se o exemplo da Tabela 3.3, deve-se testar a combinação (a) e qualquer uma de (b) a (d). Caso a combinação possua mais de duas condições é importante que seja testada uma combinação toda verdadeira, uma toda falsa e uma outra qualquer. Pelo exemplo da Tabela 3.2, testar a combinação ***(a)***, ***(b)*** e qualquer uma de ***(c)*** a ***(h).***

Para o operador lógico **"ou"**, foram definidos os seguintes critérios:

- **Todas-Combinações-OU** – Todas as condições de uma combinação devem ser testadas pelo menos uma vez. A Tabela 3.4, apresenta um exemplo de aplicação deste critério, onde avalia-se uma combinação com duas condições (C1=1 ou C2=1).

TABELA 3.4 – Exemplo de combinação para teste de condições utilizando operador lógico **"ou"**.

	C1	C2	Resultado
(a)	**0**	**0**	**F**
(b)	**1**	**1**	**V**
(c)	**1**	**0**	**V**
(d)	**0**	**1**	**V**

- **Combinação-FV** – Testar uma combinação que resulte falso e uma que resulte verdadeiro. Considerando a Tabela 3.4, deve-se testar a combinação (a) e qualquer uma de (b) a (d). Para as combinações com mais de uma condição, proceder como no critério **Combinações-VF**.

Os critérios para combinações apresentados, também, podem ser utilizados nos casos em que em uma combinação com mais de duas condições sejam utilizados os dois operadores lógicos ("e", "ou"). Neste caso, o procedimento será o mesmo dos critérios **Combinações-VF** e **Combinações-FV**, testar uma combinação toda verdadeira, toda falsa e qualquer uma das demais combinações.

O desenvolvedor pode optar pelo nível de eficiência do teste a ser realizado, tendo em vista que cada critério de cobertura aqui definido possui uma abrangência diferente. Caso queira o nível de eficiência máxima deve utilizar, de acordo com operador lógico existente, os critérios "**Todas-Combinações-E**", "**Todas-Combinações-OU**".

A verificação das condições ou combinação de condições deve ser realizada em todos os módulos do sistema que possuam métodos (modificadores[32], seletores e de interação) onde as mesmas sejam implementadas. No momento em que o projetista de software determinar quais são as condições existentes em sua aplicação, o mesmo já pode preparar os dados de teste a serem aplicados nas condições quando estas forem implementadas.

3.1.2 Informações de Teste a partir dos Métodos da Classe

Nesta seção, serão identificadas as informações que podem ser extraídas do diagrama de classes para a análise dos métodos de cada classe. Tal como na seção anterior, estas informações têm por objetivo auxiliar o desenvolvedor no momento em que for testar o *software* que está sendo desenvolvido. Nos métodos, estas informações

[32] Método que altera o valor ou estado de um objeto.

dizem respeito às suas características (visibilidade, lista de parâmetros, tipo de retorno e propriedades).

A UML faz distinção entre os termos método e operação. Uma operação é considerada como sendo algo que é executado em um objeto, o processo que a classe sabe realizar, um procedimento. Já o método é a implementação desta operação, o corpo do procedimento. Neste trabalho, por ser o nomenclatura padrão para descrever as funcionalidades da classe, será utilizado o termo método para referir-se as operações da classe. A sintaxe completa para método na UML é:

visibilidade nome (lista-de-parâmetros) :tipo-de-retorno {propriedade}

a) **Visibilidade**: como nos atributos, indica a forma como o mesmo poderá ser acessado podendo ser: pública (+) padrão (*default*); protegida (#) ou privada (-). As características da visibilidade, com exceção da pública, podem variar de acordo com a linguagem utilizada para implementar o *software*.

b) **Nome**: conjunto de caracteres que representa o nome da operação.

c) **Lista-de-parâmetros**: é uma lista de valores separada por vírgula. Trata-se de especificação de variáveis que podem ser alteradas, passadas ou simplesmente devolvidas.

d) **Tipo-de-retorno**: é uma especificação dependente de linguagem de programação sobre o tipo de implementação do valor retornado pelo método. Ex: *integer*, *string* e *date*. Quando omitido indica que o método não retorna um valor.

e) **Propriedade**: Indica as propriedades que se aplicam ao método. Caso não seja utilizada, omitem-se as chaves. Algumas propriedades são: classificação, precondição, póscondição, tipo de exceção, concorrência, transformação e estereótipo.

Quando o desenvolvedor for verificar os métodos especificados na classe em relação ao que foi implementado, o mesmo deve seguir as seguintes diretrizes:

- verificar se os métodos especificados no modelo foram devidamente declarados na implementação da classe;
  - verificar se a visibilidade dos métodos obedece à definida na especificação da classe;
  - verificar se os parâmetros especificados estão em conformidade com a implementação;
  - verificar se os tipos de retorno, caso tenham sido especificados, estão corretamente implementados.
- verificar todos os métodos *get* e *set*, para analisar se estão executando corretamente seus propósitos - leitura: devolver o valor de um campo e modificação: colocar o valor em um campo;
- analisar a correta implementação das pós e pré condições do método;
- analisar grupos de métodos que manipulem os mesmos atributos;
- verificar o método construtor – criação de objetos;
- verificar se os métodos foram implementados de acordo com sua propriedade.

Como na verificação dos atributos, na Figura 3.3 são apresentados os elementos envolvidos com a verificação dos métodos da classe, onde de um lado tem-se a especificação dos métodos da classe e do outro, parte do código em *ObjectPascal* referente à implementação. Sendo assim, pode-se comparar o que foi especificado com o que foi modelado.

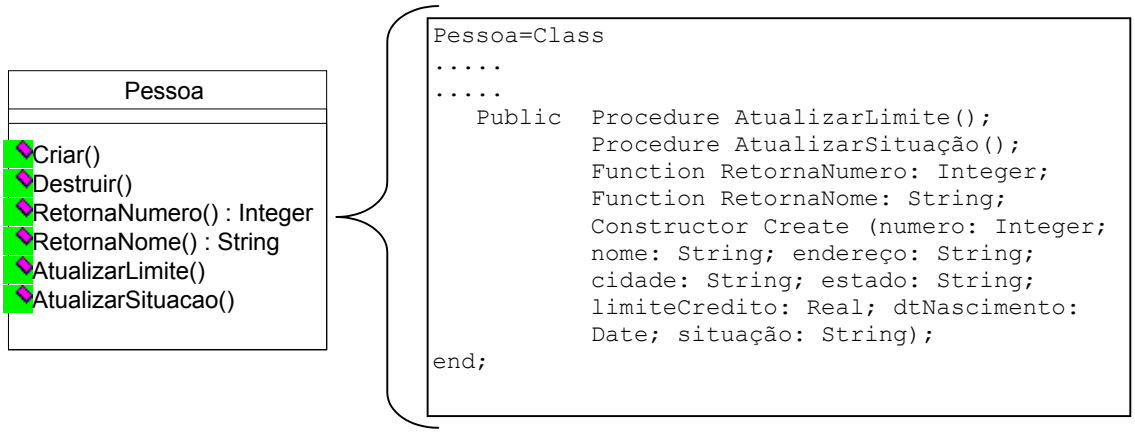


FIGURA 3.3 – Elementos envolvidos com a verificação dos métodos da classe.

A verificação nas pré e pós condições de um método, quando declaradas, devem ser verificadas conforme as diretrizes da cobertura por condições e combinação de condições apresentadas na Seção 3.1.1.1.

Na verificação dos métodos, selecionou-se e adaptou-se o critério de cobertura **"Todos-Métodos"** definido por McGregor [MCG 96], onde todos os métodos de uma classe devem ser exercitados pelo menos uma vez. Caso existam pré e pós condições estas devem ser testadas com valores válidos e inválidos. Isto torna possível a verificação da correta implementação do método.

Os principais objetivos do teste dos métodos da classe são assegurar que:

- todos os métodos foram devidamente definidos e implementados;
- todos os métodos abstratos foram avaliados segundo suas pré e pós condições, bem como a documentação sobre a responsabilidade sob sua implementação foi corretamente definida;
- todos os métodos concretos foram implementados de acordo com suas pré e pós condições.

Alguns dos aspectos que envolvem o teste de métodos estão relacionados à avaliação do relacionamento existente entre as classes. Isto será abordado na próxima seção, onde são definidas as diretrizes para a verificação do relacionamento entre classes e os critérios de cobertura que podem ser utilizados para selecionar os dados de teste. Na Seção 3.1.3.1, foram definidos os critérios de cobertura para o teste dos métodos da subclasse no relacionamento do tipo generalização.

3.1.3 Informações para o Teste baseadas nos Relacionamentos da Classe

Um aspecto importante no diagrama de classes é a capacidade de representação do relacionamento entre classes, especificando a forma como estes ocorrem. Na orientação a objetos, os quatro relacionamentos mais importantes são as dependências, as generalizações, as associações e as realizações [BOO 99]. Neste trabalho, os relacionamentos selecionados foram: (a) generalização; (b) associação simples; (c) associação do tipo agregação simples e (d) associação do tipo agregação por composição[33], observando-se, ainda, a multiplicidade e restrições destes relacionamentos.

3.2.3.1 Analisando o Relacionamento do Tipo Generalização

O relacionamento do tipo generalização expressa a relação entre um item geral (super-classe ou classe-mãe) e um tipo mais específico desse item (sub-classe ou classe-filha), onde a sub-classe herda a estrutura e o comportamento da super-classe, podendo adicionar uma nova estrutura ou comportamento, bem como modificar (sobrescrever) este comportamento. No diagrama de classes da Figura 3.1, uma *Pessoa* pode ser *Física* ou *Jurídica*. Logo, as semelhanças podem ser colocadas na classe geral *Pessoa* e as especificidades nas respectivas subclasses *Física* ou *Jurídica*.

Uma forma de testar as classes envolvidas neste tipo de relação, é utilizar alguns dos conceitos apresentados por Harrold [HAR 92] para o teste incremental[34] de subclasses. Neste tipo de teste, primeiramente testa-se a superclasse, como forma de otimizar o teste, pois na maioria das vezes, os atributos e métodos já testados, são suprimidos no teste das subclasses. Tendo isso como base, definiu-se algumas diretrizes para o teste de subclasses:

- os métodos herdados e testados na superclasse não são re-testados nas subclasses, desde que não interajam com métodos redefinidos;
- os métodos novos e os redefinidos na subclasse devem ser testados;
- os atributos testados na superclasse não são testados nas subclasses;
- os atributos novos devem ser testados;
- os atributos da superclasse, que forem manipulados por métodos modificadores das subclasses, devem ser re-testados.

Tendo como base as diretrizes apresentadas, foram definidos os seguintes critérios de cobertura para o teste de subclasses:

- **Todos-Métodos-Sub** – Testar todos os métodos na subclasse, independente de serem herdados, redefinidos ou novos. Critério com alto nível de eficiência[35], pois

[33] Associação especial, onde o objeto parte existe em função de um todo, ou seja, existe explicita definição do tempo de vida das partes em relação ao todo [BOO 99].
[34] Vide Seção 2.4.1.5 Teste de subclasses.
[35] Considera-se eficiente o critério de cobertura em que pelo menos um de seus casos de testes detectar um erro [WEY 80].

testa todos os métodos, inclusive aqueles que já foram testados na superclasse (mas poderiam apresentar problemas quando utilizados nas subclasses).

- **Todos-Métodos-NRHr** – Testar na subclasse os métodos novos, os redefinidos e os herdados que interajam com os métodos redefinidos na subclasse. Como os métodos herdados que não interagem com os métodos da subclasse já foram testados na superclasse, a probabilidade de ocorrer um erro é mínima, mas não pode ser descartada. Portanto, este critério, pode ser considerado de nível médio em relação aos demais critérios (Todos-Métodos-Sub e Todos-Métodos-NR);
- **Todos-Métodos-NR** – Testar na subclasse, somente os métodos redefinidos e os novos. Este critério possui nível de eficiência baixo, por não considerar os métodos herdados que manipulam seus métodos. Logo, deve ser utilizado em *softwares* que não sejam utilizados no gerenciamento de situações críticas.
- **Todos-Atributos-Sub** – Testar todos os atributos na subclasse, os novos e os herdados – nível eficiência alto;
- **Todos-Atributos-NH** – Testar na subclasse somente os atributos novos e os herdados que forem manipulados por métodos da subclasse – nível de eficiência médio;
- **Todos-Atributos-N** – Testar na subclasse somente os atributos novos e métodos que manipulem estes atributos – nível de eficiência baixo.

Quando o desenvolvedor for testar o *software* que foi modelado, na fase de implementação, o mesmo poderá fazer uso de alguma das ferramentas disponíveis que façam este tipo de teste. O uso de ferramentas reduz o tempo destinado aos testes, além de minimizar a probabilidade de erros oriundos do processo não automatizado.

3.1.3.2 Analisando o Relacionamento do tipo Associação

A Associação é um tipo de relacionamento que representa as relações entre ocorrências de classes, ou seja, a conexão de objetos de uma classe com os objetos de outra [BOO 99]. No diagrama da Figura 3.1 uma *Locação* tem que ser feita por uma única *Pessoa* e esta *Pessoa* pode fazer várias *Locações* ao longo do tempo. Cada associação possui duas pontas de associação, freqüentemente chamada de papéis, cada uma ligada a uma das classes da associação [FOW 2000].

Existem dois tipos especiais de associação, a agregação simples e a agregação por composição. A agregação simples é inteiramente conceitual e nada faz além de diferenciar o *todo* da *parte*, sem vincular o tempo de vida do *todo* e suas *partes*. Já na agregação por composição, o *todo* é responsável pela disposição (criação e destruição) de suas *partes*, enfatizando a dependência das *partes* em relação ao *todo*. Logo, o objeto *parte* existe em função de um *todo*, demonstrando a explicita definição do tempo de vida das *partes* em relação ao *todo* [BOO 99].

As associações simples e do tipo agregação simples, são conceituais, portanto, a única característica a ser observada é a multiplicidade existente entre as classes na associação. Este tipo de verificação será descrito na Seção 3.1.3.4 que trata da análise da multiplicidade do relacionamento.

Quando a associação do tipo agregação por composição estiver presente na relação entre duas classes, os métodos das classes "todo" e "parte" que excluam[36] seus objetos devem possuir pré e pós condições que considerem este relacionamento. A Figura 3.4 apresenta um exemplo de associação do tipo agregação por composição. Neste caso, o método *ExcluirLocação()* da classe *Locação* (todo), deve possuir como pré-condição, a verificação da existência de objetos da classe *VeículoLocado* (parte). Caso a précondição seja verdadeira, executa-se o método *ExcluirVeículoLocado()* até que não existam mais objetos associados nesta classe, permitindo com isso, a exclusão do objeto da classe *Locação*. A forma como a classe "todo" gerencia suas partes pode variar em função da linguagem de programação utilizada.

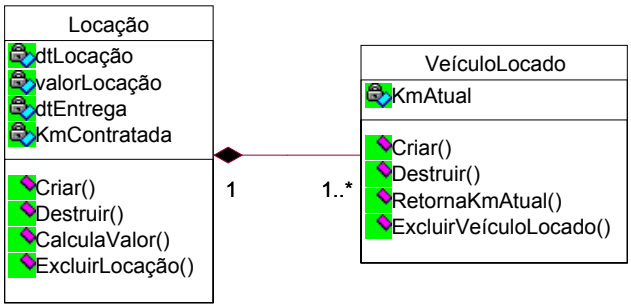


FIGURA 3.4 – Exemplo de associação do tipo agregação por composição

Outro elemento importante que pode fazer parte da associação é a restrição. Trata-se de uma relação semântica entre elementos do modelo que especifica condições e proposições que devem ser mantidas como verdadeiras, caso contrário, o sistema ou elemento descrito pelo modelo é nulo. Certos tipos de restrição são pré-definidas na UML, outras podem ser definidas pelo usuário. Uma restrição definida pelo usuário é descrita por palavras em uma determinada linguagem [OMG 99]. Na Figura 3.5, a restrição *{Física.CalculaIdade >= 18}* foi inserida na associação entre *Pessoa* e *Locação*, estabelecendo como idade mínima para que uma pessoa possa locar um veículo, dezoito anos.

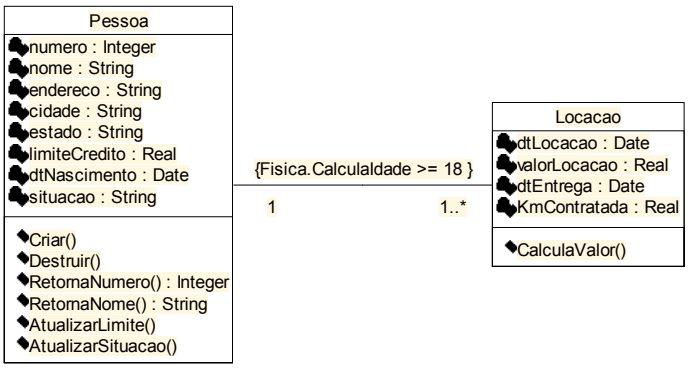


FIGURA 3.5 – Exemplo de restrição de associação.

As restrições quando forem expressas por condições devem ser testadas utilizando-se as diretrizes para o teste de condições e os critérios de cobertura baseados em condições e combinações de condição.

[36] Excluir objetos persistentes, ou seja, aqueles que estão armazenados no Banco de dados.

Com base nas características apresentadas, foram definidas as diretrizes para o teste das associações simples, do tipo agregação simples e por composição:

- verificar todas as associações e agregações simples quanto a sua multiplicidade – exercitar os métodos construtores e aqueles que efetivam a relação entre as classes de acordo com os valores máximos e mínimos da multiplicidade;
- verificar todas as associações do tipo agregação por composição quanto às suas peculiaridades;
  - verificar todos as pré e pós condições dos métodos que excluam objetos das classes todo e parte – exercitando estes métodos;
  - verificar todos os métodos construtores e destrutores das classes que façam parte de uma associação do tipo agregação por composição – exercitar os métodos considerando as características da associação e multiplicidades existentes;
- verificar todas as restrições de uma associação quanto a sua correta implementação – exercitar os métodos onde as condições das restrições tenham sido implementadas. Para gerar os dados de teste, pode ser utilizada a cobertura por condições e combinação de condições da Seção 3.1.1.1.

3.1.3.4 Analisando a Multiplicidade do Relacionamento

Um dos componentes importantes do relacionamento é a multiplicidade, cuja função é estabelecer a quantidade mínima e máxima de objetos de uma classe em relação aos objetos da outra [OMG 99]. No diagrama de classes da Figura 3.1, o "*1..**" na ponta da *Locação* na associação com a *Pessoa* indica que uma *Pessoa* pode ter uma ou mais, locações associadas a ela, enquanto que o numeral "*1*" na outra ponta indica que uma *Locação* esta associada a somente uma *Pessoa*.

A multiplicidade pode ser verificada, utilizando-se a técnica de teste apresentada por Binder [BIN 97][37] e a técnica de teste análise do valor limite de Myers [MYE 79]. Tendo como base estas técnicas, foram definidos os critérios de cobertura para o teste de multiplicidade:

- **Todas as Multiplicidades** - Testar todas as multiplicidades das associações no diagrama de classes pelo menos uma vez, em seus valores máximos e mínimos. Este critério permite que todas as multiplicidades existentes no modelo sejam verificadas.
- **Todas-Multiplicidades-Obrigatórias** - Testar somente as multiplicidades obrigatórias das associações, ou seja, aquelas em que o valor mínimo seja igual a "1". Este critério permite que pelo menos as associações obrigatórias sejam testadas.

Estes critérios de cobertura devem ser utilizados de acordo com as seguintes diretrizes definidas:

- nos casos em que os valores máximos e mínimos de uma multiplicidade sejam especificados, deve-se projetar casos de teste com os valores máximo, mínimo e outros logo acima e abaixo destes valores (conforme BVA);

[37] Vide Seção 2.4.2.3 Proposta de Binder baseada na verificação da Associação entre classes

- nos casos onde o valor máximo não seja especificado, projetar casos de teste com valores pelo menos duas vezes maiores (no mínimo) que o valor mínimo e outros logo acima e abaixo destes valores (+1 e –1).

As Tabelas 3.5 e 3.6 apresentam um conjunto de casos de teste para a multiplicidade da associação existente entre as classes *Pessoa* e *Locação* da Figura 3.1, para simular a existência de valor máximo foi estipulado que uma pessoa poderia estar associada a no máximo três locações. A Figura 3.6 mostra a associação *Pessoa*/*Locação* com a multiplicidade modificada.

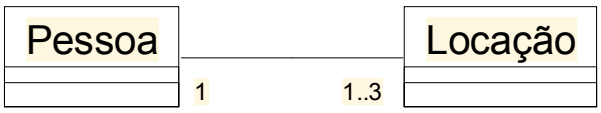


FIGURA 3.6 – Exemplo de multiplicidade com valor máximo especificado

TABELA 3.5 – Exemplo de casos de teste para objetos da classe Pessoa em relação à classe Locação

Classe A: **Pessoa**		Data Atual: **29/03/2000**		
Classe B: **Locação**				
Multiplicidade Mínima: **1**		Multiplicidade Máxima: **3**		
Parâmetro	Pessoa	Locação	Resultado Esperado	
Valor máximo	**1**	**3**	**V**	***1***
Valor mínimo		**1**	**V**	***2***
Valor máximo + 1		**4**	**F**	***3***
Valor máximo – 1		**2**	**V**	***4***
Valor mínimo +1		**2**	**F**	***5***
Valor mínimo –1		**0**	**F**	***6***

TABELA 3.6 – Exemplo de casos de teste para objetos da classe Locação em relação à classe Pessoa

Classe A: **Locação**		Data Atual: **29/03/2000**		
Classe B: **Pessoa**				
Multiplicidade Mínima: **1**		Multiplicidade Máxima: **1**		
Parâmetro	Locação	Pessoa	Resultado Esperado	
Valor máximo	**1**	**1**	**V**	***1***
Valor mínimo		**1**	**V**	***2***
Valor máximo + 1		**2**	**F**	***3***
Valor máximo – 1		**0**	**F**	***4***
Valor mínimo +1		**2**	**F**	***5***
Valor mínimo –1		**0**	**F**	***6***

Utilizando os casos de teste apresentados nas Tabelas 3.5 e 3.6, caso o resultado esperado seja falso (F), indica que na implementação do método responsável por efetivar a *Locação* deve existir uma mensagem implementada para tratar os resultados apresentados. Por exemplo, para o resultado esperado "*3*" da Tabela 3.5, deve ser implementada a mensagem *"Operação Cancelada - Cliente pode fazer no máximo 3 (três) Locações"*.

Na Tabela 3.7 apresenta-se um conjunto de casos de teste para a multiplicidade da associação existente entre as classes *Pessoa* e *Locação* da Figura 3.1 onde, a valor máximo da multiplicidade não foi especificado.

TABELA 3.7 – Exemplo de casos de teste para objetos da classe Pessoa em relação à classe Locação

		Data Atual: **29/03/2000**		
Classe A: **Pessoa**		Classe B: **Locação**		
Multiplicidade Mínima: **1**		Multiplicidade Máxima: *		
Parâmetro	Pessoa	Locação	Resultado Esperado	
Valor máximo	**1**	**4**	**V**	***1***
Valor mínimo		**1**	**V**	***2***
Valor máximo + 1		**5**	**V**	***3***
Valor máximo – 1		**3**	**V**	***4***
Valor mínimo +1		**2**	**F**	***5***
Valor mínimo –1		**0**	**F**	***6***

Observando os resultados das Tabelas 3.6 e 3.7, pode-se concluir que com um pequeno número de casos de teste, pode-se verificar a correta implementação da multiplicidade do relacionamento. Este teste pode ser feito funcionalmente pela execução dos métodos construtores e dos métodos que efetivem o relacionamento entre as classes ou analisando-se o código dos métodos que implementam estas características.

3.2 Critérios de Cobertura baseados em Diagrama de Seqüência

Nesta seção são abordados os aspectos importantes para a definição de critérios de cobertura para o teste de software com base em informações derivadas do diagrama de seqüência. Como em um diagrama de seqüência a troca de mensagens pode gerar um conjunto exorbitante de caminhos de teste, definiu-se um conjunto de critérios de cobertura, os quais servem para que o desenvolvedor selecione os caminhos que deseja exercitar e, por conseguinte, o grau de eficiência do teste que deseja obter.

O diagrama de seqüência apresenta a interação de objetos organizados em uma seqüência de tempo e de mensagens trocadas, mas não trata diretamente das associações entre estes objetos. Normalmente, representa a interação entre objetos para a realização de somente um caso de uso, ou seja, revela a interação para um cenário[38] específico durante a execução de um sistema [BOO 99].

A Figura 3.7 apresenta um exemplo de diagrama de seqüência que demonstra a interação entre os objetos para o caso de uso Manter Contrato de Locação de um sistema automatizado para locação de veículos, que exibe o seguinte cenário:

- o atendente da locadora envia uma mensagem *Criar()* para *Locação*;
- *Locação* envia a mensagem *RetornaSituação()* para *Pessoa*;

[38] Uma seqüência específica de ações que ilustra comportamentos ou interações. Vários cenários compõem um caso de uso [BOO 99].

  - se a situação do cliente for diferente de "OK" a locação é cancelada com o envio da mensagem *CancelaLocação()*;
  - caso contrário, *Locação* envia a mensagem *RetornaNome()* para *Pessoa*;
- se o tipo de pessoa for física, *Pessoa* envia a mensagem *RetornaCpf()* para *Física*. Caso contrário, será enviada a mensagem *RetornaCnpj()* para *Jurídica*;
- para cada veículo a ser locado, *Locação* envia as mensagens: *RetornaPlaca()* para *Veículo*, *Criar()* para *VeículosLocado*, *AtualizaStatus()* para *Veículo* e *CalculaValor()* para *Locação*;
- ao final, para que o contrato seja emitido é enviada a mensagem *EmitirContrato()*.

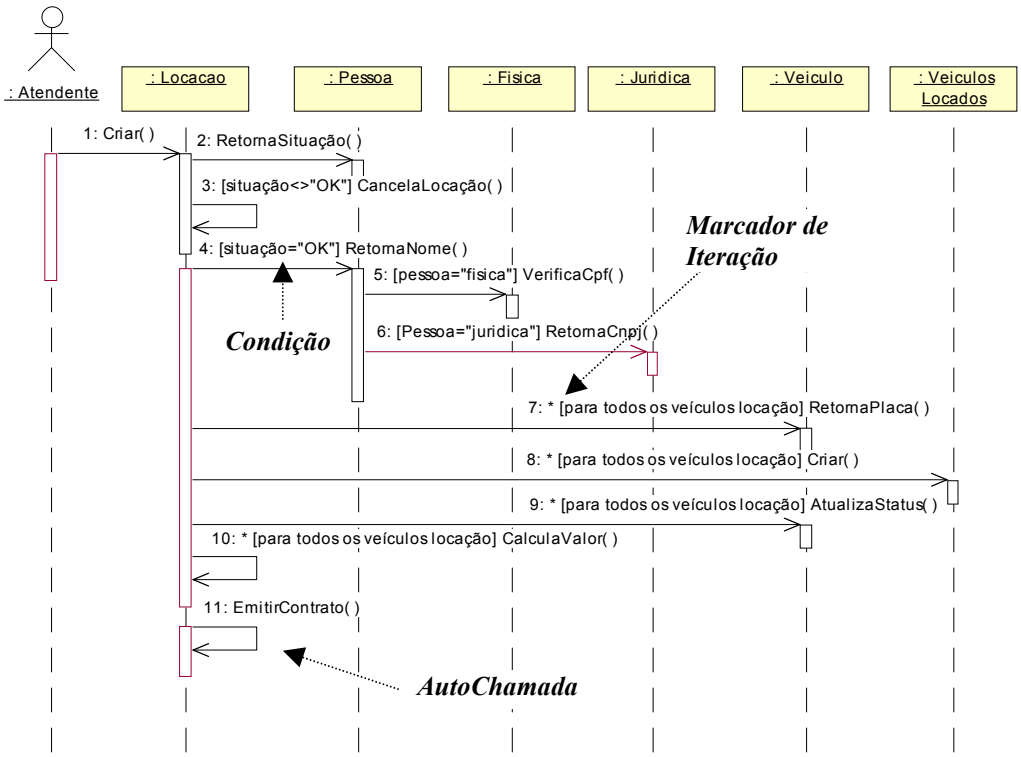


FIGURA 3.7 – Diagrama de Seqüência do Caso de Uso Manter Contrato de Locação.

Neste diagrama, um objeto é desenhado como um retângulo no topo de uma linha vertical tracejada. Essa linha, é chamada de linha de vida do objeto, representando o ciclo de vida do objeto durante uma interação. As mensagem são representadas por linhas com setas, dirigidas horizontalmente entre as linhas de vida de dois objetos. Cada mensagem é rotulada, no mínimo, com seu nome, podendo incluir os argumentos e alguma informação de controle [FOW 2000]. Dentre as informações de controle, duas são de extrema importância:

- **condição** – utilizada para indicar quando a mensagem será enviada, ou seja, caso a condição seja verdadeira, a mensagem será enviada [FOW 2000]. Como exemplo, na Figura 3.7 a condição *[situação<>"OK"]*, indica que a mensagem *3:CancelaLocação()* será enviada caso a situação seja diferente de "OK".

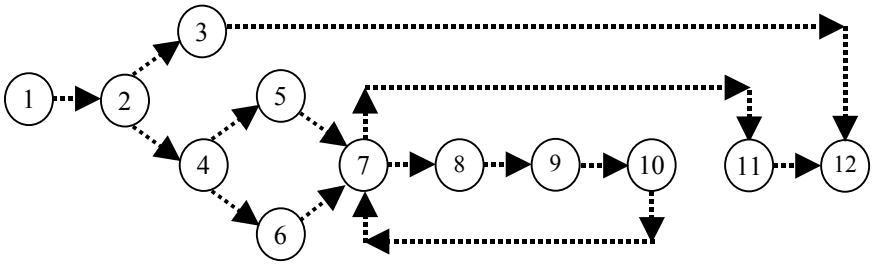
- **marcador de iteração (repetição)** – é representado por um asterisco "*" antes da mensagem iterativa, indicando que esta será enviada várias vezes para múltiplos

objetos receptores [OMG 99]. A base (especificação) da iteração deve ser colocada entre colchetes, como apresentado no diagrama de seqüência da Figura 3.7, na mensagem *7: * [para todos os veículos locação] RetornaPlaca( )* neste contexto, esta mensagem será repetida enquanto houver veículos para a locação.

Observando-se as características das informações disponibilizadas por este diagrama e utilizando-se como base os critérios procedimentais para o fluxo de controle[39], definiu-se um conjunto de critérios de cobertura para o teste utilizando o diagrama de seqüência como fonte de informações:

- **Todas-S-Mensagens** - todas as mensagens devem ser executadas pelo menos uma vez. Este critério serve para verificar a declaração e implementação das mensagens por seus respectivos métodos, verificando as condições das mensagens. Equivale ao critério de cobertura Todos os Nós de Myers (Seção 2.1.1.1).

