Teste de Integração para Sistemas Baseados em Componentes

Cidinha Costa Gouveia

Dissertação submetida à Coordenação do Curso de Pós-Graduação em Informática da Universidade Federal da Paraíba - Campus II como parte dos requisitos necessários para obtenção do grau de Mestre em Informática.

Área de Concentração: Ciência da Computação

Linha de Pesquisa: Engenharia de Software

Patrícia Duarte de Lima Machado

(Orientadora)

Jorge César Abrantes de Figueiredo

(Orientador)

Campina Grande, Paraíba, Brasil

GOUVEIA, Cidinha Costa
G719T

Teste de Integração para Sistemas Baseados em Componentes.

Dissertação de Mestrado, Universidade Federal de Campina Grande, Centro de Ciências e Tecnologia, Coordenação de Pós-Graduação em Informática, Campina Grande, Paraíba, Fevereiro de 2004.

166 p. II.

Orientadores: Patrícia Duarte de Lima Machado
Jorge César Abrantes de Figueiredo

Palavras Chave:

1. Engenharia de Software
2. Teste de Integração
3. Componentes

CDU - ~~519.683~~ 004.41(043)

"TESTE DE INTEGRAÇÃO PARA SISTEMAS BASEADOS EM COMPONENTES"

CIDINHA COSTA GOUVEIA

DISSERTAÇÃO APROVADA EM 27.02.2004

PROFª PATRICIA DUARTE DE LIMA MACHADO, Ph.D
Orientadora

PROF. JORGE CÉSAR ABRANTES DE FIGUEIREDO, D.Sc
Orientador

PROFª FRANCILENE PROCÓPIO GARCIA, D.Sc
Examinadora

PROF. ÁLVARO FRANCISCO DE CASTRO MEDEIROS, D.Sc
Examinador

CAMPINA GRANDE – PB

Resumo

Como a conectividade entre os componentes é um ponto chave do Software Baseado em Componentes, a verificação da interação entre os componentes da aplicação torna-se essencial para obter um produto final de qualidade. Dessa forma, são realizados testes com a finalidade de garantir que a combinação entre os componentes do sistema produza um comportamento esperado. Esses testes são chamados de teste de integração. Apesar de que esforços consideráveis têm sido realizados, ainda existe uma grande carência por métodos e técnicas efetivas de teste de integração de componentes que cubra todas as etapas necessárias, a fim de se ter um processo de teste completo do ponto de vista de componentes. O objetivo deste trabalho é propor um método para testar a integração entre os componentes de um sistema. Os artefatos de teste são gerados a partir de especificações em UML (Unified Modelling Language), que é uma linguagem de especificação largamente utilizada, facilitando o uso do método por grande parte da comunidade da engenharia de software. O método fornece potencial para automação, uma vez que disso depende o seu uso na prática. Um estudo de caso também é apresentado a fim demonstrar a aplicação do método.

Abstract

As the connectivity between components is a concern of Component-Based Software, verifying the interaction between these components becomes essential to get a final product quality. Tests are produced in order to guarantee that the combining of system components yields an expected behavior. These tests are known as integration tests. Although considerable efforts have been carried out, there still exists a great absence of effective methods and techniques of integration test that cover all the necessary stages, in order to have a complete test process of components point of view. The objective of this work is to supply a method for testing the integration between the components that constitute a system. This method is simple and efficient, besides being linked to a clear-cut component development process. The test devices are generated from UML (Unified Modelling Language), that is a widely used specification language, facilitating the use of this method for a large part of the software engineering community. The method supplies a potential for automation, since its pratical use is dependt on this. A case study is also presented to demonstrate the method application.

Agradecimentos

À Simone Cavalcanti Antonino e Ibrahim Buarque Antonino pelo grande apoio que me deram em Campina Grande. A minha mãe Laurizênia Maria Costa Gouveia que sempre me ajuda nos momentos críticos. Ao grande amigo Nilton Freire pela grande amizade e confiança. Ao professor e amigo Pedro Luiz Christiano (UFPB), pela contínua amizade e incentivo constante. Aos meus tios Lourival Júnior e Marcos Rogério que sempre me apóiam em minhas decisões. A Puppy que sempre me alegra quando nos momentos de cansaço. Aos professores do DSC por todos os conhecimentos transmitidos, especialmente a Dalton, Jorge e Patrícia. Aos colegas do Labpetri (Amancio, André, Carina, Cássio, Daniel Aguiar, Daniel, Emerson, Laísa, Paulo, Rodrigo, Taciano, Sandro) pelas discussões e momentos de descontração. Aos colegas do LES (Júlio, Marcos Duarte, Marcos Luiz, Rodrigo, Vanderson, Vitrúvio) pela amizade e disponibilidade em ajudar. Aos amigos especiais (Bento, Daniel, e Keizer) que estão sempre presente e dando apoio. Aos outros colegas (Carol, Robson, David, Ricardo, Victor, Augusto e Fabrício). As minhas amigas de longas datas (Larissa, Kátia e Paloma). A minha orientadora Patrícia Duarte de Lima Machado e ao meu co-orientador Jorge César Abrantes de Figueiredo, pelo grande empenho e dedicação ao longo mestrado. Aos funcionários do DSC pela boa vontade e disponibilidade no atendimento, em especial a Aninha e Zeneide.

Conteúdo

1 Introdução **1**

1.1 Contextualização 1

1.2 Trabalhos Relacionados 4

1.3 Objetivos 6

1.4 Estrutura da Dissertação 7

2 Testes: Uma visão geral **9**

2.1 Teste Funcional 9

2.2 Teste de Software Orientado a Objetos 10

2.3 Teste de Software Baseados em Componentes 12

2.4 Teste de Integração 14

2.5 Estratégia de Teste de Integração 17

2.6 Conclusão 23

3 Desenvolvimento de SBC e Teste de Componentes **25**

3.1 Introdução 25

3.2 Desenvolvimento de SBC 27

3.2.1 Requisitos 27

3.2.2 Modelagem 30

3.2.3 Fornecimento 33

3.2.4 Montagem 34

3.2.5 Teste e Distribuição 34

3.3 Teste de Componentes 34

3.3.1 Planejamento 36

3.3.2 Especificação 39
3.3.3 Construção e Execução dos Testes e Verificação dos Resultados . . 42
3.4 Conclusão 42

4 Método de Teste de Integração 44
4.1 Introdução 44
4.2 Estrutura 45
4.3 Etapas do método de teste de integração 46
4.3.1 Planejamento 46
4.3.2 Especificação 49
4.3.3 Construção, Execução e Análise dos Resultados 52
4.4 Conclusões 52

5 Estudo de Caso - Desenvolvimento da Aplicação 54
5.1 Roteiro Mínimo 54
5.2 Escolha da Aplicação 57
5.3 Desenvolvimento do Estudo de Caso 58
5.3.1 Requisitos 58
5.3.2 Modelagem 63
5.3.3 Fornecimento, Montagem e Testes 76
5.4 Conclusões 77

6 Estudo de Caso - Aplicação do Método de Teste de Integração 78
6.1 Planejamento 78
6.1.1 Construção da Análise de Risco 79
6.2 Especificação 80
6.2.1 Escolha da ordem de teste dos componentes e dos *stubs* necessários 80
6.2.2 Geração e Seleção dos Casos de Teste 85
6.2.3 Geração dos Dados e Oráculos de Teste 93
6.3 Construção, Execução e Análise dos Resultados 100
6.4 Conclusões 110

7 Considerações Finais **113**

7.1 Resultados Alcançados 113

7.2 Contribuições 114

7.3 Trabalhos Futuros 115

A Casos de Uso **120**

B Modelos de Informação **127**

C Diagramas de Seqüência referente ao Componente Jogo **132**

D Diagramas de Seqüência referente ao Componente Regras **136**

E Expressões Regulares e OCL's **144**

F Especificações das Operações das Interfaces **151**

Lista de Figuras

2.1 Modelagem dos nodos 18
2.2 Modelagem da dependência Classe para Classe 19
2.3 Tipos de Dependência 19
2.4 GTD preliminar e regra de normalização 20
2.5 CFC e as partições 20
2.6 Chamadas Recursivas e CFCs não triviais 22

3.1 Etapas do Processo de Desenvolvimento 28
3.2 Integração das Etapas de Teste de Componentes no Processo de Desenvolvimento 36
3.3 Modelo de Uso 41

4.1 Etapas do processo de teste proposto contextualizado dentro do processo de desenvolvimento Componentes UML 45
4.2 Etapas e atividades do processo de teste de componentes e de integração contextualizados dentro do processo de desenvolvimento Componentes UML . 47

5.1 Modelo de Processo do Negócio 58
5.2 Modelo Conceitual do Negócio 59
5.3 Modelo de Caso de Uso 60
5.4 Modelo de Tipo de Negócio 65
5.5 Modelo de Responsabilidade de Interfaces 66
5.6 Arquitetura Inicial dos Componentes 66
5.7 Diagrama de Colaboração da operação escolherJogo() da interface ControladorIF 70

5.8 Diagrama de Colaboração da operação mudarDirecao() da interface ControladorIF 70
5.9 Diagrama de Colaboração da operação mudarVelocidade() da interface ControladorIF 71
5.10 Diagrama de Colaboração da operação obterScore() da interface ControladorIF 71
5.11 Diagrama de Colaboração da operação rodaJogo() da interface ControladorIF 71
5.12 Especificação da operação escolherJogo(String nome) do componente Controlador 72
5.13 Especificação da operação rodaJogo() do componente Controlador 72
5.14 Especificação da operação mudarDirecao(int direcao) do componente Controlador 73
5.15 Especificação da operação mudarVelocidade(int n) do componente Controlador 73
5.16 Especificação da operação obterScore() do componente Controlador 73
5.17 Modelo de Informação da interface do Componente Cobra 73
5.18 Cenário Inicial 75
5.19 Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde o jogo termina por ter obtido a pontuação máxima, referente ao Componente Jogo 76

6.1 GTD do estudo de caso 81
6.2 GTD normalizado 82
6.3 algoritmo aplicado ao GTD 83
6.4 CFC do GTD 83
6.5 Algoritmo aplicado recursivamente ao CFC 84
6.6 Modelo de Uso do Caso de Uso Jogar Hungry Snake referente ao componente Jogo 88
6.7 Modelo de Uso do Caso de Uso Jogar Hungry Snake referente ao componente Regras 89
6.8 Modelo de Uso do Caso de Uso Jogar Gula Gula referente ao componente Jogo 90

6.9 Modelo de Uso do Caso de Uso Jogar Gula Gula referente ao componente Regras 91
6.10 Cenário Inicial comum a todos os Modelos de Uso 94
6.11 Tabela de Decisão do jogo Hungry Snake referente ao componente Jogo . . 97
6.12 Tabela de Decisão do jogo Hungry Snake referente ao componente Regras . 97
6.13 Tabela de Decisão do jogo Gula Gula referente ao componente Jogo 98
6.14 Tabela de Decisão do jogo Gula Gula referente ao componente Regras . . . 98
6.15 Tabela de Decisão do Caso de Uso Obter Ranking 98

B.1 Modelo de Informação da Interface do Componente Controlador 127
B.2 Modelo de Informação da interface do Componente Comida 128
B.3 Modelo de Informação da interface do Componente Parede 128
B.4 Modelo de Informação da interface do Componente Tabuleiro 128
B.5 Modelo de Informação da interface do Componente Ranking 129
B.6 Modelo de Informação da interface do Componente Regras 129
B.7 Modelo de Informação da interface do Componente Cobra 129
B.8 Modelo de Informação da interface do Componente Jogo 130
B.9 Modelo de Informação da interface do Componente Ponto 131

C.1 Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, onde o jogo termina por a cobra ter batido na parede, referente ao Componente Jogo . . 132
C.2 Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde a cobra bate nela mesmo, referente ao Componente Jogo 133
C.3 Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde o jogo termina por ter obtido a pontuação máxima, referente ao Componente Jogo 133
C.4 Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina por a cobra ter batido na parede, referente ao Componente Jogo 134
C.5 Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina porque a cobra bateu nela mesmo, referente ao Componente Jogo 134

C.6 Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina por ter obtido a pontuação máxima, referente ao Componente Jogo 135

D.1 Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde a cobra come uma comida, referente ao Componente Regras 136
D.2 Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde o jogo termina por a cobra ter batido na parede, referente ao Componente Regras 137
D.3 Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde o jogo termina porque a cobra bateu nela mesmo, referente ao Componente Regras 138
D.4 Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde o jogo termina por ter obtido a pontuação máxima, referente ao Componente Regras 139
D.5 Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde a cobra come uma comida, referente ao Componente Regras 140
D.6 Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina por a cobra ter batido na parede, referente ao Componente Regras 141
D.7 Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina porque a cobra bateu nela mesmo, referente ao Componente Regras 142
D.8 Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina por ter obtido a pontuação máxima, referente ao Componente Regras 143

F.1 Especificação da operação escolherJogo(String nome) do componente Controlador 151
F.2 Especificação da operação rodaJogo() do componente Controlador 151
F.3 Especificação da operação mudarDirecao(int direcao) do componente Controlador 152
F.4 Especificação da operação mudarVelocidade(int n) do componente Controlador 152

F.5 Especificação da operação obterScore() do componente Controlador 152
F.6 Especificação da operação geraPontos(int larguraTabuleiro, int alturaTabuleiro) do componente Comida 153
F.7 Especificação da operação getPontosScore() do componente Comida 153
F.8 Especificação da operação getRabo() do componente Cobra 154
F.9 Especificação da operação getCabeca() do componente Cobra 154
F.10 Especificação da operação moverCobra(int direcao) do componente Cobra . 154
F.11 Especificação da operação crescer(Ponto crescer) do componente Cobra . . 155
F.12 Especificação da operação geraPontos(int larguraTabuleiro, int alturaTabuleiro) do componente Parede 155
F.13 Especificação da operação addPonto(Ponto umPonto) do componente Parede 155
F.14 Especificação da operação getLinhas() do componente Tabuleiro 156
F.15 Especificação da operação getColunas() do componente Tabuleiro 156
F.16 Especificação da operação setLinhas(int numLinhas) do componente Tabuleiro 156
F.17 Especificação da operação setColunas(int numColunas) do componente Tabuleiro 157
F.18 Especificação da operação (String nome, int score) do componente Ranking 157
F.19 Especificação da operação setRegras(String nome) do componente Regras . 157
F.20 Especificação da operação testeCobraComeu(JogoIF jogo) do componente Regras 158
F.21 Especificação da operação testeCobraBateuParede(JogoIF jogo) do componente Regras 158
F.22 Especificação da operação testeCobraBateuNelaMesma(JogoIF jogo) do componente Regras 159
F.23 Especificação da operação jogar(JogoIF jogo) do componente Regras . . . 160
F.24 Especificação da operação testeFimJogo(JogoIF jogo) do componente Regras 161
F.25 Especificação da operação setDirecao(int direcao) do componente Jogo . . 161
F.26 Especificação da operação setScore(int score) do componente Jogo 162
F.27 Especificação da operação setStatus(boolean status) do componente Jogo . 162
F.28 Especificação da operação setVelocidade(int velocidade) do componente Jogo 162
F.29 Especificação da operação run() do componente Jogo 162

F.30 Estrutura Interna do Componente Regras 163
F.31 Estrutura Interna do Componente Jogo 164
F.32 Estrutura Interna do Componente Parede 164
F.33 Estrutura Interna do Componente Comida 165
F.34 Estrutura Interna do Componente Cobra 165
F.35 Estrutura Interna do Componente Ranking 165
F.36 Estrutura Interna do Componente Tabuleiro 166
F.37 Estrutura Interna do Componente Controlador 166
F.38 Estrutura Interna do Componente Ponto 166

Lista de Tabelas

5.1 Texto Complementar ao Modelo de Processo do Negócio 59
5.2 Caso de Uso Controlar a Direção 61
5.3 Caso de Uso Cobra Come 61
5.4 Caso de Uso Jogar Hungry Snake 62
5.5 Interfaces do Sistema, operações e Casos de Uso 64

6.1 Análise de Risco para cada Caso de Uso 79

A.1 Caso de Uso Controlar a Direção 120
A.2 Caso de Uso Cobra Come 121
A.3 Caso de Uso Jogar Hungry Snake 121
A.4 Caso de Uso Jogar Gula Gula 122
A.5 Caso de Uso Jogar Gula Gula 2 123
A.6 Caso de Uso Jogar Magic Snake 124
A.7 Caso de Uso Jogar My Snake 125
A.8 Caso de Uso Escolher Nível 126
A.9 Caso de Uso Visualizar Scores 126
A.10 Caso de Uso Escolher Jogo 126

Capítulo 1

Introdução

Este capítulo tem como objetivo principal fornecer uma visão geral do trabalho realizado e encontra-se dividido em quatro seções: a Seção 1 contextualiza o trabalho dentro da realidade da Engenharia de Software fornecendo uma visão geral sobre Software Baseado em Componentes; a Seção 2 introduz alguns trabalhos relacionados com o tema em questão; a Seção 3 elenca os objetivos do trabalho; a Seção 4 fala sobre a estrutura da dissertação, detalhando o conteúdo de cada capítulo.

1.1 Contextualização

Atingir um alto nível de qualidade de produto serviço é o objetivo da maioria das organizações. Atualmente, não é mais aceitável entregar produtos com baixa qualidade e reparar os problemas depois que os produtos foram entregues aos clientes [Ian, 2003]. Os custos relacionados com problemas encontrados em sistemas de software tornam-se maiores à medida que se aumenta o tempo gasto na detecção e reparo dos mesmos. Por isso, é importante que a qualidade esteja presente nos sistemas desenvolvidos. Contudo, a qualidade do software é um conceito complexo, que não pode ser definido de maneira simples. Classicamente, a noção de qualidade tem sido a de que o produto desenvolvido deve cumprir com sua especificação [Ian, 2003].

Devido à crescente disseminação do uso de software, produzir um software de qualidade hoje, não é mais um simples requisito e sim uma necessidade. Porém, apesar da qualidade ser um fator essencial na maioria dos casos, construir um software com este perfil ainda pode

requerer um custo bastante alto. Por isso, pesquisadores e setores da indústria têm investido bastante no desenvolvimento de novas tecnologias capazes de produzir software de maneira eficiente, que seja de fácil entendimento e manutenção, a um baixo custo e em um período mínimo de tempo.

O desenvolvimento de software baseado em componentes é uma realidade em ascensão na atual indústria de software por ser encarado como uma abordagem que permite, simultaneamente, reduzir custos no desenvolvimento, encurtar o período necessário ao desenvolvimento do software e aumentar a sua qualidade [Szyperski, 1998]. Os ganhos obtidos ao se produzir esse tipo de sistema se dá em decorrência da reutilização dos componentes, o que faz minimizar o tempo gasto no processo de desenvolvimento, e da facilidade obtida em adicionar novas funcionalidades ao sistema de acordo com a necessidade e conveniência do cliente. De forma semelhante, funcionalidades já existentes e que tenham sofrido algum tipo de mudança podem também ser substituídas sem causar nenhum impacto às outras partes do sistema.