- **Todas-C-Mensagens** - neste critério todas as condições da seqüência de mensagens devem ser testadas pelo menos uma vez com valores verdadeiros e falsos. Este critério permite a verificação da implementação das mensagens de acordo com as condições especificadas. Este critério equivale ao critério Todas as Decisões de Myers. A Figura 3.8 apresenta a representação da aplicação deste critério em um GFC construído a partir dos dados do diagrama de seqüência (cada nó corresponde a uma mensagem) da Figura 3.7, onde as seqüências A B e C especificam as possibilidades de caminhos a serem seguidos pelo teste.



Seqüência_A : 1, 2, 3, 12
Seqüência_B : 1, 2, 4, 5, 7, 8, 9, 10, 7, 11, 12
Seqüência_C : 1, 2, 4, 6, 7, 8, 9, 10, 7, 11, 12

FIGURA 3.8 – Representação do Critério Todas-C-Mensagens em um GFC.

- **Todas-Iterações** – este critério é para o teste de mensagens iterativas (*loops*).Todas as mensagens iterativas devem ser executadas para seus valores mínimos e máximos, de acordo com a multiplicidade especificada no relacionamento existente entre as classes. Normalmente, para que o teste de iteração seja satisfeito, devem ser seguidos os seguintes passos: (1) não exercitar a iteração; (2) exercitar uma vez; (3) exercitar várias vezes. A *Seqüência_A* da Figura 3.9, demostra os caminhos executados quando a iteração não é executada. Já a *Seqüência_B* e a *Seqüência_C* apresentam a execução da iteração por uma e

[39] Todos os nós, arcos e decisões, para maiores detalhes, vide Seção 2.1.1.1 Critérios de Cobertura Baseados em Fluxo de Controle

várias vezes respectivamente. Quando o valor máximo da multiplicidade for especificado, testar a iteração com o valor especificado, um logo acima e outro logo abaixo. Semelhante a técnica de Análise de Valor Limite –BVA proposta por Myers [MYE 79].

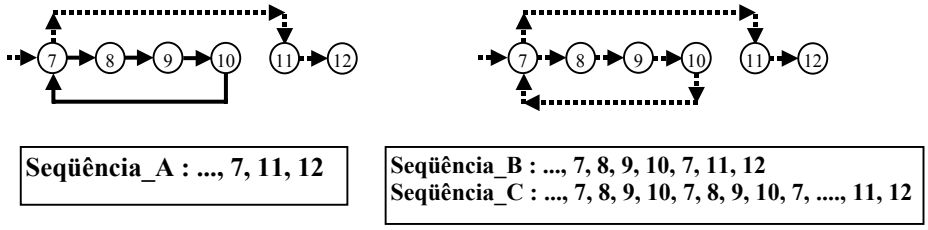


FIGURA 3.9 – Representação do Critério Todas-SCS-Mensagens em um GFC.

Os critérios de cobertura apresentados podem ser utilizados individualmente ou em conjunto. Isto será determinado pelas características do cenário especificado no diagrama de seqüência. Por exemplo, para gerar a seqüência de testes para o diagrama da Figura 3.7, devem ser utilizados todos os critérios de cobertura, pois, existem mensagens com condições e iterações.

A aplicação do critério de cobertura **Todas-Iterações**, detectou um erro de especificação no diagrama de seqüência da Figura 3.7, mesmo as iterações das mensagens 7,8,9,10 não sendo executadas, a mensagem 11 é enviada. Assim sendo, um contrato de locação sem veículos foi emitido, situação esta inexistente no processo de locação. Foram feitas três modificações no diagrama para eliminar esta anomalia:

(1) inserida uma mensagem para verificar a existência de veículos "*11:VerificaVeiculo( )*";
(2) inserida uma mensagem para cancelar a *Locação* caso não existam veículos locados "*12:[VerificaVeiculo="F"] CancelaLocação( )* " e;
(3) modificada a mensagem "*11:EmitirContrato( )*", alterando seu número de seqüência e inserindo uma condição de guarda para indicar que a mesma só será enviada caso existam veículos na locação. Sua nova representação seria: "*13:[VerificaVeículo="V"]EmitirContrato( )*".

Na Figura 3.10, podem ser observadas as mudanças realizadas no diagrama de seqüência da Figura 3.7.

Levando-se em consideração o que foi apresentado, o teste baseado nas informações de um diagrama, na maioria das vezes, é capaz de detectar erros na especificação do próprio diagrama. Daí a importância da iteratividade no ciclo de produção de software mencionado no início deste trabalho. Uma vez detectado o erro, deve-se retroceder para verificar a causa do mesmo, corrigi-lo e continuar o ciclo.

As informações de controle, quando utilizadas no diagrama de seqüência disponibilizam dados de extrema importância para o teste do software, indicando algumas vezes cenários particulares antes não observados e, até mesmo, estruturas especiais de implementação, como por exemplo o laço necessário para a entrada de

veículos na locação, demonstrado pela seqüência de mensagens iterativas 7, 8, 9 e 10 do diagrama de seqüência da Figura 3.10.

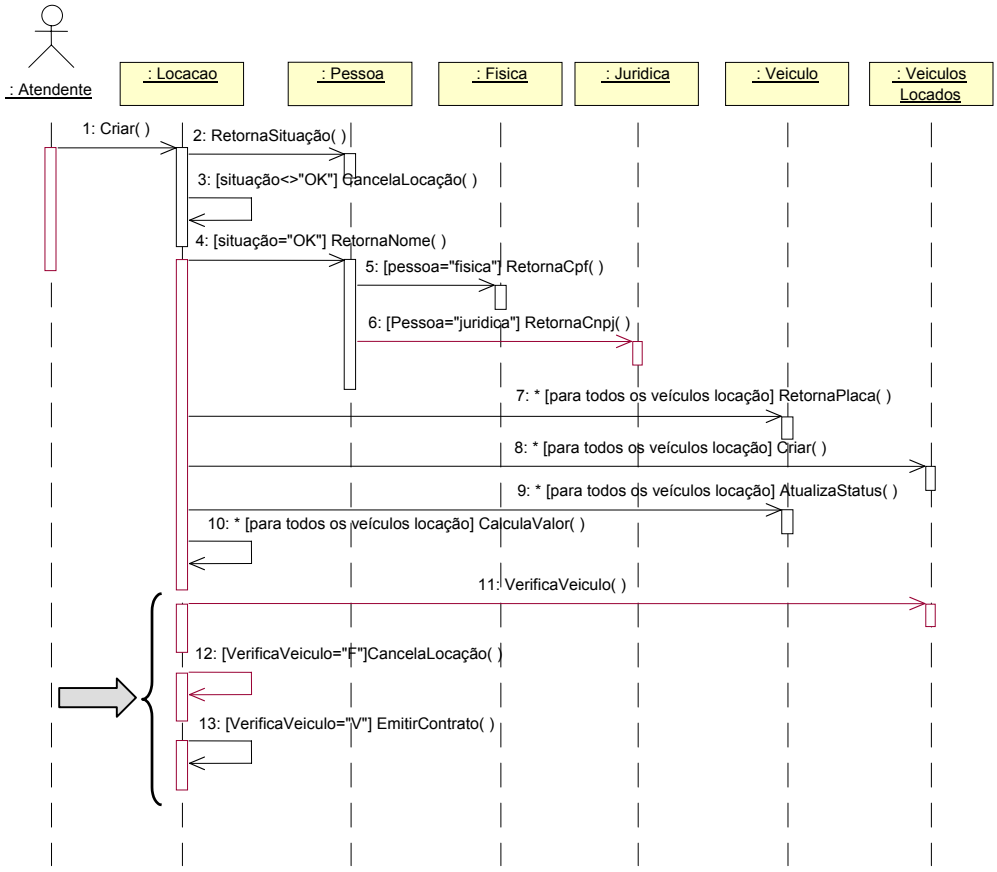


FIGURA 3.10– Diagrama de Seqüência modificado após aplicação do critério Todas-SCS-Mensagens.

Este tipo de diagrama disponibiliza ao desenvolvedor informações com as quais será possível determinar os dados para testar a implementação do cenário modelado no diagrama. Logo pode-se testar:

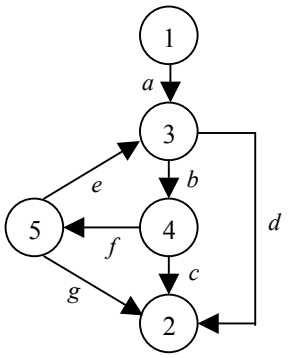
(a) a existência das classes;
(b) a implementação dos métodos;
(c) a seqüência das mensagens;
(d) o comportamento das mensagens de acordo com as condições especificadas;
(e) a implementação das condições;
(f) a implementação das iterações.

As condições existentes nas mensagens devem ser verificadas, os dados de teste necessários podem ser gerados de acordo com as diretrizes e critérios de cobertura para condições e combinação de condições apresentados na Seção 3.1.1.1. Caso a base da iteração também contenha uma condição, esta também deve ser testada.

3.2.1 Implementando o teste baseado no diagrama de seqüência

A técnica de teste escolhida para o teste baseado em diagrama de seqüência foi a apresentada por Binder [BIN 98][40], onde um diagrama é convertido em um grafo de fluxo (GFC) para que sejam gerados os caminhos de teste. O autor sugeriu que seria possível automatizar a geração de casos de teste, mas não disponibilizou detalhes de como isto poderia ser feito.

Segundo Pressman [PRE 95] uma forma de automatizar o procedimento de criação de caminhos básicos, derivados do GFC, seria a utilização de uma *matriz de grafo*. O conceito de matriz de grafo, proposto por Beizer [BEI 90], trata de uma matriz quadrada cujo número de linhas e colunas é igual ao número de nós do grafo de fluxo. Nesta matriz, cada linha e coluna correspondem a um nó identificado no GFC e as entradas da matriz equivalem à conexões (ramos) entre os nós. Na Figura 3.11 pode-se observar um exemplo de grafo de fluxo e sua matriz de grafo correspondente. Nesta, cada nó do GFC foi identificado por um número e cada ramo uma letra. A conexão entre dois nós é identificada pela inserção de uma letra na matriz. Como exemplo, o nó 4 está ligado ao nó 5 pelo ramo *f*.



Grafo de Fluxo

Nó \ Ligado ao nó	1	2	3	4	5
1			*a*		
2					
3		*d*		*b*	
4		*c*			*f*
5		*g*	*e*		

Matriz de Grafo

FIGURA 3.11 – Exemplo conversão do GFC em Matriz de Grafo [PRE 95].

Analisando o contexto do teste de Cenários de Binder e a idéia de Beizer para a automação da geração de caminhos lógicos de um GFC, concluiu-se que um diagrama de seqüência pode ser convertido em uma matriz de grafo. Com isso, os caminhos lógicos para o teste do cenário de um Diagrama de Seqüência, podem ser gerados automaticamente.

Algumas modificações foram feitas na matriz de grafo para adequá-la ao teste de cenários. Neste trabalho, esta matriz passou a ser chamada de **Matriz de Caminho**. Nesta, o número de linhas e colunas é equivalente ao número de mensagens existentes no diagrama de seqüência. Cada linha e cada coluna correspondem a uma mensagem identificada no diagrama e as entradas da matriz as suas conexões. Tendo como base o diagrama de seqüência da Figura 3.10, criou-se a matriz de caminho apresentada na Figura 3.12, permitindo a definição automática dos possíveis caminhos para o teste do cenário apresentado.

[40] Seção 2.4.2 Proposta de Binder baseada nos cenários de um diagrama de seqüência.

Matriz de Caminho

	M1	M2	M3	M4	M5	M6	M7	M8	M9	M10	M11	M12	M13
M1		C											
M2			CF	C									
M3													
M4					C	C							
M5							C						
M6							IN						
M7								I			IF		
M8									I				
M9										I			
M10							IN				IF		
M11												CF	CF
M12													
M13													

Mensagens

M1= 1:Criar()
M2= 2:RetornaSituação()
M3= 3:[situação<>"OK"] CancelaLocação()
M4= 4:[situação="OK"] RetornaNome()
M5= 5:[Pessoa="Física"] RetornaCpf()
M6= 6:[Pessoa="Jurídica"] RetornaCnpj()
M7= 7:* RetornaPlaca()
M8= 8:* Criar()
M9= 9:* AtualizaStatus()
M10= 10:* CalculaValor()
M11= 11:VerificaVeiculo()
M12= 12:[VerificaVeiculo="F"] CancelaLocação()
M13= 13:[VerificaVeiculo="V"] EmitirContrato()

Legendas

M? = Mensagem
X = Mensagem Normal
C = Mensagem com condição
CF = Condição Final
IN = Mensagem de Iteração Inicial
IF = Mensagem de Iteração Final
I = Mensagem de Iteração

FIGURA 3.12 – Matriz de Caminho do diagrama de seqüência da Figura 3.10.

Definiu-se algumas diretrizes a serem seguidas para a conversão do diagrama de seqüência em uma matriz de caminho, visto que este possui algumas características que devem ser observadas, dentre elas as condições e iterações:

- Quando uma mensagem possuir um marcador de iteração: (1) a primeira mensagens iterativas, na matriz de caminho, deve indicar a próxima mensagem e aquela que finaliza a iteração; (2) a última mensagem iterativa deve indicar a mensagem iterativa inicial e aquela que finaliza a iteração. Como exemplo, a mensagem 7, da Figura 3.7, é a primeira mensagem iterativa, na matriz de caminho, esta indica sua conexão com as mensagens 8 (seguinte) e 11 (após iteração). Assim sendo, estabelece o caminho para o teste da iteração (*loop*). A iteração pode ser representada através de diferentes estruturas de repetição (*while*, *until*, *for*). Salienta-se contudo a importância em adequar-se esta diretriz à estrutura de repetição escolhida. Por exemplo, sendo uma estrutura do tipo *until*, o final da iteração é determinado pela última mensagem desta iteração.
- Quando em uma mensagem existir uma condição, a mensagem exatamente anterior a esta deve indicar quais são as mensagens a serem escolhidas. Logo, estabelece as opções de caminho para a condição existente. Como exemplo, observe a mensagem 3 da Figura 3.7, nela existe uma condição. A mensagem 2, na matriz de caminho, é quem estabelece quais serão as possíveis mensagens executadas, no caso as mensagens 3 ou 4. O desenvolvedor deve identificar nas mensagens condicionais, aquelas que encerram a seqüência de mensagens – (condições finais). Quando em um diagrama de seqüência existirem diversas

condições, dependendo de sua complexidade, recomenda-se o desenvolvimento de um diagrama independente para representar cada cenário apresentado pelas condições [FOW 2000].

- Caso a mensagem não contenha nenhuma das características apresentadas nas diretrizes anteriores, esta deve indicar apenas qual será a próxima mensagem a ser enviada.
- As mensagens que indiquem o final da seqüência, não devem ser conectadas a nenhuma outra mensagem na matriz de caminho. Na Figura 3.12, a mensagem 3 encerra a seqüência, logo, não deve ser conectada a nenhuma outra mensagem.

É imprescindível que as características mínimas sejam especificadas no diagrama, como por exemplo, condições e marcadores de iteração. A ausência destas informações faz com que sejam geradas informações incoerentes e com isso, caminhos que na realidade não garantem o teste do cenário apresentado no diagrama de seqüência.

A Tabela 3.8 apresenta os caminhos para o teste, gerados a partir da matriz de caminho da Figura 3.12 de acordo com os critérios de cobertura definidos para este tipo de diagrama.

TABELA 3.8 – Caminhos gerados com base nos critérios de cobertura definidos para o diagrama de seqüência.

CRITÉRIOS DE COBERTURA	CAMINHOS
Todas-S-Mensagens	Caminho 1= ***M1, M2, M3*** Caminho 2= ***M1, M2, M4, M5, M7, M8, M9, M10, M7, M11,M12*** Caminho 3= ***M1, M2, M4, M6, M7, M8, M9, M10, M7, M11,M13***
Todas-C-Mensagens	Caminho 1= ***M1, M2, M3*** Caminho 2= ***M1, M2, M4, M5, M7, M8, M9, M10, M7, M11,M12*** Caminho 3= ***M1, M2, M4, M6, M7, M8, M9, M10, M7, M11,M13***
Todas-Iterações	Caminho 4= ***M1, M2, M4, M5, M7, M11, M12*** Caminho 5= ***M1, M2, M4, M6, M7, M11, M13*** *Caminho 6= ***M1, M2, M4, M5, M7, M8, M9, M10, M7, M8, M9, M10, ..., M11,M12*** *Caminho 7= ***M1, M2, M4, M6, M7, M8, M9, M10, M7, M8, M9, M10, ..., M11, M13***

*Exercitar várias vezes a iteração.

Com base nos caminhos fornecidos, o desenvolvedor de sistemas pode optar pelo nível de teste a ser executado de acordo com o tipo de cenário a ser testado, uma vez que cada critério possui um grau de eficiência específico.

A maioria das ferramentas para a modelagem automatizada de sistemas, não disponibiliza, em meio magnético, as informações sobre os diagramas implementados. Isto facilitaria a implementação de uma ferramenta para geração automática de caminhos, pois seria necessário apenas que fosse feita a leitura em um arquivo para a importação dos dados necessários. A confiabilidade do teste diminui a medida que aumenta a interação humana na geração dos caminhos e dados de teste.

A ferramenta a ser desenvolvida para a geração de caminhos, deve disponibilizar recursos para o armazenamento destes caminhos, facilitando com isso, sua recuperação no momento em que os casos de teste forem gerados. Um modelo

conceitual (diagrama de classes simplificado) para uma ferramenta de geração de caminhos é apresentado no Anexo 1 deste trabalho.

3.3 Critérios de Cobertura baseados em Diagrama de Colaboração

O diagrama de colaboração permite que, pela troca de mensagens sejam verificadas as interações dinâmicas e as associações existentes entre objetos. Os diagramas de colaboração e seqüência expressam informações semelhantes mas apresentam-nas de forma diferenciada. O primeiro exibe uma seqüência explicita de mensagens, sendo melhor utilizado para especificações em tempo real e para cenários complexos, enquanto o segundo mostra os vínculos entre objetos.

No diagrama de colaboração, a seqüência de tempo é determinada pela numeração das mensagens dentro do diagrama. Desenham-se objetos como ícones, e tal como ocorre no diagrama de seqüência, setas indicam as mensagens enviadas entre objetos para a realização de um caso de uso. Na Figura 3.13, pode-se visualizar um exemplo do diagrama de colaboração para o caso de uso Manter Contrato de Locação.

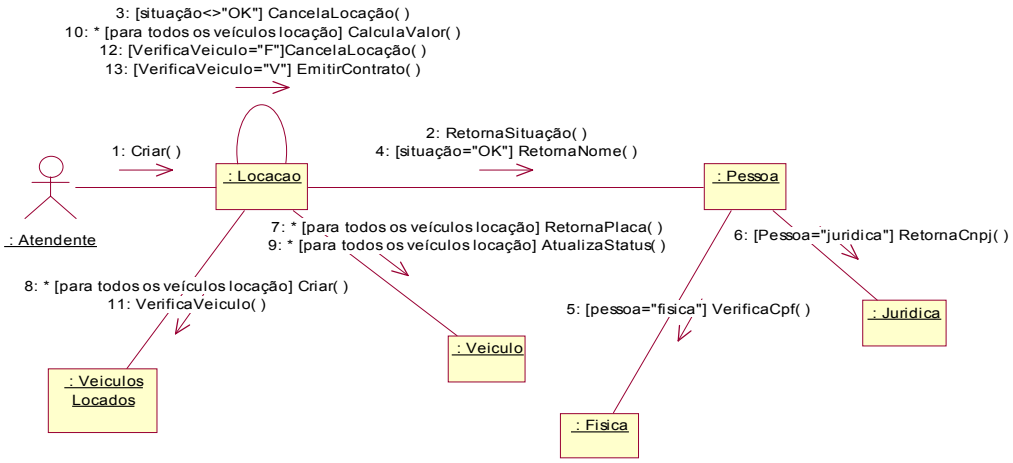


FIGURA 3.13 – Diagrama de Colaboração do Caso de Uso Manter Contrato de Locação.

Os diagramas de seqüência e colaboração são semanticamente equivalentes, portanto, um pode ser convertido no outro sem perda de informações independente da ordem em que forem construídos [OMG 99]. O projetista de software, de acordo com as características do sistema que está sendo desenvolvido, pode optar por construir apenas o diagrama de colaboração, o que não interferirá na qualidade e desenvolvimento do teste. Um exemplo disto pode ser observado no diagrama de colaboração da Figura 3.13 que foi gerado a partir do dados do diagrama de seqüência da Figura 3.7 o mesmo poderia ser feito na ordem inversa. Como observado, as mesmas informações de controle (condições e iterações) mostradas em um diagrama de seqüência, podem ser acrescentadas em um diagrama de colaboração.

Representando a interação existente entre objetos através da troca de mensagens, o diagrama de colaboração pode ser utilizado no teste de interação inter-

classes[41] [MCG 96]. Caso a mensagem possua como argumento um método de outra classe, isto indica que estas classes devem estar integradas [MCG 96]. Esta característica também pode ser observada neste diagrama, desde que os argumentos sejam especificados. Logo, o diagrama de colaboração pode ser utilizado como fonte de informações para o teste de interação e integração entre classes.

Os critérios de cobertura de teste definidos para o diagrama de colaboração foram:

- **Todas-Interações-Totais** – Testar todas as interações do diagrama de colaboração, exercitando todas as mensagens de cada interação. Caso exista uma ou mais mensagens iterativas na interação, estas devem ser testadas pelo menos uma vez. Sendo possível, exercite as iterações várias vezes. Nas mensagens condicionais, testar as condições com valor verdadeiro e falso, pelo menos uma vez.

- **Todas-Interações-Parciais** – Testar todas as interações do diagrama de colaboração, exercitando pelo menos uma mensagem de cada interação. Caso exista somente uma mensagem e esta seja condicional, testar as condições uma vez com valores falso e verdadeiro. Existindo mensagens iterativas, testá-las pelo menos duas vezes para simular iteração contínua. Caso exista um grupo de mensagens iterativas em uma interação, nenhuma das mensagens pode ser desprezada.

Existindo o diagrama de seqüência para a mesma interação, não se faz necessário aplicar os critérios definidos para o diagrama de colaboração, uma vez que os diagramas são semanticamente equivalentes. No teste da condição das mensagens, utilizar, para a definição dos dados de teste, as diretrizes e critérios de cobertura de teste definidos para o teste de condição e combinações de condições da Seção 3.1.1.1.

3.3.1 Implementando o teste baseado no diagrama de colaboração

Já que os diagramas de colaboração e seqüência são equivalentes, a técnica definida para implementação dos caminhos baseados no diagrama de seqüência pode ser também utilizada no diagrama de colaboração. A ferramenta a ser desenvolvida deve possuir uma rotina que faça a conversão entre os diagramas.

Desconsiderando o conteúdo gráfico (sua projeção), os dados disponíveis dos dois diagramas são os mesmos. Sendo assim, o módulo responsável pela geração de caminhos do diagrama de colaboração deve: fazer a leitura dos dados na base, gerar a matriz de caminho e de acordo com os critérios de cobertura, exibir a relação de caminhos possíveis para o teste. Na relação de critérios são apresentados os definidos para os dois tipos de diagrama. Cabe ao usuário definir qual será o critério utilizado, de acordo com o cenário existente.

[41] Seção 2.3.3 – Teste de interação entre classes

3.4 Critérios de Cobertura baseados em Diagrama de Estado

Esta seção trata das características do diagrama de estados implementado pela UML, as quais foram utilizadas na definição das diretrizes e critérios de cobertura de teste definidas para este tipo de diagrama. Utilizou-se como base os conceitos apresentados pelo *Object Management Group* (OMG) [OMG 99], por Booch [BOO 99], Larman [LAR 99] e Fowler [FOW 2000]. Considerou-se no estudo deste tipo de diagrama, somente as características básicas de estados e transições, as quais, disponibilizam recursos para que sejam modelados diversos tipos de comportamentos. Desconsiderou-se os aspectos relativos a transições e estados avançados (ações de entrada e saída, transições internas e eventos adiados), subestados (estados aninhados), subestados seqüenciais (disjunção de estados compostos), estados de histórico (armazena histórico de ativação de subestados de um estado composto) e subestados concorrentes (estados executados em paralelo).

Segundo Larman, um diagrama de estados pode ser aplicado a diversos elementos da UML, dentre eles, classes, tipos (conceitos) e casos de uso. O diagrama de estados modelando casos de uso apresenta a descrição da seqüência legal de eventos de sistema externos que são reconhecidos e tratados por um sistema, no contexto de um caso de uso. Um exemplo deste tipo de diagrama de estado (versão resumida) pode ser observado na Figura 3.14 onde são apresentados os eventos para o caso de uso *Locar Veículo* de uma aplicação de locação de veículos. O diagrama apresentado mostra que o evento *receberVeículo* será efetivado somente depois que o evento *terminarLocação* gerar a transição do sistema para o estado *EsperandoDevolução*.

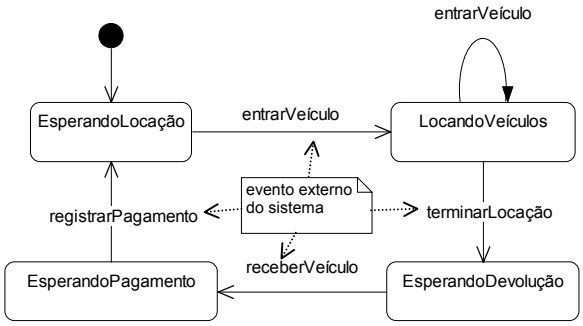


FIGURA 3.14 – Diagrama de estados do caso de uso *Locar Veículo*.

O diagrama de estados de um sistema é utilizado para visualizar as transições de estado para todos os eventos do sistema entre todos os casos de uso. É considerado uma variação do diagrama de estado de casos de uso, ou seja, a união de diversos diagramas deste tipo.

Quando descreve o comportamento de um objeto, o diagrama de estados apresenta todos os estados possíveis que este objeto pode assumir em resposta a eventos. Para facilitar o entendimento do modelo, a UML recomenda que os diagramas de estados sejam projetados para uma única classe, demonstrando o comportamento de um único objeto ao longo de seu tempo de vida. Segundo Fowler, os diagramas de estados devem ser utilizados somente em classes que exibam algum

comportamento interessante, o qual deva ser melhor detalhado para auxiliar na compreensão do que está acontecendo.

O diagrama de estados é constituído basicamente de estados, eventos, transições, atividades e ações:

- **estado -** uma condição ou situação na vida de um objeto durante a qual ele satisfaz alguma condição, realize alguma atividade ou aguarde um evento. Um objeto permanece em um estado por uma quantidade de tempo finita.
- **transição** - é o relacionamento entre dois estados, indicando que um objeto no primeiro estado realizará certas ações e quando ocorrer um evento especificado que atenda as condições especificadas, o mesmo entrará no segundo estado;
- **evento** - especificação de uma ocorrência significativa que tem uma localização no tempo e no espaço, com capacidade para ativar uma transição de estado. Os eventos podem ser externos – causados por algo fora da fronteira do sistema; internos – ativados por algo dentro da fronteira do sistema, como por exemplo, chamada a um método via mensagem; temporais – causados pela ocorrência de uma data e hora específicos ou pela passagem de tempo;
- **atividades** - são associadas a estados. Trata-se de execuções não-atômicas em andamento enquanto o objeto permanece em um estado. As atividades podem ser interrompidas por algum evento;
- **ações** são associadas a transições. Representam computações atômicas executáveis (por não poderem ser interrompidas) que podem agir diretamente no objeto que está sendo modelado e indiretamente em outros objetos que estão visíveis ao objeto. Na implementação do *software*, podem representar a invocação de um método da classe do diagrama de estado.

Uma transição pode ser observada como sendo constituída de cinco partes:
(1) **Estado de origem** – estado afetado pela transição;
(2) **Evento de ativação** – evento que faz com que a transição seja ativada, desde que a condição de guarda, se existir, seja satisfeita. Sua sintaxe é *nome_evento(lista de argumentos)* Os argumentos devem ser separados por vírgula e podem expressar seu tipo (*arg_1 : tipo, arg_2*);
(3) **Condição de guarda** – é uma expressão booleana entre colchetes que é avaliada quando a transição é iniciada pela recepção do evento de ativação. A transação ocorre somente se a condição for verdadeira;
(4) **Ação** – será executada se a transição for efetivada. As ações podem incluir: chamadas a métodos (do objeto especificado ou de outro visível), chamadas à criação ou destruição de outros objetos ou o envio de um sinal a um objeto;
(5) **Estado de destino** – o estado ativado após a conclusão da transição.

A notação básica da transição pode ser assim representada por seus parâmetros: *nome_evento (lista de argumentos) [condição de guarda] / ação.* Todos os parâmetros da transição são opcionais, inclusive seu nome (quando sua ocorrência for evidente). Exemplificando o diagrama de estados, modelou-se na Figura 3.15, os possíveis estados de um objeto da classe *Veículo*, de um sistema de locação de automóveis. Um veículo pode estar em três estados *Disponível*, *Locado* e *Indisponível*. A transição inicial para o estado *Disponível* é *cadastrar* com as seguintes ações: *status:=disponível; placa:=HRJ4889; cor:=preto; modelo:=FiestaCLX; ano:=1997; fabricante:=Ford; KmAtualizada:=65457*. Estando *Disponível*, o veículo pode ser *Locado* para isso, a transição rotulada pelo evento *locar* precisa ser ativada.

Estando neste estado, o veículo pode voltar ao estado *Disponível*. Para que isto ocorra, o evento *devolução* deve acionar sua transição. O veículo pode tornar-se *Indisponível* com a ativação do evento *manutenção*, e retornar ao estado anterior com o evento *disponibilizar*.

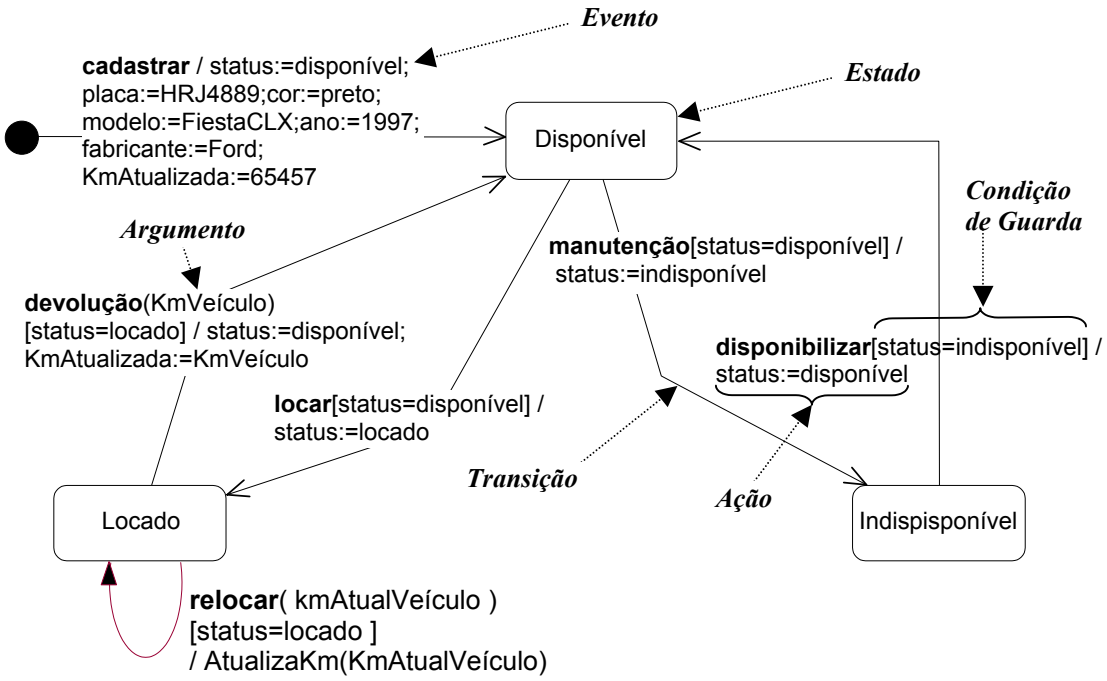


FIGURA 3.15 – Diagrama de Estados para um objeto da classe Veículo do sistema de Locação de Veículos.

A diversidade de informações disponibilizadas pelo diagrama de estados permite a verificação dos seguintes aspectos:

- criação dos objetos pelas transições iniciais de estado – verificar os métodos construtores;
- a especificação e implementação de todos os métodos envolvidos em uma transição – verificando as características das condições de guarda (pré condições) dos métodos (utilizar as diretrizes apresentadas na Seção 3.1.2 e 3.1.3 para o teste de métodos e relacionamentos da classe);
- a implementação das condições de guarda do evento em uma transição – analisar os resultados pela utilização de dados de teste falsos e verdadeiros (para a geração dos dados de teste, utilizar as diretrizes e critérios para o teste de condições e combinações de condições apresentado na Seção 3.1.1.1);
- a liberação de memória quando os objetos forem destruídos por atingirem um estado final – verificar o funcionamento dos métodos destrutores;
- a especificação e implementação das ações em uma transição – verificar os métodos onde as ações forem implementadas;
- a especificação e implementação de todas as atividades em um estado – verificar os métodos onde as atividades foram implementadas;
- a especificação das mensagens envolvidas nas transições;
- o comportamento de um objeto quanto ao estado no qual se encontra – verificar se os resultados obtidos estão de acordo com os esperados.

Utilizando o diagrama de estados, foram definidos os critérios de cobertura de teste de estados. Trata-se de uma adaptação feita nos critérios definidos por McGregor [MCG 96] e apresentados na Seção 2.3.4.1:

- **Todos os Estados** – todos os estados de um diagrama de seqüência devem ser visitados, pela execução de pelo menos uma transição para cada estado. Caso existam atividades nas transições internas de um estado, deve-se verificar sua realização, testando sua interrupção e implementação nos métodos da classe. Com base na especificação, pode-se verificar se todos os métodos especificados no diagrama em determinado estado, foram corretamente implementados. Daí a importância em verificar-se o comportamento do objeto em cada estado, comparando os resultados encontrados com os que foram especificados. Os métodos podem ser testados de acordo com as diretrizes definidas na Seção 3.2.2;
- **Todas as Transições** – exercitar todas as transições pelo menos uma vez, testando a ocorrência de todos os eventos. Caso existam condições de guarda, estas devem ser testadas com valores falsos e verdadeiros. Também deve ser testada a realização das ações e sua correta implementação;
- **Todas-N-Transições** – Testar todas as combinações (seqüências) de transições do diagrama de estados. Exercitar as transições várias vezes;
- **Todos os Caminhos** – Testar todos os caminhos do diagrama de estados pelo menos uma vez. Ao utilizar este critério, exercitam-se todas as transições pela realização de seus eventos. Sendo assim, todos os estados são visitados, podendo com isso, confirmar as suas implementações.

3.4.1 Implementando o teste baseado no diagrama de estados

Esta seção apresenta uma proposta para a implementação de uma ferramenta que faça a geração automática dos caminhos de teste baseados no diagrama de estados. Utilizou-se como base a técnica apresentada por Binder que traduz um diagrama de transição de estados em uma árvore de transições e a partir desta define a seqüência dos caminhos de teste [BIN 95].

A ferramenta deverá prover uma estrutura que permita o armazenamento dos dados relacionados ao diagrama de estados (estados e transições). Na Figura 3.16, é apresentado um modelo conceitual para exemplificar uma possível estrutura de armazenamento destes dados. Neste modelo, para cada transição de um estado registram-se: nome do evento, argumentos, condição de guarda e ações e pelo relacionamento com a classe estado, determinam-se seus estados de origem e destino. Com estas informações é possível gerar a árvore de transições e consequentemente os caminhos de teste. O modelo conceitual apresentado segue o princípio de Fowler, que sugere a criação de um modelo de estados para somente uma classe [FOW 2000] e [OMG 99].

Considerou-se nesta proposta que um estado que não é destino para nenhuma transição, é considerado o estado inicial. Em contrapartida, o estado que não é origem em nenhuma transição é considerado um estado final.

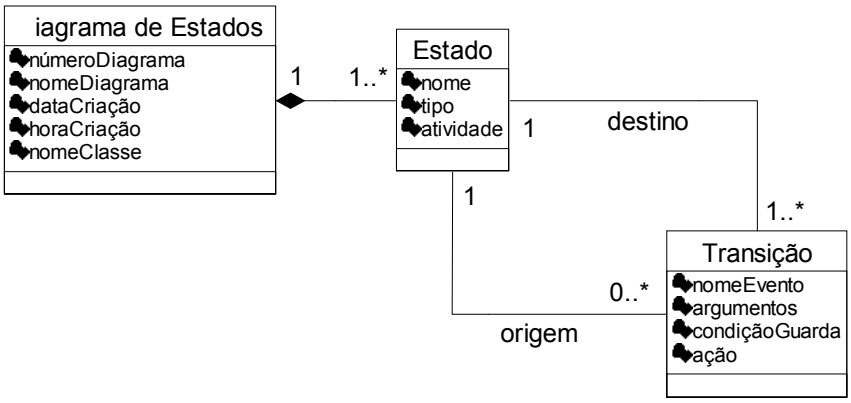


FIGURA 3.16 – Modelo conceitual para armazenamento dos dados do diagrama de estados.

Na Tabela 3.9 foram agrupados os dados sobre as transições do diagrama de estados da Figura 3.15, permitindo com isso, a geração da árvore de transições da Figura 3.17. Foram definidas algumas diretrizes para que o desenvolvedor da ferramenta possa implementar a conversão dos dados armazenados, em uma árvore de transições:

1) Pesquisar a transição que possua o estado de origem como *inicial* e designar o estado de destino desta transição como sendo o estado raiz (primeiro estado) da árvore de transições;
2) Pesquisar a próxima transição que possua o estado atual como sendo o estado de origem. Caso existam transições com o mesmo estado de origem, escolher uma em que o estado de destino não seja igual ao de origem. Executar este passo até que o estado de destino já tenha sido um estado de origem neste mesmo caminho ou o estado destino não seja estado de origem em nenhuma outra transição;
3) Encerrado um caminho, caso ainda existam transições, pesquisar uma transição que possua como estado de origem o primeiro estado da árvore ou algum estado depois deste;
4) Repitir os passos 2 e 3 até que todos os caminhos tenham sido gerados.

TABELA 3.9 – Informações armazenadas do diagrama de estados da Figura 3.15.

NomeEvento	Argumento	Condição de Guarda	Ações	Estado de Origem	Estado de Destino
cadastrar	∅	∅	status:=disponível; placa:=HRJ4889; cor:=preto; modelo:=FiestaCL; ano:=1997; fabricante:=Ford; KmAtualizada:=65457	Inicial	Disponível
Locar	∅	status=disponível	status:=locado	Disponível	Locado
Manutenção	∅	status=disponível	status:=indisponível	Disponível	Indisponível
Devolução	KmVeículo	status=locado	status:=disponível; KmAtualizada:=KmVeículo	Locado	Disponível
Disponibilizar		status=indisponível	status:=disponível	Indisponível	Disponível
Relocar	kmAtualVeículo	[status=locado]	AtualizaKm(KmAtualVeículo)	Locado	Locado

∅ = vazio.

Caso exista alguma transição associada à classe em teste, que não possua ligação com nenhum estado da árvore, isto indica que esta transição foi mal especificada ou ocorreu um erro na transferência dos dados do diagrama.

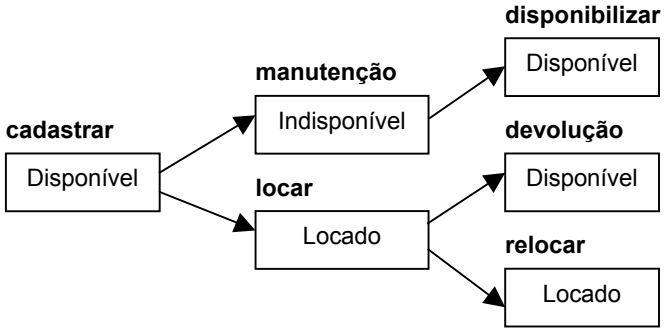


FIGURA 3.17 – Árvore de transições do diagrama de estados da Figura 3.15.

Analisando a árvore de transições derivada do diagrama de estados e apresentada na Figura 3.17, é possível gerar três seqüências distintas de caminhos de teste. Entende-se por seqüência de caminho, o conjunto de estados que formam o caminho de teste, partindo-se do estado raiz até o estado final do ramo. Na Figura 3.18, apresentam-se os três caminhos extraídos da árvore de transições.

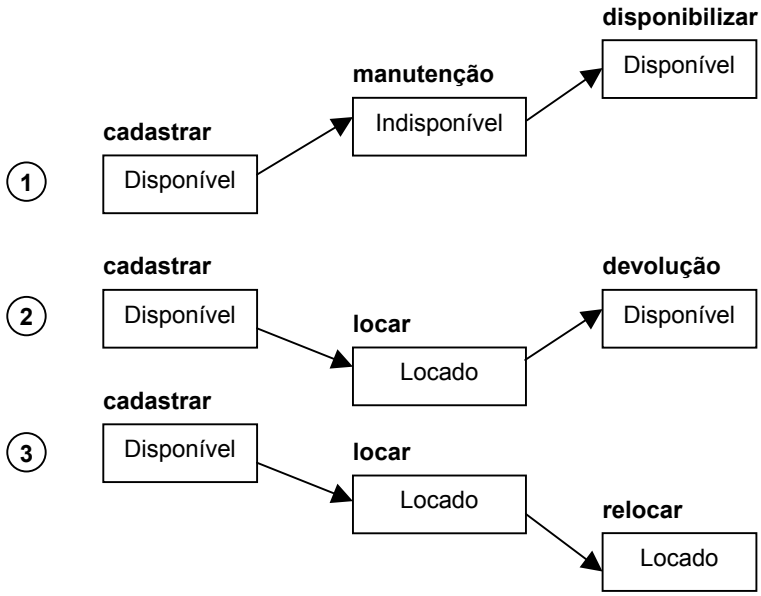


FIGURA 3.18 – Caminhos de teste gerados pela árvore de transições da Figura 3.17.

Tendo os caminhos, o próximo passo será a definição dos dados de teste para que estes caminhos sejam exercitados, permitindo a realização de todos os eventos e podendo com isso analisar as transições de estados do objeto que foi modelado. Um exemplo da geração de casos de teste pode ser observado na Seção 2.3.4 no teste baseado em estados.

Esta proposta de implementação aborda os aspectos básicos apresentados nesta seção. O desenvolvedor pode optar por outra alternativa em função da linguagem de programação utilizada para sua codificação.

A eficiência dos critérios está associada à qualidade da especificação desenvolvida. Quanto mais detalhes forem acrescentados aos diagramas, maior será o número de informações disponibilizadas para a geração dos dados de teste. No teste baseado em especificações diagramáticas deve-se considerar as informações obtidas na fase de análise de requisitos, pois neste momento, registram-se as expectativas do usuário em relação ao *software* que está sendo produzido. Observa-se com isso a importância da interação com o usuário no processo de desenvolvimento. A qualidade do produto final depende deste grau de participação e do desempenho do projetista de software no momento em que forem levantadas as informações necessárias.

Observou-se, durante o desenvolvimento das diretrizes e critérios, que a maioria das informações utilizadas no teste de *software* pode ser obtida pela análise de especificações desenvolvidas em diagramas. Com isso, o teste de pequenas aplicações pode ser facilitado, uma vez que a maioria das pequenas e médias empresas de desenvolvimento não possuem recursos suficientes para a aquisição de ferramentas e nem tempo para aplicar técnicas complexas de teste. Logo, utilizam o teste funcional como instrumento para a avaliação de seus produtos [MET 2000].

Na Tabela 3.10, relacionou-se os critérios de cobertura de teste definidos neste trabalho para os diagramas de classe, seqüência, colaboração e estados.

TABELA 3.10 – Relação de Critérios de Cobertura definidos para o teste baseado em especificações diagramáticas UML.

Critérios	Aplicação – Diagrama
Todas-Combinações-E **Combinações-VF** **Todas-Combinações-OU** **Combinações-FV**	- Condições e Combinação de Condições - Condição de guarda de transição - Estados - Condição de mensagens – Seqüência e Colaboração - Pré e Pós Condições dos Métodos – Classes - Restrições de Atributos – Classes - Restrições de Associação – Classes
Todos-Métodos **Todos-Métodos-Sub** **Todos-Métodos-NRHr** **Todos-Métodos-NR** **Todos-Atributos-Sub** **Todos-Atributos-NH** **Todos-Atributos-N**	- Métodos - Classe; - Associação do Tipo Generalização - Classe
Todas as Multiplicidades **Todas-Multiplicidades-Obrigatórias**	- Multiplicidades do Relacionamento - Classe
Todas-S-Mensagens **Todas-C-Mensagens** **Todos-Mensagens-Iterativas**	- Seqüência de Mensagens – Seqüência e Colaboração
Todas-Interações-Totais **Todas-Interações-Parciais**	- Interações – Colaboração
Todos os Estados **Todas as Transições** **Todas-N-Transições** **Todos os Caminhos**	- Transições – Estado

Neste trabalho também foram definidas diretrizes de teste para alguns tipos de diagramas. Isto foi feito para nortear os testes depois que a aplicação já estivesse

sendo implementada. Na Tabela 3.11 foram relacionadas as diretrizes, bem como os diagramas para os quais foram elaboradas.

TABELA 3.11 – Relação das Diretrizes definidas para o teste baseado em especificações diagramáticas UML.

Diagrama *(elemento) – Local Texto*	**Diretrizes de Teste**
Diagrama de Classes ***(atributos) - Seção 3.1.1***	• verificar se os atributos especificados no modelo foram devidamente declarados na implementação da classe; ▪ observar se a visibilidade dos atributos obedece à definida na especificação da classe; ▪ verificar se o tipo do atributo declarado é igual ao especificado na classe; • verificar se as condições da restrição do atributo, quando declaradas, foram devidamente implementadas nos métodos que alteram o valor deste atributo; • nos casos em que for especificado valor inicial para o atributo, verificar se esta característica foi corretamente implementada; • verificar se o atributo foi implementado de acordo com sua propriedade (*changeable, addOnly, frozen*). Caso esta seja declarada.
Diagrama de Classes ***(métodos) - Seção 3.1.2***	• verificar se os métodos especificados no modelo foram devidamente declarados na implementação da classe; ▪ verificar se a visibilidade dos métodos obedece à definida na especificação da classe; ▪ verificar se os parâmetros especificados estão em conformidade com a implementação; ▪ verificar se os tipos de retorno, caso tenham sido especificados, estão corretamente implementados. • verificar todos os métodos *get* e *set*, para analisar se estão executando corretamente seus propósitos - leitura: devolver o valor de um campo e modificação: colocar o valor em um campo; • analisar a correta implementação das pós e pré condições do método; • analisar grupos de métodos que manipulem os mesmos atributos; • verificar o método construtor – criação de objetos; • verificar se os métodos foram implementados de acordo com sua propriedade.
Diagrama de Classes ***(Relacionamento Generalização) - Seção 3.2.3.1***	• os métodos herdados e testados na superclasse não são re-testados nas subclasses, desde que não interajam com métodos redefinidos; • os métodos novos e os redefinidos na subclasse devem ser testados; • os atributos testados na superclasse não são testados nas subclasses; • os atributos novos devem ser testados; • os atributos da superclasse, que forem manipulados por métodos modificadores das subclasses, devem ser re-testados.
Diagrama de Classes ***(Associação Simples, Agregação Simples e por Composição) - Seção 3.1.3.2***	• verificar todas as associações e agregações simples quanto a sua multiplicidade – exercitar os métodos construtores e aqueles que efetivam a relação entre as classes de acordo com os valores máximos e mínimos da multiplicidade; • verificar todas as associações do tipo agregação por composição quanto às suas peculiaridades; ▪ verificar todos as pré e pós condições dos métodos que excluam objetos das classes todo e parte – exercitando estes métodos; ▪ verificar todos os métodos construtores e destrutores das classes que façam parte de uma associação do tipo agregação por composição – exercitar os métodos considerando as características da associação e multiplicidades existentes; • verificar todas as restrições de uma associação quanto a sua correta implementação – exercitar os métodos onde as condições das restrições tenham sido implementadas.
Diagrama de Classes ***(Multiplicidade) - Seção 3.1.3.4***	• nos casos em que os valores máximos e mínimos de uma multiplicidade sejam especificados, deve-se projetar casos de teste com os valores máximo, mínimo e outros logo acima e abaixo destes valores (conforme BVA); • nos casos onde o valor máximo não seja especificado, projetar casos de teste com valores pelo menos duas vezes maiores (no mínimo) que o valor mínimo e outros logo acima e abaixo destes valores (+1 e –1).
Diagrama de Seqüência ***(Conversão em Matriz Caminho) - Seção 3.2.1***	• Quando uma mensagem possuir um marcador de iteração: (1) a primeira mensagens iterativas, na matriz de caminho, deve indicar a próxima mensagem e aquela que finaliza a iteração; (2) a última mensagem iterativa deve indicar a mensagem iterativa inicial e aquela que finaliza a iteração. Assim sendo, estabelece o caminho para o teste da iteração (*loop*). • Quando em uma mensagem existir uma condição, a mensagem exatamente anterior a esta deve indicar quais são as mensagens a serem escolhidas. Logo, estabelece as opções de caminho para a condição existente. • Caso a mensagem não contenha nenhuma das características apresentadas nas diretrizes anteriores, esta deve indicar apenas qual será a próxima mensagem a ser enviada. • As mensagens que indiquem o final da seqüência, não devem ser conectadas a nenhuma outra mensagem na matriz de caminho.

No próximo capítulo, para demonstrar a aplicação dos critérios de cobertura e das diretrizes para o teste com base em especificações diagramáticas UML, desenvolveu-se um estudo de caso tendo como base a modelagem de um módulo de sistema para o controle da emissão de ordens de serviço em aeronaves. Foram elaborados os diagramas de casos de uso, classes, estados, seqüências e colaborações.

Aplicando-se as técnicas de teste apresentadas neste trabalho, foram geradas as tabelas com os possíveis caminhos de teste em função dos critérios utilizados.

4 Estudo de Caso: Módulo de controle para emissão de ordens de serviço em aeronaves

Com o propósito de verificar o uso das diretrizes e dos critérios de cobertura em uma situação real, realizou-se um estudo de caso para a Tacape Oficina de Aviões Ltda. Os diagramas utilizados, neste estudo de caso, fazem parte do projeto CONTROLER que prevê a completa automação das rotinas desenvolvidas pela empresa.

O projeto CONTROLER encontra-se na fase de especificação do sistema. Neste estudo de caso, será utilizada a modelagem do módulo de controle para emissão de Ordens de Serviço. As informações de teste definidas neste capítulo serão utilizadas no decorrer do projeto, quando for iniciada a implementação do sistema. Os diagramas utilizados neste estudo de caso foram: diagrama de classes, diagramas de seqüência, diagramas de colaboração e diagramas de estado.

Na próxima seção será analisado o diagrama de classes, para a derivação das informações de teste tendo como base as diretrizes e critérios de cobertura para este tipo de diagrama.

4.1 Analisando o Diagrama de Classes do Estudo de Caso

Considerando a quantidade de classes existentes no diagrama de classes e a similaridade de suas características, serão analisadas somente as classes *Aeronave*, *Ordem de serviço*, *Mecânico*, *Credenciado*, *Aprendiz*, *HéliceAeronave* e *MotorAeronave*, juntamente com seus atributos, métodos, multiplicidades e relacionamentos.

Na tentativa de facilitar a apresentação do diagrama de classes, as classes submetidas ao teste serão apresentadas individualmente. O diagrama de classes (visão simplificada) para o módulo de controle da emissão de ordens de serviço, pode ser visualizado na Figura 4.1. Todos os diagramas utilizados no estudo de caso (visão completa) podem ser visualizados nos anexos (2 – 17) deste trabalho. A ordem em que serão aplicadas as diretrizes e critérios de cobertura de teste será a mesma do Capítulo 3 (atributos, métodos, relacionamentos). A Figura 4.2, apresenta a classe *Aeronave* com seus atributos e métodos.

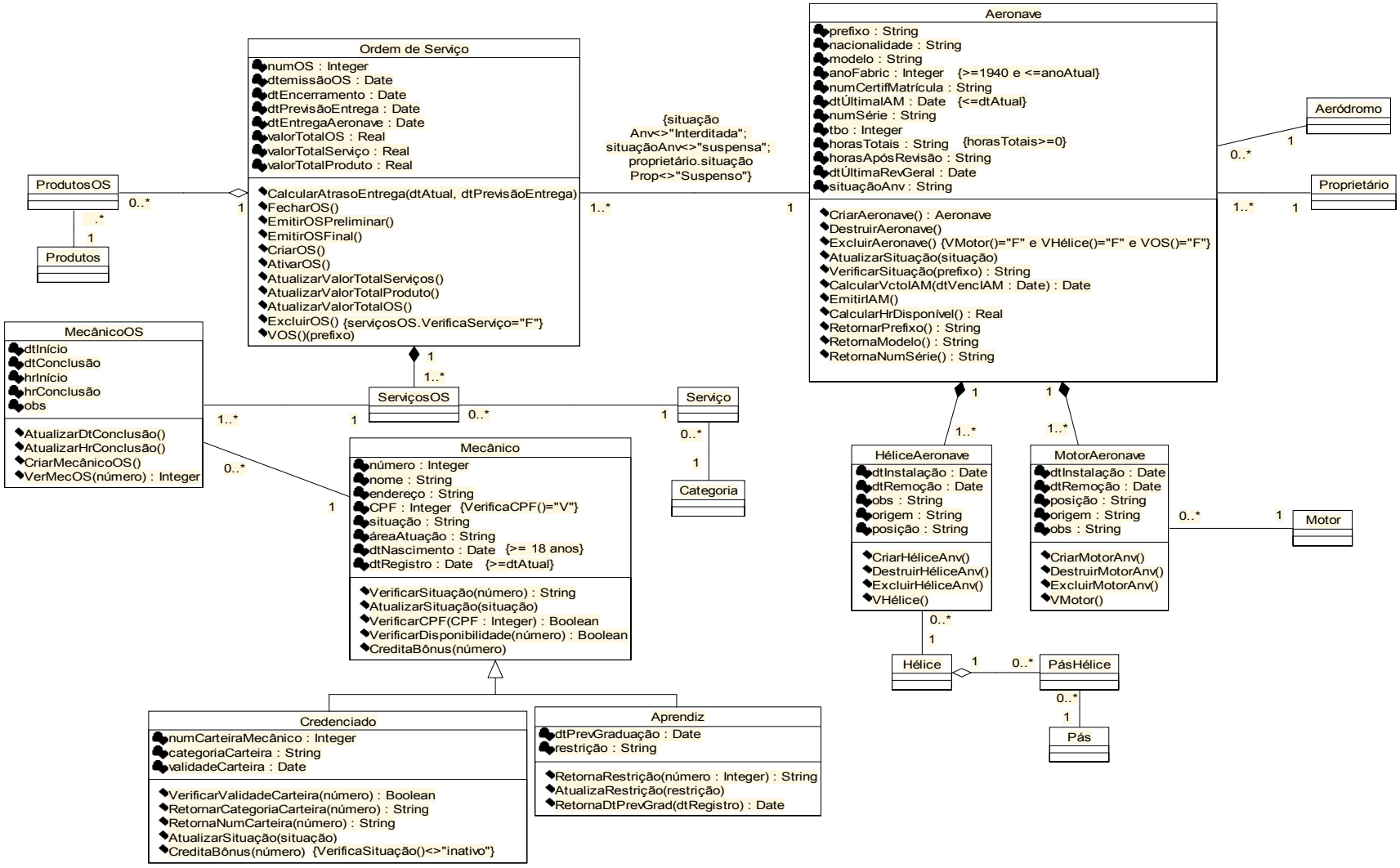


FIGURA 4.1 – Diagrama de Classes do estudo de caso.

4.1.1 Cobertura de Restrições sobre Atributos

Observando a especificação da classe *Aeronave*, e seguindo as diretrizes para verificação de atributos de um diagrama de classes (Seção 3.1.1), é possível verificar por comparação se os atributos especificados foram devidamente implementados quanto a sua visibilidade e tipo.

Alguns atributos da classe possuem restrições com condições e, para testá-las, foram utilizadas as diretrizes e critérios de cobertura por condições e combinação de condições definidos na Seção 3.1.1.1. Esta cobertura estabelece que para o teste dos operadores relacionais devem ser verificados os limites dos valores (limite, limite+1 e limite-1) e, caso haja combinação de condições, utilizar os critérios de acordo com o operador lógico existente.

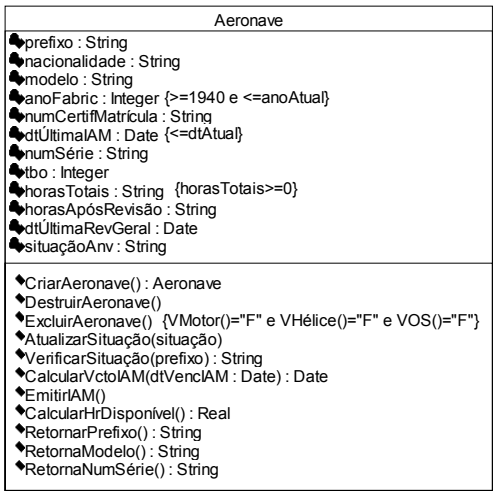


FIGURA 4.2 – Classe *Aeronave* com seus atributos e métodos.

As restrições simples (somente uma condição) que não utilizam operadores lógicos (e/ou) devem ser testados quanto a implementação dos operadores relacionais. Nas Tabelas 4.1 e 4.2 são apresentados os casos de teste definidos para verificar a restrição dos atributos *dtÚltimaIAM* e *horasTotais*.

TABELA 4.1 – Casos de teste para o atributo *dtÚltimaIAM* da classe *Aeronave*.

Atributo: ***dtÚltimaIAM***		*Data Atual:* ***29/03/2000***		
Restrição(ões): ***dtÚltimaIAM <= dtAtual***				
*Parâmetro**		*Valores de Teste*	*Resultado Esperado*	*Resultado Obtido*
1	*Valor Limite*	***29/03/2000***	**V**	
2	*Valor Limite + 1*	***30/03/2000***	**F**	
3	*Valor Limite – 1*	***28/03/2000***	**V**	

* Em relação ao atributo sendo verificado.

TABELA 4.2 – Casos de teste para o atributo *horasTotais* da classe *Aeronave*.

Atributo: ***horasTotais***		*Data Atual:* ***29/03/2000***	
Restrição(ões): ***horasTotais >=0***			
*Parâmetro**	*Valores de Teste*	*Resultado Esperado*	*Resultado Obtido*
1 *Valor Limite*	***0:00***	***V***	
2 *Valor Limite + 1*	***0:01***	***V***	
3 *Valor Limite – 1*	***-0:00***	***F***	

* Em relação ao atributo sendo verificado.

Ao testar a restrição do atributo, o desenvolvedor, utilizando a ficha de avaliação, deve comparar o resultado obtido com o esperado. Havendo discrepância, deve verificar na implementação a possível causa do erro.

Na classe *Aeronave*, o atributo *anoFabric* possui uma restrição que é uma combinação com duas condições *{>=1940 e <=AnoAtual}* incluindo o operador lógico "e". Como nas condições existem operadores relacionais e seus limites devem ser testados, utilizou-se a ficha para avaliação de condições simples para determinar os valores válidos e inválidos para os casos de teste. As Tabelas 4.3 e 4.4 apresentam os casos de teste definidos para as duas condições.

TABELA 4.3 – Casos de teste com os valores válidos e inválidos para o atributo *anoFabric* em sua restrição *{>=1940}*.

Atributo: ***anoFabric***		*Data Atual:* ***29/03/2000***	
Restrição(ões): ***anoFabric>= 1940***			
*Parâmetro**	*Valores de Teste*	*Resultado Esperado*	*Resultado Obtido*
1 *Valor Limite*	***1940***	**V**	
2 *Valor Limite + 1*	***1941***	**V**	
3 *Valor Limite – 1*	***1939***	**F**	

* Em relação ao atributo sendo verificado.

TABELA 4.4 – Casos de teste com os valores válidos e inválidos para o atributo *anoFabric* em sua restrição *{<=AnoAtual}*.

Atributo: ***anoFabric***		*Data Atual:* ***29/03/2000***	
Restrição(ões): ***anoFabric<= AnoAtual***			
*Parâmetro**	*Valores de Teste*	*Resultado Esperado*	*Resultado Obtido*
1 *Valor Limite*	***2000***	**V**	
2 *Valor Limite + 1*	***2001***	**F**	
3 *Valor Limite – 1*	***1999***	**V**	

* Em relação ao atributo sendo verificado.

Para verificar a combinação com o operador lógico "e", pode-se optar por utilizar um dos critérios de cobertura para este tipo de operador (**Todas-Combinações-E** ou **Combinações-VF**). Na Tabela 4.5, utilizando os valores definidos nas Tabelas 4.3 e 4.4, são apresentadas as combinações com os quais pode-se avaliar as condições do atributo *anoFabric*.

TABELA 4.5 – Combinação de valores para o teste das condições do atributo *anoFabric*.

	Combinação		Resultado Esperado	Resultado Obtido
	>=1940	**<=anoAtual**		
(a)	**1940**	**2000**	**V**	
(b)	**1939**	**2001**	**F**	
(c)	**1941**	**2001**	**F**	
(d)	**1939**	**1999**	**F**	

Optando pelo critério de cobertura **Todas-Combinações-E**, devem ser testadas todas as combinações apresentadas. De acordo com o critério de cobertura **Combinações-VF**, deve ser testada a combinação ***(a)*** e qualquer uma de ***(b)*** a ***(d)***. Cabe ao desenvolvedor, de acordo com as características do *software* em teste, determinar qual dos critérios de cobertura será utilizado. Para este sistema, o ano de fabricação da aeronave é utilizado em todo o sistema determinando o tipo de serviço a ser realizado, os boletins a serem aplicados, Portanto, deve-se optar pelo critério de cobertura mais forte **Todas-Combinações-E**.