De acordo com uma estimativa feita por Jeff Offutt, em [Wu et al., 2003], grande parte dos sistemas de software desenvolvidos recentemente são Sistemas Baseados em Componentes (SBC). Esses sistemas consistem em um conjunto de repositórios de componentes, os quais acoplados, permitem uma integração rápida, facilitando a montagem do sistema. Os componentes podem ser escritos em diversas linguagens de programação, executados em várias plataformas operacionais e distribuídos através de diferentes países. Alguns desses componentes podem ser produzidos para uso próprio e outros podem ser adquiridos de terceiros, mais conhecidos como *COTS (Commercial off-the-shelf Components)*, componentes comerciais prontos para serem reutilizados, cujos códigos-fonte normalmente não se encontram disponíveis para os desenvolvedores da aplicação.

Nem sempre o termo componente é utilizado com significados coincidentes. Freqüentemente encontramos referências associadas aos mais diversos conceitos, tais como: unidade de projeto, item de configuração num sistema de gestão de configurações, ou porção de código-fonte passível de ser reutilizado. No contexto deste trabalho, a definição proposta em [Szyperski, 1998], parece ser a mais adequada: "Um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas. Um componente de software pode ser distribuído independentemente e

está sujeito a composição com outras partes".

As características apresentadas nos SBCs podem ajudar a produzir software rapidamente e com um alto nível de qualidade. Entretanto, além dos problemas enfrentados no paradigma de desenvolvimento tradicional, elas também introduzem novos problemas para realização dos testes, tais como:

- A ausência do código-fonte dos componentes
  Muitos componentes são fornecidos sem os seus códigos-fonte.

- A heterogeneidade dos componentes
  Diferentes componentes podem ser produzidos em diferentes linguagens de programação e por diferentes plataformas de hardware.

- Descrições informais dos serviços providos
  Os serviços providos pelos componentes, na grande maioria dos casos, são dados através de descrições informais juntamente com a assinatura dos métodos que invocam esses serviços. Por não serem formais, essas descrições dificultam a automatização dos testes.

- Especificação com informações insuficientes
  As especificações normalmente não provêem informações suficientes sobre a compatibilidade de suas interfaces e sobre as possíveis interações do componente, conseqüentemente restringindo a verificação das propriedades dos sistemas como um todo.

- Dificuldade em identificar os relacionamentos existentes entre os componentes e em gerenciá-los.
  Ainda não se tem respostas concretas para algumas questões: i) como o comportamento do componente pode afetar ou não outros componentes e/ou o sistema como um todo; ii) quais os fatores que determinam cooperação e dependência entre componentes; iii) sobre que circunstâncias podem surgir problemas nesses relacionamentos. Por essas e outras questões, torna-se difícil identificar e gerenciar as conexões existentes entre os componentes que constituem o sistema.

Diante das dificuldades apresentadas anteriormente, é notório que para obter sucesso no reuso dos componentes, é necessário testar as funcionalidades do componente tanto por parte

do fornecedor quanto do cliente. O fornecedor deverá testar o componente com propósitos gerais, ou seja, independente do contexto onde ele será inserido. Já o cliente deverá testar o componente dentro de um contexto específico, onde ele será utilizado. Teste é uma atividade que pode fornecer fortes indicadores sobre a qualidade e confiabilidade do produto.

Com o intuito de testar SBC, pesquisadores e desenvolvedores podem assumir que os componentes que irão constituir o sistema se encontram totalmente testados pelos fornecedores. Porém, ao enquadrá-los em um novo contexto, resultados inesperados podem vir a ocorrer. Uma integração imperfeita pode resultar em despesas significantes, por isso, qualquer redução das falhas ocorridas durante a junção dos componentes pode ser bastante lucrativa. Assim sendo, a integração adequada dos componentes é a chave para o sucesso de um SBC.

1.2 Trabalhos Relacionados

Nos últimos anos algumas abordagens e técnicas de teste de integração têm sido desenvolvidas no intuito de contribuir para uma melhoria da qualidade dos sistemas que vêm sendo produzidos. A seguir, é apresentada uma visão geral de alguns trabalhos relacionados com algumas dessas técnicas.

Como UML é uma linguagem padrão para análise e design orientado a objetos, além de ser facilmente aceita e entendida pela grande maioria dos usuários dessa tecnologia, as técnicas de teste baseadas em UML têm se tornado bastante atrativas dentro da comunidade de engenharia de software. Os trabalhos apresentados em [Hanh et al., 2001], [Traon et al., 1999] e [Traon et al., 2000] são exemplos de técnicas de teste baseadas em UML. O primeiro trata de uma comparação empírica do desempenho dos algoritmos demonstrados em cada estratégia de integração descrita. Os dois últimos apresentam uma metodologia para planejar o teste de integração a partir de um modelo UML, o qual consiste na geração de um grafo que representa as dependências existentes no sistema e serve como base para que as classes e métodos sejam testados em uma seqüência que minimize os esforços gastos para realizar o teste de integração.

Existem algumas técnicas de teste integração focadas em teste de caixa-branca. Porém, não são muito atrativas devido à incerteza da presença do código-fonte dos componentes.

Contudo vale ressaltar algumas delas, como as apresentadas em [Jin and Offutt, 1998] e [Tsai et al., 2001]. A primeira baseia-se no acoplamento dos componentes. Consiste em testar os valores esperados de uma variável antes e depois de uma interação entre componentes. Já a segunda, baseia-se na construção de *thin threads*, que são estruturas representadas em forma de árvore e consistem na representação de um cenário completo, do ponto de vista de um usuário final. Essas estruturas podem ser identificadas tanto através da estrutura do sistema (caixa-branca) quanto das funcionalidades do sistema (caixa-preta).

Algumas técnicas também são desenvolvidas no intuito de se testar os componentes individualmente, favorecendo o fornecedor do componente, como é o caso do trabalho apresentado em [Vieira et al., 2001] que propõe uma linguagem de especificação para descrever os serviços providos pelos componentes. Em [Farias, 2003] é proposto um método de teste para componentes que testa o componente independente do contexto onde o mesmo estará inserido. Este método se destaca dos outros por uma série de vantagens, tais como: faz uso de artefatos UML, que é uma linguagem de especificação bastante utilizada, possui potencial para automação, uma vez que faz uso de uma linguagem formal de especificação (OCL[1]), está atrelado a um processo de desenvolvimento, o qual é uma adaptação de Componentes UML.

O desenvolvimento de ferramentas que ajudam a automação dos testes também é um ponto de suma importância no ambiente da engenharia de software. O trabalho apresentado em [Dellarocas, 1997] retrata a criação do Shyntesis, um ambiente de desenvolvimento que tem o objetivo de minimizar o esforço manual requerido para integrar componentes desenvolvidos independentemente em novas aplicações. Uma outra abordagem interessante é apresentada em [Harrold and Soffa, 1991], trata de um técnica de teste de integração implementada para programas desenvolvidos em Pascal. Apesar da técnica ter sido desenvolvida para programas em Pascal, a princípio não há nada que impeça seu uso para testar a integração de componentes. Porém, ainda é preciso por isso em prática e analisar a viabilidade de usá-la em sistemas de grande porte.

Poucas são as abordagens que se preocupam com a possibilidade de automação. Algumas fazem uso de UML, como: [Hanh et al., 2001], [Traon et al., 1999] e [Traon et al., 2000], o

[1]OCL é linguagem formal de especificação que vem sendo cada vez mais utilizada nas especificações de sistemas.

que facilita um pouco por se tratar de uma especificação formal, entretanto nenhuma delas enfatiza o problema da automação como uma preocupação relevante.

Diante das técnicas discutidas acima, pode se concluir que ainda existe muito trabalho a ser realizado para se obter uma técnica adequada e aceitável de teste de integração dentro do atual padrão de desenvolvimento. Apesar de algumas abordagens interessantes já terem sido propostas, a maioria das abordagens de teste de integração de componentes existentes apresentam alguns fatores que dificultam sua utilização na prática: a falta de potencial para automação, a falta de uma padronização na especificação dos componentes oferecidos, a dificuldade de utilizá-las, a falta de contextualização dentro de um processo de desenvolvimento. Além disto, não cobrem todas as etapas de um processo de teste completo. É importante ressaltar que o processo de desenvolvimento é independente do processo de teste, mas é interessante que ocorram em paralelo para uma maior eficiência dos testes. Assim sendo, o método aqui proposto tem o intuito de suprir as dificuldades explicitadas acima de maneira prática e efetiva.

1.3 Objetivos

O trabalho aqui proposto tem o objetivo de fornecer um método com as as seguintes características:

- Ser definido dentro do escopo da metodologia de desenvolvimento Componentes UML[2] adaptado em [Farias, 2003] com a inclusão de um método de teste de componentes isolados.
- Ter potencial para automação, visto que disto dependerá a aplicabilidade do mesmo na prática.
- Fazer uso, se possível, apenas de artefatos produzidos pelo processo de desenvolvimento utilizado. Isto permite que o método tenha um impacto reduzido no total de esforço a ser empreendido no processo como um todo.

[2]Componentes UML é um processo de desenvolvimento proposto por John Cheesman e John Daniels em [Cheesman and Daniels, 2001], que propõe algumas adaptações à linguagem UML a fim de especificar software baseado em componentes de uma forma simples

- Dar suporte à: geração dos casos de teste de integração, através da definição da ordem de integração de componentes; execução dos casos de teste gerados e análise dos resultados obtidos através da execução.

- Fazer uso efetivo de resultados e artefatos de teste gerados no teste isolado de cada componente, considerando a utilização do método proposto em [Farias, 2003].

1.4 Estrutura da Dissertação

O restante deste documento foi organizado da seguinte forma:

Capítulo 2: Testes: Uma visão geral Este capítulo introduz o conceito de teste funcional bem como a sua importância; enfatiza as principais diferenças existentes ao se testar software orientado a objeto, bem como alguns desafios enfrentados por esta tecnologia; apresenta algumas características do teste de software baseado em componente e alguns problemas encontrados ao testar esse tipo de software; retrata a importância do teste de integração, bem como alguns problemas enfrentados pelo mesmo e apresenta ainda uma estratégia de teste de integração.

Capítulo 3: Desenvolvimento e Teste dos Componentes Este capítulo tem como objetivo principal descrever o método de teste, apresentado em [Farias, 2003], que será utilizado para testar os componentes individualmente. Entretanto, para realizar os testes dos componentes, é necessário que os mesmos sejam desenvolvidos e testados ao longo do processo de desenvolvimento. Por isso, será apresentado também o processo de desenvolvimento Componentes UML, com algumas adaptações realizadas, o qual está intimamente atrelado ao método de teste.

Capítulo 4: Método de Teste de Integração Neste capítulo será descrito o método de teste de integração para sistemas baseados em componentes proposto neste trabalho. O método será explanado e contextualizado dentro do processo de desenvolvimento especificado no capítulo anterior. As atividades de teste propostas no método serão descritas seguindo um processo tradicional de teste de integração.

Capítulo 5: Estudo de Caso - Desenvolvimento da Aplicação No Capítulo 5 será explanado como foi aplicado o processo de desenvolvimento no estudo de caso realizado, uma vez que o processo de teste está atrelado ao mesmo, relatando as dificuldades encontradas, bem como os pontos positivos.

Capítulo 6: Estudo de Caso - Aplicação do Método de Teste de Integração O Capítulo 6 apresentará a aplicação do método de teste de integração proposto no estudo de caso realizado, demonstrando as principais vantagens e desvantagens encontradas.

Capítulo 7: Considerações Finais Baseado nos resultados obtidos, o último capítulo está destinado a apresentação das conclusões finais, relatando as principais contribuições perspectivas de trabalhos futuros.

Apêndice A No Apêndice A, são apresentados os Casos de Uso do estudo de caso realizado.

Apêndice B No Apêndice B, são apresentados os Modelos de Informação construídos para o estudo de caso realizado.

Apêndice C No Apêndice C, encontram-se os Diagramas de Seqüência referente ao Componente Jogo do estudo de caso realizado.

Apêndice D No Apêndice D, encontram-se os Diagramas de Seqüência referente ao Componente Regras do estudo de caso realizado.

Apêndice E No Apêndice E, são apresentadas as Expressões Regulares construídas no estudo de caso, bem como suas condições de execução expressas em OCL.

Apêndice F No Apêndice F, estão as Especificações das Operações das Interfaces dos componentes referente ao estudo de caso.

Capítulo 2

Testes: Uma visão geral

O objetivo deste capítulo é apresentar de forma sucinta conceitos e estado da arte das principais áreas de pesquisa que estão envolvidas neste trabalho. A Seção 2.1 introduz o conceito de teste funcional bem como a sua importância. A Seção 2.2 enfatiza as principais diferenças existentes ao se testar software orientado a objeto, bem como alguns desafios enfrentados por esta tecnologia. A Seção 2.3 apresenta algumas características do teste de software baseado em componente e alguns problemas encontrados ao testar esse tipo de software. A Seção 2.4 retrata a importância do teste de integração, bem como alguns problemas enfrentados pelo mesmo. A Seção 2.5 detalha a estratégia de integração apresentada em [Traon et al., 2000]. Por fim, a Seção 2.6 apresenta as conclusões.

2.1 Teste Funcional

O processo de teste de software hoje, infelizmente, ainda é visto por alguns, como um processo que tem início ao final do processo de desenvolvimento. Nesses casos, os problemas enfrentados em estágios anteriores têm sido resolvidos fazendo-se uso do tempo e dinheiro reservados para teste, não sobrando recursos suficiente para se testar adequadamente o software. Dessa forma, o teste se torna bastante curto e os sistemas produzidos tornam-se não confiáveis [Jeff, 1995].

Quanto mais tarde o teste for realizado, ou seja, quanto mais tarde os erros forem detectados, mais difícil e mais dispendioso eles se tornarão para serem reparados. Isto acontece porque além dos recursos necessários para corrigir os erros originais, serão necessários re-

cursos adicionais para depurar e corrigir os erros que têm sido propagados, pelo erro original, para os estágios subseqüentes.

Ao contrário do teste estrutural, no teste funcional, também conhecido como teste de caixa-preta ou teste baseado na especificação, os casos de teste são construídos baseados apenas na especificação do software e não em como ele é implementado [Beizer, 1999]. Sendo assim, o teste funcional se apresenta como sendo de suma importância no processo de desenvolvimento por permitir que o processo de teste seja iniciado cedo, antes mesmo que seja realizada qualquer implementação.

O teste funcional também é de extrema importância em SBCs para o desenvolvimento dos componentes. Através dele, o fornecedor do componente pode ser capaz de garantir que as propriedades especificadas na interface sejam capazes de se adequar a qualquer contexto que o componente possa ser inserido. Além disso, o usuário poderá ser capaz de testar o componente dentro da sua aplicação, sem dispor do seu código-fonte [Harrold et al., 1999].

Apesar de existirem inúmeros estudos e técnicas de teste baseados em teste funcional, são poucos aqueles de uso prático. As técnicas existentes devem ser melhoradas e novas técnicas criadas, com o objetivo de possibilitar a automação da geração de casos de teste e dos dados de teste. Além disso, também se sente falta de ferramentas que possam executar esses casos de teste e analisar os resultados obtidos através de tal execução. Esses resultados, de alguma forma, ainda devem ser empacotados junto ao componente para que o usuário possa vê-los a fim de poder testar melhor sua aplicação.

2.2 Teste de Software Orientado a Objetos

Não são apenas mudanças na linguagem de programação que afetam o teste, mas também mudanças no processo de desenvolvimento e mudanças no foco da análise e *design*. Uma das principais diferenças em se desenvolver e testar software orientado a objeto, está no modo como ele é designado, como um conjunto de objetos que essencialmente modelam um problema e colaboram entre si, a fim de efetivar uma solução. Um programa no qual o seu *design* é estruturado a partir de um problema, e não em uma solução imediatamente requerida, será mais adaptável a futuras mudanças. Um grande benefício desta abordagem é que o modelo de análise ajuda a gerar o modelo de *design* e, o modelo de *design* ajuda

na geração do código. Sendo assim, pode-se começar a testar na fase de análise e refinar os testes realizados na análise para testar o *design*[1]. Da mesma forma, os testes realizados na fase de *design* podem ser refinados para gerar o teste de implementação. Dessa forma, pode-se perceber que o processo de teste deve estar interligado ao processo de desenvolvimento [McGregor and Sykes, 2001].

Testar os modelos de análise e *design* pode trazer algumas vantagens, como por exemplo:

- Casos de teste podem ser identificados cedo, à medida em que os requisitos estão sendo determinados, o que ajuda os analistas e projetistas a entender melhor os requisitos do sistema e conseqüentemente expressá-los de forma mais correta.

- Falhas podem ser identificadas cedo no processo de desenvolvimento, economizando tempo, custos e esforço.

- Os casos de teste podem ser revisados o quanto antes para obter uma melhor corretude do sistema, o que ajuda os testadores e os desenvolvedores a ter um entendimento mais consistente dos requisitos do sistema.

Como já vimos anteriormente, é de suma importância que o teste seja realizado o mais cedo possível, a fim de evitar que os erros se propaguem para estágios posteriores e, dessa forma, amortizar os custos e o tempo envolvidos no processo de desenvolvimento. Por essas razões, várias pesquisas como as descritas em [Jeff, 1995] e [Briand and Labiche, 2001] têm defendido a idéia de integrar o processo de teste ao processo de desenvolvimento do software orientado a objetos, isto é, fazer com que estas duas etapas ocorram paralelamente e não uma após a outra. Em [Jeff, 1995], é proposta uma abordagem que sugere uma seqüência de atividades a serem seguidas ao longo processo do desenvolvimento. Porém, não foi divulgado nenhum estudo de caso que colocasse em prática tal técnica. Já em [Briand and Labiche, 2001], é proposta uma abordagem baseada em UML para testar sistemas orientado a objetos que consiste em derivar requisitos de teste a partir de artefatos produzidos no final da fase de análise, como o Diagrama de Caso de Uso, descrição dos Casos de Uso, Diagramas de Interação (seqüência e colaboração) associados com cada Caso de Uso e o Diagrama de Classe. Tais requisitos de teste podem ser usados para gerar casos de teste. Outra

[1]Testes de modelos de análise e *design* são normalmente feitos usando inspeções e revisões guiadas por casos de teste funcionais.

grande vantagem desta metodologia é que ela já possui algoritmos prontos para a realização de algumas de suas etapas, além de fazer uso da linguagem OCL para contratos e condições de guarda, o que facilita bastante para uma posterior automação do teste, que é um critério muito importante a se considerar, já que grandes sistemas são muito complexos para se testar e, requerem uma boa estratégia de teste para suportar a enorme quantidade de casos de teste que são gerados.

"Testar cedo, testar sempre, testar o suficiente [McGregor and Sykes, 2001]!". Esta frase resume os conceitos mais importantes que abrangem o mundo do teste. Com base nessa idéia sabe-se que, testar os modelos propostos na fase de análise e *design* não apenas ajuda a descobrir problemas cedo (quando são mais fáceis e mais barato de consertá-los), como ajuda a se ter noção do tamanho do esforço necessário para testar o que realmente precisa ser testado e, conseqüentemente, planejar os recursos relevantes durante todo o processo de desenvolvimento.

2.3 Teste de Software Baseados em Componentes

Teste de sistemas baseados em componentes tem sido investigado com base no uso de diferentes notações para modelagem e considerando a integração dentro de processos de desenvolvimento, como defendido em: [Wu et al., 2001], [Martins et al., 2001], citeGao03, [Kim and Carlson, 2001]. Na maioria dos casos, diagramas de estado são adotados. Nesta seção, focalizaremos em trabalhos relacionados ao uso de UML.

Dentre as várias técnicas propostas nos últimos anos, para se testar software baseado em componentes e, que envolve o teste do componente no ambiente onde ele será inserido, pode-se destacar a técnica proposta em [Beydeda and Gruhn, 2001]. Trata-se de uma técnica onde os componentes são especificados através de máquinas especiais de estado finito, também chamadas de *Component State Machine*, as quais possuem transições que utilizam código Java para permitir uma geração automática de protótipos executáveis. Uma vez gerados tais protótipos, é produzida uma representação gráfica, chamada de *Component-Based Software Flow Graph*, através da qual é possível visualizar informações provindas tanto da especificação quanto da implementação. Após ter em mãos esta representação gráfica, pode-se fazer uso de técnicas bem conhecidas de teste estrutural para identificar os casos

de teste. A abordagem em questão trata do teste de integração de componentes do ponto de vista do desenvolvedor de aplicações, não se fazendo necessário a utilização do código-fonte do componente e, fazendo-se uso de um subconjunto de funcionalidade providas pelo componente. Esta abordagem poderia ser ainda mais eficiente se os casos de teste fossem gerados automaticamente a partir do grafo construído, podendo também ser utilizada de uma maneira mais específica, obtendo casos de teste que cobrissem determinados caminhos e, ainda poderia ser também estendida para outros tipos de teste como o teste de regressão por exemplo.

Em [Farias, 2003], é proposto um método de teste funcional aplicável a componentes de software. Já que o método de teste proposto é intimamente dependente do processo de desenvolvimento, o mesmo se apóia na metodologia de desenvolvimento Componentes UML [Cheesman and Daniels, 2001], a qual segue os seguintes passos: Definição de Requisitos, Modelagem de Componentes, Materialização de Componentes, Montagem da Aplicação, Testes, Entrega da Aplicação. O método se torna interessante por permitir que o componente possa ser testado com relação à sua especificação, ou seja, sem ter acesso ao seu código-fonte. Esta especificação é definida através da linguagem OCL, o que abre caminhos para uma posterior automação das atividades de teste. Além disso, os artefatos de teste são exportados junto com os componentes a fim de facilitar o teste do mesmo por parte de clientes.

Uma outra técnica encontra-se apresentada em [Hartmann et al., 2000]. A técnica propõe a geração e execução dos casos de teste de forma automática para sistemas baseados em componentes COM/DCOM e CORBA e especificados utilizando UML. A técnica impõe que o comportamento dinâmico dos componentes bem como a comunicação entre eles seja descrito através de diagramas de estado UML. O trabalho também propõe uma notação própria para rotular as transições dos diagramas de estados, uma vez que a notação UML atual não contempla o aspecto de comunicação entre os componentes.

A técnica propõe que sejam construídos diagramas de estados individuais de cada componente, os quais são vistos como máquinas de estados finitas Mealy, onde as saídas produzidas em função da ocorrência de um evento são associadas à transição e não ao estado [Beizer, 1999]. A partir desses diagramas é construído um modelo de comportamento global do sistema, representado por uma máquina de estado composta.

Assim como outras técnicas, essa também possui suas vantagens e desvantagens. Apesar

de ter a vantagem de possuir um suporte ferramental, exige que sejam construídas máquinas de estados para representar o comportamento global do sistema. Isso não é bom, uma vez que, normalmente, na modelagem não são construídas essas especificações. O ideal seria utilizar modelos que já tivessem sidos desenvolvidos durante a especificação do sistema. Outra desvantagem é que a técnica não se encontra integrada a nenhum processo de desenvolvimento.

2.4 Teste de Integração

Num sistema baseado em componentes, os mesmos trocam informações com o objetivo de prover as funcionalidades do sistema. Geralmente os componentes são construídos com a finalidade de oferecer serviços mais abstratos em um sistema de forma a facilitar possíveis modificações futuras e reutilizações do mesmo serviço. Esta composição cria interações que promovem dependências entre os componentes. Assumindo que os componentes encontram-se previamente testados, cabe agora se preocupar com a integração deles, de modo a garantir que o sistema funcione como esperado.

Teste de integração será definido como sendo o modo que o teste é conduzido para integrar componentes em um sistema. Refere-se ao teste das interações entre unidades e módulos para garantir que eles possuam considerações consistentes e se comunicam corretamente [Jin and Offutt, 1998], sendo responsável por verificar se todas as partes de uma aplicação, ao serem juntas, funcionam como esperado. Existem duas estratégias clássicas de teste de integração, são elas: "bottom-up"e "top-down". A abordagem *bottom-up* consiste em testar os módulos em um baixo nível e continuar os testes seguindo uma hierarquia de módulos até que o módulo de mais alto nível seja testado. Já na abordagem *top-down*, ocorre exatamente o contrário, o sistema é testado do alto nível para um baixo nível. O programa é representado como um único componente abstrato com sub-componentes representados por *stubs*. Um *stub* tem como objetivo simular o comportamento real de um componente que ainda não está integrado no sistema. *Stubs* possuem a mesma interface dos componentes, porém, com funcionalidades limitadas. Após o componente de alto nível ter sido testado, seus sub-componentes são implementados e testados da mesma maneira. O processo se repete até que todos os componentes de baixo-nível tenham sido implementados. Desta forma todo o

sistema será completamente testado. Dependendo de uma série de fatores como: o tipo de software que está sendo produzido, os recursos disponíveis, dentre outros, pode se utilizar uma ou outra abordagem, ou ainda, como é muito utilizado, uma mistura das duas abordagens.

Quando qualquer modificação tiver de ser feita no componente, ou em sua interface, poderão surgir algumas dificuldades significantes, já que na grande maioria das vezes o cliente do componente não tem acesso ao seu código-fonte e, por conseguinte, não entende as consequências das mudanças propostas. Por isso, já que o desenvolvimento de SBC vem se tornando padrão, novas técnicas devem ser produzidas para se testar software desse tipo.

Segundo [Hanh et al., 2001], as principais dificuldades para se realizar uma integração de componentes a um custo/benefício eficiente são:

- a minimização do número de *stubs* a serem escritos, já que um *stub* é usado apenas para simular o comportamento de um componente real que ainda não foi integrado (custo);
- o número de passos necessários para realizar a integração (duração).

O paradigma da engenharia de software baseado em componentes ainda apresenta alguns problemas mais complexos do que o paradigma do desenvolvimento tradicional. Muitos desses problemas são resultantes da heterogeneidade dos componentes, ou seja, pelo fato de que podem ser desenvolvidos por diferentes pessoas, em diferentes linguagens de programação e em diferentes plataformas de hardware e, das restritas informações que são fornecidas junto ao componente, já que, geralmente, são distribuídos como uma caixa-preta.

Várias tecnologias como CORBA da OMG, COM/COM+ da Microsoft e Enterprise Java Beans da Sun suportam o desenvolvimento baseado em componentes. A maior parte das tecnologias existentes possuem mecanismos para resolver alguns problemas relacionados com a heterogeneidade, permitindo uma melhor interoperabilidade. Entretanto, não suportam uma descrição precisa do componente. No geral, os componentes são fornecidos com uma descrição informal dos serviços providos, juntamente com a assinatura dos métodos que invocam esses serviços. Essas especificações normalmente não provêem informações suficientes para uma boa interação do componente com o sistema e com outros componentes.

Componentes são expressos em termos de interfaces visíveis, sem implementação, ou seja, o cliente sabe *o que* o componente faz mas não *como* ele o faz. Uma interface define quais serviços um componente provê e requer para ser utilizado em um sistema.

Na grande maioria dos casos, as interfaces são descritas através da assinatura de suas operações. Existem algumas IDL's (*Interface Definition Language*) como CORBA IDL, que são usadas para descrever algumas interfaces de componentes. Contudo, elas provêem suporte apenas para descrever assinaturas dos nomes e tipos dos atributos e das operações dos componentes, ignorando aspectos relacionados com o comportamento. Os aspectos comportamentais do componente são importantes para analisar as propriedades do sistema de uma forma genérica e, para realização das atividades de teste de integração. Algumas abordagens tentam especificar o comportamento do componente através de pré e pós-condições. Pré-condições expressam as propriedades que devem ser mantidas sempre que uma operação for chamada. Já as pós-condições descrevem as propriedades que devem ser garantidas após uma determinada operação ter sido executada. Caso haja alguma falha, há algum problema com o componente.

Normalmente a especificação de uma interface não estipula o que pode acontecer depois que um de seus métodos são invocados. Por isso, com o intuito de se ter uma documentação de alto nível relacionada com as dependências dos componentes Marlon Vieira, Márcio Dias e Debra Richardson [Vieira et al., 2001] propuseram uma linguagem de especificação para descrever os serviços providos pelos componentes. A linguagem cria um nova descrição ao invés de simplesmente estender uma IDL. Isso traz um efeito positivo à medida que permite dar suporte para um extração automática desta descrição a partir do código-fonte, além de manter uma abordagem mais genérica, independente de alguma IDL específica, permitindo uma maior compatibilidade com diferentes tecnologias. Ao usar esta descrição provida pelos diversos componentes que compõem o sistema, pode se construir um grafo de dependência mostrando os componentes e suas ações. Isso facilitaria o entendimento das interações de cada componente com o sistema e com outros componentes, ajudando o desenvolvedor a determinar se uma mudança em um determinado componente pode afetar outros componentes e a escolher quais componentes precisam ser re-testados após uma determinada funcionalidade ser modificada.

Zhenyi Jin e Jefferson Offutt, em [Jin and Offutt, 1998], desenvolveram uma técnica de

teste de integração baseada no acoplamento dos componentes que constituem o software. A técnica apresenta alguns critérios de cobertura para três tipos de acoplamento e descreve uma análise que, através de algoritmos, mede a cobertura dos testes. Para uma maior utilidade da mesma, poderia-se desenvolver uma ferramenta de análise de cobertura, aproveitando para isto, os algoritmos que foram apresentados. Além disso, seria bastante interessante a automação da geração de alguns dados de teste para a técnica baseada em acoplamento.

A forma de integração/montagem de componentes, por exemplo, pode influenciar os tipos de teste a serem feitos. Por isto, é tão importante considerar o processo de teste associado ao processo de desenvolvimento. Uma estratégia ruim pode levar a construção de muitos *stubs* ou à combinação de componentes não testados previamente, dificultando o processo de localização de erros.

2.5 Estratégia de Teste de Integração

O trabalho apresentado em [Traon et al., 2000], apresenta uma estratégia de teste de integração aplicada a sistemas orientados a objetos. A estratégia consiste na construção de um grafo, gerado a partir de diagramas de classe UML, que representa as dependências de teste existentes entre as classes e métodos que compõem o sistema. Este grafo, também conhecido como Grafo de Teste de Integração (GTD), serve como base para ordenar as classes e métodos a serem testados com o propósito de integração, ou seja, visando a minimização da construção de *stubs*. Diante da complexidade em otimizar a redução do número de *stubs*, um algoritmo é aplicado ao grafo, fornecendo uma eficiente ordem de teste para otimizar esta redução. Dessa forma, a técnica pode ser aplicada com base nas especificações normalmente produzidas na modelagem do sistema, não havendo necessidade de se desprender um esforço adicional para construção de especificações com notações pouco utilizadas na prática. Além disso, para geração da ordem de integração é aplicado um algoritmo ao grafo gerado, o que possibilita a automação da estratégia.

O GTD é um grafo cujos vértices representam classes e/ou métodos inclusos (dependendo do nível de detalhe do *design*) e os arcos representam dependências de teste. O mesmo é gerado a partir de algumas regras aplicadas ao diagrama de classe, onde classes e métodos são transformados em nodos, como se pode observar na Figura 2.1. Com o intuito de se

entender o processo de geração do GTD, faz necessário a compreensão dos seguintes tipos de dependências:

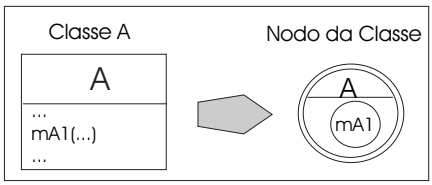

Figura 2.1: Modelagem dos nodos

- Classe - Classe: Representa a dependência existente entre classes. Cada classe é modelada por um nodo em um GTD e interligadas por arcos. A Figura 2.2 mostra uma modelagem de uma dependência Classe - Classe.

- Método - Classe: Representa a dependência existente entre um método e uma classe. Ocorre, por exemplo, quando um determinado método *m* possui um objeto de uma classe *A* como parâmetro de entrada em sua assinatura. Na Figura 2.3 pode se visualizar este tipo de dependência.

- Método - Método: Pode ocorrer quando um determinado método *m1* de uma certa classe *A* usa um método *m2* de uma determinada classe *B* através de um objeto da classe *B*. Isto pode ser visualizado na Figura 2.3.

Após criado o GTD, deve se aplicar algumas regras de normalização a fim de transformar o GTD preliminar em uma representação que seja possível a aplicação de uma estratégia de integração baseada em algoritmos clássicos de teste. A aplicação de uma estratégia de integração tem como objetivo facilitar o entendimento das relações existentes entre as entidades do sistema e estabelecer uma ordem de integração baseada na redução do número de *stubs* a serem construídos. A normalização consiste em separar os nodos que representam os métodos dos nodos de classes, nos quais eles estavam incluídos na primeira representação do grafo. Na Figura 2.4 pode se observar um exemplo de um GTD preliminar e após aplicado a normalização.

O algoritmo que será utilizado como estratégia de integração encontra-se descrito em [Traon et al., 2000] e, é similar ao algoritmo de Busca em Profundidade, que é bastante conhecido pela comunidade de Engenharia de Software.

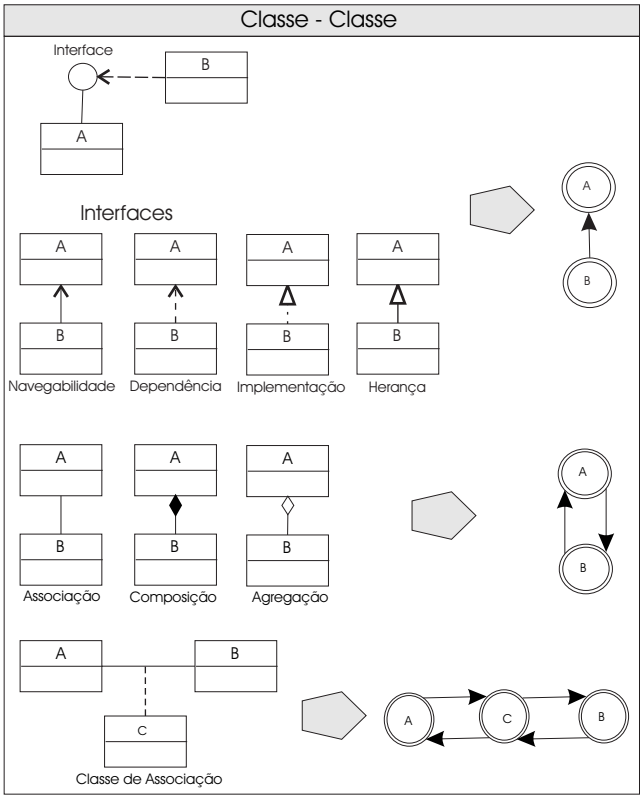

Figura 2.2: Modelagem da dependência Classe para Classe

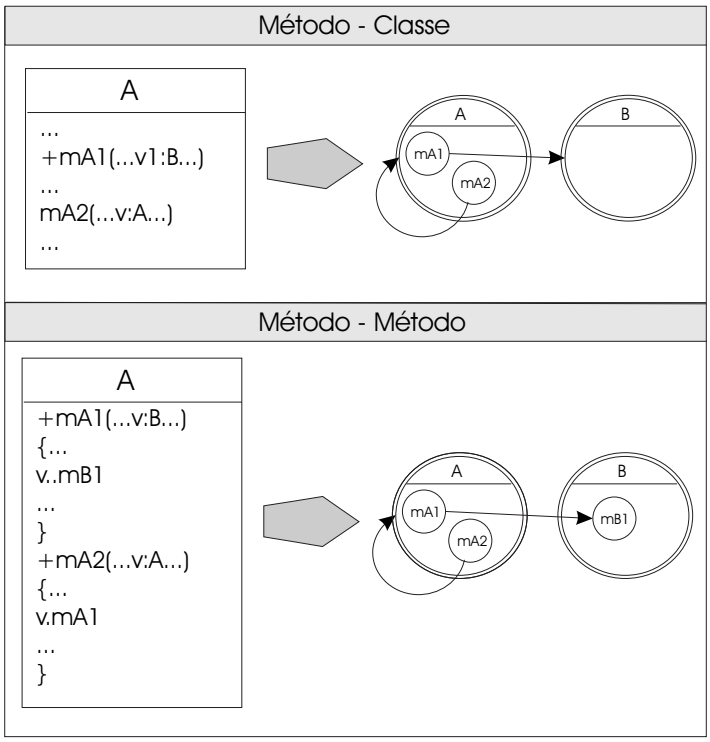

Figura 2.3: Tipos de Dependência

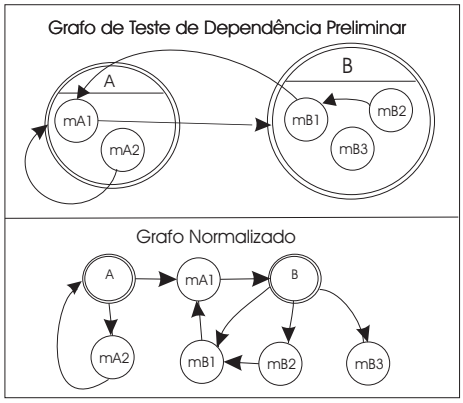

Figura 2.4: GTD preliminar e regra de normalização

O algoritmo consiste em numerar os nodos como vértices e em definir os tipos de arestas (ligações) de acordo com sua primeira visita ao grafo. Essas arestas podem ser classificadas como: *Tree, Frond, Cross* e *Forward*. Como pode se observar na Figura 2.5: as arestas *Tree*, são as que partem de um vértice para um vértice ainda não visitado, como por exemplo a aresta que vai de "a"para "b"; as arestas do tipo *Frond* partem de um descendente para um ancestral, como por exemplo a aresta que vai de "d"para "b"; as arestas do tipo *Cross* partem de um vértice para diferentes ramificações, como por exemplo a aresta que vai de "h", que vem da ramificação "a","f","g","h", para "e", que vem da ramificação "a","b","c","e"; e as arestas do tipo *Forward* que partem de um vértice para um descendente que já foi visitado e faz parte da mesma ramificação, como por exemplo a aresta que vai de "b"para "e".