Quando existe combinação de condições, a determinação dos dados de teste deve ser feita em duas etapas, primeiro são definidos os casos de teste para os limites e depois, com base nestes valores, as combinações.

4.1.2 Cobertura de Restrições sobre Métodos

Utilizando a especificação da classe *Aeronave* da Figura 4.1 e as diretrizes para a verificação dos métodos especificados na classe, pode-se comparar a implementação destes métodos quanto a sua visibilidade, tipo de retorno, passagem de parâmetros e propriedades.

As restrições dos métodos, tais como as dos atributos, também devem ser testadas. O método *ExcluirAeronave()* possui como restrição uma combinação de condições *{Vmotor()="F" e Vhélice="F" e VOS="F"}*, a qual estabelece que para excluir uma aeronave, a mesma não pode estar associada à ordem de serviço, motor e hélice. Como o operador relacional utilizado na condição é o de igualdade, basta testá-la com um valor falso e um verdadeiro. A tabela 4.6 apresenta as combinações para testar a combinação da restrição do método *ExcluirAeronave()*.

Analisando as combinações da Tabela 4.6, pelo critério de cobertura **Todas-Combinações-E** devem ser testadas todas as combinações, mas pelo critérios **Combinações-VF**, testar a combinação ***(a)*** e qualquer uma de ***(b)*** a ***(h)***. Na combinação do método *ExcluirAeronave(),* existem mais de duas condições, portanto, como sugerido nas diretrizes para o teste de condições, é importante testar uma combinação que resulte verdadeiro, uma que resulte falso e qualquer uma das outras.

TABELA 4.6 – Combinação de valores para o teste das condições do método *ExcluirAeronave()*.

	Combinação			Resultado Esperado	Resultado Obtido
	Vmotor()	*Vhélice()*	*VOS()*		
(a)	**F**	**F**	**F**	**V**	
(b)	**V**	**V**	**V**	**F**	
(c)	**V**	**V**	**F**	**F**	
(d)	**V**	**F**	**F**	**F**	
(e)	**F**	**V**	**V**	**F**	
(f)	**F**	**F**	**V**	**F**	
(g)	**F**	**V**	**F**	**F**	
(h)	**V**	**F**	**V**	**F**	

4.1.3 Cobertura de Teste para a Herança de classes

A forma como os atributos e métodos de uma classe são verificados é dependente do tipo de relacionamento existente entre as classes. O relacionamento do tipo generalização existente entre as classes *Mecânico*, *Credenciado* e *Aprendiz* é apresentado na Figura 4.3.

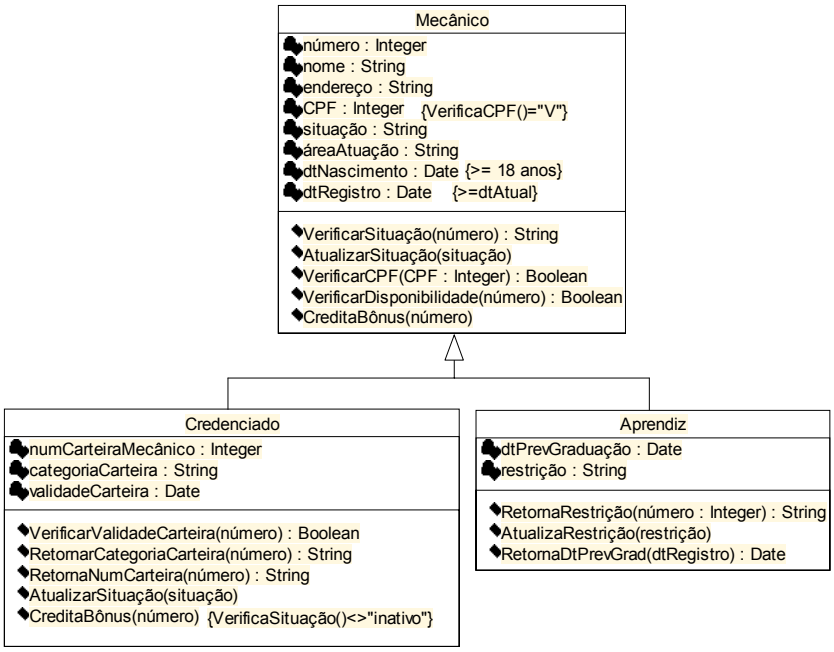


FIGURA 4.3 – Classes *Mecânico*, *Credenciado* e *Aprendiz* com relacionamento do tipo generalização.

Quando existir este tipo de relacionamento, representando a herança entre classes, as subclasses não podem ser testadas isoladamente. Seguindo as diretrizes para o teste deste tipo de relacionamento, primeiramente deve ser testada a superclasse

e depois as subclasses. No teste da subclasse *Credenciado*, de acordo com as diretrizes de teste, os seguintes elementos devem ser verificados:

- os atributos herdados *nome*, *endereço*, *CPF*, *áreaAtuação*, *dtNascimento* e *dtRegistro* não precisam ser reconsiderados pois já devem ter sido testados na superclasse e os métodos da subclasse não manipulam estes atributos (neste caso os atributos possuem visibilidade privada);
- os atributos herdados *número* e *situação* devem ser retestados pois são manipulados pelos métodos da subclasse (pois são parâmetros dos métodos da subclasse);
- os métodos redefinidos *AtualizarSituação()* e *CreditaBônus* devem ser testados;
- os demais atributos e métodos definidos na subclasse devem ser testados.

No teste da subclasse *Aprendiz* verificar os seguintes elementos:

- os atributos herdados *nome*, *endereço*, *CPF*, *situação*, *áreaAtuação* e *dtNascimento* não precisam ser novamente testados pois já foram testados na superclasse e os métodos da subclasse não manipulam estes atributos (neste caso os atributos possuem visibilidade privada na superclasse);
- os atributos herdados *número* e *dtRegistro* devem ser retestados pois são manipulados pelos métodos da subclasse (pois são parâmetros dos métodos da subclasse);
- os demais atributos e métodos definidos na subclasse devem ser testados.

Tendo como base os critérios de cobertura definidos para o teste de subclasses da Seção 3.1.3.1, foram definidos, na Tabela 4.7, os atributos e métodos de cada uma das subclasses que devem ser testados.

TABELA 4.7 – Atributos e métodos das subclasses *Credenciado* e *Aprendiz* selecionados pelos critérios de cobertura para o teste de subclasses.

Critério de Cobertura	Subclasse *Credenciado*	Subclasse *Aprendiz*
Todos-Métodos-Sub	VerificarSituação() AtualizarSituação() VerificarCPF() VerificarDisponibilidade() CreditaBônus() VerificaValidadeCarteira() RetornaCategoria() RetornaNumCarteira() AtualizarSituação() CreditaBônus()	VerificarSituação() AtualizarSituação() VerificarCPF() VerificarDisponibilidade() RetornaRestrição() AtualizaRestrição() RetornaDtPrevGrad()
Todos-MétodosNRHr	VerificaSituação() VerificaValidadeCarteira() RetornaCategoria() RetornaNumCarteira() AtualizarSituação() CreditaBônus()	RetornaRestrição() AtualizaRestrição() RetornaDtPrevGrad()
Todos-Métodos-NR	VerificaValidadeCarteira() RetornaCategoria() RetornaNumCarteira() AtualizarSituação() CreditaBônus()	RetornaRestrição() AtualizaRestrição() RetornaDtPrevGrad()
Todos-Atributos-Sub	número, nome, endereço, CPF, situação, áreaAtuação, dtNascimento, dtRegistro, numCarteiraMecânico, categoriaCarteira, validadeCarteira	número, nome, endereço, CPF, situação. ÁreaAtuação, dtNascimento, dtRegistro, dtPrevGraduação, restrição

Critério de Cobertura	Subclasse *Credenciado*	Subclasse *Aprendiz*
Todos-Atributos-NH	número, situação, dtRegistro, numCarteiraMecânico, categoriaCarteira, validadeCarteira	número, dtRegistro, dtPrevGraduação, restrição
Todos-Atributos-N	numCarteiraMecânico, categoriaCarteira, validadeCarteira	DtPrevGraduação, restrição

Conforme o critério de cobertura selecionado, o desenvolvedor, pode elaborar os casos de teste para verificar os atributos e exercitar os métodos das subclasses.

4.1.3 Cobertura de Restrição sobre Associações

A associação simples existente entre as classes *Aeronave* e *Ordem de serviço* possui uma restrição que combina de três condições, indicando que a emissão de uma ordem de serviço está condicionada à situação da aeronave e do seu proprietário. O teste das restrições da associação deve seguir as diretrizes e os critérios de cobertura para o teste de condições. A Figura 4.4 apresenta a associação entre as duas classes.

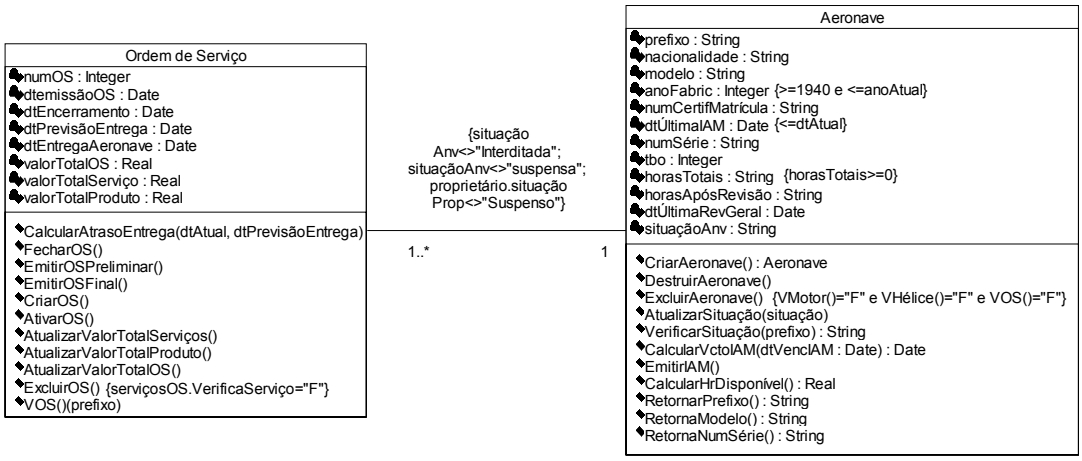


FIGURA 4.4 – Associação simples com restrição entre as classes *Aeronave* e *Ordem de Serviço*.

Na Tabela 4.8 são apresentadas as combinações de valores para testar as condições da restrição da associação *Aeronave/Ordem de Serviço*.

TABELA 4.8 – Casos de teste da restrição da associação entre as classes *Aeronave* e *Ordem de Serviço*.

	Combinação			**Resultado Esperado**	**Resultado Obtido**
	SituaçãoAnv() <>"Interditada"	*SituaçãoAnv() <>"Suspensa"*	*Proprietário. SituaçãoProp() <> "Suspenso"*		
(a)	**Liberada**	**Liberada**	**Ativo**	**V**	
(b)	**Interditada**	**Suspensa**	**Suspenso**	**F**	
(c)	**Liberada**	**Liberada**	**Suspenso**	**F**	
(d)	**Liberada**	**Suspensa**	**Suspenso**	**F**	
(e)	**Interditada**	**Liberada**	**Ativo**	**F**	
(f)	**Interditada**	**Suspensa**	**Ativo**	**F**	

	Combinação			Resultado Esperado	Resultado Obtido
	SituaçãoAnv() <> "Interditada"	*SituaçãoAnv() <> "Suspensa"*	*Proprietário. SituaçãoProp() <> "Suspenso"*		
(g)	**Interditada**	**Liberada**	**Suspenso**	**F**	
(h)	**Liberada**	**Suspensa**	**Ativo**	**F**	

Os critérios de cobertura para esta combinação de condições foram os mesmo utilizados para o teste dos atributos e métodos (Todas-Combinações-E e Combinações-VF).

4.1.4 Cobertura da Associação do Tipo Agregação Composta

Outro tipo de relacionamento, existente neste diagrama de classes, é a associação de agregação por composição. Os métodos construtores, destrutores e aqueles que excluem objetos persistentes destas classes, devem ser verificados quanto às suas pré e pós condições. Na Figura 4.5 é apresentada a relação do tipo agregação por composição entre a classe *Aeronave* (todo) e suas partes *Hélice Aeronave* e *MotorAeronave*.

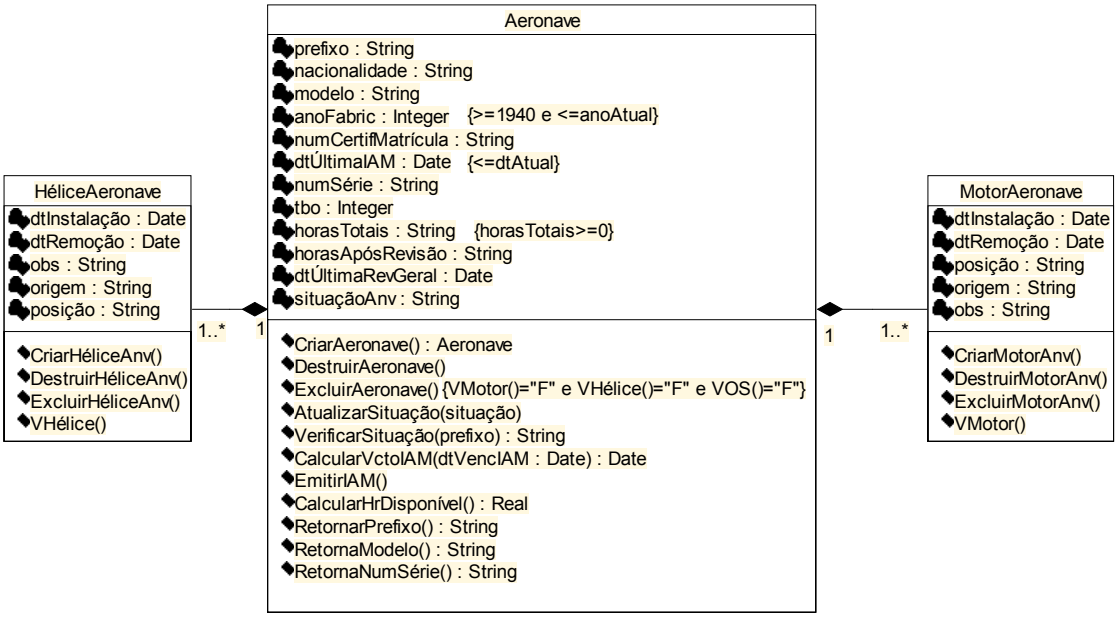


FIGURA 4.5 – Associação de agregação por composição entre as classes *Aeronave*, *Hélice Aeronave* e *MotorAeronave*.

O teste deste tipo de associação é realizado de acordo com a implementação desta característica. Normalmente, a classe *todo* é responsável por gerenciar suas *partes*. Portanto, ao executar o método *CriarAeronave()* da classe *Aeronave*, este deve executar os métodos *CriarHéliceAnv()* e *CriarMotorAnv()*. Executando o método *DestruirAeronave()*, este deve executar *DestruirHéliceAnv()* e *DestruirMotorAnv()*. Outro aspecto importante é a exclusão de objetos persistentes, caso seja executado o método *ExcluirAeronave()*. O efeito cascata deve ocorrer, ou seja, os métodos *ExcluirHéliceAnv()* e *ExcluirMotorAnv()* devem ser executados.

4.1.5 Cobertura da Multiplicidade do Relacionamento

A multiplicidade da associação é uma característica de extrema importância, pois através desta pode-se verificar o funcionamento dos métodos construtores das classes envolvidas. No diagrama de classes deste estudo de caso existem 16 (dezessete) associações entre classes, ou seja, 32 (trinta e dois) pares de multiplicidade. A Figura 4.6 apresenta o diagrama de classes simplificado (sem atributos e métodos) para a visualização das multiplicidades.

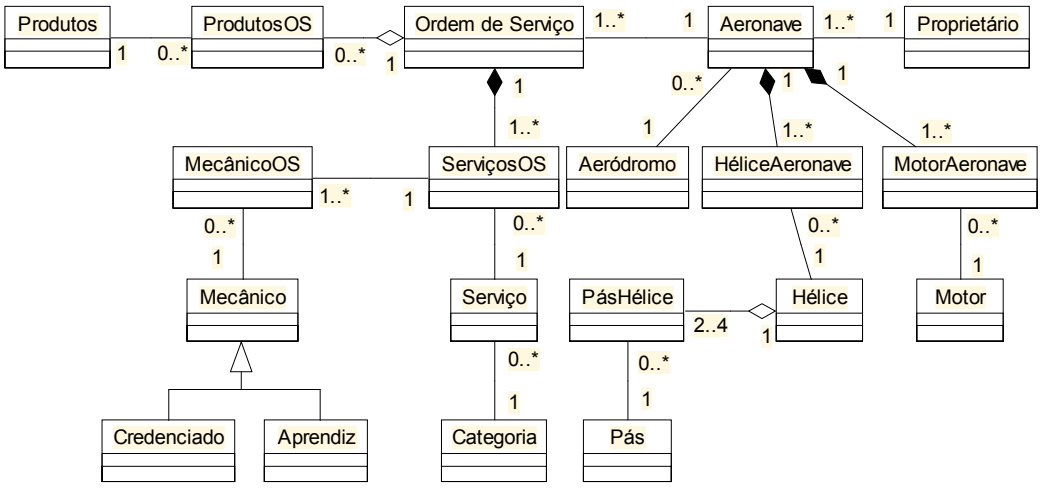


FIGURA 4.6 – Multiplicidades do diagrama de classes do estudo de caso.

Com base nos critérios de cobertura para o teste de multiplicidades, selecionou-se as multiplicidades segundo o critério **Todas as Multiplicidades** (Tabela 4.9) e critério **Todas-Multiplicidades-Obrigatórias** (Tabela 4.10):

TABELA 4.9 – Relação de multiplicidades para o critério Todas as Multiplicidades.

Associação		Multiplicidade Origem	Multiplicidade Destino
Origem	Destino		
Aeronave	Ordem de Serviço	1	1..*
Ordem de Serviço	Aeronave	1..*	1
Aeronave	Aeródromo	0..*	1
Aeródromo	Aeronave	1	0..*
Aeronave	Proprietário	1..*	1
Proprietário	Aeronave	1	1..*
Aeronave	HéliceAeronave	1	1..*
HéliceAeronave	Aeronave	1..*	1
Aeronave	MotorAeronave	1	1..*
MotorAeronave	Aeronave	1..*	1
HéliceAeronave	Hélice	0..*	1
Hélice	HéliceAeronave	1	0..*
MotorAeronave	Motor	0..*	1
Motor	MotorAeronave	1	0..*
Hélice	PásHélice	1	2..4
PásHélice	Hélice	0..*	1
PásHélice	Pás	0..*	1

Associação		Multiplicidade Origem	Multiplicidade Destino
Origem	Destino		
Pás	PásHélice	1	0..*
Ordem de Serviço	ProdutosOS	1	0..*
ProdutosOS	Ordem de Serviço	0..*	1
ProdutosOS	Produtos	0..*	1
Produtos	ProdutosOS	1	0..*
Ordem de Serviço	ServiçoOS	1	1..*
ServiçoOS	Ordem de Serviço	1..*	1
ServiçoOS	Serviço	0..*	1
Serviço	ServiçoOS	1	0..*
Serviço	Categoria	0..*	1
Categoria	Serviço	1	0..*
ServiçoOS	MecânicoOS	1	1..*
MecânicoOS	ServiçoOS	1..*	1
MecânicosOS	Mecânico	0..*	1
Mecânico	MecânicoOS	1	0..*

TABELA 4.10 – Relação de multiplicidades para o critério Todas-Multiplicidades-Obrigatórias.

Associação		Multiplicidade Origem	Multiplicidade Destino
Origem	Destino		
Aeronave	Ordem de Serviço	1	1..*
Ordem de Serviço	Aeronave	1..*	1
Aeronave	Aeródromo	0..*	1
Aeronave	Proprietário	1..*	1
Proprietário	Aeronave	1	1..*
Aeronave	HéliceAeronave	1	1..*
HéliceAeronave	Aeronave	1..*	1
Aeronave	MotorAeronave	1	1..*
MotorAeronave	Aeronave	1..*	1
HéliceAeronave	Hélice	0..*	1
MotorAeronave	Motor	0..*	1
Hélice	PásHélice	1	2..4
PásHélice	Hélice	2..4	1
PásHélice	Pás	0..*	1
ProdutosOS	Ordem de Serviço	0..*	1
ProdutosOS	Produtos	0..*	1
Ordem de Serviço	ServiçoOS	1	1..*
ServiçoOS	Ordem de Serviço	1..*	1
ServiçoOS	Serviço	0..*	1
Serviço	Categoria	0..*	1
ServiçoOS	MecânicoOS	1	1..*
MecânicoOS	ServiçoOS	1..*	1
MecânicosOS	Mecânico	0..*	1

A diferença foi de dez multiplicidades a menos para o segundo critério. Isto reduz o número de casos de teste e garante que pelo menos as associações obrigatórias, que são as mais importantes, sejam verificadas. O teste de multiplicidade deve ser feito nos dois sentidos da associação. Serão apresentados apenas os casos de teste com características diferentes. Nas Tabelas 4.11 a 4.13 são apresentados alguns dos casos de teste definidos para o teste de multiplicidades.

TABELA 4.11 – Casos de teste para multiplicidade da associação Aeronave/OS

Classe A: **Aeronave**		Data Atual: **29/03/2000**		
Classe B: **Ordem de Serviço**				
Multiplicidade Mínima: **1**		Multiplicidade Máxima: *****		
Parâmetro	Aeronave	OS	Resultado Esperado	ct
Valor máximo	**1**	**3**	**V**	***1***
Valor mínimo		**1**	**V**	***2***
Valor máximo + 1		**4**	**V**	***3***
Valor máximo – 1		**2**	**V**	***4***
Valor mínimo +1		**2**	**V**	***5***
Valor mínimo –1		**0**	**F**	***6***

ct = número de casos de teste

TABELA 4.12 – Casos de teste para multiplicidade da associação OS/Aeronave

Classe A: **Ordem de Serviço**		Data Atual: **29/03/2000**		
Classe B: **Aeronave**				
Multiplicidade Mínima: **1**		Multiplicidade Máxima: **1**		
Parâmetro	OS	Aeronave	Resultado Esperado	ct
Valor máximo	**1**	**1**	**V**	***1***
Valor mínimo		**1**	**V**	***2***
Valor máximo + 1		**2**	**F**	***3***
Valor máximo – 1		**0**	**F**	***4***
Valor mínimo +1		**2**	**F**	***5***
Valor mínimo –1		**0**	**F**	***6***

ct = número de casos de teste

TABELA 4.13 – Casos de teste para multiplicidade da associação Hélice/PásHélice

Classe A: **Hélice**		Data Atual: **29/03/2000**		
Classe B: **PásHélice**				
Multiplicidade Mínima: **2**		Multiplicidade Máxima: **4**		
Parâmetro	Hélice	PásHélice	Resultado Esperado	ct
Valor máximo	**1**	**4**	**V**	***1***
Valor mínimo		**2**	**V**	***2***
Valor máximo + 1		**5**	**F**	***3***
Valor máximo – 1		**3**	**V**	***4***
Valor mínimo +1		**3**	**V**	***5***
Valor mínimo –1		**1**	**F**	***6***

ct = número de casos de teste

Os casos de teste serão usados pelos métodos que efetivam a relação entre as classes.

Tendo sido definidas as informações para o teste dos atributos, métodos e relacionamentos das classes, deve-se verificar a interação entre os objetos para a realização dos casos de uso definidos para o módulo do sistema em estudo. A análise das interações serão apresentadas na próxima seção.

4.2 Analisando os Diagramas de Seqüência do Estudo de Caso

Nesta seção serão analisados os diagramas de interação do módulo para o controle da emissão de ordens de serviço do projeto CONTROLER. Os diagramas de interação, na maioria das vezes, são construídos para demonstrar a interação, pela troca de mensagens, entre objetos para a realização de um caso de uso. Neste módulo, foram definidos seis casos de uso e, para estes, elaborados os diagramas de seqüência e colaboração. A Figura 4.7 apresenta o diagrama de casos de uso desenvolvido para este módulo do projeto. Neste estudo de caso serão apresentados somente três diagramas de cada tipo (seqüência e colaboração).

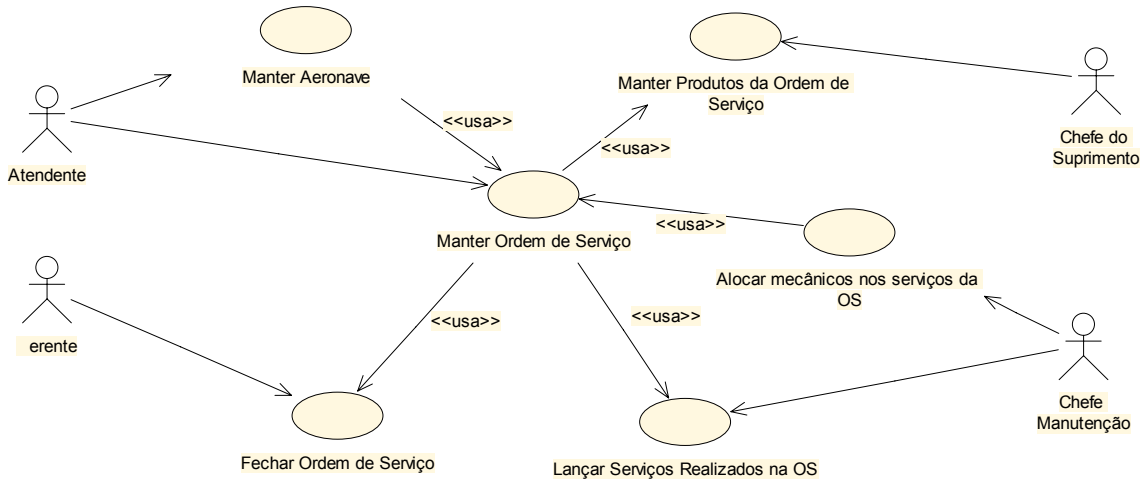


FIGURA 4.7 – Diagrama de casos de uso para o módulo de Controle de emissão de Ordens de Serviço.

A Figura 4.8 apresenta o diagrama de seqüência para o caso de uso "Manter Ordem de Serviço". Nesta interação, estão envolvidos os objetos *Ordem de Serviço*, *Aeronave*, *Proprietário*, *Serviço* e *ServiçoOS*. Observa-se que na especificação existem mensagens condicionais e iterativas, e estas devem ser verificadas. Na maioria das vezes, as condições já foram analisadas em restrições no diagrama de classes. Como é o caso das condições das mensagens 3, 4, 5 e 6, que controlam a situação da aeronave e do proprietário para que a Ordem de Serviço seja emitida. Estas condições fazem parte da restrição de associação entre Aeronave e Ordem de Serviço analisadas na seção anterior. Utilizando a técnica apresentada na Seção 3.2 para a analise do diagrama de seqüência, foram identificados os caminhos de teste para a análise dos cenários apresentados no diagrama de seqüência da Figura 4.8. A Tabela 4.14 apresenta estes caminhos de acordo com os critérios de cobertura definidos para este tipo de teste.

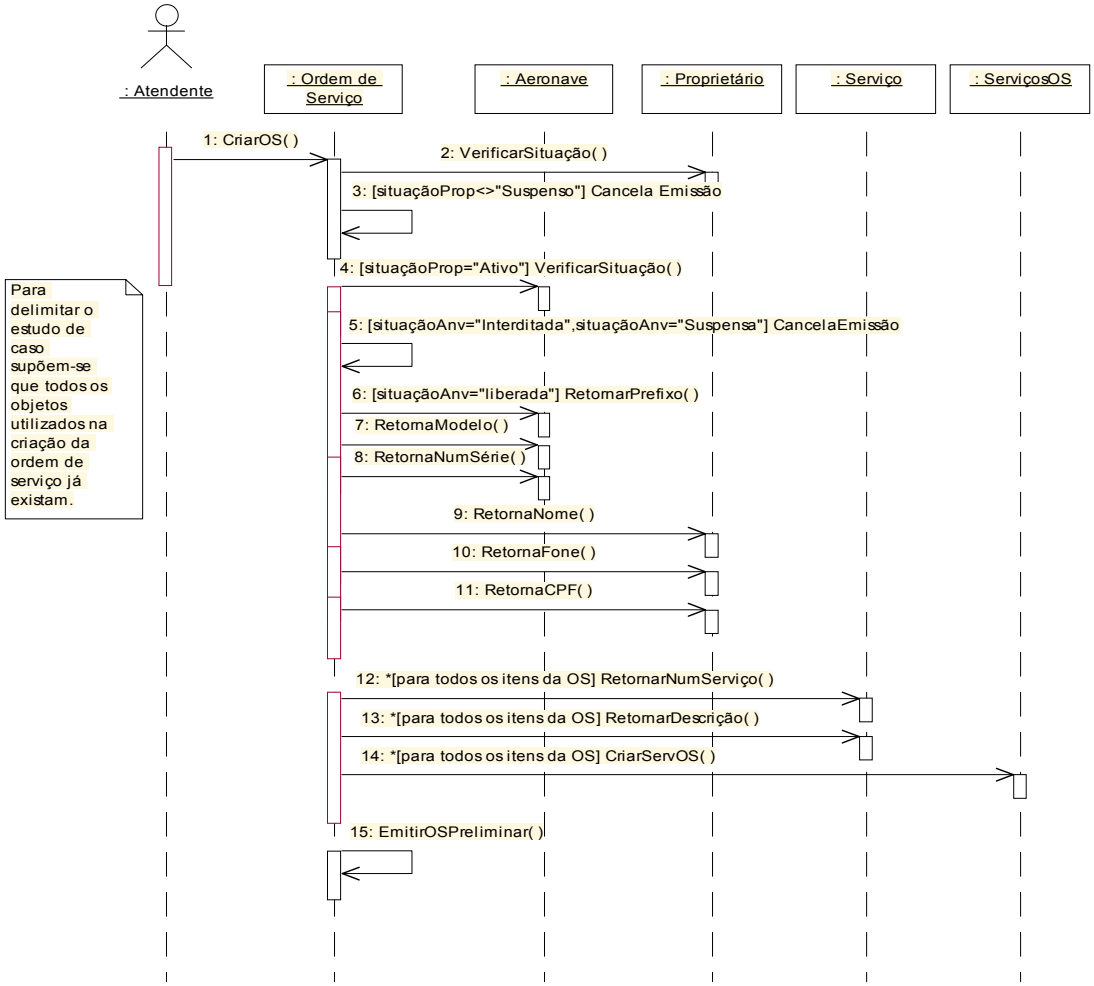


FIGURA 4.8 – Diagrama de seqüência para o caso de uso "*Manter Ordem de Serviço*".

TABELA 4.14 – Caminhos derivados da Figura 4.8 de acordo com os critérios de cobertura Todas-S-Mensagens, Todas-C-Mensagens e Todas-Iterações.

CRITÉRIOS DE COBERTURA	CAMINHOS
Todas-S-Mensagens	Caminho 1= ***M1, M2, M3*** Caminho 2= ***M1, M2, M4, M5*** Caminho 3= ***M1, M2, M4, M6, M7, M8, M9, M10, M11,M12, M13, M14, M12, M15***
Todas-C-Mensagens	Caminho 1= ***M1, M2, M3*** Caminho 2= ***M1, M2, M4, M5*** Caminho 3= ***M1, M2, M4, M6, M7, M8, M9, M10, M11,M12, M13, M14, M12, M15***
Todas-Iterações	Caminho 1= ***M1, M2, M4, M6, M7, M8, M9, M10, M11,M12, M15*** Caminho 2= ***M1, M2, M4, M6, M7, M8, M9, M10, M11,M12, M13, M14, M12, M15*** Caminho 3= ***M1, M2, M4, M6, M7, M8, M9, M10, M11,M12, M13, M14, M12, M13, M14,M12,....., M12, M15***

Como não existem condições especiais (*else* vazio) no diagrama de seqüência, os critérios **Todas-S-Mensagens** e **Todas-C-Mensagens** possuem os mesmos caminhos. Devido a existência de iterações consideradas importantes, para testar este cenário, devem ser exercitados os caminhos definidos pelo critério **Todas-Iterações**. Na Tabela 4.15 estão relacionados os casos de teste para os caminhos selecionados.

A associação entre as classes *Ordem de Serviço* e *ServiçosOS* possui a multiplicidade "1..*", ou seja, um objeto da *Ordem de Serviço* deve estar associado a pelo menos um objeto de *ServiçoOS*. Logo, a iteração deve ser assim testada:

- desviar da iteração – não entrar no laço para testar o mínimo – 1;
- exercitar a iteração uma vez – para testar o mínimo;
- exercitar a iteração várias vezes para testar o valor máximo, uma vez que este não foi definido.

TABELA 4.15 – Casos de teste para o caminhos dos Critérios Todas-Iterações da Tabela 4.14.

	Caminho 1	Caminho 2	Caminho 3
Mensagem	**Cria(1)**	**Cria(2)**	**Cria(3)**
Dados	(1, 29/03/00, 03/04/00)	(2, 29/03/00, 03/04/00)	(3, 29/03/00, 03/04/00)
Mensagem	**Proprietário.VerificaSituação()**	**Proprietário.VerificaSituação()**	**Proprietário.VerificaSituação()**
Dados	Situação="Ativo"	Situação="Ativo"	Situação="Ativo"
Mensagem	**Aeronave.VerificaSituação()**	**Aeronave.VerificaSituação()**	**Aeronave.VerificaSituação()**
Dados	Situação="Liberada"	Situação="Liberada"	Situação="Liberada"
Mensagem	**Aeronave.RetornaPrefixo()**	**Aeronave.RetornaPrefixo()**	**Aeronave.RetornaPrefixo()**
Dados	("PT-NAN")	("PT-NAN")	("PT-NAN")
Mensagem	**Aeronave.RetornaModelo()**	**Aeronave.RetornaModelo()**	**Aeronave.RetornaModelo()**
Dados	("EMB710C")	("EMB710C")	("EMB710C")
Mensagem	**Aeronave.RetornaNumSérie()**	**Aeronave.RetornaNumSérie()**	**Aeronave.RetornaNumSérie()**
Dados	(710006)	(710006)	(710006)
Mensagem	**Proprietário.RetornaNome()**	**Proprietário.RetornaNome()**	**Proprietário.RetornaNome()**
Dados	("João Vitor")	("João Vitor")	("João Vitor")
Mensagem	**Proprietário.RetornaFone()**	**Proprietário.RetornaFone()**	**Proprietário.RetornaFone()**
Dados	(677414004)	(677414004)	(677414004)
Mensagem	**Proprietário.RetornaCPF()**	**Proprietário.RetornaCPF()**	**Proprietário.RetornaCPF()**
Dados	(51962845168)	(51962845168)	(51962845168)
Mensagem	**Serviço.RetornaNUmServciço()**	**Serviço.RetornaNUmServciço()**	***Serviço.RetornaNUmServciço()**
Dados	(235)	(null)	(235)
Mensagem	**Serviço.RetornaDescrição()**	**EmitirOSPreliminar()**	***Serviço.RetornaDescrição()**
Dados	("Verificar Freio")		("Verificar Freio")
Mensagem	**ServiçoOS.CriarServOS()**		***ServiçoOS.CriarServOS()**
Dados	(29/03/00,15:00)		(29/03/00,15:00)
Mensagem	**Serviço.RetornaNUmServciço()**		**.....**
Dados	(null)		.....
Mensagem	**EmitirOSPreliminar()**		**Serviço.RetornaNUmServciço()**
Dados			(null)
Mensagem			**EmitirOSPreliminar()**
Dados			(5)

* Repetir várias vezes.

O diagrama de seqüência apresentado na Figura 4.9 para o caso de uso *Fechar Ordem de Serviço*, possui dois grupos de mensagens iterativas (2 – 5 e 6 – 8) e duas condições especiais nas mensagens 1 e 6.

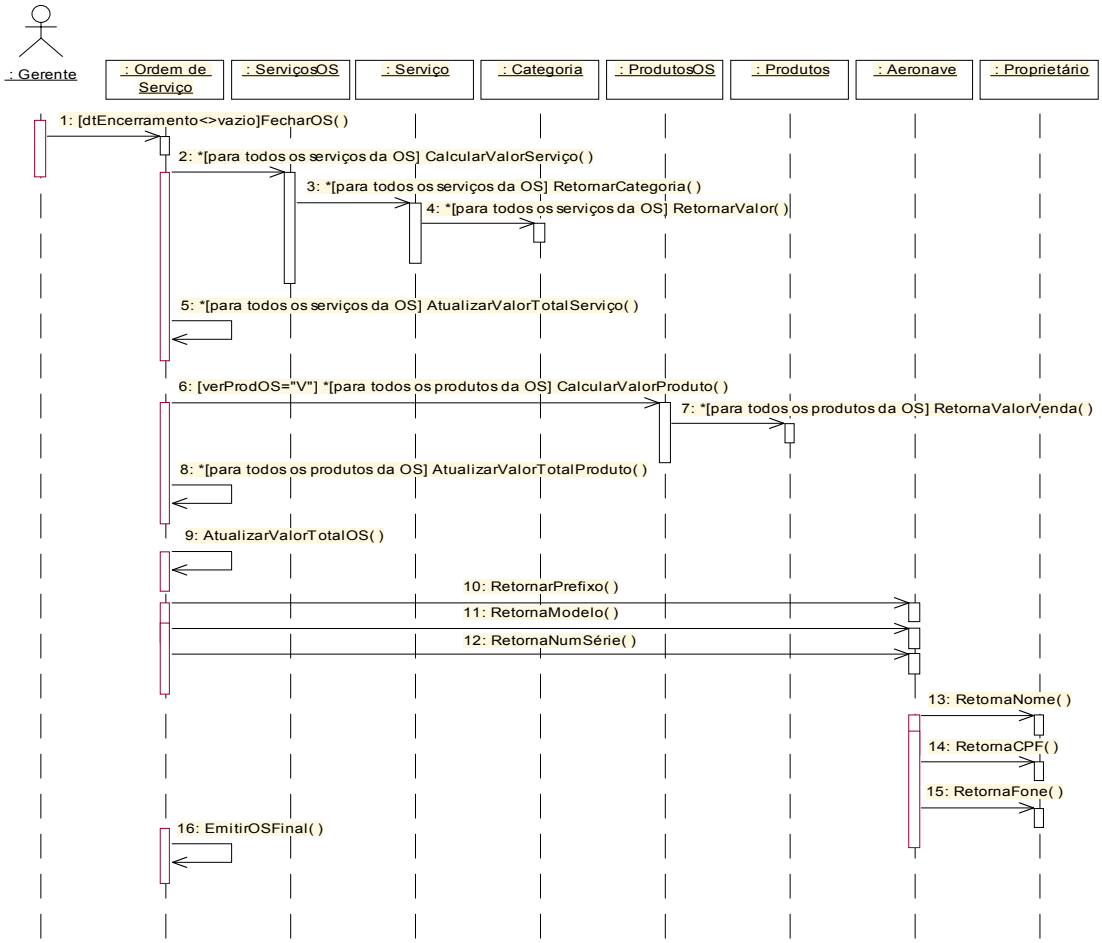


FIGURA 4.9 – Diagrama de seqüência para o caso de uso "*Fechar Ordem de Serviço*".

Nas Tabelas 4.16, 4.17 e 4.18, são apresentados os caminhos de teste, os caminhos selecionados e os casos de teste para este diagrama de seqüência.

TABELA 4.16 – Caminhos derivados do Diagrama de seqüência para o caso de uso "*Fechar Ordem de Serviço*".

CRITÉRIOS DE COBERTURA	CAMINHOS
Todas-S-Mensagens	Caminho 1= ***M1, M2, M3, M4, M5,M2, M6, M7, M8, M6, M9, M10, M11, M12, M13, M14, M15, M16*** Caminho 2= ***M1, M2, M3, M4, M5,M2, M6, M9, M10, M11, M12, M13, M14, M15, M16***
Todas-C-Mensagens	Caminho 1= ***M1*** Caminho 2= ***M1, M2, M3, M4, M5,M2, M6, M7, M8, M6, M9, M10, M11, M12, M13, M14, M15, M16*** Caminho 3= ***M1, M2, M3, M4, M5,M2, M6, M9, M10, M11, M12, M13, M14, M15, M16***

CRITÉRIOS DE COBERTURA	CAMINHOS
Todas-Iterações	Caminho 1= ***M1, M2, M6, M9, M10, M11, M12, M13, M14, M15, M16*** Caminho 2= ***M1, M2, M3, M4, M5,M2, M6, M7, M8, M6, M9, M10, M11, M12, M13, M14, M15, M16*** Caminho 3= ***M1, M2, M3, M4, M5,M2, ..., M2, M6, M7, M8, M6, ..., M6, M9, M10, M11, M12, M13, M14, M15, M16***

Conforme características do diagrama de seqüência apresentado, foram selecionados os caminhos de teste definidos pelos critérios de cobertura **Todas-C-Mensagens** e **Todas-Iterações**. Desconsiderou-se os caminhos que possuíam a mesma seqüência de mensagens.

TABELA 4.17 – Caminhos selecionados para o diagrama de seqüência da Figura 4.9.

Caminho	Mensagens
1	M1, M2, M3, M4, M5,M2, M6, M7, M8, M6, M9, M10, M11, M12, M13, M14, M15, M16
2	M1, M2, M3, M4, M5,M2, M6, M9, M10, M11, M12, M13, M14, M15, M16
3	M1
4	M1, M2, M6, M9, M10, M11, M12, M13, M14, M15, M16
5	M1, M2, M3, M4, M5,M2, ..., M2, M6, M7, M8, M6, ..., M6, M9, M10, M11, M12, M13, M14, M15, M16

OBS: Foram desconsiderados os caminhos repetidos.

TABELA 4.18 – Casos de teste para os caminhos selecionados da Tabela 4.17.

	Caminho 1	Caminho 2	Caminho 3	Caminho 4	Caminho 5
Mensagem	**FecharOS(1)**	**FecharOS(2)**	**FecharOS(3)**	**FecharOS(2)**	**FecharOS(1)**
Dados	(1, 29/03/00,29/03/00)	(2, 29/03/00,29/03/00)	(null)	(4, 29/03/00,29/03/00)	(5, 29/03/00,29/03/00)
Mensagem	**ServiçoOS.CalculaValorServiço()**	**ServiçoOS.CalculaValorServiço()**		**ServiçoOS.CalculaValorServiço()**	***ServiçoOS.CalculaValorServiço()**
Dados	(123)	(123)		(null)	(123)
Mensagem	**Serviço.RetornaCategoria()**	**Serviço.RetornaCategoria()**		**ProdutosOS.CalculaValorProduto()**	***Serviço.RetornaCategoria()**
Dados	(12)	(12)		(null)	(12)
Mensagem	**Categoria.RetornaValor()**	**Categoria.RetornaValor()**		**AtualizaValorTotalOS()**	***Categoria.RetornaValor()**
Dados	(12.10)	(12.10)		()	(12.10)
Mensagem	**AtualizaValorTotalServiço()**	**AtualizaValorTotalServiço()**		**Aeronave.RetornaPrefixo()**	***AtualizaValorTotalServiço()**
Dados	(12.10)	(12.10)		(“PT-NAN”)	(12.10)
Mensagem	**ServiçoOS.CalculaValorServiço()**	**ServiçoOS.CalculaValorServiço()**		**Aeronave.RetornaModelo()**	**ServiçoOS.CalculaValorServiço()**
Dados	(null)	(null)		(“EMB710C”)	(124)
Mensagem	**ProdutosOS.CalculaValorProduto()**	**ProdutosOS.CalculaValorProduto()**		**Aeronave.RetornaNumSérie()**	**....**
Dados	(“AN3-10A")	(null)		(710006)	....
Mensagem	**Produtos.RetornaValorProduto()**	**AtualizaValorTotalOS()**		**Proprietário.RetornaNome()**	***ProdutosOS.CalculaValorProduto()**
Dados	(1.20)	()		(“João Vitor”)	(“AN3-10A")
Mensagem	**AtualizaValorTotalProduto()**	**Aeronave.RetornaPrefixo()**		**Proprietário.RetornaCPF()**	***Produtos.RetornaValorProduto()**
Dados	(1.20)	(“PT-NAN”)		(51962845168)	(1.20)
Mensagem	**ProdutosOS.CalculaValorProduto()**	**Aeronave.RetornaModelo()**		**Proprietário.RetornaFone()**	***AtualizaValorTotalProduto()**
Dados	(null)	(“EMB710C”)		(677414004)	(1.20)
Mensagem	**AtualizaValorTotalOS()**	**Aeronave.RetornaNumSérie()**		**EmitirOSFinal()**	**ProdutosOS.CalculaValorProduto()**
Dados	()	(710006)		(4)	(“AN960-10”)
Mensagem	**Aeronave.RetornaPrefixo()**	**Proprietário.RetornaNome()**			**....**

	Caminho 1	Caminho 2	Caminho 3	Caminho 4	Caminho 5
Dados	(“PT-NAN”)	(“João Vitor”)			....
Mensagem	**Aeronave.RetornaModelo()**	**Proprietário.RetornaCPF()**			**AtualizaValorTotalOS()**
Dados	(“EMB710C”)	(51962845168)			()
Mensagem	**Aeronave.RetornaNumSérie()**	**Proprietário.RetornaFone()**			**Aeronave.RetornaPrefixo()**
Dados	(710006)	(677414004)			(“PT-NAN”)
Mensagem	**Proprietário.RetornaNome()**	**EmitirOSFinal()**			**Aeronave.RetornaModelo()**
Dados	(“João Vitor”)	(2)			(“EMB710C”)
Mensagem	**Proprietário.RetornaCPF()**				**Aeronave.RetornaNumSérie()**
Dados	(51962845168)				(710006)
Mensagem	**Proprietário.RetornaFone()**				**Proprietário.RetornaNome()**
Dados	(677414004)				(“João Vitor”)
Mensagem	**EmitirOSFinal()**				**Proprietário.RetornaCPF()**
Dados	(1)				(51962845168)
Mensagem					**Proprietário.RetornaFone()**
Dados					(677414004)
Mensagem					**EmitirOSFinal()**
Dados					(5)

* Repetir várias vezes.

Ao elaborar os casos de teste, observou-se que na maioria das vezes o critério **Todas-S-Mensagens** exercita a maioria das mensagens, com exceção das iterações especiais (evitar a iteração) cujo teste é previsto somente no critério **Todas-Iterações**.

As condições existentes nas mensagens do diagrama de seqüência analisados nesta seção, em sua maioria, são simples e utilizam o operador relacional de igualdade ou diferença, ou seja, não possuem uma combinação de condições. Pelas diretrizes da cobertura por condições, nestes casos, as condições devem ser testadas com um valor falso e outro verdadeiro. Este conceito foi aplicado no momento em que foram elaborados os casos de teste.

Na próxima seção serão analisados os diagramas de colaboração com a especificação da colaboração existente entre os objetos para a realização de um caso de uso.

4.3 Analisando os Diagramas de Colaboração do Estudo de Caso

Nesta seção são analisados os diagramas de colaboração desenvolvidos para a visualização da interação entre objetos visando a implementação de um caso de uso específico. Como os diagramas de colaboração são semanticamente equivalentes aos de seqüência, serão apresentadas, somente, as características que não foram analisadas na seção anterior. A Figura 4.10 apresenta o diagrama de colaboração para o caso de uso *Manter Aeronave*.

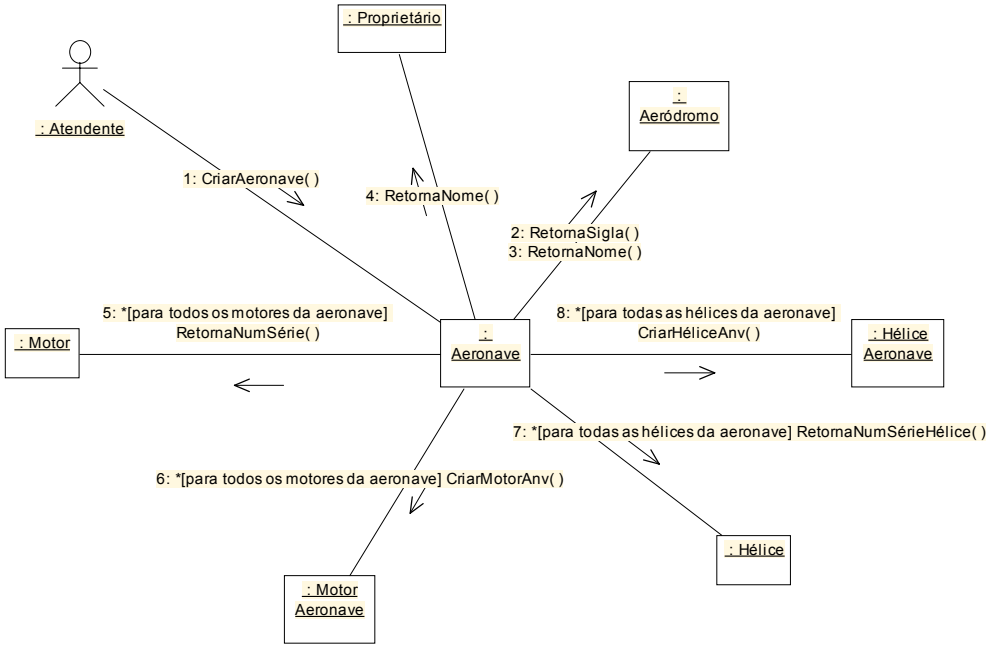


FIGURA 4.10 – Diagrama de colaboração para o caso de uso "*Manter Aeronave*".

Uma das técnicas disponíveis para derivar casos de teste de um diagrama de colaboração é converte-lo em um diagrama de seqüência. A técnica utilizada nesta seção avalia as interações diretamente no diagrama de colaborações. Na Tabela 4.19 são apresentadas as mensagens utilizadas em cada uma das interações, para as quais serão definidos os casos de teste.

TABELA 4.19 – Relação das mensagens existentes na colaboração apresentada na Figura 4.10.

Interação		**Mensagens**
Origem	**Destino**	
Atendente	Aeronave	1: CriarAeronave()
Aeronave	Aeródromo	2: RetornaSigla()
Aeronave	Aeródromo	3: RetornaNome()
Aeronave	Proprietário	4: RetornaNome()
Aeronave	Motor	5: *[para todos os motores da aeronave] RetornaNumSérie( )
Aeronave	MotorAeronave	6: *[para todos os motores da aeronave] CriarMotorAnv( )
Aeronave	Hélice	7: *[para todas as hélices da aeronave] RetornaNumSérieHélice( )
Aeronave	HéliceAeronave	8: *[para todas as hélices da aeronave] CriarHéliceAnv( )

Modelando as interações para a definição dos casos de teste, observou-se que primeiramente as mensagens devem ser colocadas em ordem. Desta maneira é possível exercitar as mensagens em ordem temporal, tal como ocorre no diagrama de seqüência. A estrutura condicional não fica muito clara pois não é possível verificar o encerramento da ativação[42] da operação. O que pode ser feito é considerar para o teste cada mensagem de acordo com sua condição. As mensagens iterativas também devem

[42] Determina o tempo no qual um objeto está executando diretamente uma operação [OMG 99].

ser consideradas, bastando observar a base da iteração. Caso esta base seja a mesma, isso indica que as mensagens fazem parte da mesma iteração. Como exemplo, as mensagens 5 e 6, possuem a mesma base *[para todos os motores da aeronave]*, portanto, fazem parte da mesma iteração. Sendo assim para realizar o teste basta exercitar as mensagens que pertençam ao mesmo grupo (iteração).

A relação das mensagens consideradas pelos critérios de cobertura definidos para este tipo de diagrama estão relacionadas na Tabela 4.20.

TABELA 4.20 – Relação das mensagens de acordo com os critérios de cobertura de teste definidos para o diagrama de colaboração da Figura 4.10.

Critério de Cobertura	Mensagens
Todas-Interações-totais	1: CriarAeronave() 2: RetornaSigla() 3: RetornaNome() 4: RetornaNome() 5: *[para todos os motores da aeronave] RetornaNumSérie( ) 6: *[para todos os motores da aeronave] CriarMotorAnv( ) 7: *[para todas as hélices da aeronave] RetornaNumSérieHélice( ) 8: *[para todas as hélices da aeronave] CriarHéliceAnv( )
Todas-Interações-parcias	1: CriarAeronave() 2: RetornaSigla() 4: RetornaNome() 5: *[para todos os motores da aeronave] RetornaNumSérie( ) 6: *[para todos os motores da aeronave] CriarMotorAnv( ) 7: *[para todas as hélices da aeronave] RetornaNumSérieHélice( ) 8: *[para todas as hélices da aeronave] CriarHéliceAnv( )

No diagrama de colaboração apresentado na Figura 4.10, a interação entre os objetos é feita, na maioria das vezes, pela troca de uma única mensagem, daí a pequena diferença existente entre os critérios. Os casos de teste para avaliar a colaboração existente no diagrama da Figura 4.10 estão relacionados na Tabela 4.21.

TABELA 4.21 – Casos de teste para as mensagens selecionados da Tabela 4.20.

	Todas-Interações-totais	Todas-Interações-Parciais
Mensagem	**Aeronave.CriarAeronave()**	**Aeronave.CriarAeronave()**
Dados	(“PT-NAN”,”Bras”,....)	(“PT-NAN”,”Bras”,....)
Mensagem	**Aeródromo.RetornaSigla()**	**Aeródromo.RetornaSigla()**
Dados	(SBCG)	(SBCG)
Mensagem	**Aeródromo.RetornaNome()**	**Motor.RetornaNumSérie( )**
Dados	(“Aeroporto Campo Grande”)	(“L72908-2L”)
Mensagem	***Motor.RetornaNumSérie( )**	**MotorAeronave.CriarMotorAnv( )**
Dados	(“L72908-2L”)	(29/03/00, “D”, “Original”)
Mensagem	***MotorAeronave.CriarMotorAnv( )**	**Motor.RetornaNumSérie( )**
Dados	(29/03/00, “D”, “Original”)	(“L72908-2L”)
Mensagem	***Motor.RetornaNumSérie( )**	**MotorAeronave.CriarMotorAnv( )**
Dados	(“L72908-2L”)	(29/03/00, “D”, “Original”)
Mensagem	***MotorAeronave.CriarMotorAnv( )**	**RetornaNumSérieHélice( )**
Dados	(29/03/00, “D”, “Original”)	(“PHC464872-R”)
Mensagem	**....**	**CriarHéliceAnv( )**
Dados	....	(29/03/00, “D”, “Original”)
Mensagem	***RetornaNumSérieHélice( )**	**RetornaNumSérieHélice( )**
Dados	(“PHC464872-R”)	(“PHC464872-R”)
Mensagem	***CriarHéliceAnv( )**	**CriarHéliceAnv( )**

	Todas-Interações-totais	Todas-Interações-Parciais
Dados	(29/03/00, “D”, “Original”)	(29/03/00, “D”, “Original”)
Mensagem	***RetornaNumSérieHélice( )**	
Dados	(“PHC464872-R”)	
Mensagem	***CriarHéliceAnv( )**	
Dados	(29/03/00, “D”, “Original”)	
Mensagem	**.....**	
Dados	.....	

* Repetir várias vezes.

Pela importância da colaboração apresentada, as iterações devem ser testadas de acordo com o critério de cobertura **Todas-Iterações** definido para o diagrama de seqüência. Neste critério, as iterações são testadas pelos limites estabelecidos na multiplicidade existente entre as classes envolvidas na interação.

No diagrama de colaboração da Figura 4.10, a aeronave possui pelo menos um motor e uma hélice, portanto, deve ser verificado o comportamento das mensagens para zero (para testar o mínimo), uma e várias iterações (para testar o máximo).

A Figura 4.11 apresenta o diagrama de colaboração para o caso de uso “*Manter Ordem de Serviço* ”.

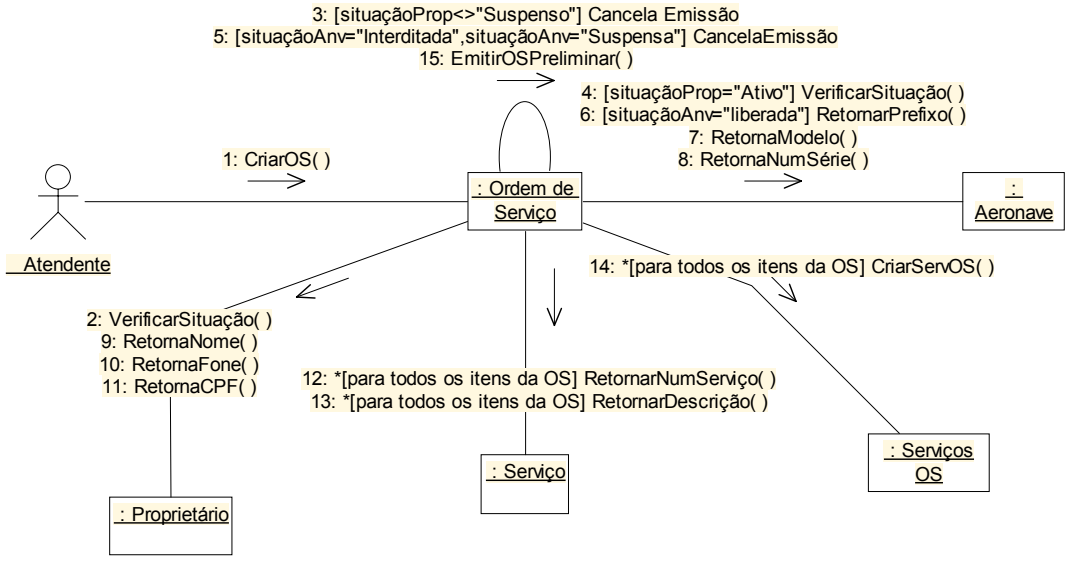


FIGURA 4.11 – Diagrama de colaboração para o caso de uso “*Manter Ordem de Serviço*”.

Nas Tabelas 4.22, 4.23 e 4.24, são apresentadas as mensagens especificadas para as interações do diagrama da Figura 4.11, a relação com as mensagens selecionadas de acordo com cada um dos critérios de cobertura e os casos de teste.

TABELA 4.22 – Relação das mensagens existentes na colaboração apresentada na Figura 4.11.

Interação		Mensagens
Origem	**Destino**	
Atendente	Ordem de Serviço	1: CriarOS()
Ordem de Serviço	Proprietário	2: VerificarSituação( )
Ordem de Serviço	Ordem de Serviço	3: [situaçãoProp<>"Suspenso"] Cancela Emissão
Ordem de Serviço	Aeronave	4: [situaçãoProp="Ativo"] VerificarSituação( )
Ordem de Serviço	Ordem de Serviço	5: [situaçãoAnv="Interditada",situaçãoAnv="Suspensa"] CancelaEmissão
Ordem de Serviço	Aeronave	6: [situaçãoAnv="liberada"] RetornarPrefixo( )
Ordem de Serviço	Aeronave	7: RetornaModelo( )
Ordem de Serviço	Aeronave	8: RetornaNumSérie( )
Ordem de Serviço	Proprietário	9: RetornaNome( )
Ordem de Serviço	Proprietário	10: RetornaFone( )
Ordem de Serviço	Proprietário	11: RetornaCPF( )
Ordem de Serviço	Serviço	12: *[para todos os itens da OS] RetornarNumServiço( )
Ordem de Serviço	Serviço	13: *[para todos os itens da OS] RetornarDescrição( )
Ordem de Serviço	ServiçoOS	14: *[para todos os itens da OS] RetornarDescrição( )
Ordem de Serviço	Ordem de Serviço	15: EmitirOSPreliminar( )

TABELA 4.23 – Relação das mensagens de acordo com os critérios de cobertura de teste definidos para o diagrama de colaboração da Figura 4.11.

Critério de Cobertura	Mensagens
Todas-Interações-totais	1: CriarOS() 2: VerificarSituação( ) 3: [situaçãoProp<>"Suspenso"] Cancela Emissão 4: [situaçãoProp="Ativo"] VerificarSituação( ) 5: [situaçãoAnv="Interditada",situaçãoAnv="Suspensa"] CancelaEmissão 6: [situaçãoAnv="liberada"] RetornarPrefixo( ) 7: RetornaModelo( ) 8: RetornaNumSérie( ) 9: RetornaNome( ) 10: RetornaFone( ) 11: RetornaCPF( ) 12: *[para todos os itens da OS] RetornarNumServiço( ) 13: *[para todos os itens da OS] RetornarDescrição( ) 14: *[para todos os itens da OS] CriarServOS( ) 15: EmitirOSPreliminar( )
Todas-Interações-parcias	1: CriarOS() 2: VerificarSituação( ) 3: [situaçãoProp<>"Suspenso"] Cancela Emissão 4: [situaçãoProp="Ativo"] VerificarSituação( ) 12: *[para todos os itens da OS] RetornarNumServiço( ) 13: *[para todos os itens da OS] RetornarDescrição( ) 14: *[para todos os itens da OS] CriarServOS()

Neste diagrama, é possível observar uma diferença considerável entre as mensagens selecionadas pelos dois critérios:

- Todas-Interações-totais – 15 mensagens
- Todas-Interações-parcias. – 7 mensagens

TABELA 4.24 – Casos de teste para as mensagens selecionados da Tabela 4.23.