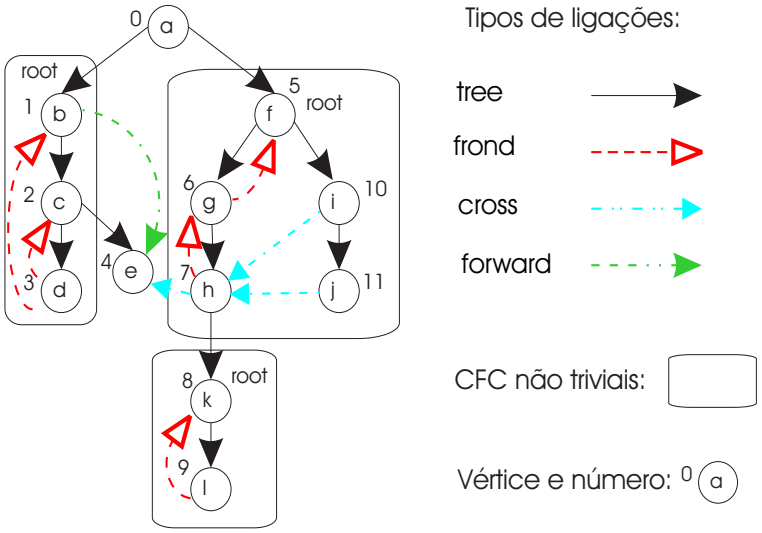

Figura 2.5: CFC e as partições

No caso mais simples onde o GTD é acíclico, para o propósito de teste de integração, a estratégia natural é testar os nodos (classes e métodos), dos seus descendentes para os seus ancestrais, no grafo. Contudo, nos casos mais gerais, o GTD pode conter ciclos de dependências (loops). É exatamente nesses ciclos onde se encontra um dos maiores problemas do teste de integração. Diante destes ciclos, fica difícil decidir qual das entidades que constituem o ciclo, deve ser testada primeiro, já que uma sempre tem dependência com outra.

Visando sempre a minimização do número de *stubs* a serem construídos, a idéia é que um *stub* deva quebrar o maior número de ciclos possíveis. Como os arcos do tipo Frond partem de um descendente para um ancestral, eles representam sempre um ciclo. Portanto, o vértice que possui o maior número de entradas e saídas de arcos do tipo frond é o vértice que possui mais ciclos. Se um conjunto de componentes está incluido em um ciclo de dependências pode-se dizer que esses componentes pertencem ao mesmo Componente Fortemente Conectado ou CFC. Com base nesta idéia, para cada CFC, é escolhido um vértice, onde são deletadas todas as arestas que nele chegam. A partir deste vértice são aplicadas chamadas recursivas do algoritmo. Este rocedimento deve seguido até que não existam mais ciclos.

A aplicação do algoritmo no exemplo da Figura 2.5 é detalhado na Figura 2.6. A primeira chamada identifica 3 CFC's. No CFC $\{b,c,d\}$ o vértice selecionado é d. Numa chamada recursiva c $\rightarrow$ d é deletado e o grafo torna-se acíclico. Em $\{f,g,h,i,j\}$ o vértice g é selecionado, f $\rightarrow$ g e h $\rightarrow$ g são deletados, e o grafo torna-se acíclico. Em $\{k,l\}$, l $\rightarrow$ k é deletado e o grafo também torna-se acíclico. Isto resulta no lado direito da Figura 2.6. Em termos de teste, a seguinte estratégia é fornecida:

- l é testado usando *stub(k)*,
- k é testado,
- e é testado.

Para o CFC b,c,d,

- c é testado usando *stub(d)* e e,
- b é testado usando c e e,
- d é testado usando c.

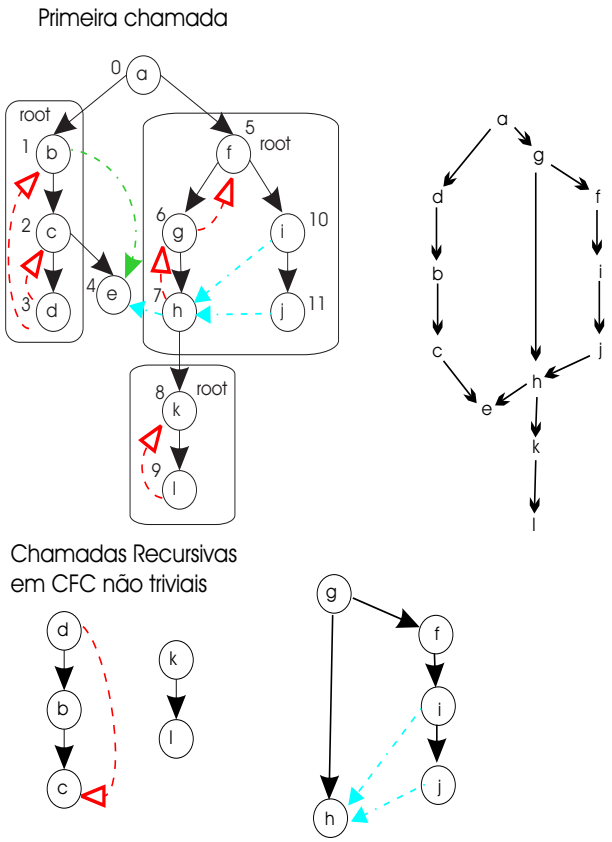

Figura 2.6: Chamadas Recursivas e CFCs não triviais

Para o CFC f,g,h,i,j,

- *h* é testado primeiramente usando *stub(g)*, *e*, e *k*,
- *j* é testado usando *h*,
- *i* é testado usando *h*,
- *f* é testado usando *stub(g)* e *i*,
- *g* é testado usando *h* e *f*.

Por fim:

- *a* é testado usando *b* e *f*.

A estratégia de integração apresentada, mostra os relacionamentos existentes entre as classes e métodos do sistema, o que facilita o entendimento da integração entre estas entidades. Além disso, determina uma ordem de integração baseada na redução da construção de *stubs*. Isso faz com que os recursos utilizados no sistema sejam otimizados.

2.6 Conclusão

Este capítulo apresentou uma visão geral sobre alguns conceitos relevantes e o estado da arte das principais áreas de foco deste trabalho, com destaque no teste de integração e alguns problemas enfrentados neste tipo de teste. Além disso, foi ilustrada uma estratégia de integração para sistemas orientados a objetos, apresentada em [Traon et al., 2000].

Um bom método de teste de integração deve possuir algumas características importantes que o diferencia dos outros, tais como:

- identificar casos de teste cedo, o que pode ser feito através de modelos de análise e *design*, ajudando a entender e expressar melhor os requisitos do sistema;
- identificar falhas o quanto antes, ainda no processo de desenvolvimento, economizando tempo, custo e esforço;
- possuir um potencial para para automação, podendo ser através de especificações formais, podendo ter mais perspectivas de ser aproveitado na prática;

- possuir alguma forma de representação, que sirva para entender melhor os relacionamentos existentes entre os componentes da aplicação, e gerenciá-los facilmente;
- definir, de forma efetiva, a ordem de integração e realização de testes.

Capítulo 3

Desenvolvimento de SBC e Teste de Componentes

Neste capítulo será apresentado um método de teste de componentes, encontrado em [Farias, 2003], o qual possui um conjunto de atividades que são desenvolvidas dentro do contexto de cada uma das etapas de um processo de teste tradicional. Como todo bom método de teste, este também encontra-se atrelado a um processo de desenvolvimento. Como o foco deste trabalho é voltado para o uso de componentes, o processo de desenvolvimento utilizado para descrição do método é Componentes UML. A Seção 3.1 fornece uma breve introdução do capítulo. Na Seção 3.2, encontram-se descritas as fases do processo de desenvolvimento, e na Seção 3.3 estão descritas as atividades propostas pelo método de teste de componentes. A Seção 3.4 explana uma breve conclusão do capítulo.

3.1 Introdução

Uma das principais finalidades de um componente é dividir um problema em vários pedaços menores, de forma a resolvê-lo de maneira mais elaborada baseando-se em soluções mais simples [Cheesman and Daniels, 2001].

Um processo de desenvolvimento ideal assumiria um mundo estável e que nunca sofresse mudanças, ou seja, todos os requisitos seriam reunidos, todo o sistema seria especificado de forma a alcançar os requisitos, seria feito o design de todas as partes do software, e depois essas partes seriam implementadas e integradas. Teoricamente isto seria perfeito, não

havendo necessidade de modificar o software depois que ele estivesse pronto, uma vez que os custos envolvidos nas modificações costumam ser bastante elevados. Infelizmente, não se vive num mundo imutável. Praticamente todos os dias, o dólar muda de preço, o clima muda, as pessoas inventam uma nova gíria, cientistas fazem novas descobertas, o mundo muda em alguma coisa. Mudanças influenciam nossos hábitos, nossa rotina e nossa forma de pensar e de agir. É praticamente impossível achar que os requisitos de um sistema nunca irão sofrer qualquer tipo de mudança. Por isso, uma das maiores motivações para usar uma abordagem baseada em componentes é poder gerenciar melhor as mudanças que possam ocorrer nos requisitos de uma sistema, ou seja, um componente deve ser facilmente substituído.

O desenvolvimento baseado em componentes possui algumas características importantes que não são muito enfatizadas num processo de desenvolvimento que não seja baseado em componentes [Cheesman and Daniels, 2001]. Como por exemplo:

- Um componente deve se ajustar a um tipo de ambiente padrão, ou seja, deve seguir um conjunto pré-estabelecido de padrões básicos. Enterprise JavaBeans (EJB) e COM+ são exemplos de tais padrões.

- Um componente deve possuir uma clara especificação de suas interfaces, para que possamos saber exatamente o que cada parte faz.

- Deve existir uma separação bastante clara entre a especificação e a implementação do componente. Isto contribui para que o componente possa ser substituído mais facilmente.

É válido ressaltar ainda que, assim como os serviços providos pelo componente são importantes, a informação gerenciada por este componente também é de suma importância. Como por exemplo, se fosse necessário substituir um componente que gerenciasse milhares de contas bancárias por um novo componente que fornecesse os mesmos serviços de gerenciamento de contas, mas não tivesse nenhum conhecimento sobre estas contas, seria extremamente complicado. Por isso, é fundamental se referenciar a componentes, tendo em mente que está se referenciando a serviços e estado do componente juntos, ou seja, um princípio chave do conceito de objeto.

UML é uma linguagem padrão para análise e design orientado a objetos, além de ser facilmente aceita e entendida pela grande maioria dos usuários dessa tecnologia. Porém, in-

felizmente, UML não suporta todos os conceitos de componentes discutidos acima. Por isso, John Cheesman e John Daniels, em [Cheesman and Daniels, 2001], propuseram um processo de desenvolvimento designado como Componentes UML que propõe algumas adaptações à linguagem UML para que se possa especificar software baseado em componentes de uma forma simples e de fácil entendimento.

3.2 Desenvolvimento de SBC

Por fazer uso de abordagem UML, a qual se trata de uma linguagem de especificação padrão e, por procurar entender e representar a relação entre os componentes, Componentes UML foi escolhido como processo de desenvolvimento a ser adotado.

O processo de desenvolvimento proposto em Componentes UML segue uma seqüência de atividades descritas nas seguintes etapas: Requisitos, Especificação, Fornecimento, Montagem, Teste e Distribuição. A fase de Requisitos define os requisitos necessários para gerar entradas para a Especificação, sendo esta a de maior prioridade, já que o intuito do processo de desenvolvimento é especificar software baseado em componentes. Ela se divide em três sub-fases: Identificação do Componente, Interação do Componente e Especificação do Componente. As fases de Fornecimento e Montagem não são muito exploradas em Componentes UML, mas estão presentes apenas para dar idéia de como seria a implementação do sistema. As fases de teste e distribuição também são muito importantes, apesar de não serem a razão principal do processo de desenvolvimento.

A Figura 3.1 mostra as etapas envolvidas no processo de desenvolvimento. A seguir, serão descritas cada uma dessas etapas.

3.2.1 Requisitos

O principal objetivo da fase de requisitos é produzir os artefatos necessários para serem usados na fase seguinte que é a Especificação. Tais artefatos consistem em:

- Modelo de Processo do Negócio
- Modelo Conceitual do Negócio
- Modelos de Casos de Uso

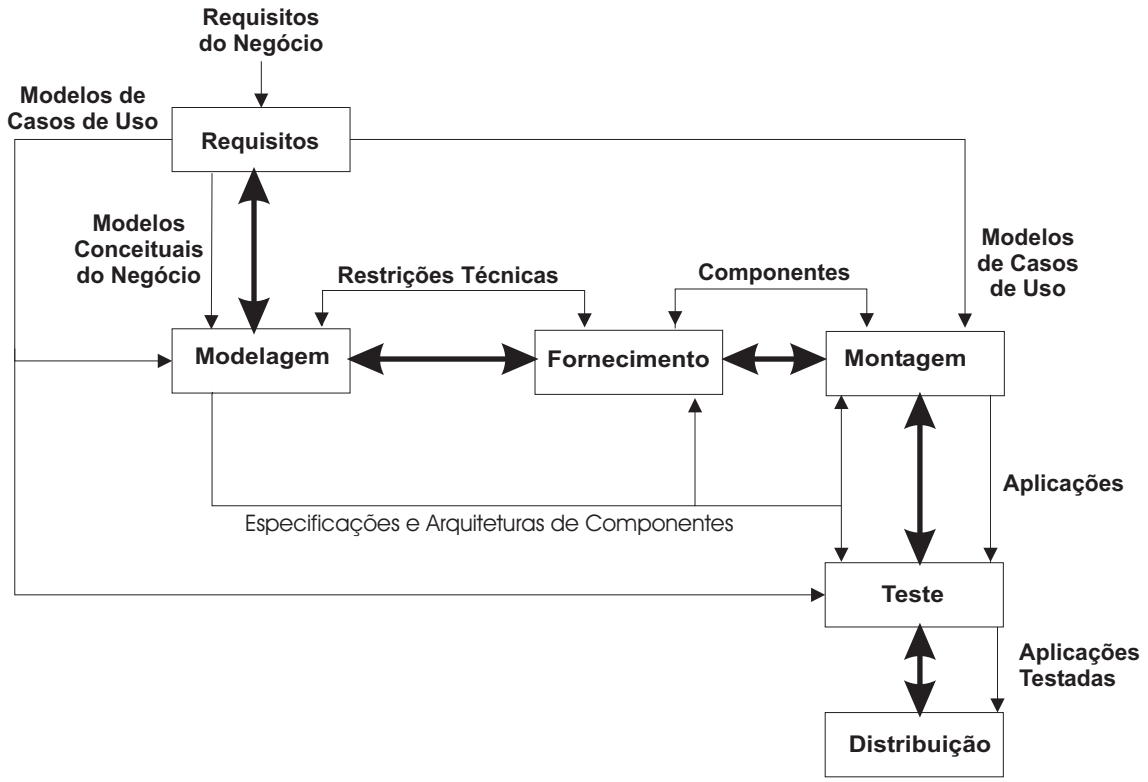

Figura 3.1: Etapas do Processo de Desenvolvimento

Modelo de Processo do Negócio

O Modelo de Processo do Negócio é um diagrama de alto nível que fornece uma visão geral do funcionamento do negócio modelado. Para representar tal diagrama pode se usar a notação do diagrama de atividades UML. Um diagrama de atividades UML é uma espécie de fluxograma que fornece uma rica notação para demonstrar uma seqüência de atividades. Cada bloco no diagrama representa uma atividade e cada transição é disparada automaticamente após a execução da atividade. Segundo proposto em [Farias, 2003], pode-se ainda, produzir um texto para complementar o diagrama, descrevendo em alto nível, quais serão as principais funções do sistema e, como o sistema deverá funcionar do ponto de vista de seus usuários.

Modelo Conceitual do Negócio

O Modelo Conceitual do Negócio é um modelo conceitual da informação existente no domínio do problema. Seu principal propósito é criar um vocabulário comum entre as pessoas que fazem parte do negócio e que estão envolvidas no projeto, capturar conceitos e

identificar relacionamentos. É importante ressaltar que para construir este modelo a concentração deve ser voltada para o problema a ser solucionado e não para a solução do mesmo. Pode ser que existam elementos no modelo que não necessariamente façam parte do sistema a ser desenvolvido. Aos poucos, esses elementos são eliminados do modelo, à medida que o modelo é refinado em outras etapas.

Modelos de Casos de Uso

Os Modelos de Casos de Uso servem para especificar alguns aspectos dos requisitos funcionais do sistema. Descrevem a interação do usuário (ou qualquer ator externo) com o sistema. É uma projeção dos requisitos do sistema, expressa em termos de interações que devem ocorrer através dos limites do sistema. O sistema é visto como uma caixa preta que aceita estímulos de atores (quem inicia o caso de uso) e gera responsabilidades. UML provê o Modelo de Caso de Uso para uma modelagem semi-formal da interação do usuário com o sistema. A mínima estrutura textual apresentada pela abordagem que está sendo descrita para tal modelo é a seguinte:

- um nome ou número identificador;
- o nome do ator iniciante;
- uma rápida descrição do objetivo do Caso de Uso;
- uma simples seqüência numérica de passos que descrevem o cenário principal de sucesso.

O cenário principal pode ser entendido como uma seqüência específica de ações e interações entre atores e o sistema [Larman, 1999]. Ele descreve a situação mais comum do caso de uso, onde tudo acontece como esperado. Contudo, podem existir cenários alternativos, que descrevem outras situações. Os cenários alternativos também devem ser descritos no Caso de Uso e vêm logo após o cenário principal, sendo descritos como extensões deste cenário. Cada cenário alternativo é descrito como extensões, as quais devem conter as seguintes informações:

- O número do passo, no cenário principal, ao qual a extensão se aplica;
- Uma seqüência numerada dos passos que descrevem a extensão.

3.2.2 Modelagem

Esta etapa do processo, como dito anteriormente, é dividida em: Identificação do Componente, Interação do Componente e Especificação do Componente.

Identificação do Componente

Este é o primeiro estágio da fase de Modelagem. O objetivo desta etapa do processo é identificar um conjunto inicial de interfaces do negócio para os componentes do negócio e um conjunto inicial de interfaces do sistema para os componentes do sistema, depois juntar tudo isso para obter uma arquitetura inicial do componente. Para isto, serão utilizados dois dos modelos gerados anteriormente: o Modelo de Caso de Uso e o Modelo Conceitual do Negócio.

Descobrindo Interfaces do Sistema

As interfaces do sistema e suas operações iniciais emergem de algumas considerações feitas no Modelo de Caso de Uso. Inicialmente, para cada Caso de Uso é gerada uma interface do sistema. Em seguida, cada passo é analisado para verificar se existe, ou não, alguma responsabilidade do sistema que deva ser modelada. Caso exista, elas serão representadas como uma ou mais operações do sistema. Alguns casos de uso relacionados podem ser agrupados, originando uma só interface do sistema.

Descobrindo Interfaces do Negócio

O Modelo Conceitual do Negócio é utilizado para ajudar a centralizar as informações associadas ao processo que o sistema irá gerenciar. Este modelo representa uma visão do mundo vista por olhos humanos e, será refinado para dar origem ao Modelo de Tipo do Negócio que representa uma visão do mundo vista pelo sistema. O Modelo de Tipo de Negócio será então usado para desenvolver o conjunto de interfaces do negócio e é representado por um diagrama de classe UML.

O Modelo de Tipo de Negócio contém informações específicas do negócio que devem ser usadas para que o sistema seja especificado. É construído através de uma cópia do Modelo Conceitual do Negócio sendo adicionado ou removido elementos até chegar ao escopo correto. Em seguida, são colocados alguns atributos, se necessário e, regras de multiplici-

dade, bem como alguma especificação OCL, se for o caso. Logo mais, são identificados os *core types*, ou seja, quais informações são independentes de qualquer outra para existir. Para cada *core type* identificado, se cria uma interface de negócio. Ao adicionar essas novas interfaces ao Modelo de Tipo de Negócio, gera-se um novo diagrama chamado de Diagrama de Responsabilidades de Interface.

Neste ponto já se pode pensar em uma arquitetura inicial da especificação do componente. Para cada interface de negócio temos uma especificação diferente do componente. Cada especificação dessa irá fazer parte da especificação do sistema como um todo.

Interação do Componente

Já tendo definido o conjunto de interfaces e componentes que irá se trabalhar, esta etapa decide como os componente irão interagir de forma a fornecer a funcionalidade requerida. Para isto é usado o Diagrama de Colaboração UML.

Com os Casos de Uso foram descobertas as operações das interfaces do sistema. Nesta etapa essas operações são exploradas, desenvolvendo diagramas de colaboração para cada uma dessas operações. A partir desses diagramas são descobertas as operações necessárias, relacionadas com as interfaces de negócio. Ainda não é hora de se deter aos detalhes de como devem ser implementadas essas operações, apenas é identificado um conjunto inicial de operações, e suas assinaturas, que devem ser fornecidas pelas interfaces. Neste ponto, o entendimento sobre as responsabilidades de cada interface é consolidado.

Especificação do Componente

Com o intuito de certificar que cada pedaço de software desenvolvido em tempos diferentes, por diferentes pessoas e possivelmente por diferentes organizações, possam trabalhar juntos com sucesso, é importante que sejam definidos os contratos. Numa definição bem geral, um contrato é um acordo formal entre duas ou mais partes. É importante ressaltar que um contrato não diz como as coisas serão feitas, mas sim o que será feito. Em Componentes UML, são definidos dois tipos de contrato: o Contrato de Uso e o Contrato de Realização. O Contrato de Uso é definido pela especificação da interface, ou seja, descreve o relacionamento entre uma interface do componente e seus clientes. Já o Contrato de Realização é definido pela especificação do componente, ou seja, descreve o relacionamento entre uma

especificação do componente e sua implementação.

Contrato de Uso

Para se obter o contrato de uso, faz-se necessário a especificação da interface que é dada pelos seguintes ítens:

- Operações

  A especificação de operações inclui: parâmetros de entrada, parâmetros de saída, pré e pós-condições. A pós-condição define os efeitos da operação, enquanto que a pré-condição defini as condições nas quais cada pós-condição é garantida. Em UML estas condições contratuais podem ser especificadas usando OCL por se tratar de uma linguagem declarativa e bastante precisa que permite construir expressões lógicas de maneira formal. Em Componentes UML não é proposto, mas será seguida a idéia expressa em [Farias, 2003], de que o nome da operação e a descrição do seu objetivo são informações importantes para melhor compreensão da operação e, por isso, essas informações serão acrescentadas à especificação da operação.

- Modelo de Informação

  É um modelo dos possíveis tipos de estado de um objeto de um componente. Para cada interface temos um Modelo de Informação de Interface. O modelo deve possuir informação suficiente para que cada operação da interface seja especificada.

- Invariantes

  São restrições associadas a um tipo de informação que deve ser verdadeira para todas as instâncias desse tipo.

Contrato de Realização

Um contrato de realização é definido por uma especificação de componente. A especificação do componente deve ser constituída pelas especificações das interfaces oferecidas, e usadas, pelo componente, e pelas interações entre o componente e outros componentes que oferecem seus serviços.

Na fase de Identificação do Componente, é definido um conjunto de interfaces oferecidas e usadas pelo componente. Nesta fase são construídos os diagramas de especificação dos componentes de sistema e de negócio, e a arquitetura inicial do sistema. Ainda nesta mesma etapa do processo, são definidas as interações entre os componentes, as quais são

representadas por meio de diagramas de colaboração. Estes diagramas são construídos com a finalidade de se descobrir as operações fornecidas por cada componente e, de demonstrar como os componentes interagem entre si para entregar alguma funcionalidade do sistema.

O processo de desenvolvimento Componentes UML sugere que sejam construídos novos diagramas de colaboração, contendo todas as interações encontradas, apenas no componente que está sendo especificado, sem levar em consideração as interações existentes entre outros componentes, que também são importantes, mas não fazem parte do contexto do componente sendo especificado. Dessa forma, seguindo o que foi proposto em [Farias, 2003], não serão construídos esses diagramas por representarem um ponto de redundância no projeto, o que poderia causar inconsistência na documentação. Ao invés deles, serão construídos diagramas de seqüência, a fim de representar as interações entre as classes que fazem parte do componente. Além disso, esses diagramas servem ainda para auxiliar nas atividades de teste que serão descritas no Capítulo 4.

Assim sendo, para cada Caso de Uso, são construídos diagramas de seqüência que representam o funcionamento interno do componente. O objetivo principal de construir estes diagramas é descobrir as operações que devem ser oferecidas por cada classe.

3.2.3 Fornecimento

De posse das especificações dos componentes, esta etapa tem como finalidade decidir quais componentes devem ser implementados e/ou quais componentes podem ser adquiridos de forma a obedecer a especificação fornecida.

É necessário escolher um ambiente de desenvolvimento, de acordo com as regras que eles impõem e as regras que devem ser obedecidas pelo componente. Os dois ambientes mais conhecidos são: Microsoft COM+ e Enterprise JavaBeans (EJB).

A escolha da tecnologia a ser utilizada para implementar o sistema afeta o modo como irá se passar da especificação para implementação. Alguns dos pontos que podem ser afetados são:

- parâmetros passados
- tratamento de erros e exceções
- herança de interface

Ainda nesta etapa, os componentes devem ser testados de forma individual, um a um. Isto será detalhado mais adiante, na Seção 3.3.

3.2.4 Montagem

Após adquirido e/ou desenvolvidos todos os componente necessários, é preciso integrá-los, para que juntamente com uma interface de usuário, seja possível obter uma aplicação.

Para certificação de que os componentes funcionam quando postos em conjunto é importante que sejam realizados os testes de integração, de forma a garantir que o sistema funcionará corretamente após todas as peças serem juntas.

3.2.5 Teste e Distribuição

Estas etapas não são detalhadas na apresentação original de Componentes UML. Elas são semelhantes às etapas de teste e distribuição propostas no RUP, onde as atividades de teste envolvem a verificação do sistema como um todo, com testes de integração e conformidade com os requisitos especificados e, a distribuição envolve o empacotamento, distribuição e instalação.

O presente trabalho, bem como o trabalho apresentado em [Farias, 2003], propõem uma definição mais detalhada para a atividade de teste, incluindo a realização de teste de integração e teste de componentes individuais, respectivamente, em paralelo com as atividades de desenvolvimento.

Após testada, a aplicação encontra-se pronta para ser distribuída.

3.3 Teste de Componentes

O teste de componentes, segundo proposto em [Farias, 2003], é constituído por uma série de atividades que são desenvolvidas dentro do contexto de cada uma das etapas de um processo de teste tradicional, e dispostas ao longo das fases do processo de desenvolvimento. Vale salientar que o método de teste de componentes, proposto em [Farias, 2003] e apresentado nesta seção, está sendo automatizado em [Barbosa, 2003].

Normalmente, um processo de teste tradicional, é constituído pelas seguintes etapas: Planejamento, Especificação, Construção, Execução e Análise dos Resultados.

- Planejamento

  Durante o planejamento são definidos quais os tipos de testes que serão realizados e o que se pretende obter mediante os testes realizados.

- Especificação

  Na especificação são gerados os modelos de teste, a partir dos quais são derivados os casos de teste, bem como os dados de teste e os oráculos.

- Construção

  Nesta etapa os casos de teste e os oráculos gerados anteriormente são implementados podendo ser utilizada alguma linguagem de programação. São criados também os artefatos necessários para execução dos testes.

- Execução

  De posse dos dados de teste selecionados durante a fase de especificação, os casos de teste são então executados.

- Análise dos Resultados

  De acordo com os oráculos gerados durante a construção, é feita uma análise com o intuito de verificar se os testes foram realizados com sucesso ou não.

As atividades de teste, como pode ser observado na Figura 3.2, são inseridas no processo da seguinte forma:

- O planejamento dos testes pode ser feito durante a definição dos requisitos. Esta atividade tem como objetivo definir os tipos de teste que devem ser realizados e o que se espera destes testes.

- A especificação dos testes é feita à medida que a modelagem é realizada. Durante a etapa de modelagem são selecionados os casos de teste, através a técnica TOTEM (Testing Object-orienTed systEms with the unified Modeling language) [Briand and Labiche, 2001], combinada com o aspecto estatístico do CleanRoom [Prowell et al.,

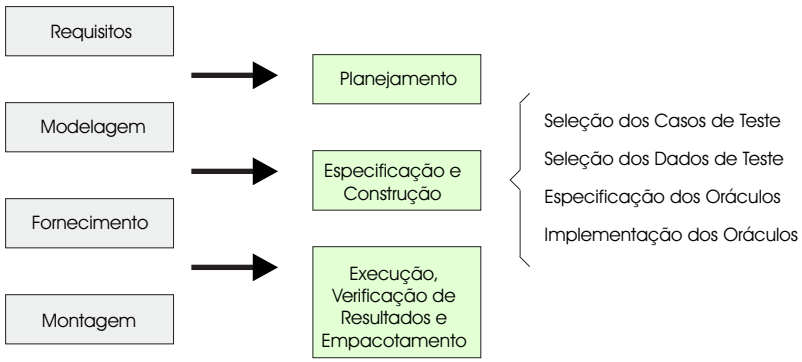

Figura 3.2: Integração das Etapas de Teste de Componentes no Processo de Desenvolvimento

1999]. Os dados de teste também são selecionados durante esta etapa. E por fim, os oráculos são gerados com base na técnica TOTEM, a partir das especificações de pré e pós-condições feitas em OCL.

- A construção dos testes pode ser realizada em paralelo ou ao final de sua especificação.
- A execução dos testes e a análise dos resultados é feita após o fornecimento dos componentes.

A seguir, estão descritas cada uma dessas etapas.

3.3.1 Planejamento

O planejamento pode ser inicializado logo que são produzidos os artefatos de análise, os quais representam informações importantes para o planejamento dos testes. O desenvolvimento do plano de teste no início do processo de desenvolvimento ajuda a dar idéia da dimensão da tarefa de teste a ser realizada fazendo com que os recursos sejam distribuídos de maneira mais racional. Nesta etapa deve se responder questões do tipo [McGregor and Sykes, 2001]:

Quem executará os testes? Os testes podem ser realizados tanto pelos desenvolvedores quanto por uma equipe especializada de teste. Em sistemas de pequeno porte, normalmente, é adotada a primeira opção, onde muitas vezes é realizada a troca de testes entre os desenvolvedores, onde cada um é responsável por testar a parte do outro. Isso evita que erros devido ao mal entendimento do funcionamento do negócio sejam levados

adiante. Já em sistemas críticos, como os que envolvem risco de vida, normalmente existe uma equipe responsável por realizar os testes. Ou ainda pode haver soluções que envolvam ambos os casos, como por exemplo: o testador poderá especificar os testes e os programadores poderão construí-los e executá-los.

Que partes serão testadas? Esta é uma das respostas mais difíceis de ser dada. Testar tudo? Não testar? Ou testar uma parte do software? Testar tudo parece ser o ideal quando se tem tempo e recursos disponíveis. Porém, não é o que acontece na maioria dos casos. Por isso, geralmente, testar uma amostra pode ser uma alternativa viável. Nesse caso, a seleção dos casos de teste vem a se tornar uma tarefa importante no processo de teste. Os casos de teste podem ser selecionados ao acaso, o que não é uma boa estratégia, uma vez que podem não testar funções comuns do software; podem ser selecionados baseados nos usos mais prováveis do sistema, ou seja, as entradas mais comuns; e ainda pode se dar ênfase aos casos patológicos, onde se assume que, se os desenvolvedores prestaram atenção nos casos mais obscuros é porque entenderam todos os requisitos e, portanto, o restante deve estar correto.

Quando serão executados os testes? Quanto mais cedo o problema for identificado mais fácil e mais barato será para consertá-lo. Dentro desta linha de raciocínio, temos três opções: testar sempre, testar à medida que os componentes são desenvolvidos ou testar tudo no final do processo. Testar sempre é uma ótima idéia, porém, é muito mais caro, e em geral não existem recursos suficientes para fazer uso desta opção. Testar tudo no fim do processo, normalmente é realizado quando existem poucos desenvolvedores envolvidos e quando conhecem bem os requisitos. Testar à medida que forem desenvolvidos os componentes pode retardar um pouco o processo de desenvolvimento. Por outro lado, pode se evitar maiores problemas no futuro, como por exemplo: integrar partes não testadas em um grande sistema. Esta última estratégia possui a vantagem de reduzir os custos de testar cada parta individualmente. Contudo, o sucesso dependerá da complexidade de cada parte e do custo que irá ter para consertar caso algum erro seja encontrado.

Como os testes serão realizados? Os testes podem ser realizados de duas maneiras: baseados na especificação, onde os testes nos diz o que o software faz; baseados na im-

plementação, onde nos diz como ele faz o que ele deve fazer; ou ainda uma mistura dos dois, baseados na especificação e na implementação. Os testes baseados na especificação evitam que os erros existentes se propaguem para a implementação, o que reduz os custos uma vez que os erros são encontrados cedo. Existem casos onde os testes baseados na implementação também se faz necessário, como por exemplo para componentes de alto risco, onde é importante ter certeza que cada linha está sendo executada com sucesso.

Quanto devemos testar? Essa decisão está relacionada à cobertura dos testes, ou seja, o quanto os testes exercitarão cada funcionalidade do software. Pode-se, então, não testar a funcionalidade, testá-la de forma exaustiva ou testá-la parcialmente. Testar exaustivamente se torna quase sempre inviável devido às limitações de recursos. Normalmente, os testes continuam até que os custos de não cobrir falhas são balanceados pela grande qualidade do produto.

É importante ressaltar que nenhuma dessas decisões é melhor ou pior que a outra. Cada uma é mais adequada em determinadas situações. Um bom exemplo para compreender isso melhor é comparar com as cores. Não existe uma cor melhor ou pior que outra. Mas pode se dizer que amarelo ou vermelho é melhor que preto para se pintar uma lanchonete, ou que branco é melhor que marrom para se pintar um hospital.

Elaborando o planejamento

Com o propósito de determinar o que e o quanto será testado, além dos artefatos produzidos na análise de requisitos, o planejamento de testes deve levar em conta também a análise de riscos. A realização dos testes baseados na análise de riscos tem como objetivo dar um grau de importância maior às partes do projeto que apresentam riscos mais elevados. Para isso, será utilizado o planejamento baseado em risco discutido em [McGregor and Sykes, 2001]. O objetivo principal da análise de risco é identificar o risco referente a cada caso de uso. Para realizar esta análise é necessário se desenvolver três tarefas:

- identificar os riscos que cada caso de uso representa ao desenvolvimento do software
- quantificar o risco através de uma análise de risco e,

- elaborar uma lista dos casos de uso ordenada pelo grau de risco.

Para estimar a quantidade de riscos referente a cada Caso de Uso deve se ter níveis suficientes para separar os Casos de Uso em grupos de tamanho razoável. A idéia é que os Casos de Uso que se enquadram no nível de grau mais elevado recebam uma atenção maior que os outros e conseqüentemente um maior número de casos de teste.

3.3.2 Especificação

Assim que a especificação do componente for iniciada, a especificação dos testes também pode ser iniciada . A especificação do componente contém informações importantes sobre a solução adotada para o problema e é usada para derivar os casos de teste, dados e oráculos.

Selecionando os Casos de Teste

Na fase de modelagem dos componentes foram construídos alguns diagramas de seqüência, para cada caso de uso, com o intuito de descobrir os métodos que cada classe que compõe o componente deveria fornecer para que a funcionalidade proposta no caso de uso pudesse ser entregue. Para a atividade de teste, Esses diagramas possuem a finalidade de orientar a seleção dos casos de teste. Já que os diagramas de seqüência fornecem uma visão dos diversos cenários de uso do componente, eles são usados para decidir quais cenários serão testados. A análise de riscos realizada durante o planejamento fornece a quantidade de cenários, de cada funcionalidade do componente, que serão testados, mas não diz quais são esses cenários. Dessa forma, é usada a combinação de duas técnicas de teste existentes, TOTEM e Cleanroom, para selecionar os cenários de uso do componente que serão testados.

A técnica TOTEM consiste em expressar o diagrama de seqüência na forma de expressões regulares, que são uma forma mais compacta e analisável dos diagramas. O alfabeto das expressões são os métodos públicos dos objetos presentes nos diagramas. As expressões são então formadas de termos que apresentam o formato Operação$_{classe}$, denotando a operação que está sendo executada seguida da classe a qual se encontra tal operação. A técnica TOTEM tem como idéia principal gerar uma única expressão regular, que representa os cenários principal e os alternativos do caso de uso, através da qual será possível extrair automaticamente todos os possíveis cenários.