	Todas-Interações-totais	Todas-Interações-Parciais
Mensagem	**CriarOS(**3**)**	**CriarOS(**3**)**
Dados Mensagem	(1, 29/03/00, 03/04/00) **Proprietário.VerificaSituação()**	(1, 29/03/00, 03/04/00) **Proprietário.VerificaSituação()**

	Todas-Interações-totais	Todas-Interações-Parciais
Dados	Situação=”Ativo”	Situação=”Ativo”
Mensagem	**Cancela Emissão()**	**Cancela Emissão()**
Dados	(1)	(1)
Mensagem	**Aeronave.VerificaSituação()**	**Aeronave.VerificaSituação()**
Dados	Situação=”Liberada”	Situação=”Liberada”
Mensagem	**CancelaEmissão()**	**Serviço.RetornaNUmServciço()**
Dados	(1)	(235)
Mensagem	**Aeronave.RetornaPrefixo()**	**Serviço.RetornaDescrição()**
Dados	(“PT-NAN”)	(“Verificar Freio”)
Mensagem	**Aeronave.RetornaModelo()**	**ServiçoOS.CriarServOS()**
Dados	(“EMB710C”)	(29/03/00,15:00)
Mensagem	**Aeronave.RetornaNumSérie()**	**Serviço.RetornaNUmServciço()**
Dados	(710006)	(235)
Mensagem	**Proprietário.RetornaNome()**	**Serviço.RetornaDescrição()**
Dados	(“João Vitor”)	(“Verificar Freio”)
Mensagem	**Proprietário.RetornaFone()**	**ServiçoOS.CriarServOS()**
Dados	(677414004)	(29/03/00,15:00)
Mensagem	**Proprietário.RetornaCPF()**	
Dados	(51962845168)	
Mensagem	***Serviço.RetornaNUmServciço()**	
Dados	(235)	
Mensagem	***Serviço.RetornaDescrição()**	
Dados	(“Verificar Freio”)	
Mensagem	***ServiçoOS.CriarServOS()**	
Dados	(29/03/00,15:00)	
Mensagem	**....**	
Dados	....	
Mensagem	**EmitirOSPreliminar()**	
Dados	(3)	

* Repetir várias vezes.

Os últimos diagramas analisados no estudo de caso foram os baseados nos estados da classe. Na próxima seção, serão apresentados os resultados da aplicação das diretrizes e critérios de cobertura de teste para os diagramas de estado.

4.3.5 Analisando os Diagramas de Estados do Estudo de Caso

Nesta seção, são analisados os diagramas de estado elaborados para as classes do módulo de controle de emissão de ordens de serviço. As classes envolvidas neste módulo, em sua maioria, não possuem características que justifiquem a criação de diagramas de estado, com exceção das classes *Aeronave* e *Proprietário*. Na Figura 4.12 é apresentado o diagrama de estados da classe *Proprietário*.

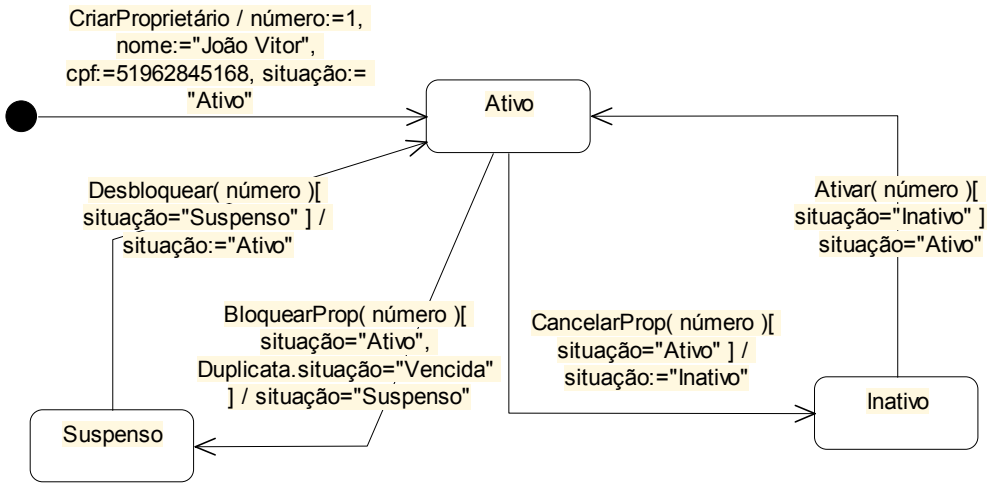


FIGURA 4.12 – Diagrama de estados da classe *Proprietário*.

Tendo como base o diagrama de estados da Figura 4.12, foi gerada a árvore de transições apresentada na Figura 4.13.

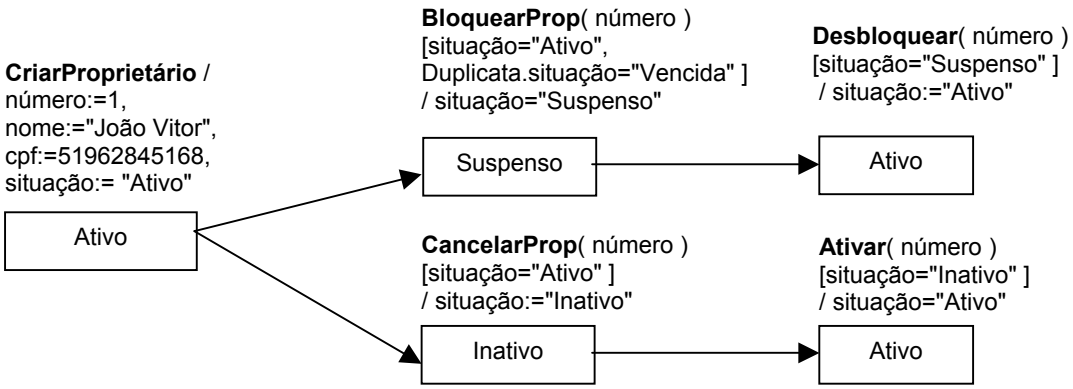


FIGURA 4.13 – Árvore de transições do diagrama de estados da classe *Proprietário*.

A estrutura da árvore de transições disponibiliza um conjunto de caminhos para que sejam aplicados os casos de teste no diagrama de estados. Na Tabela 4.24 são apresentados os casos de teste para os caminhos derivados da árvore de transições da Figura 4.13.

TABELA 4.25 – Casos de teste para os caminhos derivados da árvore de transições da Figura 4.13.

	1	2
Operação	CriarProprietário(1)	CriarProprietário(2)
Dados	(“1”,”João Vitor”,51962845168,”Ativo”)	(“2”,”João Vitor”,51962845168,”Ativo”)
Operação	BloquearProp(1)	CancelarProp(1)
Dados	("Ativo","Vencida") ("Suspenso")	("Ativo") ("Inativo")
Operação	Desbloquear(1)	Ativar(1)
Dados	("Suspenso") ("Ativo")	("Inativo") ("Ativo")

O diagrama de estados da Figura 4.14 apresenta os possíveis estados da classe *Ordem de Serviço*. A árvore de transições deste diagrama é apresentado na Figura 4.15.

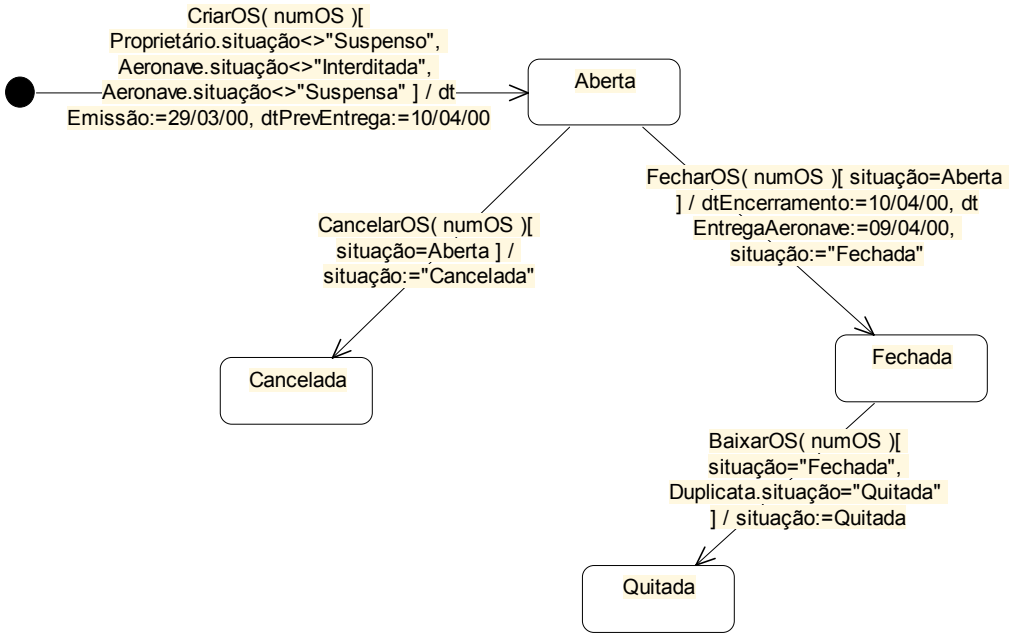


FIGURA 4.14 – Diagrama de estados da classe *Ordem de Serviço*.

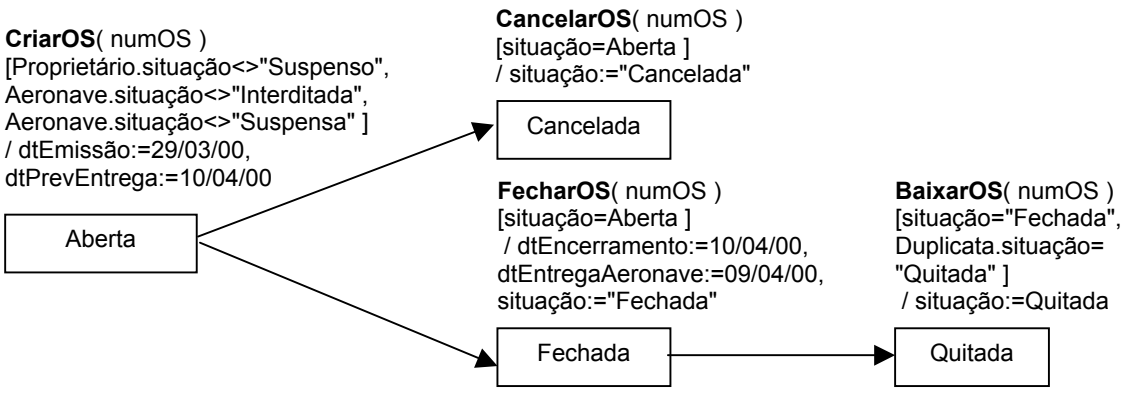


FIGURA 4.15 – Árvore de transições do Diagrama de estados da classe *Ordem de Serviço*.

A Tabela 4.26 apresenta os casos de teste para os caminhos derivados da árvore de transições do diagrama de estados de estados da Figura 4.14.

TABELA 4.26 – Casos de teste para o caminhos derivados da árvore de transições da Figura 4.15.

	1	2
Operação	CriarOS(1)	CriarOS(2)
Dados	("Ativo",”Liberada") (29/03/00, 10/04/00)	("Ativo",”Liberada") (29/03/00, 10/04/00)
Operação	CancelarOS(1)	FecharOS(2
Dados	("Ativo","Vencida") ("Suspenso")	(“Aberta”) (10/04/00, 09/04/00,"Fechada")
Operação		BaixarOS(2)
Dados		("Fechada", "Quitada") (“Quitada”)

Durante a aplicação dos critérios de teste baseados no diagrama de estados observou-se que a seleção dos dados de teste é melhor evidenciada em função da complexidade das transições representadas no diagrama.

No próximo capítulo serão apresentadas as conclusões sobre o trabalho desenvolvido, considerando:

- o Teste de Software Orientado a Objetos (suas características e particularidades);
- os Critérios de Cobertura de Teste disponíveis na Literatura e utilizados no desenvolvimento deste trabalho;
- os Critérios de Cobertura e Diretrizes de Teste definidos ou adaptados neste trabalho; e
- os resultados obtidos com a aplicação dos Critérios e Diretrizes no Estudo de Caso.

5 Conclusão

A atividade de teste é utilizada no processo de desenvolvimento de *software* para garantir a qualidade dos produtos gerados. O teste consiste em verificar dinamicamente um *software,* executando-o e observando o seu comportamento em relação aos requisitos definidos na especificação do sistema. Para orientar o teste e estabelecer o momento em que este foi suficientemente testado, faz-se uso dos critérios de cobertura. Estes critérios determinam o que testar e conseqüentemente, o nível de confiabilidade de um sistema pode variar em função da abrangência do critério de cobertura de teste utilizado.

Já na fase de especificação podem ser definidos os elementos a serem testados quando o software for implementado. A documentação de um sistema, na maioria das vezes, é realizada por diagramas, cujo principal objetivo é disponibilizar ao desenvolvedor uma visão da aplicação antes que a mesma seja implementada, permitindo que sejam feitos os ajustes e definidas as informações para sua implementação. Neste momento, a maioria das informações necessárias para o teste já estão disponíveis e estas devem ser utilizadas na definição do que testar.

Neste sentido, este trabalho teve como principal objetivo a definição de um conjunto de diretrizes e de critérios de cobertura de teste derivados de diagramas implementados pela UML. Para isso, foram buscados os seguintes objetivos específicos:

- **Estudar as técnicas de teste orientado a objetos** – Foram pesquisadas as técnicas de teste disponíveis na literatura que atendessem as características do trabalho proposto. Verificou-se que a maioria das técnicas de teste, faziam referência a critérios de cobertura procedimentais. Neste sentido, estudou-se as técnicas e critérios procedimentais para avaliar como estes poderiam ser utilizados na construção dos critérios baseados em especificações.
- **Estudar os Critérios de Cobertura existentes** – Como a proposta deste trabalho era a definição de critérios de cobertura baseado em especificação, fez-se uma pesquisa para identificar possíveis critérios de cobertura com estas característica ou que pudessem ser adaptados a esta proposta. A maioria destes dizia o que testar mas não poderiam ser diretamente aplicados, a menos que fossem adaptados ao contexto do trabalho.
- **Estudar a linguagem de modelagem UML** – A UML tem sido muito utilizada e estabelece padrões para a modelagem de especificações orientadas a objeto. Por isso, optou-se por utilizá-la como base para a definição das diretrizes e critérios de cobertura. Foram analisados os seus diagramas para identificar aqueles que poderiam ser utilizados no desenvolvimento deste trabalho. Selecionou-se quatro diagramas: classes, seqüência, colaboração e estados. Sobre estes fez um estudo aprofundado determinando os aspectos a serem considerados no teste.
- **Definir os Critérios de Cobertura para o teste** Tendo os critérios de cobertura existentes, as técnicas de teste definidas para software OO e os aspectos relevantes dos diagramas, desenvolveu-se um conjunto de diretrizes com objetivo de estabelecer regras para o teste com base nas informações dos diagramas e para apoiar na geração dos caminhos e casos de teste, ou seja, o que testar. Foram elaborados critérios de cobertura de teste para cada um dos diagramas analisados.

- **Elaborar uma proposta de implementação da seleção de caminhos e casos de teste com base nos diagramas UML**. Durante o levantamento de informações para a definição dos critérios, observou-se que era possível elaborar uma ferramenta que fizesse a geração automática dos caminhos de teste e com base nos critérios, definiu-se os casos de teste. Corroborando com isso, desenvolveu-se uma proposta de implementação da seleção de caminhos e casos de teste com base em diagramas UML.
- **Aplicar Critérios de Cobertura em estudo de caso.** Finalmente, para demonstrar a funcionalidade das diretrizes e critérios definidos, estes foram aplicados à modelagem de um módulo de sistema para a emissão de ordens de serviço em aeronaves, permitindo com isso, a geração dos dados necessários para teste da aplicação quando esta for implementada.
  - Na análise de condições e combinações de condições – os critérios definidos atenderam ao propósito de selecionar os casos de teste.
  - Na análise do relacionamento do tipo generalização – houve redução substancial dos casos de teste em função dos critérios de cobertura definidos e utilizados.
  - Na análise das multiplicidades – (a) observou-se que a quantidade de relacionamentos pode influenciar na derivação de casos de teste, influenciando a sua viabilidade; (b) a aplicação do critério de cobertura *Todas-Multiplicidades-Obrigatórias* reduziu substancialmente o número de associações a serem submetidas a teste. É de suma importância a automatização deste tipo de teste.
  - No diagrama de seqüência – (a) a aplicação das diretrizes definidas pôde auxiliar na verificação da própria modelagem, permitindo a detecção de erros de especificação; (b) observou-se a diferença dos caminhos gerados em função dos critérios definidos e utilizados.
  - No diagrama de colaboração – (a) foi possível observar a redução dos elementos a serem analisados pela seleção dos critérios em função da complexidade do cenário modelado nos diagramas; (b) os critérios definidos para o diagrama de seqüência podem ser aplicados, não sendo necessário a conversão dos diagramas.
  - No diagrama de estados – observou-se que a diferença entre os critérios não pôde ser observada em diagramas com poucas transições e estados, ou seja, a eficácia dos critérios de cobertura definidos para este tipo de diagrama está associada a complexidade da especificação.

A eficiência dos critérios desenvolvidos neste trabalho pode variar em função da qualidade da especificação da aplicação. Observou-se durante a aplicação dos critérios que quanto mais detalhes possuir um diagrama, melhor será a qualidade dos casos de teste gerados. Como por exemplo, as pré condições de um método são muito importantes pois determinam o momento de sua realização. A ausência da condição de guarda em uma transição pode levar um objeto para um estado indesejado, pela execução de um evento indevido. O mesmo ocorre em um diagrama de seqüência. Quando a condição da mensagem não é especificada, a mensagem pode ser enviada e com isso, uma operação inválida será executada.

5.1 Trabalhos Futuros

Extensões a este trabalho envolvem: a validação dos critérios de cobertura propostos, a extensão dos critérios propostos e o desenvolvimento de ferramentas voltadas ao teste de aplicações orientadas a objetos. Neste sentido, sugere-se como extensão a este trabalho :

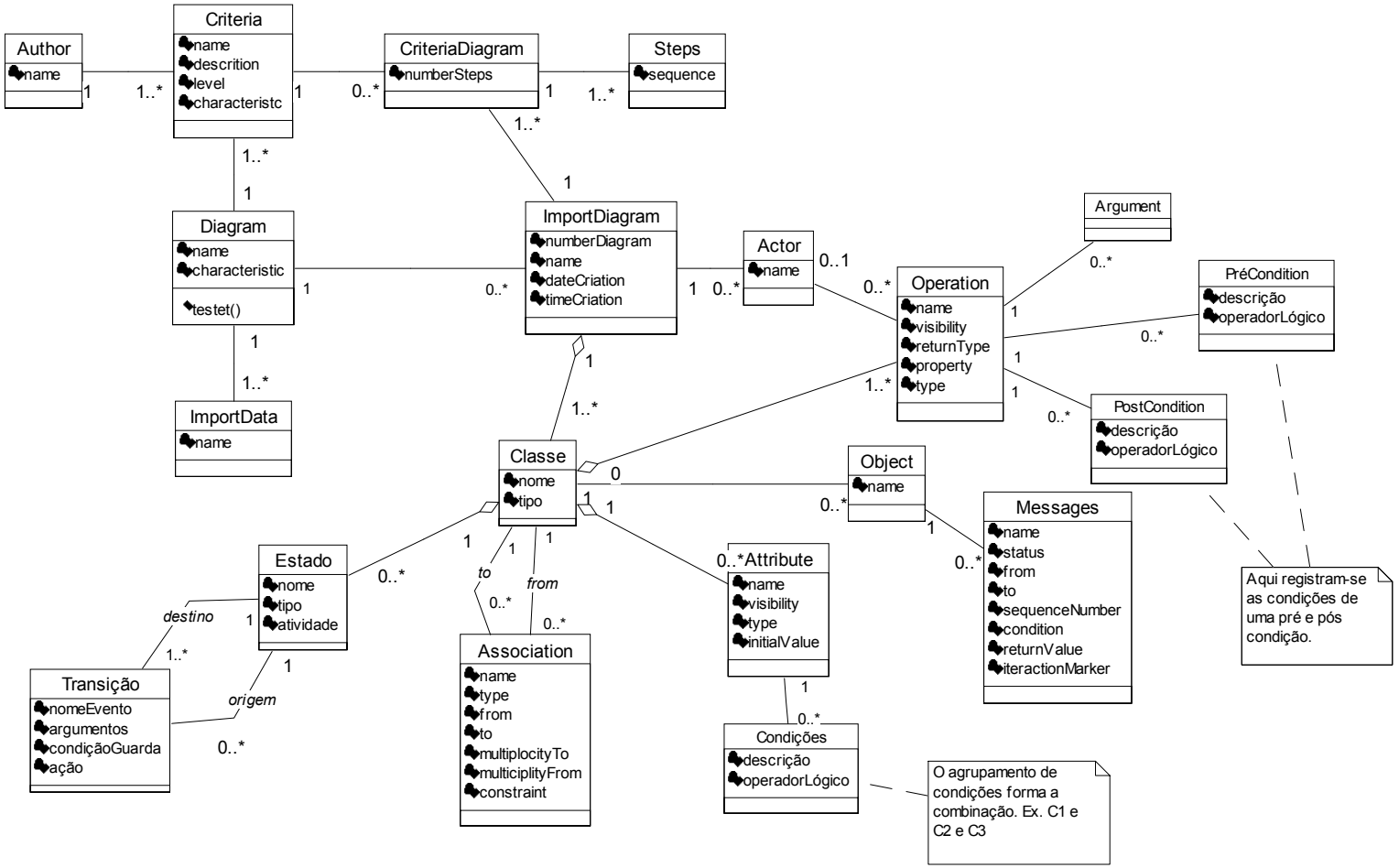
- **Definição de novos critérios ou extensão dos definidos neste trabalho para que permitam a derivação de informações para o teste com base nos outros diagramas da UML**. O diagrama de casos de uso é uma excelente ferramenta para o teste de aceitação e de sistema pois serve para comparar os requisitos do sistema com o produto final, uma vez que se tem a visão dos agentes externos. Com base nos diagramas de componentes podem ser definidos critérios para testar a usabilidade do sistema, verificando se o mesmo utiliza todos os recursos dos quais dispõe e se a sua interface foi adequadamente implementada.

- **Extensão dos critérios de cobertura definidos neste trabalho considerando outras particularidades dos diagramas.** No diagrama de estados, pode-se considerar os aspectos avançados das transições e dos estados (ações de entrada e saída, transições internas, eventos adiados, subestados, subestados seqüenciais, estados de histórico e subestados concorrentes). Nos diagramas de seqüência e colaboração, pode-se considerar o processamento concorrente de classes ativas durante a transferência de mensagens. No diagrama de classes, pode-se avaliar os qualificadores, as dependências, a herança múltipla, a navegabilidade e modelagem de interfaces. Pode-se também, considerar as particularidades dos sistemas em tempo real como o tratamento de eventos, simultaneidade e sincronização.

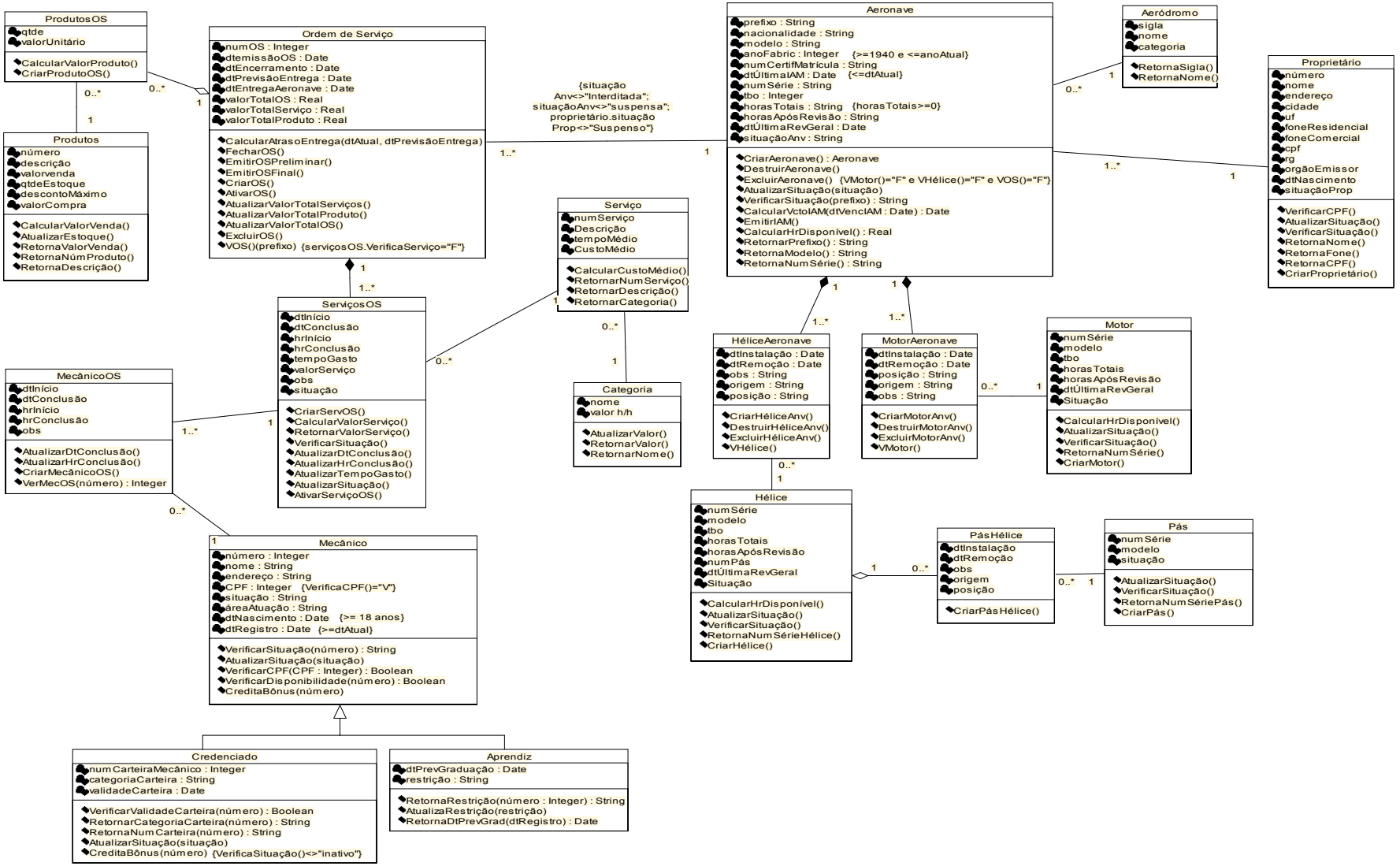
- **Desenvolvimento de uma ferramenta para automatizar a geração de caminhos de teste.** Utilizando a proposta de implementação sugerida neste trabalho, é possível implementar uma ferramenta que (a) gere automaticamente os caminhos de teste; (b) armazene as informações sobre os caminhos gerados em um banco de dados para que possam ser definidos os casos de teste; (c) disponibilize recursos para a importação de informações sobre os diagramas modelados em outras ferramentas; (d) emita relatórios com as fichas de avaliação dos atributos, métodos, condições (simples e múltiplas) e multiplicidades, bem como outros que sejam pertinentes a atividade de testes.

- **Avaliação dos critérios propostos neste trabalho**. Fazer a análise comparativa entre os critérios de cobertura definidos neste trabalho e outros existentes na literatura. Uma das formas de avaliar e comparar os critérios está relacionada à definição de um conjunto de métricas de qualidade e eficiência que considere as particularidades do Teste de *Software* Orientado a Objetos.

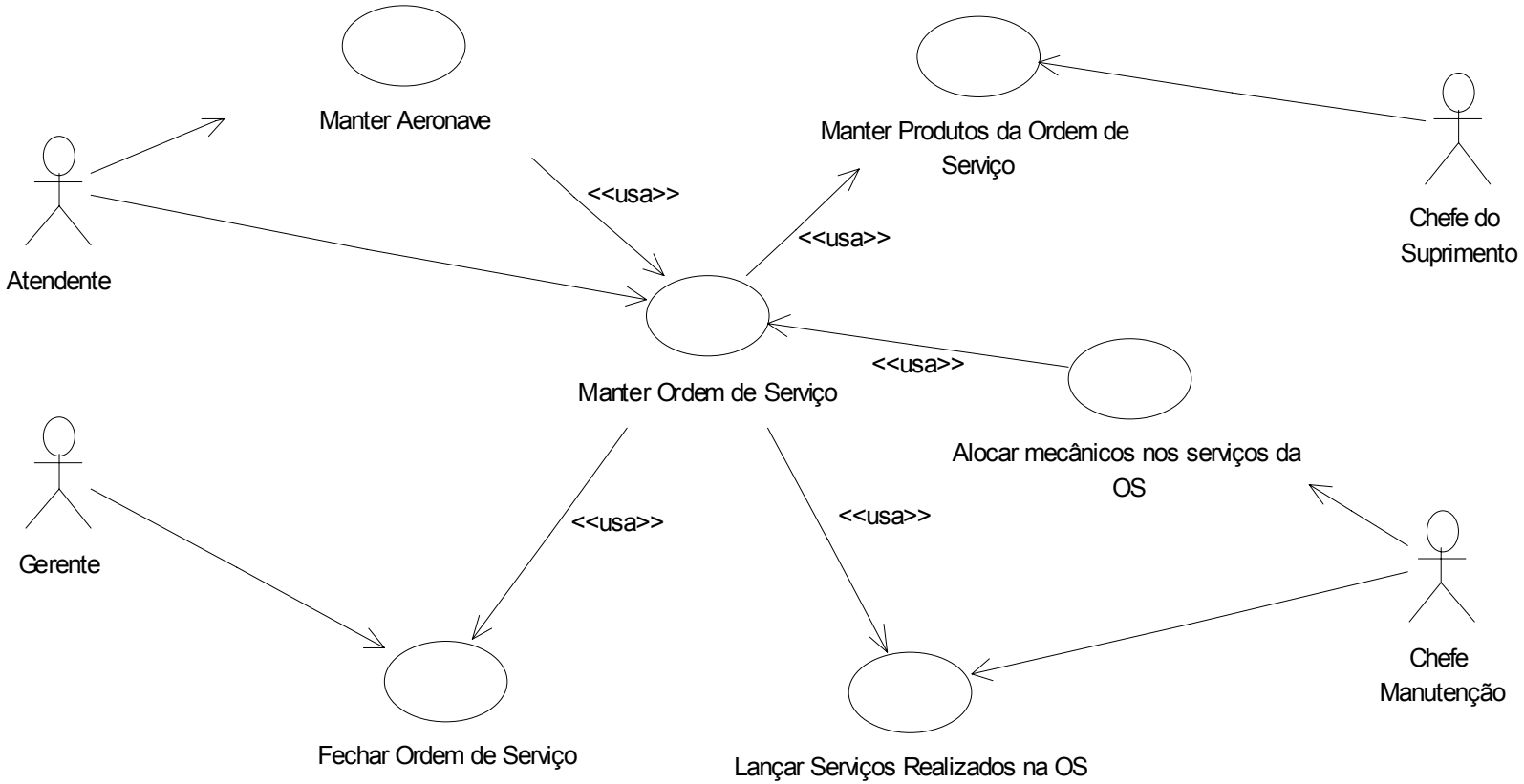
Anexo 1 Proposta de Modelo Conceitual para uma Ferramenta de Geração de Caminhos de teste



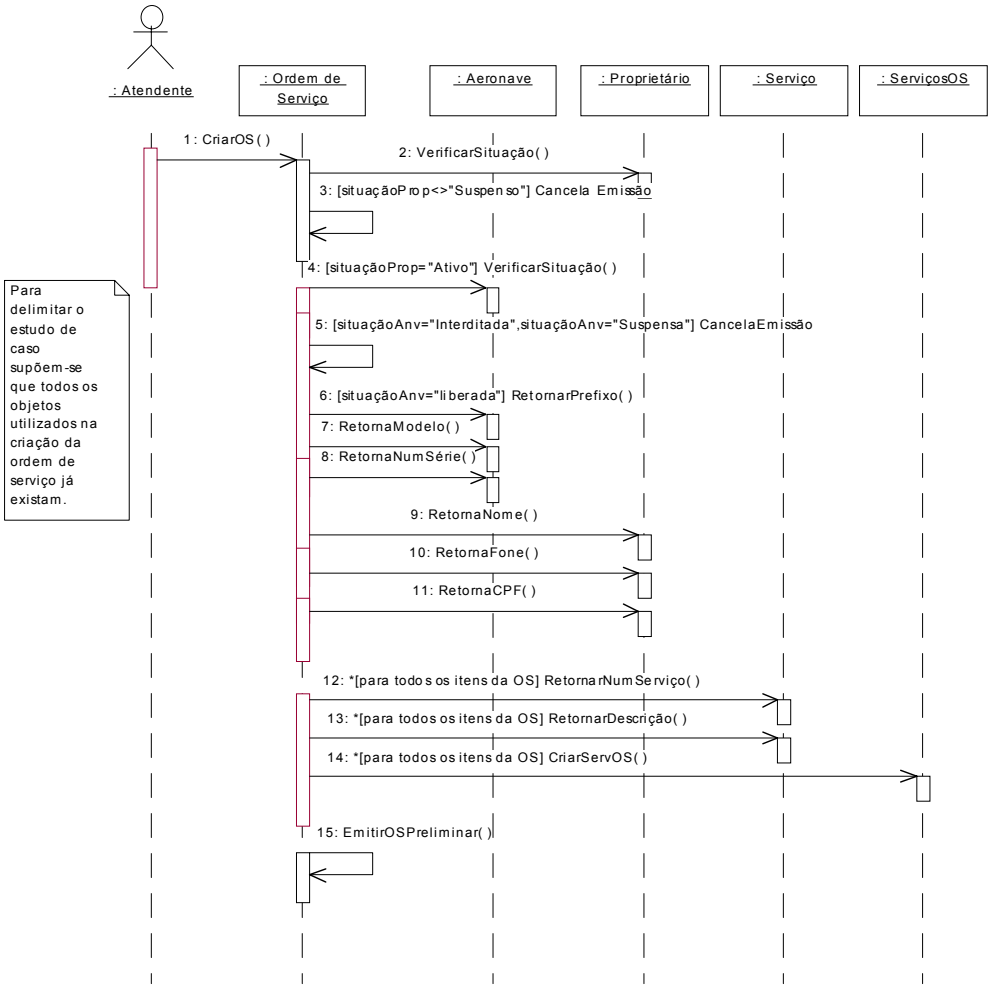
Anexo 2 Diagrama de Classes Módulo Emissão de Ordens de Serviço



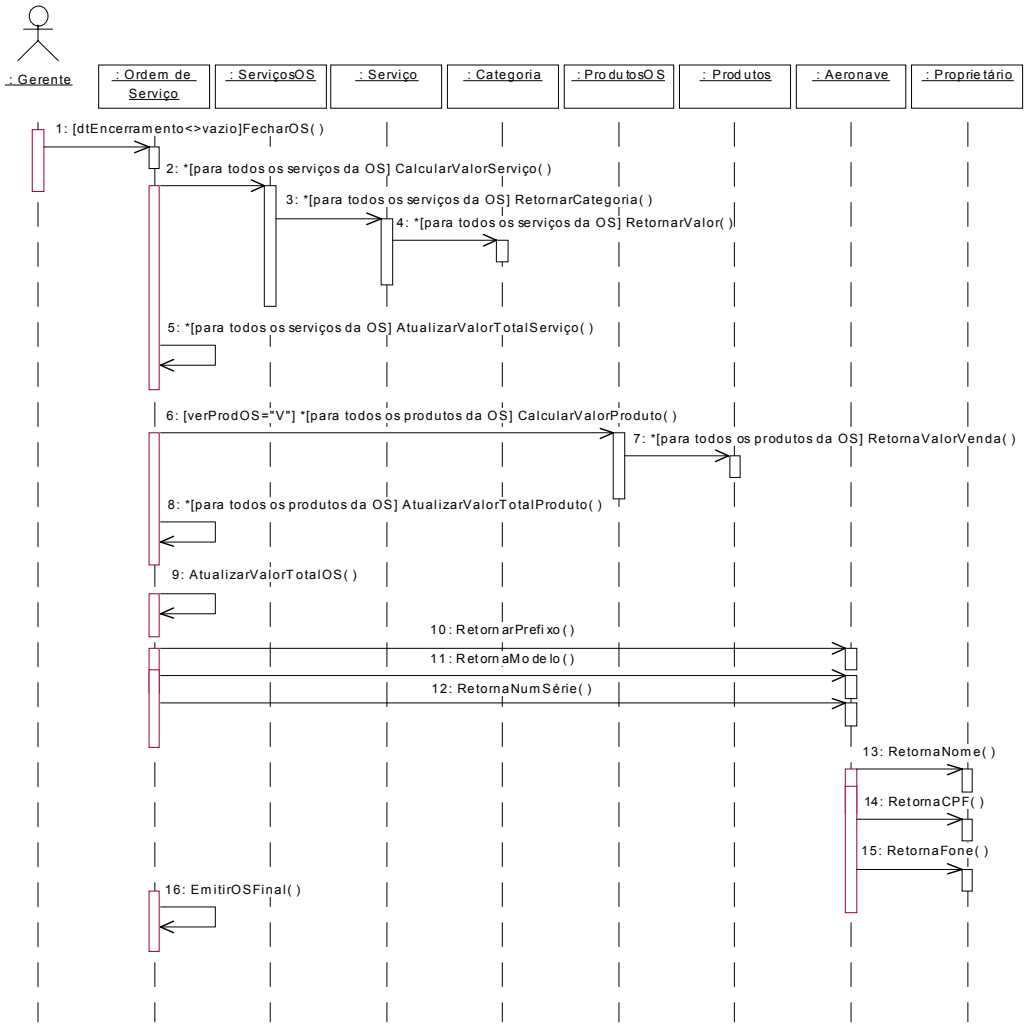
Anexo 3 Diagrama de Casos de Uso do Módulo Emissão de Ordens de Serviço



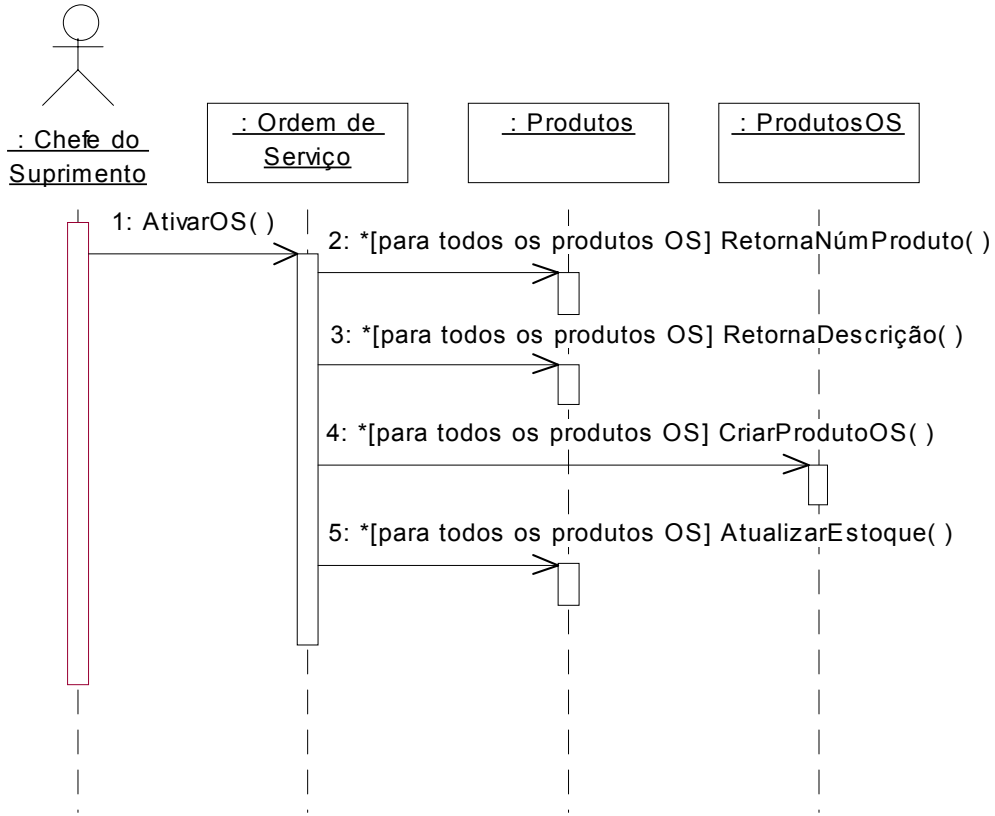
Anexo 4 Diagrama de Seqüência Manter Ordem de Serviço do Estudo de Caso



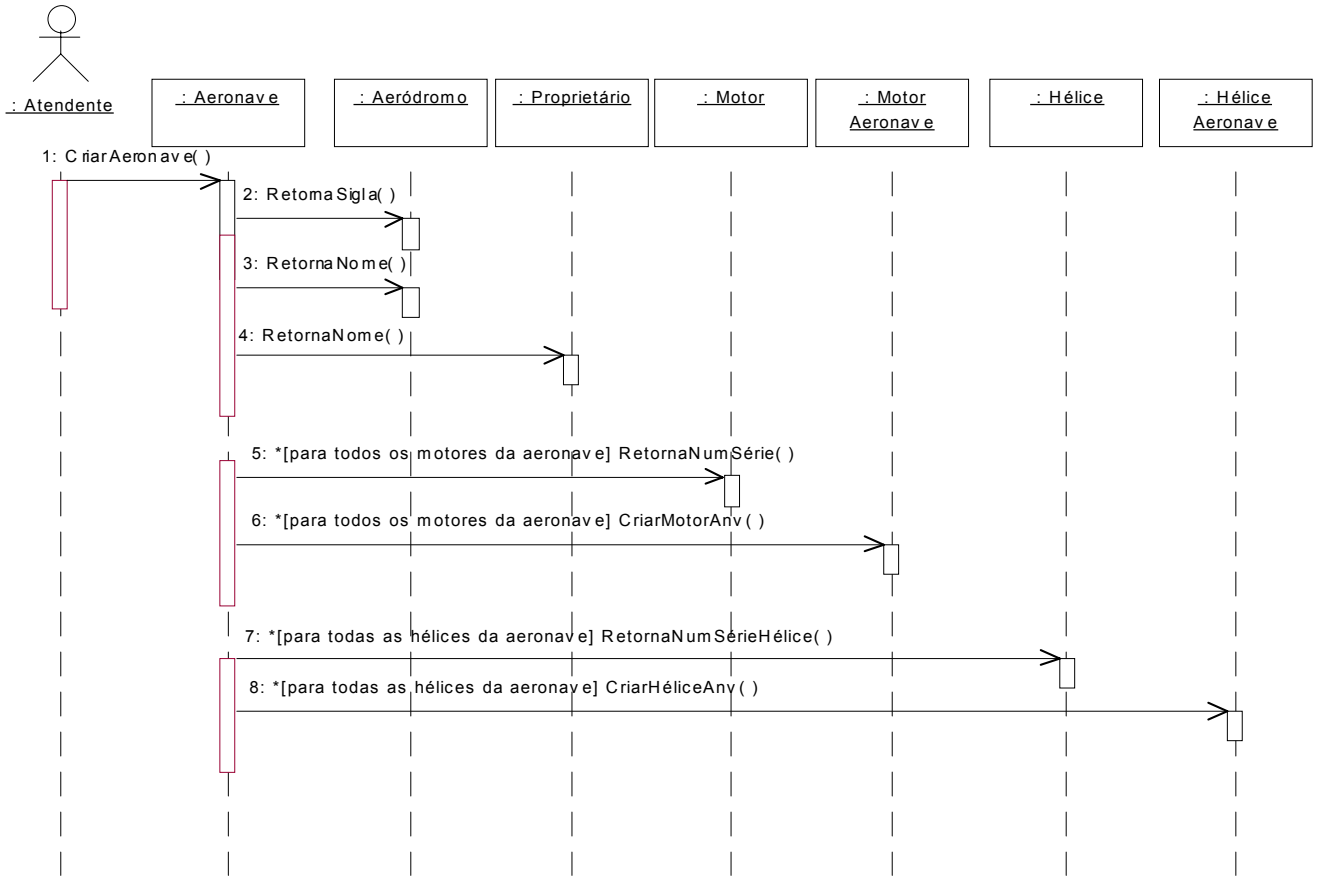
Anexo 5 Diagrama de Seqüência Fechar Ordem de Serviço do Estudo de Caso



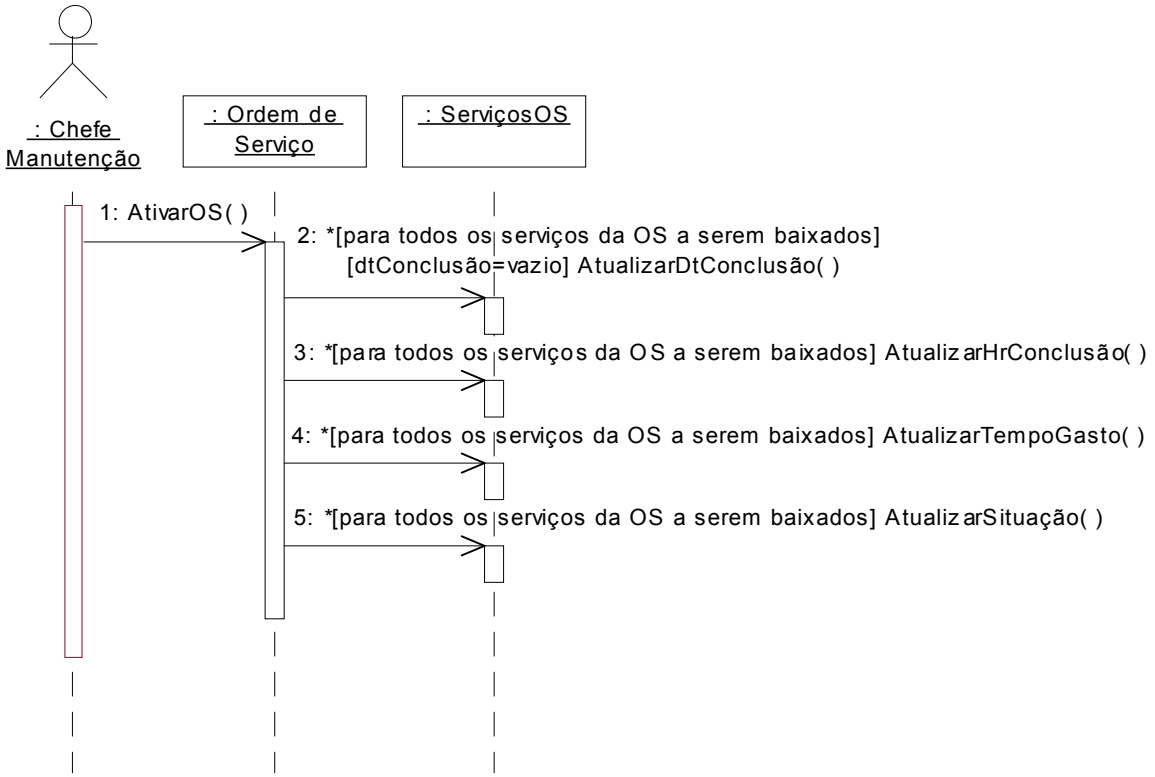
Anexo 6 Diagrama de Seqüência Manter Produtos da OS do Estudo de Caso



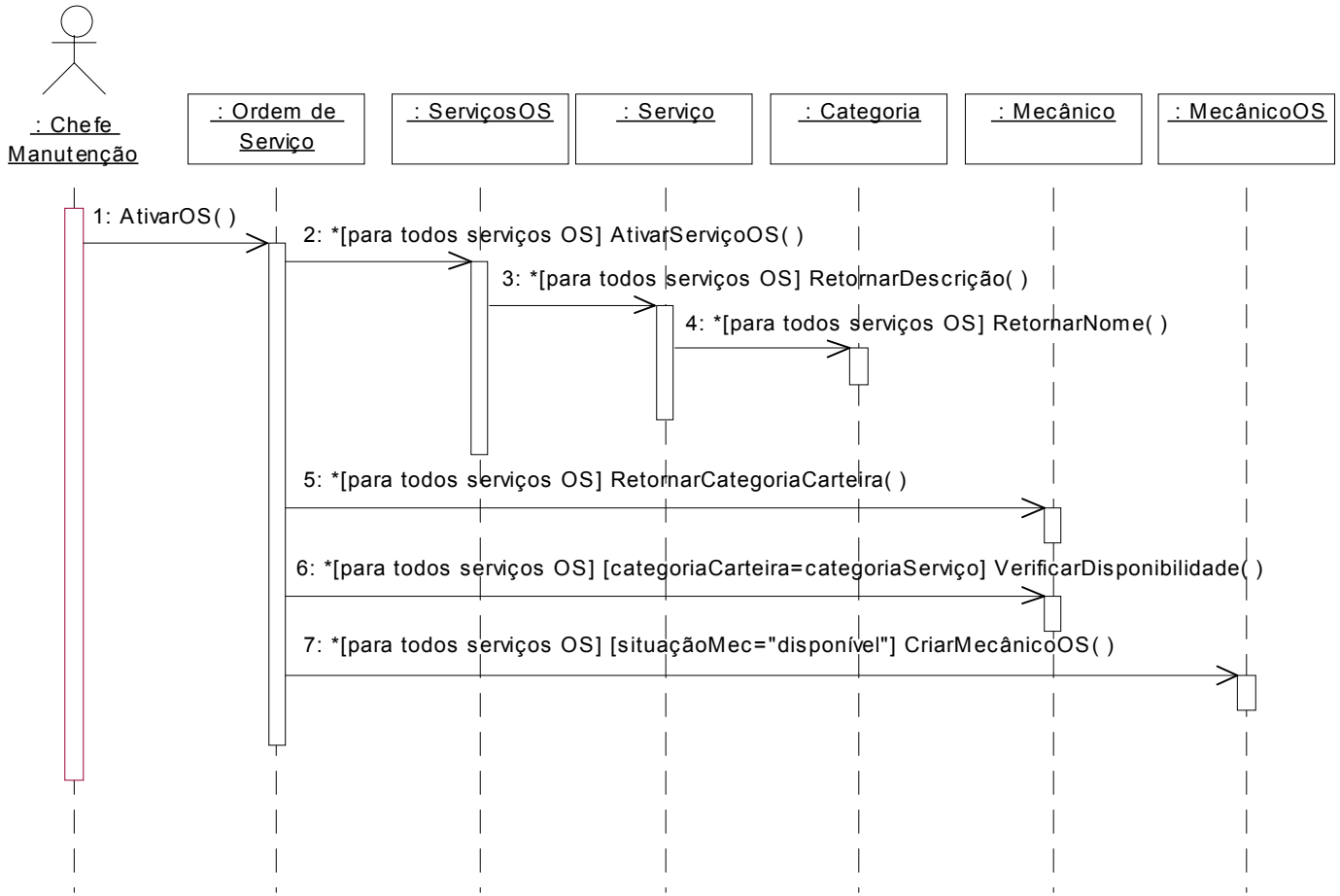
Anexo 7 Diagrama de Seqüência Manter Aeronave do Estudo de Caso



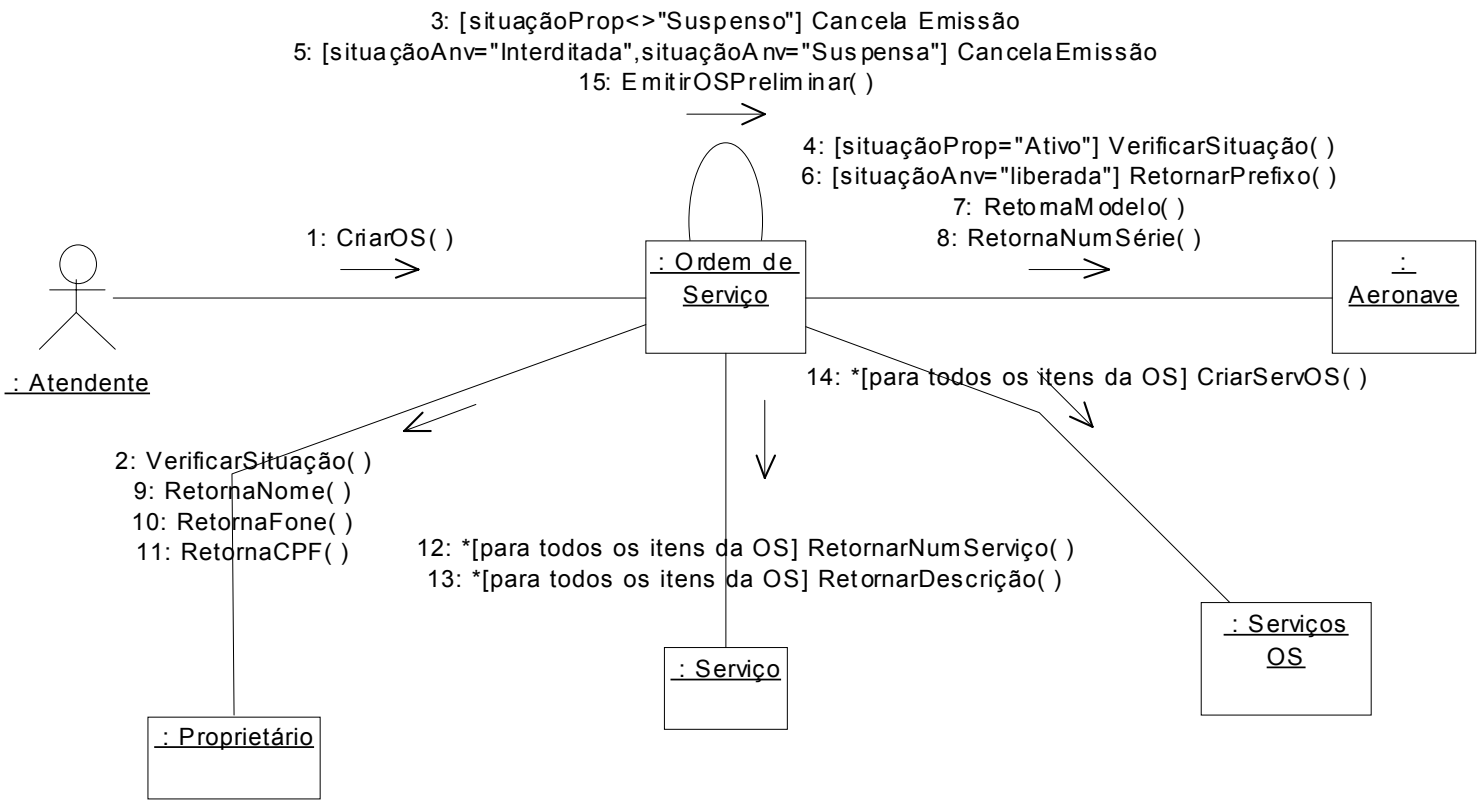
Anexo 8 Diagrama de Seqüência Lançar Serviços Realizados na OS do Estudo de Caso



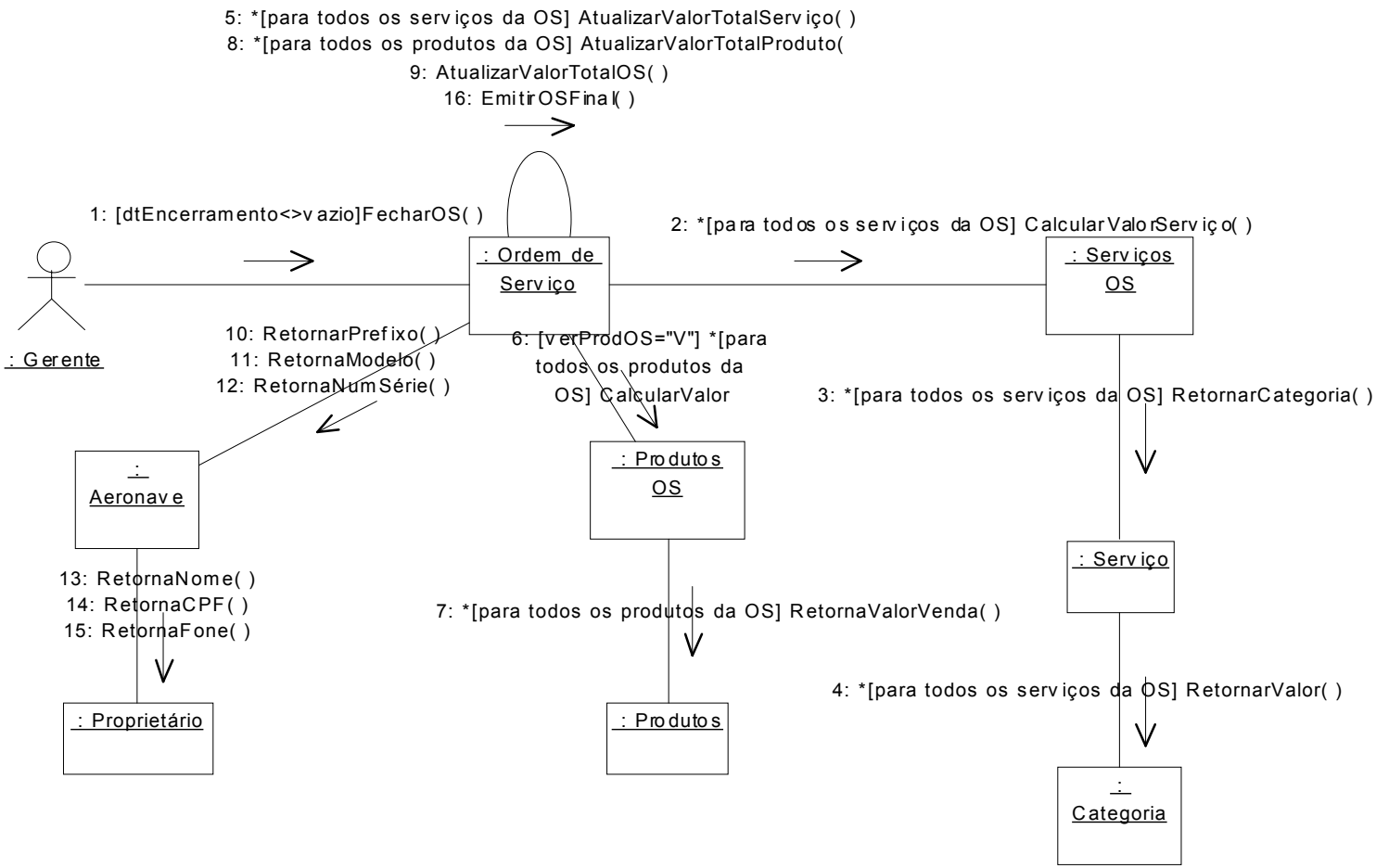
Anexo 9 Diagrama de Seqüência Alocar Mecânico nos Serviços da OS do Estudo de Caso



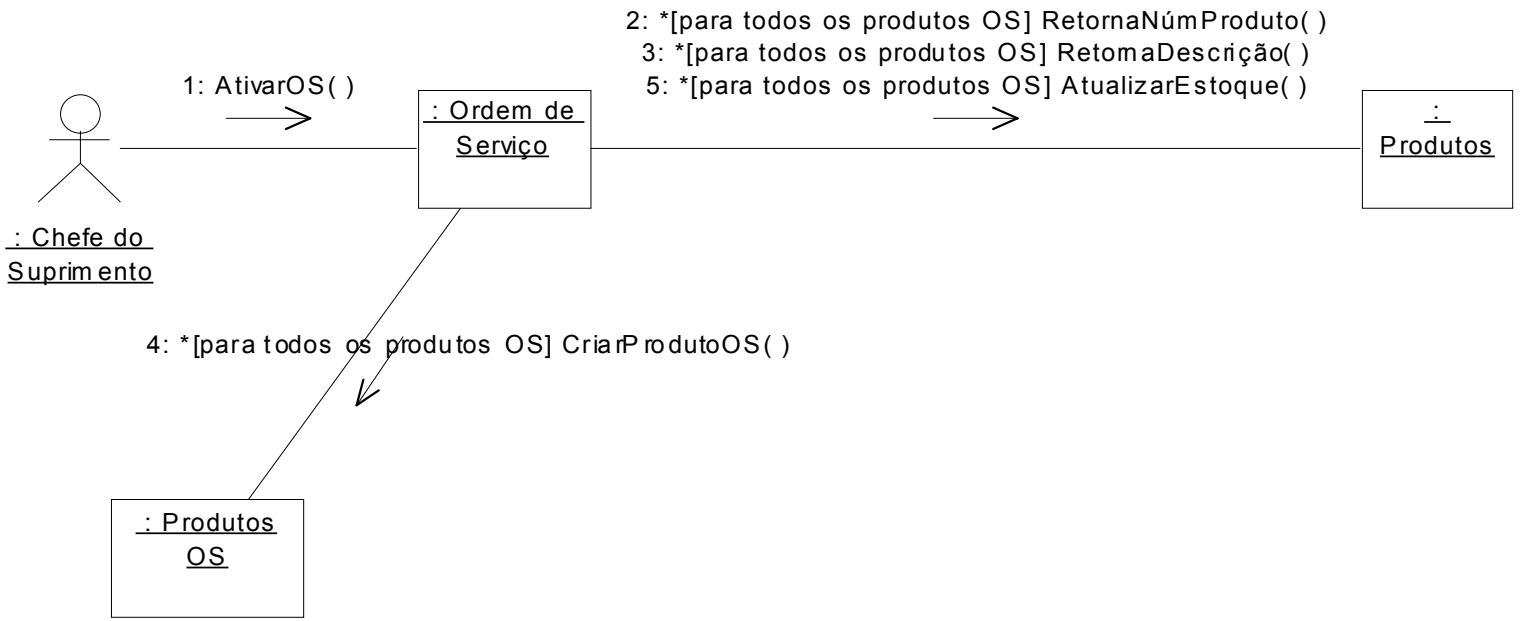
Anexo 10 Diagrama de Colaboração Manter Ordem de Serviço do Estudo de Caso