Porém, como a construção de diagramas de seqüência que englobam tanto o cenário principal quanto os cenários alternativos não é uma prática comum, nem tão pouco é uma atividade trivial, o método de teste de componentes em questão propõe adaptar a técnica para gerar a expressão regular a partir de vários diagramas de seqüência. Neste caso, cada diagrama deverá representar um cenário de uso diferente, e a expressão regular gerada no final representará então todos os possíveis cenários de uso extraídos dos diagramas de seqüência.

A seleção dos cenários é feita com a incorporação à técnica TOTEM, de alguns aspectos estatísticos utilizados na técnica de teste usada no Cleanroom, com o intuito de melhorar os critérios de seleção dos cenários, possibilitando uma seleção mais automática e direcionada dos cenários de uso.

A técnica de seleção de casos de teste usada no Cleanroom propõe a construção de um Modelo de Uso do sistema o qual representa todos os possíveis usos do sistema e suas probabilidades de ocorrência. Essas probabilidades são definidas através de uma função de distribuição de probabilidade. O modelo de Uso é expresso normalmente por meio de um grafo direcionado, onde um conjunto de estados são conectados através de arcos de transição. Cada arco possui o passo a ser seguido e um valor de probabilidade associado a cada passo, representando um estímulo para o sistema, fazendo com que o mesmo mude de estado. Ao se percorrer o modelo a partir do seu estado inicial até o estado final, pode se gerar os casos de teste. De acordo com as probabilidades das transições, é definida uma seqüência de estímulos que leva o sistema do seu estado inicial ao estado final, através de um determinado caminho no modelo.A Figura 3.3 mostra um exemplo de um modelo de uso construído a partir da técnica de teste usada no Cleanroom.

Com o objetivo de utilizar a técnica de seleção de casos de teste do CleanRoom, são gerados Modelos de Uso, a partir das expressões regulares obtidas anteriormente, seguindo a técnica TOTEM. Para representar o início e o fim da seqüência da troca de mensagens entre os objetos são inseridos dois vértices. A cada troca de mensagens um novo vértice é criado. As transições existentes são rotuladas com mensagens, seguindo o formato Operação$_{classe}$ descrito pelas expressões regulares. Cada transição deve possuir uma probabilidade de ocorrência, que varia de 0 a 1, indicando as chances de ser disparada. Para finalizar uma seqüência de troca de mensagens, é determinada uma última transição, que liga o vértice da última chamada de operação ao vértice que indica o fim da seqüência.

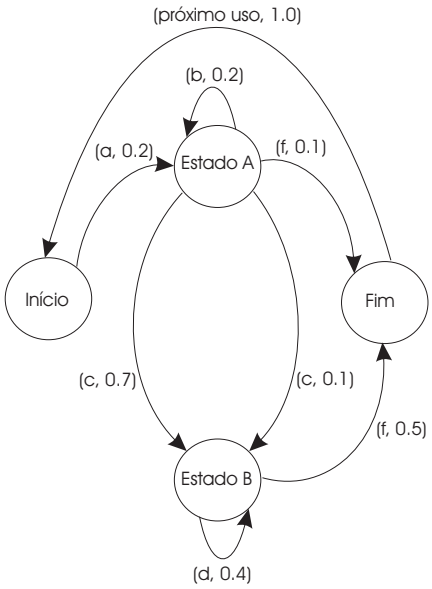

Figura 3.3: Modelo de Uso

Uma vez pronto o grafo de Modelo de Uso deve se selecionar alguns caminhos, de acordo com uma função de probabilidade, que será usada nos casos de teste. O número de casos de teste que devem ser retirados do grafo, são definidos na fase de planejamento, onde para cada caso de uso existe um nível de prioridade e conseqüentemente um certo de número de casos de teste a ser realizados.

Gerando os Oráculos

Para automatizar a atividade de teste é imprescindível uma definição precisa dos oráculos. De acordo com a técnica TOTEM, para a construção dos oráculos, é proposto que seja construída uma tabela de decisão para cada expressão regular que representa os cenários de uso do componente. A tabela é composta pelas condições de realização do uso e pelas ações que serão tomadas pelo componente diante da ocorrência do uso. Para cada termo de cada expressão regular são identificadas suas condições de execução, as quais são expressas em OCL. As ações são compostas pelas mudanças de estado, que podem ocorrer no componente com a execução de cada cenário, e, pelas mensagens que podem ser retornadas para o ator do Caso de Uso.

Selecionando os Dados

Para seleção dos dados de teste, é usada uma técnica denominada teste de domínio, proposta por [Beizer, 1999]. Essa técnica parte do princípio que os dados de entrada de um programa podem ser agrupados em classes as quais apresentam características comuns e, que o programa se comporta da mesma forma para todos os membros de uma mesma classe. Portanto, para se selecionar os dados de teste deve se identificar as partições e escolher dados particulares dentro de cada partição. A identificação das partições se baseia na especificação e documentação do software. As condições de execução definidas durante a geração dos oráculos podem servir para identificar partições adequadas ao teste. A escolha dos dados tanto pode ser feita de forma aleatória, como de forma mais direcionada, a fim de obter dados mais prováveis de revelar erros. Na escolha direcionada são considerados os dados encontrados nos limites da partição, por representarem normalmente valores atípicos, e dados considerados típicos, encontrados no meio da partição.

3.3.3 Construção e Execução dos Testes e Verificação dos Resultados

A implementação dos casos de teste deve ser feita com base nas informações contidas nas Tabelas de Decisão. Para cada versão definida na tabela, devem ser implementadas as condições de execução definidas na tabela. Por fim, deve ser verificada a mensagem que é retornada, se existir alguma e, a ocorrência ou não de mudança de estado. Após a execução do teste os resultados obtidos devem ser comparados com as definições da tabela, o que irá indicar o sucesso, ou insucesso, do teste. Para analisar os resultados dos testes também pode se fazer uso das especificações OCL fornecidas junto com o componente, com o objetivo de verificar se os componentes funcionam como esperado, ou seja, se eles produzem a resposta certa ao final e seguem de forma correta o caminho pré-definido.

No Capítulo 5 será detalhado o processo de desenvolvimento do estudo de caso realizado.

3.4 Conclusão

Neste capítulo foi apresentado o processo de desenvolvimento Componentes UML, com algumas adaptações propostas em [Farias, 2003], descrevendo cada etapa do processo: Re-

quisitos, Modelagem, Fornecimento, Montagem, Testes e Distribuição.

Na fase de requisitos são gerados alguns artefatos: Modelo de Processo do Negócio Modelo Conceitual do Negócio e Modelos de Casos de Uso. A fase de Modelagem consiste em identificar os componentes, descobrindo quais as interfaces de sistema, através dos Modelos de Casos de Uso e quais as interfaces de negócio, através do Modelo de Tipo de Negócio; definir a interação entre os componentes, através da construção de diagramas de colaboração para cada operação das interfaces de sistema e; fornecer a especificação do componente, através das definições dos contratos de uso e de realização. O contrato de uso é definido pela especificação das interfaces que inclui as especificações das operações, os Modelos de Informação e as Invariantes. Já o contrato de realização é definido pela especificação dos componentes, a qual inclui artefatos que representam o funcionamento interno dos componentes.

Foram apresentadas também as atividades de teste propostas em [Farias, 2003], para o teste de componentes. Estas atividades ocorrem em paralelo com o processo de desenvolvimento e foram descritas dentro das etapas de um processo de teste tradicional: Planejamento, Especificação, Construção, Execução e Análise dos Resultados. No Planejamento, realizado durante a definição de requisitos, é definido os tipos de teste realizados e o que se espera de cada um deles. Além disso, é realizada uma análise de risco para definir quantos casos de teste serão construídos para cada funcionalidade do sistema. Na especificação, realizada durante a fase de modelagem, são selecionados os casos de teste, através a técnica TOTEM e de alguns aspectos estatísticos do Cleanroom. São selecionados também os dados de teste e, são gerados os oráculos com base na técnica TOTEM, a partir das especificações de pré e pós-condições feitas em OCL. A construção dos testes é realizada durante ou no final da especificação, e é realizada com base nos artefatos gerados na etapa anterior. A execução dos testes e a análise dos resultados é feita logo após o fornecimento dos componentes, e é feita com o auxílio de alguma ferramenta já existente.

Como pode ser percebido, o processo de teste de componentes foi realizado ao longo do processo de desenvolvimento, o que pode gerar vantagens, como por exemplo a possível redução dos custos envolvidos no projeto, uma vez que os erros podem ser encontrados cedo, evitando a sua propagação para fase posteriores do projeto.

Capítulo 4

Método de Teste de Integração

Este capítulo tem a finalidade de descrever o método de teste de integração proposto. A Seção 4.1 faz uma breve introdução do capítulo. Na Seção 4.2, encontra-se o detalhamento da estrutura do método contextualizado dentro do precesso de desenvolvimento adotado. Na Seção 4.3 encontra-se descritas as etapas do método de teste prosposto. Na Seção 4.4 estão algumas conclusões e comentários a respeito do método.

4.1 Introdução

A principal idéia de um desenvolvimento baseado em componentes é usar componentes já prontos para se produzir software. O sistema resultante pode possuir algumas características que podem complicar os testes, como por exemplo a ausência de código-fonte dos componentes. Por isso, normalmente, um processo de teste de integração de sistemas baseados em componentes se preocupa primeiramente em entender as dependências existentes entre os componentes que constituem o sistema. Essas dependências normalmente são representadas por alguns grafos construídos a partir de especificações das interfaces fornecidas pelos componentes, sejam elas dadas através de máquinas de estado finito [Beydeda and Gruhn, 2001], ou através de linguagens próprias de especificação [Vieira et al., 2001] ou através de diagramas UML, como é mais comumente utilizado [Hanh et al., 2001], [Traon et al., 1999], [Traon et al., 2000]. A partir dessa representação gráfica das dependências dos componentes e com o auxílio de outras informações possivelmente fornecidas pelos componentes, como por exemplo os oráculos de teste, são gerados os casos de teste de integração baseados na

construção de *stubs* [Hanh et al., 2001], [Traon et al., 1999], [Traon et al., 2000].

4.2 Estrutura

O método de teste de integração aqui proposto, similarmente ao método de teste de componentes isolados, também possui um conjunto de atividades que são desenvolvidas dentro do contexto de cada uma das etapas de um processo de teste tradicional. Como foi adotada a filosofia de que o processo de teste está integrado ao processo de desenvolvimento, cabe agora definir as atividades de teste propostas e contextualizá-las dentro do processo de desenvolvimento descrito no capítulo anterior. As etapas do método de teste proposto (blocos cinzas), estão contextualizadas dentro do processo desenvolvimento Componentes UML (blocos brancos), como mostra a Figura 4.1, da seguinte forma:

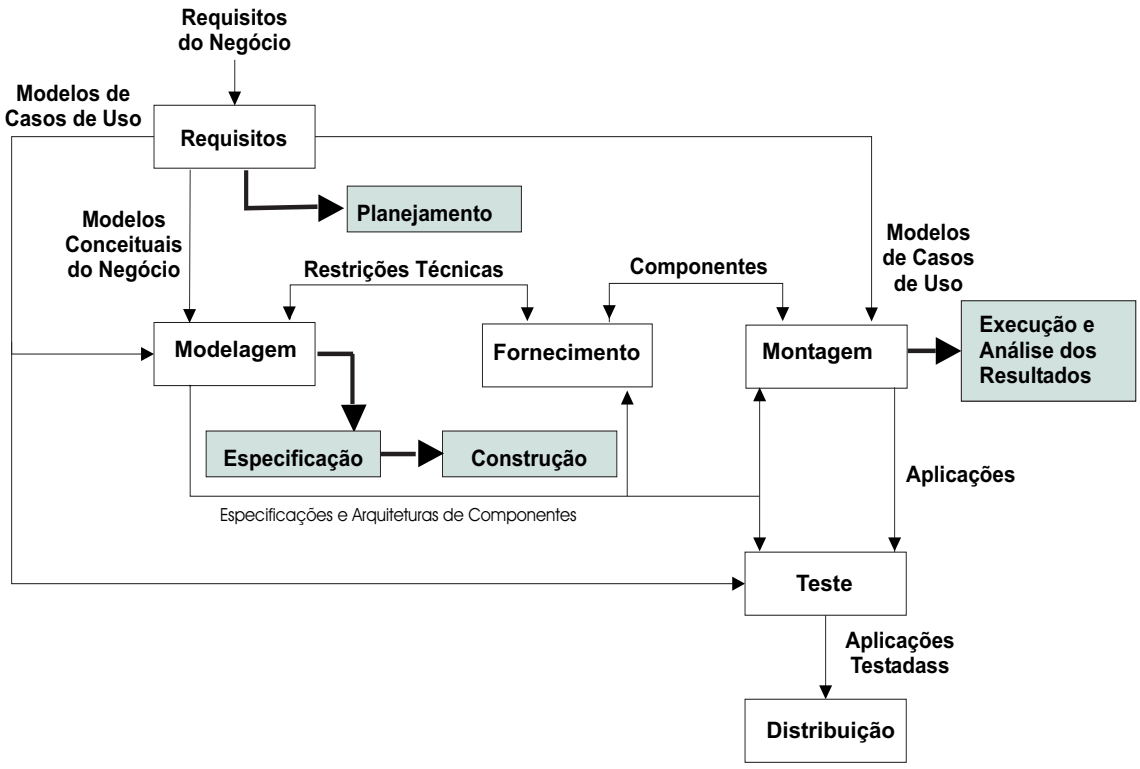

Figura 4.1: Etapas do processo de teste proposto contextualizado dentro do processo de desenvolvimento Componentes UML .

- O planejamento dos testes pode ser feito durante a definição dos requisitos, definindo o que se espera dos testes que devem ser realizados.

- A especificação dos testes é feita à medida que a modelagem é realizada. Esta etapa consiste na escolha da ordem de teste dos componentes, na geração e seleção dos casos de teste de cada componente dentro do contexto onde está inserido e na geração dos dados e oráculos de teste.

- A construção dos testes pode ser realizada em paralelo ou ao final de sua especificação.

- A execução dos testes e a análise dos resultados é feita durante a montagem dos componentes.

É válido observar que a execução dos testes e análise dos resultados, do método de teste individual de componentes, são etapas realizadas após a fase de Fornecimento do processo de desenvolvimento. Já no método de teste de integração, essas etapas são realizadas após a fase de Montagem do processo de desenvolvimento. Isso acontece porque no teste individual de componentes, à medida que um componente fica pronto, seus testes já podem ser executados. Entretanto, para executar o teste de integração dos componentes, não basta que apenas um componente esteja pronto, é preciso que pelo menos dois componentes estejam prontos para dar início a execução dos testes e análise de seus resultados. Por isso, é durante a fase de Montagem do processo de desenvolvimento, que os testes de integração podem começar a ser executados. A contextualização das etapas do processo do método de teste individual de componentes, bem como do método de teste de integração, podem ser observadas na Figura 4.2.

4.3 Etapas do método de teste de integração

4.3.1 Planejamento

O planejamento dos testes de integração pode ser feito durante a definição dos requisitos.

O planejamento dos testes tem o intuito de determinar o que e o quanto será testado. Para isto, são utlizados os artefatos produzidos na análise de requisitos, mais precisamente os Casos de Uso, e a partir deles é realizada uma análise de riscos. A realização dos testes baseados na análise de riscos, assim como no teste isolado de componentes, tem como objetivo dar um grau de importância maior às partes do projeto que apresentam riscos mais elevados. Para

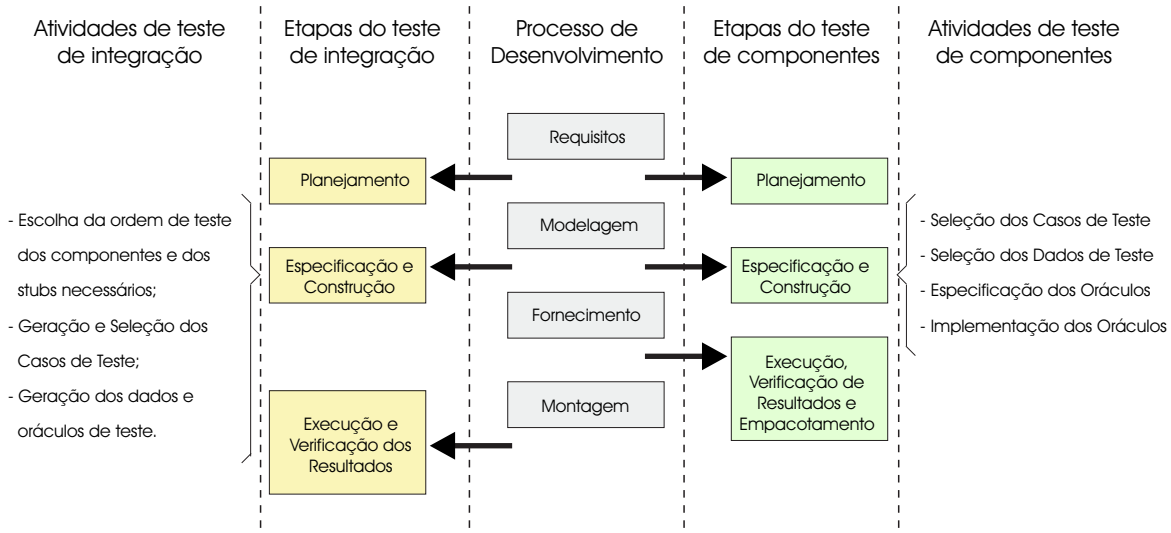

Figura 4.2: Etapas e atividades do processo de teste de componentes e de integração contextualizados dentro do processo de desenvolvimento Componentes UML .

isso, também será utilizado o planejamento baseado em risco discutido em [McGregor and Sykes, 2001]. A análise de risco tem como objetivo principal identificar os riscos referente a cada Caso de Uso. Para realizar esta análise, primeiramente deve se identificar os riscos que cada Caso de Uso representa ao desenvolvimento do software. Em seguida, quantifica-se o risco através de uma análise do grau do risco. Logo mais, deve ser elaborada uma lista dos Casos de Uso ordenada pelo grau de risco.

Com o intuito de estimar quantos riscos devem ser elencados para cada Caso de Uso, deve se ter diferentes níveis suficientes para separar os Casos de Uso em grupos de tamanho razoável. Os Casos de Uso que possuirem o nível de grau mais elevado devem receber uma atenção maior que os outros, obtendo-se também um maior número de casos de teste. Dessa forma, são adquiridos os números de casos de testes que devem ser desenvolvidos para cada funcionalidade requerida.