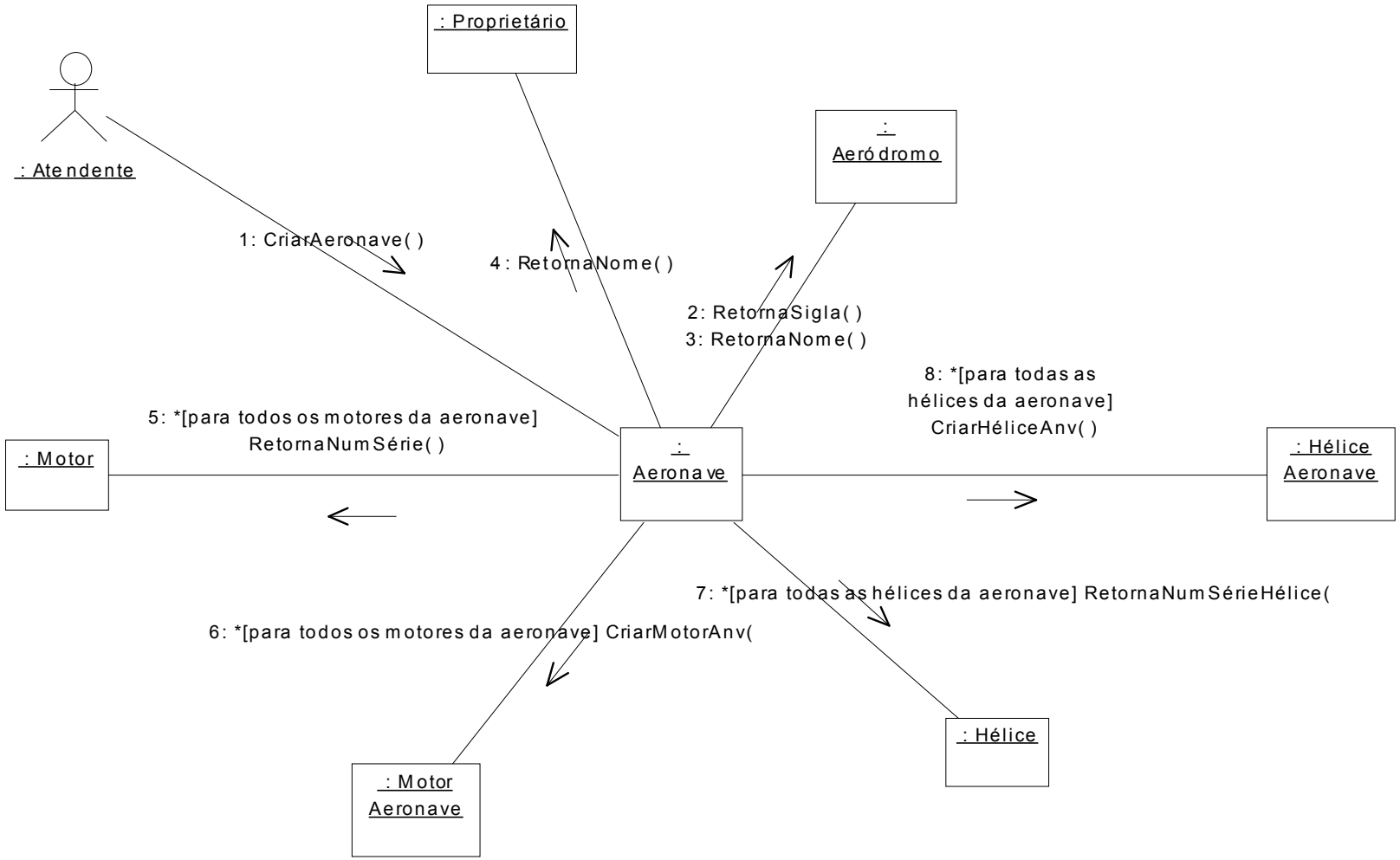
Anexo 11 Diagrama de Colaboração Fechar Ordem de Serviço do Estudo de Caso



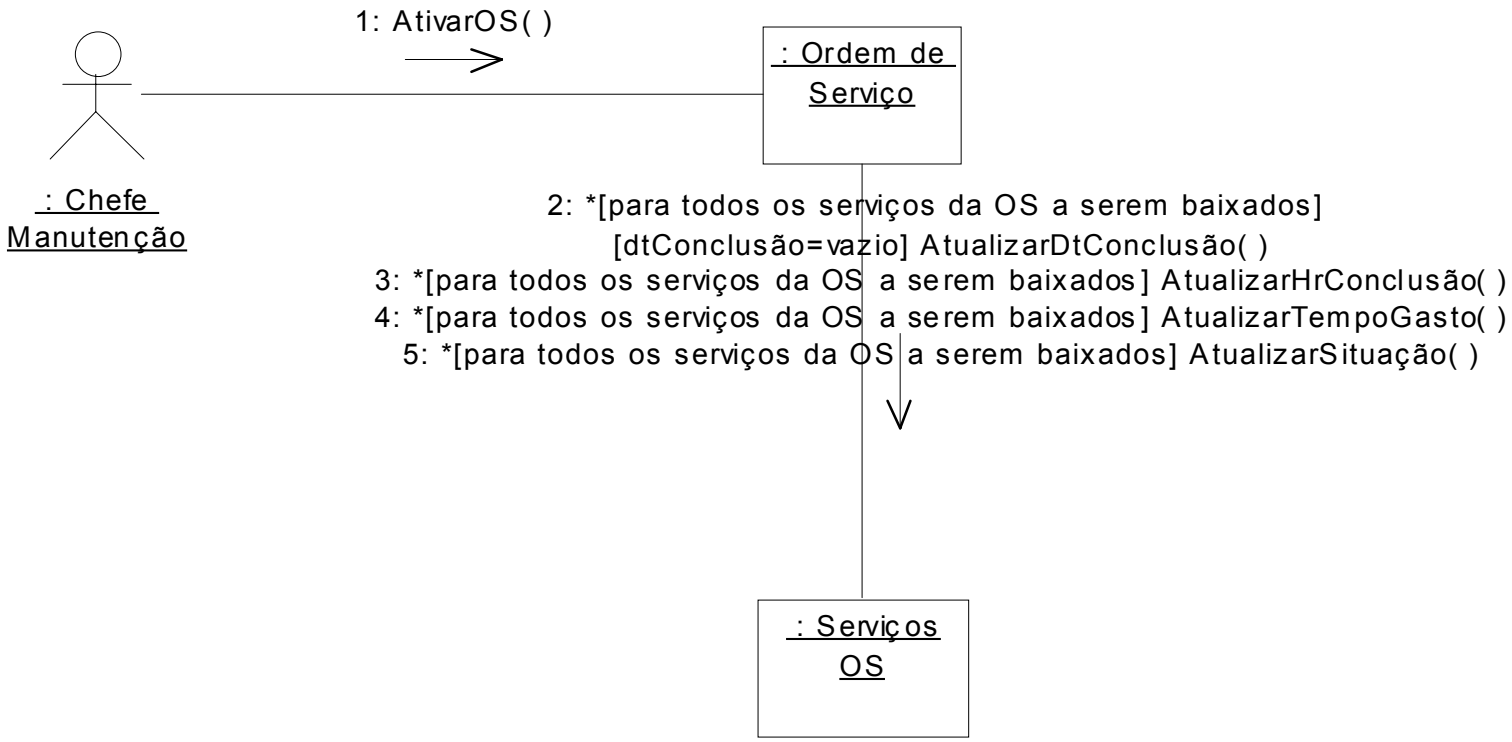
Anexo 12 Diagrama de Colaboração Manter Produtos da OS do Estudo de Caso



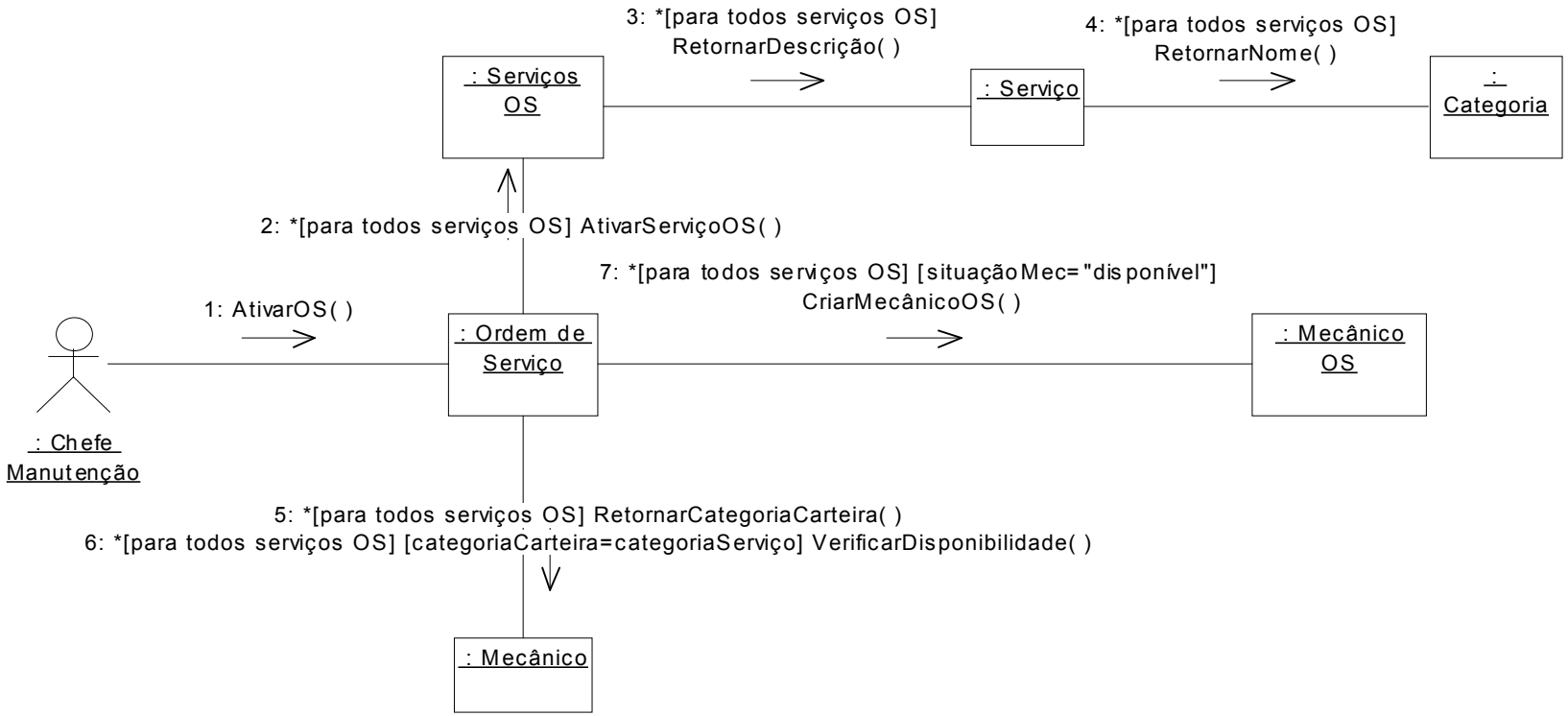
Anexo 13 Diagrama de Colaboração Manter Aeronave do Estudo de Caso



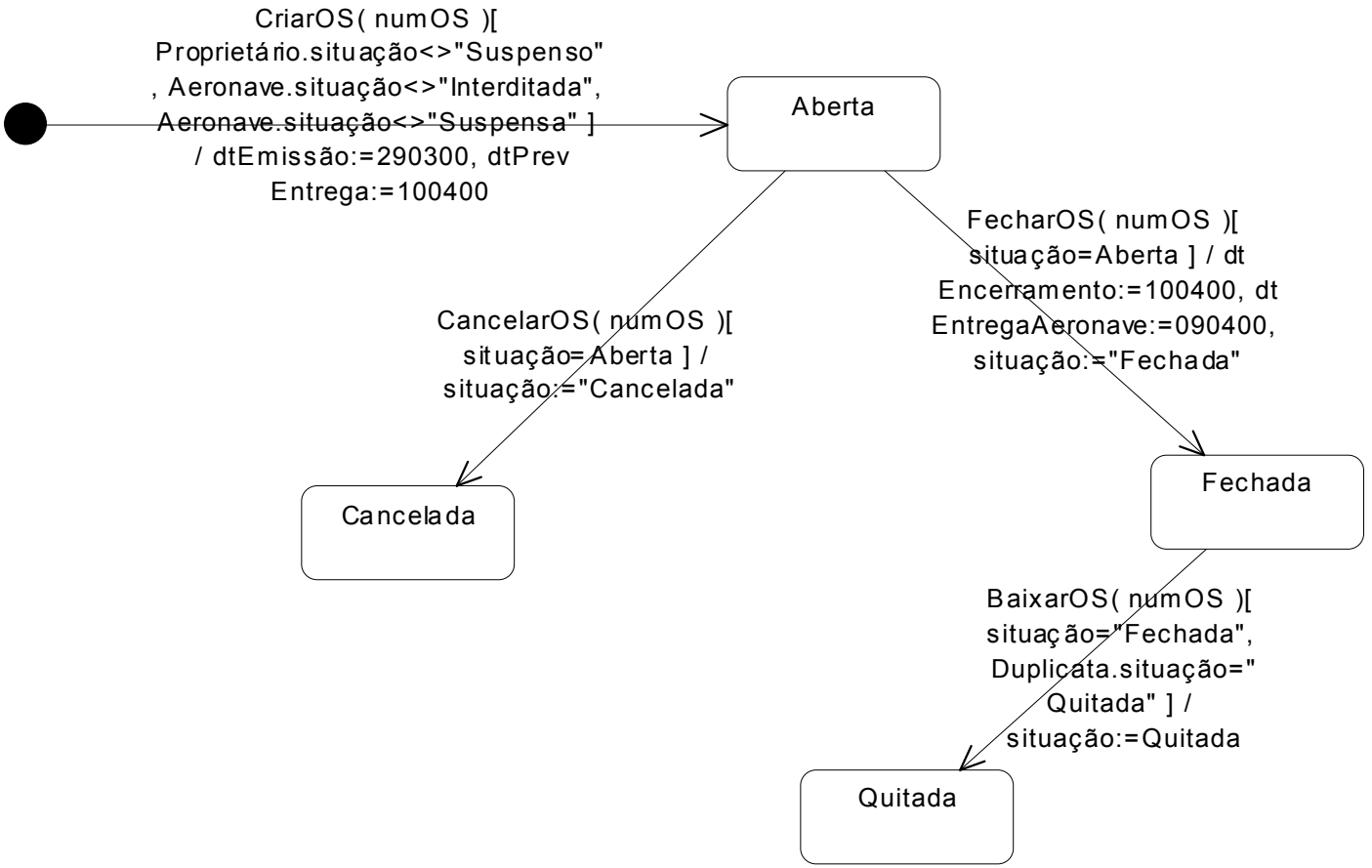
Anexo 14 Diagrama de Colaboração Lançar Serviços Realizados na OS do Estudo de Caso



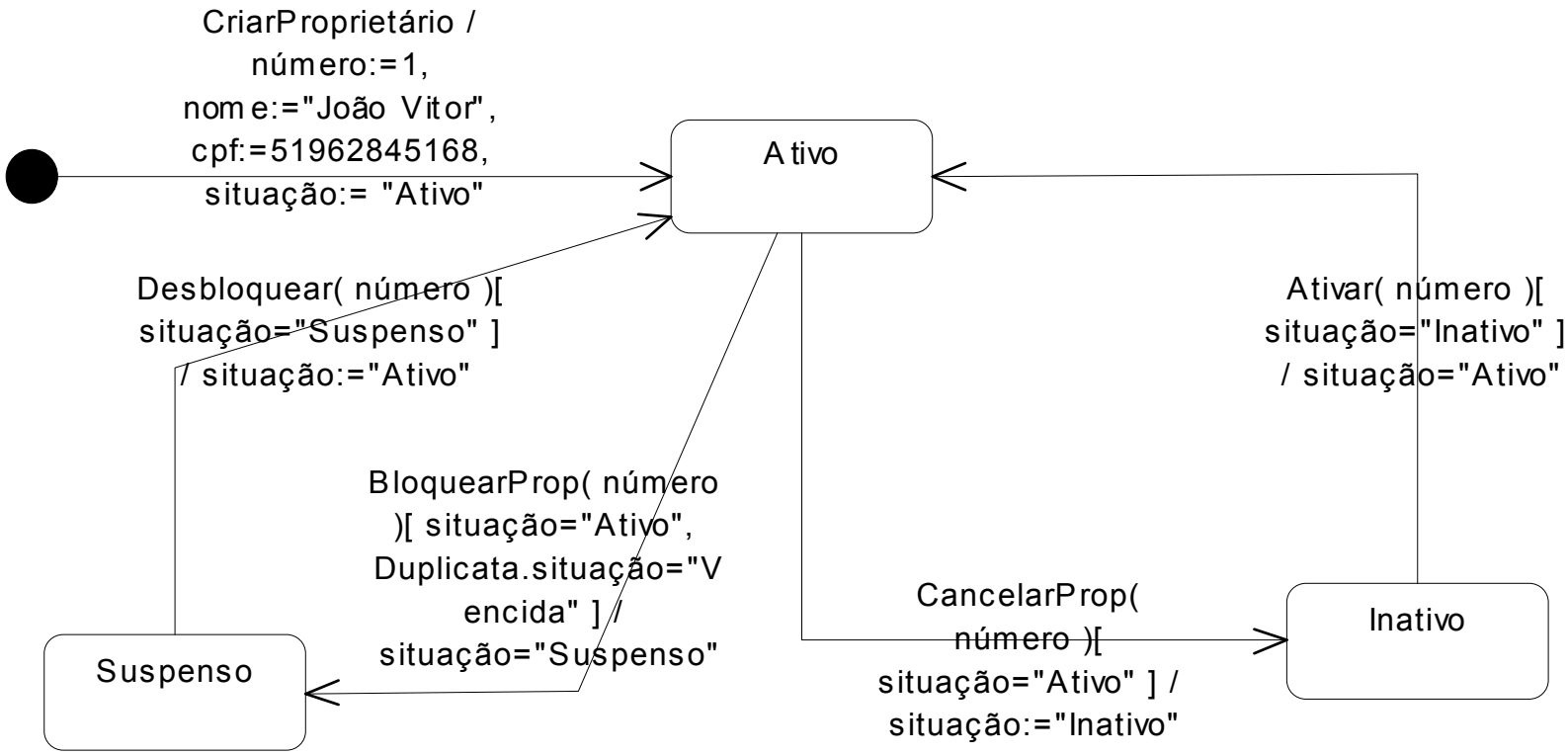
Anexo 15 Diagrama de Colaboração Alocar Mecânico nos Serviços da OS do Estudo de Caso



Anexo 16 Diagrama de Estados da Classe Ordem de Serviço do Estudo de Caso



Anexo 17 Diagrama de Estados da Classe Proprietário do Estudo de Caso



Bibliografia

[BEH 96] BEHFOROOZ, A.; HUDSON, J. F. **Software engineering fundamentals**. New York: Oxford University Press, 1996.

[BEI 90] BEIZER, B. **Software testing techniques.** Boston: International Thomson Computer Press, 1990.

[BIN 95] BINDER, R. V. State-based testing. **Object Magazine,** [S.l.], v.5, n.6, p.75-78, July/Aug. 1995.

[BIN 95a] BINDER, R. V. Trends in testing object-oriented software. **Computer**, New York, v.28, n.10, p.68-69, Oct. 1995.

[BIN 97] BINDER, R. V. Verifying class associations. **Object Magazine,** [S.l.], p.18-20, Nov. 1997.

[BIN 98] BINDER, R. V. How to test UML sequence diagram scenarios. **Object Magazine,** [S.l.], v.8, n.1, p.16-19, Mar. 1998.

[BOO 99] BOOCH, G.; RUMBAUGH, J.; JACOBSON, I. **The unified modeling language user guide.** USA: Addison-Wesley, 1999.

[BRU 82] BRUCE, P.; PENDERSON, S. **The software development project**. New York: John Wiley & Sons, 1982.

[CHO 78] CHOW, T.S. Testing software design modeled by finite state machines. **IEEE Transactions on Software Engineering**, New York, v.4, n.3, p.178-187, May 1978.

[COL 98] COLANZI, T. E.; MASIERO, P. C. Uma abordagem de teste baseada em especificação UML: um estudo de caso. In: WORKSHOP DE TESTES EM ENGENHARIA DE SOFTWARE, 1998, Maringá. **Anais...** Maringá: [s.n.], 1998. p.17-20.

[COL 2000] COLANZI, T.E.; MASIERO, P.C.; MALDONADO, J.C. **ProDeS/UML:** Um processo de desenvolvimento e teste de software para UML. In: WORKSHOP IBEROAMERICANO DE ENGENHARIA DE REQUISITOS E AMBIENTES DE SOFTWARE, 2000, Cancún, México. **Memorias...** Morelos: Cenidet, 2000. p.508-509.

[CON 79] CONSTANTINE, L. L.; YOURDON, E. **Structured design**. Englewood Cliffs: Prentice-Hall, 1979.

[COO 94] COOK, S.; DANIELS, J. **Designing object systems:** object-oriented modeling with syntropy. USA:Prentice Hall, 1994.

[COR 99] CORAZZA, E. A. M**. Critérios de cobertura para o teste de software orientado a objetos**: trabalho individual. Porto Alegre: PGCC da UFRGS, 1999. (TI-833).

[ERI 98] ERIKSSON, H. ; PENKER, M. **UML tollkit**. New York: John Willey & Sons, 1998.

[FAY 95] FAYAD, M. E.; TSAI, W. Object-oriented experiences. **Communications of ACM**, New York, v.38, n.10, p.51-53, Oct. 1995.

[FOW 2000] FOWLER, M. **UML distilled:** A brief guide to the standard object modeling language. 2nd ed. Reading: Addison Wesley, 2000.

[FRA 88] FRANKL, P. G.; WEYUKER, E. J. An applicable family of data flow testing criteria. **IEEE Transactions on Software Engineering**, New York, v.14, n.10, p.1483-1498, Oct. 1988.

[HAR 92] HARROLD, J. M.; McGREGOR, J.; FITZPATRICK, J. K. Incremental testing of object-oriented class structures. In: INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 14., 1992, Melbourne, Australia. **Proceedings...** Los Alamitos: IEEE, 1992. p.68-80.

[HEM 76] HERMAN, P. M. A data flow analysis approach to program testing. **Australian Computer Journal**, [S.l.], v.8, n.3, Nov. 1976.

[HER 97] HERBERT, J. S. **Teste de depuração de software orientado a objetos:** exame de qualificação. Porto Alegre: CPGCC da UFRGS, 1997.

[HET 87] HETZEL, W. **Guia completo de teste de software**. Rio de Janeiro: Campus, 1987.

[HUA 75] HUANG, J.C. An aproach to program testing. **Computing Surveys**, New York, v.7, n.3, p.111-128, Sept. 1975.

[JAC 92] JACOBSON, I. et al. **Object-oriented software engineering**: A use case drive approach. Reading: Addison Wesley, 1992.

[JIN 96] JIN, Z.; OFFUTT, J. Coupling-based integration testing. In: IEEE INTERNATIONAL CONFERENCE ON ENGINEERING OF COMPLEX COMPUTER SYSTEMS, 2., 1996. **Proceedings...** [S.l.:s.n.], 1996. p.10-17. Disponível em: <http://www.isse.gmu.edu/faculty/ofut/rsrch/abstracts/complex.html>. Acesso em: 04 mar. 1999.

[JIN 98] JIN, Z., OFFUTT, J. **Coupling-based criteria for integration testing**. 1998. Disponível em: <http://www.isse.gmu.edu/faculty/ofut/rsrch/abstracts/couptest.html>. Acesso em: 04 mar. 1999.

[KAN 99] KANER, C.; FALK, J.;NGUYEN, H. Q. **Testing computer software**. 2nd ed. New York: John Wiley & Sons, 1999.

[KUN 96] KUNG, D. et al. On regression testing of object-oriented programs. In: **Journal of systems and software**. [S.l.], v.32, n.1, p.21-40, Jan. 1996.

[LAR 99] LARMAN, C. **Applying UML and patterns:** An introduction to object-oriented analysis and design. Englewood Cliffs: Prentice-Hall, 1999.

[LAS 83] LASKI, J. W.; KOREL B. A data flow oriented program testing strategy. **IEEE Transactions on Software Engineering**, New York, v.9, n.3, May 1983.

[MAL 91] MALDONADO, J. C. **Critérios potenciais usos:** Uma contribuição ao teste estrutural de *Software*. Campinas: DCA-FEE-UNICAMP, 1991. Tese de Doutorado.

[MAR 95] MARICK, B. **The craft of software testing.** Englewood Cliffs: Prentice-Hall, 1995.

[MCG 94] McGREGOR, J. D.; KORSON, T. D. Integrated Object-Oriented Testing and Development Processes. **Communications of the ACM,** New York, v.37, n.9, p.59-77, Sept. 1994.

[MCG 96] McGREGOR, J. D. Testing Object-Oriented Componentes. In: EUROPEAN CONFERENCE ON OBJECT-ORIENTED PROGRAMING, 10., 1996. **Tutorial Notes**. [S.l.:s.n.], 1996.

[MEN 98] MENEZES, P. F. B. **Linguagem formais e autômatos**. Porto Alegre: Instituto de Informática da UFRGS: Sagra Luzzatto, 1998.

[MET 2000] MENZIES, T.; CUKIC, B. When to test less. **IEEE Software**, [S.l.], v.17, n.5, p.96-101, Sept. 2000.

[MEY 97] MEYER, B. **Object-Oriented software construction**. 2nd ed. Upper Saddle River: Prentice-Hall, 1997.

[MYE 79] MYERS, G.J. **The art of software testing**. New York: John Willey & Sons, 1979.

[NTA 84] NTAFOS, S. C. On required element testing. **IEEE Transactions on Software Engineering**, New York, v.10, n.6, Nov. 1984.

[NTA 88] NTAFOS, S. C. A comparison of some structural testing strategis. **IEEE Transactions on Software Engineering**, New York, v.14, n.6, p.868-873, June 1998.

[OFF 93] OFFUTT, A. J.; HARROLD, M. J. A software metric system for module coupling. **The Journal of Systems and Software**, [S.l.], v.20, n.3, p.295-308, Mar. 1993.

[OMG 99] OMG. **Unified modeling language specification version 1.3**. June 1999. Disponível em: <http://www.rational.com/uml/resources/documentation>. Acesso em: 23 set.1999.

[PAG 80] PAGE-JONES, M. **The practical guide to structured system design**. New York: Yourdon Press, 1980.

[PAG 95] PAGE_JONES, M. **What every programmer should know about object-oriented design**. New York: Dorset House, 1995.

[PIN 96] PINTO, I. M. **Inteligência artificial no teste de software**: trabalho individual. Porto Alegre: CPGCC da UFRGS, 1996. (TI-543).

[PRE 95] PRESSMAN, R.S. **Engenharia de software**. São Paulo: Makron Books, 1995.

[QAI 2000] QAI. **Status of software testing**. Florida:QAI, 1999. Disponível em: <www.qaiusa.com>. Acesso em: 29 out. 2000.

[RAP 82] RAPPS, S.; WEYUKER, E. J. Data Flow analysis techniques for test data selection. In: THE INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING, 1982. **Proceedings...** [S.l.:s.n.], 1982. p.272-278.

[RAP 85] RAPPS, S.; WEYUKER, E. J. Selecting Software test data using data flow information. **IEEE Transactions on Software Engineering**, New York, v.11, n.4, p.367-375, Apr. 1985.

[ROM 96] ROMBALDI V. **Heurísticas para geração de dados de teste**. Porto Alegre: CPGCC/UFRGS, 1995. Dissertação de Mestrado.

[RUS 2000] RUSS, M. L.; MCGREGOR, J. D. A software development process for small projects. **IEEE Software**, [S.l.], v.17, n.5, p.96-101, Sept. 2000.

[SIL 95] SILVA, J. B. da. **PROTESTE+:** Ambiente de validação automática de qualidade de software através de técnicas de teste e de métricas de complexidade. Porto Alegre: CPGCC/UFRGS, 1995. Dissertação de Mestrado.

[SMI 92] SMITH, M. D.; ROBSON, D. J. A framework for testing object-oriented programs. **Journal of Object-Oriented Programming**, [S.l.], v.5, n.3, p.45-53, June 1992.

[SPO 97] SPOTO, E. S. et al. Teste estrutural baseado em fluxo de dados de software aplicativo de banco de dados relacional. In: WORKSHOP DO PROJETO VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO,

1997, Águas de Lindóia. **Anais...** Águas de Lindóia: [s.n.], 1997. p.163-176.

[TAI 89] TAI, K. C. What to do beyong branch testing. **ACM Software Engineering Notes**, New York, v.14, n.2, p.58-61, Apr. 1989.

[VER 94] VERGÍLIO, S. R. et al. Caminhos não-executáveis no teste de integração: caracterização, previsão e determinação. In: SIMPÓSIO BRASILEIRO DE ENGENHARIA DE SOFTWARE, 8., 1994. Curitiba. **Anais...** Curitiba: [s.n.], 1994.

[VER 96] VERGILIO, S. R. **Definição de critérios estruturais restritos**. Campinas: DCA/FEE/UNICAMP, 1996. Technical Report.

[VER 97] VERGILIO, S. R. et al. Aumentando a eficácia dos critérios estruturais através da utilização de critérios restritos. In: WORKSHOP DO PROJETO VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO, 1997, Águas de Lindóia. **Anais...** Águas de Lindóia: [s.n.], 1997. p.147-159.

[VIL 97a] VILELA, P. R. S. el al. Uma visão sobre o teste estrutural baseado em análise de fluxo de dados. In: WORKSHOP DO PROJETO VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO, 1997, Águas de Lindóia. **Anais...** Águas de Lindóia: [s.n.], 1997. p.03-14.

[VIL 97b] VILELA, P. R. S. et al. Data flow based testing of programs with pointers. In: WORKSHOP DO PROJETO VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO, 1997, Águas de Lindóia. **Anais...** Águas de Lindóia: [s.n.], 1997. p. 15-26.

[VIN 97] VINCENZI, A. M. R. et al. Critério análise de mutantes: estado atual e perspectivas. In: WORKSHOP DO PROJETO VALIDAÇÃO E TESTE DE SISTEMAS DE OPERAÇÃO, 1997, Águas de Lindóia. **Anais...** Águas de Lindóia: [s.n.], 1997. p.137-146.

[VLI 97] VLIET, J. C. V. **Software engineering:** principles and practice. England: John Wiley, 1993.

[WEY 80] WEYUKER, E. J.; OSTRAND, T. J. Theories of program testing and the application of revealing subdomais. **IEEE Transactions on Software Engineering**, New York, v.6, n.3, p.236-246, May 1980.

[WIN 98] WINTER. M. Managing object-oriented integration and regression testing. In: EUROSTAR, 1998, Munich. **Proceedings...** [S.l.:s.n.], 1998.

[YAT 89] YATES, D.F., MALEVRIS, N. Reducing the effects of infeasible paths in branch testing. **ACM Computer Surveys**, New York, p.48-54, 1989.