Algumas perguntas essenciais também precisam ser respondidas, com base no teste de integração, para uma melhor elaboração do planejamento:

Quem executará os testes? Os testes de integração podem ser executados por uma equipe de teste específica ou pelos projetistas responsáveis pela integração (montagem do sistema). Dependendo do tamanho e do quanto crítico é o sistema a ser testado, poderá ser escolhida uma das opções. Em caso de sistemas pequenos, os próprios projetis-

tas podem ser responsáveis pelos testes, podendo haver troca de trabalho, onde cada um pode testar a parte do outro, evitando a propagação de erros devido ao mal entendimento do funcionamento da aplicação. Em casos de sistemas mais complexos e críticos, é interessante que se tenha uma equipe específica para se testar a aplicação.

Que partes serão testadas? Serão testadas as interações entre componentes, geradas por chamadas a métodos de interface ou passagem de parâmetros. Normalmente, são escolhidas as interações que contribuam para a realização das principais funcionalidades da aplicação expressas nos casos de uso.

Quando serão executados os testes? Os testes de integração devem, no geral, serem realizados à medida que a montagem do sistema for feita. Portanto, a ordem de teste pode influenciar a ordem de integração/montagem do sistema.

Como os testes serão realizados? O ideal é verificar a integração de um componente por vez, a fim de facilitar a localização de erros. Para isto, uma ordem de integração precisa ser definida. Geralmente, *stubs* são construídos para simular o comportamento de componentes ainda não desenvolvidos ou não previamente testados.

Quanto devemos testar? A quantidade de testes deve ser guiada pela cobertura dos cenários da aplicação e a quantidade de interações existentes na arquitetura de componentes. As iterações podem ser definidas a nível contratual, onde é considerada apenas uma visão *black-box*, onde um conjunto de testes é definido independente da implementação, ou uma visão *white-box*, onde chamadas dentro das implementações de cada método também são consideradas, bem como métodos privados. A vantagem da primeira é que os testes podem ser reutilizados face a modificações no código. A segunda apesar de mais rigorosa implica em uma modificação dos testes sempre que a implementação de um método for mudada. Ambas são recomendáveis, o que vai determinar qual a melhor a ser utilizada será o contexto do projeto de teste, podendo ainda haver uma mistura das duas.

4.3.2 Especificação

A especificação dos testes deverá ser realizada durante a fase de Modelagem do processo de desenvolvimento, onde são produzidos os diagramas de interação, os quais possuem informações importantes a serem utilizadas nesta etapa. Neste momento serão realizadas as seguintes atividades:

- Escolha da ordem de teste dos componentes e dos *stubs* necessários;
- Geração e seleção dos casos de teste;
- Geração dos dados e oráculos de teste.

Escolha da ordem de teste dos componentes e dos *stubs* necessários

Nesta etapa é definida a ordem de integração dos componente e, conseqüentemente, quais *stubs* deverão ser construídos. Como um *stub* não é um componente real e não será usado no produto final, se faz importante a minimização dos esforços gastos para sua criação. Portanto, quanto menor for o número de *stubs* a serem construídos menor são os custos relacionados com o teste de integração.

A fim de determinar a ordem de integração e os *stubs* a serem construídos, de preferência a partir de artefatos já produzidos durante o processo de desenvolvimento do sistema, é utilizada a estratégia de integração apresentada em [Traon et al., 2000] e detalhada no Capítulo 2. A estratégia consiste na construção de um grafo, a partir de artefatos UML, que representa as dependências existentes entre classes e métodos de um determinado sistema. Em seguida, é aplicado um algoritmo ao grafo, para que dele possa ser extraída a ordem de integração dos componentes, baseando-se sempre na minimização do número de *stubs*. De acordo com a estratégia abordada, o GTD pode ser gerado através de refinamentos feitos a partir de diagramas de classe UML, com o propósito de testar a integração entre as classes e métodos que compõem o sistema. O método aqui proposto possui um intuito um pouco diferente do intuito da estratégia de teste de integração adotada em [Traon et al., 2000]. Ao invés de pretender testar as relações existentes entre todas as classes e métodos do sistema, o método tem o intuito de testar a integração entre os componentes do sistema. Além disso, de acordo com o processo de desenvolvimento adotado, não se tem um diagrama de classe disponível

para se gerar o GTD, da mesma forma que em um sistema tradicional. Por isso, algumas adaptações são feitas à estratégia para que seja possível usá-la a nível de componentes.

Um componente pode ser composto por várias classes e interfaces, contendo pelo menos uma interface de negócio através da qual o componente irá se comunicar com o meio externo. Essas interfaces devem conter todas as operações necessárias para fornecer os serviços especificados pelos componentes. Como o Modelo de Informação define um conjunto de informações oferecidas pelas interfaces de negócio dos componentes, ele deve ser usado, ao invés do diagrama de classes, para geração do GTD. Cada interface representada pelo Modelo de Informação deve funcionar como uma interface do diagrama de classes. Os Diagramas de Colaboração, anteriormente gerados, também podem ajudar na geração do grafo, ajudando a identificar as relações existentes entre os componentes.

Com a utilização dos Modelos de Informação e dos Diagramas de Colaboração para a geração do GTD, a estratégia de integração utilizada faz uso apenas de artefatos já produzidos durante o processo de desenvolvimento, não se fazendo necessária a construção de nenhuma especificação adicional para utilização da estratégia. Isso é um ponto positivo do método, uma vez que não é desperdiçado tempo com a geração de artefatos apenas com o propósito de teste.

Geração e Seleção dos Casos de Teste

Uma vez gerada a ordem de integração e escolhidos os *stubs* a serem construídos, cabe agora se preocupar com a geração e seleção dos casos de teste para cada componente a ter sua integração testada. Para isto, será utilizada para cada componente, a técnica de geração e seleção de casos de teste detalhada no Capítulo 3. A técnica consiste na aplicação da técnica TOTEM em conjunto com alguns aspectos estatísticos do Cleanroom.

É recomendável que os casos de teste sejam gerados apenas para os componentes não triviais. Os triviais podem ser considerados como funcionalidade correta, dada que uma análise estática, por exemplo, tenha sido realizada.

Para a seleção dos casos de teste, algumas considerações adicionais precisam ser feitas:

- O esforço de teste de integração é bem maior que o do teste de um componente individual, ou seja, uma seleção mais rigorosa para o primeiro precisa ser feita a fim

de escolher um número mínimo de testes de acordo com as funcionalidades esperadas dentro de um contexto. Nem todas as funcionalidades fornecidas por um componentes são de interesse do contexto da aplicação, principalmente, se estiver sendo reutilizado.

- Após o componente ter sido previamente testado, durante a fase de materialização, casos de teste previamente selecionados e executados devem ser analisados, bem como o modelo de testes desenvolvido pode ser aproveitado e extendido para incluir novos cenários de interesse. Dependendo da semelhança do contexto da aplicação com o(s) usado(s) para testar o componente, casos de teste podem se tornar redundantes e não precisarem ser repetidos.

- Caso o componente não tenha sido testado previamente, faz-se necessário aplicar as técnicas de geração e seleção para o mesmo, com enfoque no contexto da aplicação (funcionalidades de interesse da aplicação).

Geração dos Dados e Oráculos de Teste

A definição precisa dos oráculos de teste é imprescindível para automação da atividade de teste. Seguindo o método de teste de componentes isolados, são geradas as tabelas de decisão para cada componente. E, para cada cenário de teste presente na tabela, são expressas suas condições de execução em OCL.

A tarefa de geração de dados de teste consiste na seleção de pontos de cada subdomínio com a finalidade de satisfazer um determinado critério, revelando um maior número de erros possíveis. Apesar da automação dessa tarefa ser importante, não existe um algoritmo de propósito geral para determinar um conjunto de dados de teste que satisfaça um certo critério.

Um conjunto de dados adequado deve ser grande o suficiente para englobar todos os valores válidos do domínio e suficientemente pequeno para que se possa testar elementos de cada tipo de entrada do conjunto.

Embora existam várias estratégias para selecionar os dados de teste, propõe-se que seja utilizada a geração aleatória dos dados. Esta estratégia não garante a seleção dos melhores dados, mas permite gerar grandes conjuntos de dados de teste a baixo custo. Além disso, a técnica aleatória elimina qualquer possível influência do testador em conduzir a geração dos dados de teste conforme o conhecimento prévio dos programas utilizados, o que pode levar

a falsas conclusões na análise dos dados obtidos nessa atividade [Domingues, 2002].

4.3.3 Construção, Execução e Análise dos Resultados

A construção dos testes poderá ser realizada assim que os casos de teste estiverem prontos, ou seja, após a modelagem dos componentes. A execução e análise dos resultados poderá ser realizada durante a fase de montagem, onde os componentes são integrados.

A implementação dos casos de teste se dará da mesma forma que no método de teste de componentes, afinal, são os casos de teste dos componentes que estão sendo implementados, porém, desta vez, obedecendo a ordem de integração dos componentes e dando uma visão mais abrangente aos casos de teste. À medida em que vão sendo implementados os casos de teste, vai-se obedecendo a ordem de integração. Caso algum caso de teste, para ser implementado, necessite de alguma funcionalidade que ainda não foi implementada, será construído um *stub*, de acordo com a ordem de integração gerada.

Como se trata de um processo iterativo e incremental é importante ressaltar que na prática, se torna interessante o uso de ferramentas para auxiliar esse processo, uma vez que, à medida que novos componentes são desenvolvidos, são também testados dentro do contexto onde está inserido, fazendo com que o processo se torne um pouco trabalhoso por ser repetitivo.

Para analisar os resultados pode também se fazer uso das especificações OCL fornecidas, bem como das tabelas de decisão, onde são encontradas informações a respeito de como deverá ser o estado do componente e quais mensagens podem ser exibidas, após a execução de um determinado cenário de teste.

4.4 Conclusões

Este capítulo apresentou, as etapas envolvidas no método de teste proposto, bem como as atividades realizadas dentro de cada uma dessas etapas. As atividades envolvidas no método se encontram definidas dentro das seguintes etapas:

- **Planejamento:** Com o objetivo de ter noção da dimensão da atividade de teste a ser realizada, consiste em explorar os Casos de Uso e escolher as funcionalidades de alto

nível a serem testadas, na aplicação com um todo.

- **Especificação:** Esta fase consiste nas seguintes atividades: na escolha da ordem de integração e de quais *stubs* precisam ser construídos, através da geração do GTD e da aplicação de um algoritmo ao mesmo; na geração e seleção dos casos de teste, partindo de especificações UML e; na geração dos dados e oráculos de teste.

- **Construção, Execução e Análise dos Resultados:** Esta etapa consiste na construção sistemática de todos os casos de teste selecionados na etapa anterior, bem como na sua execução, de acordo com a ordem prescrita pelo algoritmo. A análise dos resultados é realizada de acordo com alguns artefatos gerados na especificação.

O método aqui proposto poderá ser utilizado seguindo um outro processo de desenvolvimento que não seja o utilizado neste trabalho, desde que:

- Os componentes que irão compor o sistema possuam interfaces bem definidas, especificadas em OCL e sejam testados utilizando o método de teste proposto em [Farias, 2003], ou equivalente, que produza os mesmos resultados (implementação de testes empacotadas);

- O sistema esteja especificado em UML, possuindo os diagramas de seqüência e de colaboração.

Ainda não é possível determinar qual o cenário de melhor resultado para o uso do método proposto. Para isto, seria necessário o desenvolvimento de mais estudos de caso, para que de posse de mais resultados concretos, pudesse ser feita uma análise mais detalhada, chegando a conclusões mais completas. Contudo, acredita-se que o método de teste proposto, se seguido cuidadosamente, poderá fornecer melhorias significativas na qualidade final do produto, fazendo com que falhas que possam aparecer ao longo do processo sejam reduzidas.

Capítulo 5

Estudo de Caso - Desenvolvimento da Aplicação

O desenvolvimento de um estudo de caso é bastante interessante quando se tem o intuito de validar algum método e demonstrar de forma prática a aplicação do mesmo.

Uma vez descrita a metodologia de desenvolvimento utilizada e o método de teste de integração de componentes, resta facilitar o seu entendimento e comprovar sua aplicação através da apresentação do estudo de caso realizado. Esta apresentação encontra-se dividida em duas partes: no desenvolvimento da aplicação, seguindo a metodologia de desenvolvimento anteriormente descrita e; na aplicação do método de teste de integração de componentes proposto. A primeira parte encontra-se descrita neste capítulo, já a segunda parte está descrita no Capítulo 6. Na Seção 5.1 encontra-se descrito um roteiro mínimo necessário para a aplicação do método. A Seção 5.2 fala sobre a escolha da aplicação. Na Seção 5.3 é detalhado o desenvolvimento da aplicação. E por fim, na Seção 5.4 encontra-se a conclusão do capítulo.

5.1 Roteiro Mínimo

Com o intuito de esclarecer os principais passos necessários para aplicação do método proposto, foi descrito um roteiro mínimo para uso do mesmo.

Primeiramente, deve ser escolhida uma metodologia de desenvolvimento dos componentes, que poderá ser diferente de Componentes UML, desde que forneça as entradas necessárias para aplicação do método. Neste caso, de acordo com a metodologia escolhida

para o desenvolvimento dos componentes, deverá ser analisada em que fase da metodologia de desenvolvimento deverá ser aplicada cada etapa do método de teste proposto, uma vez que o método proposto deve ser aplicado paralelarmente a metodologia de desenvolvimento.

Uma vez escolhida a metodologia de desenvolvimento a ser utilizada e definida onde irá ser aplicada cada etapa do método de teste proposto, já pode se partir para aplicação de cada uma delas. Na fase de Planejamento o intuito é ter idéia das tarefas de testes que se deseja realizar e elaborar uma análise de risco baseada nos Casos de Uso da aplicação. Assim, tem-se idéia da quantidade de casos de teste que devem ser realizados. Em seguida, na fase de Especificação são realizadas as seguintes atividades: escolha da ordem de teste dos componentes e dos *stubs* necessários; geração e seleção dos casos de teste; geração dos dados e oráculos de teste. Para escolher a ordem de teste dos componentes e dos *stubs* necessários, o método faz uso do Modelo de Informação e dos Diagramas de Colaboração criados durante o desenvolvimento da aplicação, para geração do GTD. Neste caso, seria interessante que a metodologia de desenvolvimento escolhida fizesse uso esses diagramas, pois a idéia é reaproveitar os artefatos criados durante o desenvolvimento. Caso contrário, poderia se adaptar algum artefato criado durante o desenvolvimento, que contivesse o conjunto de informações oferecidas pelas interfaces de negócio dos componentes, de modo a gerar apropriadamente o GTD. Para a geração e seleção dos casos de teste é utilizada a técnica TOTEM em conjunto com alguns aspectos estatísticos utilizados na técnica de teste do Cleanroom. A técnica TOTEM faz uso dos Diagramas de Sequência, também criados durante a fase de desenvolvimento de Componentes UML. Sendo assim, caso seja utilizada uma outra metodologia de desenvolvimento, é importante que nela sejam desenvolvidos tais diagramas. Caso contrário, eles devem ser construidos durante o processo de teste, o que irá demandar mais tempo gasto na fase de teste. Isto não é bom, uma vez que a idéia é reutilizar artefatos anteriormente produzidos de modo a evitar retrabalho e minimizar o esforço gasto na fase de teste. Para geração dos dados e oráculos de teste são construidas as tabelas de decisão. O método proposto sugere que seja seguido um determinado processo de teste individual de componentes. Neste processo são criadas as Tabelas de Decisão e, as mesmas são reaproveitadas para o teste de integração. Caso seja utilizado um outro método de teste individual de componentes, as Tabelas de Decisão devem ser construidas neste momento, o que não é muito interessante, pois demandaria mais tempo durante a aplicação do método.

Dessa forma, de acordo com os Casos de Teste elaborados, com os dados de teste selecionados, com as Tabelas de Decisão criadas e, seguindo também a ordem de integração obtida através do GTD, os testes podem ser construidos e executados. A ferramenta utilizada pode ser qualquer uma, desde que os testes possam ser implementados e executados sem maiores problemas.

No que se refere à automação, a maioria dos métodos existentes não são automatizados e nem possuem um potencial para automação. Dessa forma, a automação do método é um ponto importante e que deve ser bem analisado. Apesar do método proposto ainda não se encontrar automatizado, ele se preocupa com a possibilidade de automação e possui várias etapas que podem ser automatizadas, tais como:

- a geração do GTD. Uma vez prontos o Modelo de Informação e os Diagramas de Colaboração, a geração do GTD poderia ser automática;
- a geração da ordem de integração, uma vez que o algoritmo para isto já encontra-se bem definido.
- a atividade de teste propriamente dita, a partir das tabelas de decisão criadas, as quais possuem as condições de realização do uso e as ações que serão tomadas pelo componente diante da ocorrência do uso, expressas em OCL.

No tocante a aplicação do método em outro processo de desenvolvimento teria que se avaliar com cautela o impacto causado por esta mudança. Dependendo da metodologia de desenvolvimento escolhida, muitos diagramas utilizados pelo método, que a princípio seriam reutilizados, como o Modelo de Informação, o Diagrama de Sequência, o Diagrama de Colaboração, podem não terem sidos constuídos durante o processo de desenvolvimento. Dessa forma, eles teriam que ser desenvolvidos durante o processo de teste, o que demandaria muito mais tempo na aplicação do método. Além disso, teria que se avaliar as etapas existentes na metodologia de desenvolvimento adotada e tentar sincronizar o método de teste com tais etapas. Por isso, é interessante que seja seguida a metodologia recomendada pelo método. Caso contrário, quanto mais distante da metodologia de desenvolvimento adotada estiver a metodologia escolhida, maior será a dificuldade aplicar o método proposto.

5.2 Escolha da Aplicação

A aplicação que serviu como estudo de caso foi escolhida de forma a obedecer uma série de parâmetros, tais como:

- possuir vários componentes interagindo entre si, a fim de demonstrar o método com mais clareza. A aplicação escolhida possui nove componentes que interagem entre si;
- não envolver demasiadamente problemas específicos relativos à tecnologias como: banco de dados, sistemas distribuídos, sistemas web, dentre outros, de modo que o foco do trabalho não seja perdido, envolvendo outros tipos de testes. A aplicação escolhida trata de um jogo simples que funciona localmente, não possuindo banco de dados, sistema web ou qualquer outro tipo de tecnologia que pudesse envolver outros tipos de testes;
- ser de fácil entendimento para que o foco seja dado a compreensão do método, e não ao funcionamento da aplicação. A aplicação escolhida é um jogo bem simples, não necessitando um grande esforço para compreensão do mesmo.

Seguindo os requisitos especificados acima, a aplicação escolhida é uma aplicação bastante conhecida que é o *Snake Game*, o conhecido jogo da cobrinha, muito encontrado em aparelhos celulares e jogos de computadores. O jogo trata de uma aplicação de funcionamento bastante simples, onde a cobra sai andando dentro de um tabuleiro, em uma determinada direção, a qual pode ser controlada pelo jogador. As comidas são geradas aleatoriamente dentro do tabuleiro e o objetivo principal do jogo é fazer com que a cobra coma o maior número possível de comidas.

No Snake Game, existem cinco tipos de jogos: Hungry Snake, Gula Gula, Gula Gula 2, Magic Snake e My Snake. Cada um deles possui regras diferentes, como por exemplo: comidas diferentes onde uma pode valer mais pontos que outras, obstáculos dispostos em posições variadas, etc. Essas regras não são relevantes neste exato momento, mas ficarão mais claras no decorrer deste capítulo. A seguir, será abordado todo o processo de desenvolvimento seguido para a elaboração do estudo de caso.

5.3 Desenvolvimento do Estudo de Caso

Para dar início ao estudo de caso, são seguidos os passos propostos no processo de desenvolvimento descrito no Capítulo 3, que se refletem nas seguintes etapas: Requisitos, Modelagem, Fornecimento, Montagem, Teste e Distribuição. Ao longo deste desenvolvimento são realizadas também, as atividades de teste propostas no método de teste de integração descrito no Capítulo 4, as quais estão descritas no capítulo seguinte.

5.3.1 Requisitos

Inicialmente, são gerados os artefatos que constituem a fase de Requisitos: o Modelo de Processo do Negócio, o Modelo Conceitual do Negócio e, os Modelos de Casos de Uso.

Na Figura 5.1, encontra-se o Modelo de Processo do Negócio referente ao Snake Game, representando uma visão geral do funcionamento da aplicação. O Modelo é bem simples e é construído utilizando o diagrama de atividades UML. Uma explicação mais detalhada de como funciona este modelo pode ser encontrada no Texto Complementar produzido na Tabela 5.1, como sugerido em [Farias, 2003].

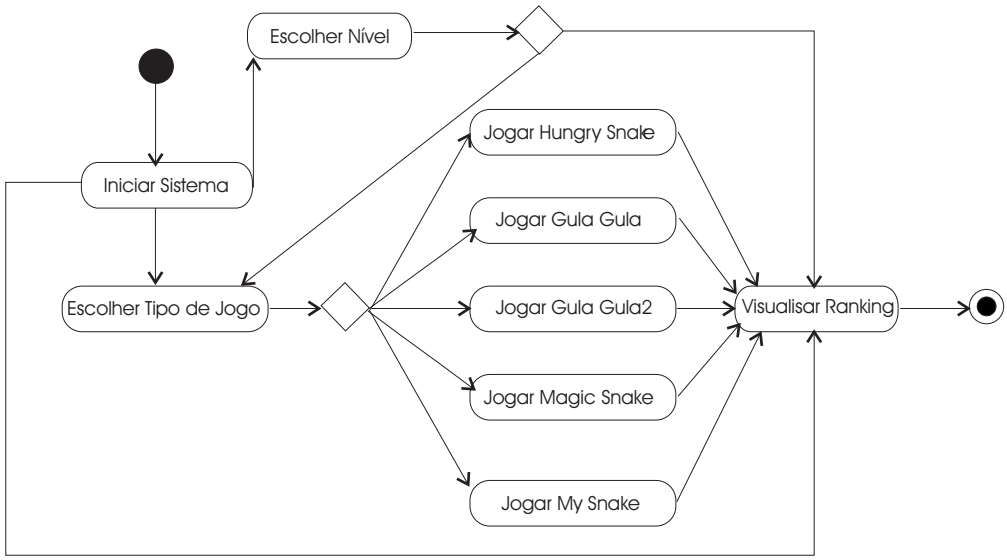

Figura 5.1: Modelo de Processo do Negócio

O próximo artefato construído a ser construído é o Modelo Conceitual do Negócio. Como sua principal função é estreitar os laços existentes entre as pessoas que fazem parte do negócio e aquelas que são responsáveis pelo projeto, através do descobrimento de alguns

Texto Complementar:
O sistema possui 5 tipos de jogos. O usuário poderá escolher qual tipo de jogo deseja jogar e o nível do jogo desejado. Existem 3 níveis: nível 1, nível 2 e nível 3, que indicam a velocidade do jogo. A qualquer momento, desde que não esteja jogando, o usuário poderá pedir para visualisar o ranking. Durante o jogo, o objetivo é sempre comer o maior número de comidas possíveis. A diferença entre um jogo e outro são as regras que cada um possui. Um tem tipos de comidas diferentes, onde uma vale mais pontos que outras; outro possui mais obstáculos; outro a cobra pode ficar invencível, não morrendo por um determinado tempo, e assim por diante.

Tabela 5.1: Texto Complementar ao Modelo de Processo do Negócio

conceitos importantes e dos relacionamento existentes entre eles, são detectadas algumas palavras-chave que se tornaram conceitos de relevância para a elaboração do modelo. Como se pode observar, Comida, Parede, Regras, Jogo, Ranking, Cobra, Jogador, Tipos de Comida e Tabuleiro são considerados conceitos importantes para o Snake Game e estão representados no Modelo Conceitual do Negócio, que pode ser encontrado na Figura 5.2.

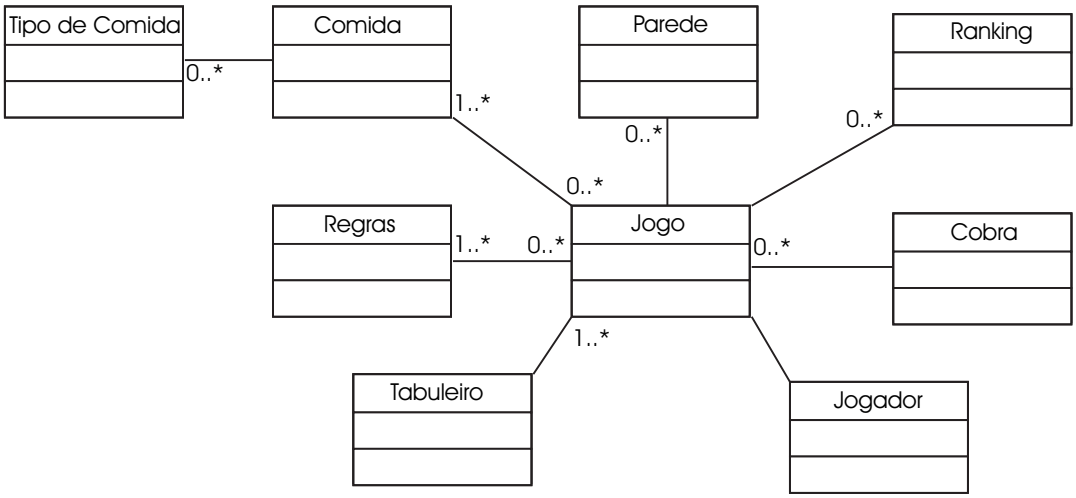

Figura 5.2: Modelo Conceitual do Negócio

O último artefato da fase de Requisitos é o Modelo de Caso de Uso, abordado de uma forma geral na Figura 5.3. Para Cada Caso de Uso são descritos o cenário principal e os cenários alternativos.

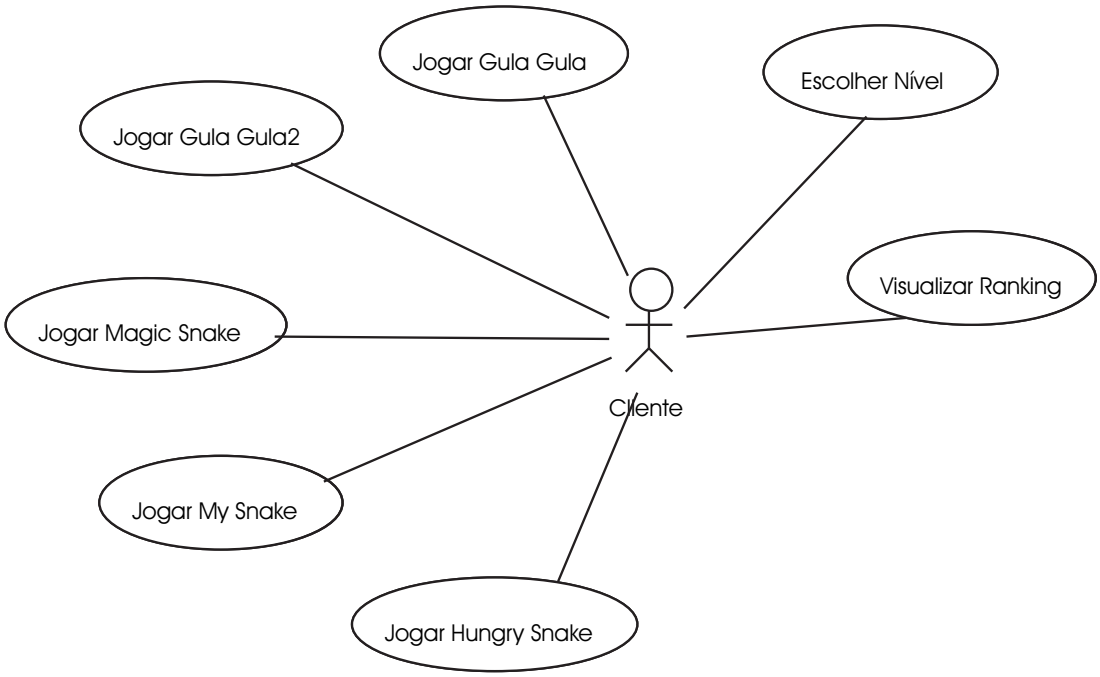

Figura 5.3: Modelo de Caso de Uso

Na Tabela 5.4 pode ser encontrada e descrição textual do Caso de Uso Jogar Hungry Snake, onde existe o cenário principal numerado de 1 à 7, representando o caso onde o jogo é iniciado e termina por atingir a pontuação máxima. Logo abaixo do cenário principal estão as extensões. A primeira extensão corresponde ao caso onde o jogo termina por a cobra bater nela mesmo. Este caso inicia na seqüência 1, 2, 3 e 4 do cenário principal e, em vez de seguir com a seqüência 5, 6 e 7 ainda do cenário principal, segue para a seqüência 5, 6 e 7 da extensão. O mesmo ocorre para a extensão seguinte, onde a cobra bate na parede e o jogo é finalizado. Nas Tabelas 5.2 e 5.3, estão representados dois Sub-Casos de Uso do Caso de Uso Jogar Hungry Snake, Controlar Direção e Cobra Come, respectivamente.

A descrição textual dos outros Casos de Uso da aplicação em questão, incluindo os cenários principal e alternativos, podem ser encontradas em anexo no Apêndice A.

Uma vez prontos os três modelos construídos nesta fase, parte-se agora para a próxima fase do processo de desenvolvimento que é a fase de Modelagem.

Sub-Caso de Uso Nº: 1

Nome: Controlar a Direção

Ator: Usuário

Objetivo: O usuário controla a direção na qual a cobra se move.

Cenário Principal:

1. Usuário solicita mudança de direção
2. A cobra se move na direção escolhida

Tabela 5.2: Caso de Uso Controlar a Direção

Sub-Caso de Uso Nº: 2

Nome: Cobra Come

Ator: Usuário

Objetivo: A cobra passa por cima de uma comida, comendo a mesma e aumentando seu tamanho.

Cenário Principal:

1. A Cobra passa por cima de uma comida.
2. A cobra aumenta de tamanho.

Tabela 5.3: Caso de Uso Cobra Come

Caso de Uso Nº: 1

Nome: Jogar Hungry Snake

Ator: Usuário

Objetivo: Jogar o Hungry Snake onde quanto mais comidas a cobra comer, maior os seus pontos. Após a cobra ter eliminado uma comida, outra comida é exibida no tabuleiro e a cobra aumenta de tamanho.

Cenário Principal:

1. O sistema cria o jogo apenas com o tabuleiro, uma comida e a cobra.
2. O jogo é iniciado.
3. Sub-Caso de Uso 1
4. Sub-Caso de Uso 2
5. A pontuação atingida é igual a 15.
6. O jogo termina.
7. O ranking é exibido.

Extensões:

5. A cobra bate nela mesma.
6. O jogo termina.
7. O ranking é exibido.

5. A cobra bate na parede.
6. O jogo termina.
7. O ranking é exibido.

Tabela 5.4: Caso de Uso Jogar Hungry Snake

5.3.2 Modelagem

Identificação do Componente

De posse do Modelo de Caso de Uso e do Modelo Conceitual do Negócio cabe agora identificar quais são as interfaces de sistema e as interfaces de negócio.

Descobrindo Interfaces do Sistema

A aplicação estudada possui 8 Casos de Uso. A princípio, poderia se pensar em elaborar uma interface para cada um deles, seguindo a regra geral. Porém, 5, dentre os 8 Casos de Uso existentes, correspondem a jogar um determinado tipo de jogo, sendo muito parecidos entre si. Por isso, foi decidido agrupar esses 5 Casos de Uso e gerar uma só interface de sistema, a qual foi chamada de *ControladorIF*. Ao analisar os passos dos Casos de Uso, são detectadas algumas responsabilidades do sistema, as quais dão origem a algumas operações da interface *ControladorIF*. Para que o jogo seja inicializado, é criada a operação *rodaJogo()*. Durante a execução do jogo o usuário pode interferir no mesmo, mudando a direção em que a cobra se move. Tal ação é atribuída a mais uma nova operação: *mudarDirecao(int direcao)*.

Mais uma vez, poderia se pensar em mais 3 interfaces de sistema para os outros 3 Casos de Uso restantes, que são: *Visualizar Ranking*, *Escolher Nível* e *Escolher Jogo*. O primeiro diz respeito a visualizar os pontos obtidos no jogo até o presente momento. Já o segundo, permite que você escolha o nível do jogo que deseja, ou seja, qual a velocidade que a cobra deve começar a andar no início do jogo, independente do tipo de jogo selecionado. O terceiro e último Caso de Uso permite que seja escolhido qual dos 5 jogos se deseja jogar. Porém, por se tratar de Casos de Uso bastante simples, onde existiria apenas uma operação em cada uma das interfaces possivelmente criadas, eles também são agrupados junto aos outros, e mais 3 operações são acrescentadas à interface *ControladorIF*: *mudarVelocidade(int n)*, *obterScore()* e *escolherJogo(String nome)*.

Para um sistema simples e de pequeno porte, como este, onde as funcionalidades acessadas pelo cliente, através das interfaces de sistema, são poucas, e não há perspectivas de que o sistema possa crescer muito, não há problemas sérios em se agrupar os Casos de Uso em uma só interface. Porém, isto não é uma solução legal caso o sistema possua um horizonte maior de crescimento. Neste caso, todas as funcionalidades acessadas pelo cliente seriam chamadas através de uma única interface, o que poderia comprometer a compreensão e o

Interfaces do Sistema	Operações	Casos de Uso
ControladorIF	escolherJogo(String nome) rodaJogo() mudarDirecao(int direcao) mudarVelocidade(int n) obterScore() escolherJogo(String nome)	Jogar Hungry Snake Jogar Gula Gula Jogar Gula Gula2 Jogar Magic Snake Jogar My Snake Visualisar Ranking Escolher Nível Escolher Jogo

Tabela 5.5: Interfaces do Sistema, operações e Casos de Uso

crescimento, de forma organizada, do sistema.

Na Tabela 5.5, pode ser encontrado um resumo da Interface do Sistema identificada, contendo suas operações e os Casos de Uso relacionados.

Descobrindo Interfaces do Negócio

Para descobrir as interfaces de negócio, são seguidos os seguintes passos:

- Criação de um Modelo de Tipos de Negócio ;
- Refinamento do Modelo de Tipos de Negócio;
- Especificação de regras de negócio;
- Identificação dos tipos centrais do negócio;
- Criação de interfaces de negócio para cada tipo central do negócio e inclusão das mesmas ao modelo de tipos do negócio, dando origem a um diagrama de responsabilidades de interface.

O Modelo de Tipo do Negócio, apresentado na Figura 5.4, é criado a partir do Modelo Conceitual do Negócio, retirando do mesmo a entidade Jogador, uma vez que o intuito agora é representar as informações do negócio que o sistema deve especificar e Jogador não faz parte deste negócio. Em seu lugar é colocada a entidade Controlador, a qual fará

o controle de todos os comandos enviados pelo usuário, servindo de comunicação entre as ações(funcionalidades) requeridas pelo cliente e a lógica do negócio. Também é criada uma

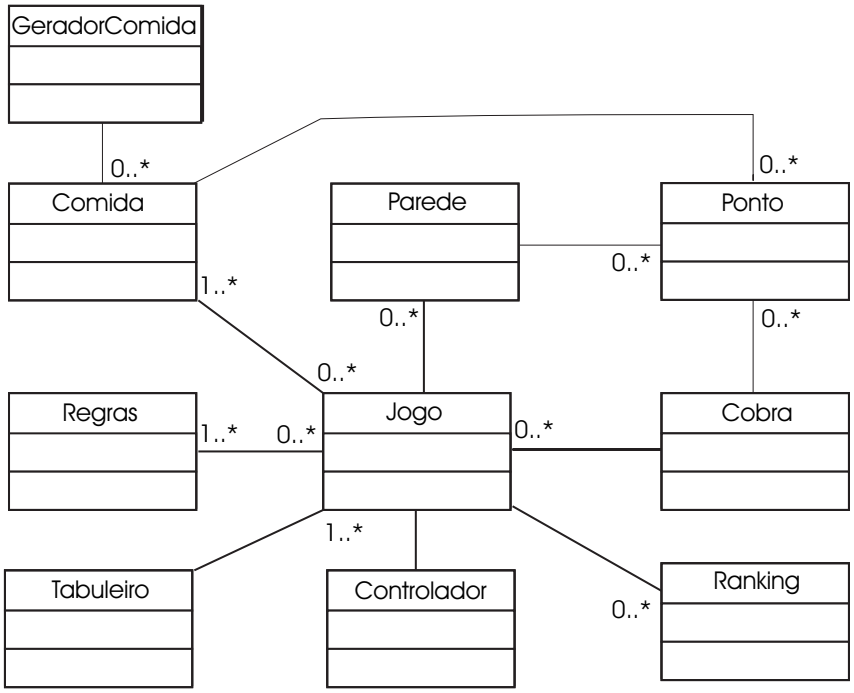

Figura 5.4: Modelo de Tipo de Negócio

outra entidade chamada Ponto, com a finalidade de ser responsável pelas coordenadas dos pontos gerados para a cobra, comida e parede. Em seguida, são acrescentadas algumas regras de multiplicidade entre as entidades, a fim de entender um pouco mais como elas se relacionam. Logo mais, são identificadas quais informações independem de outras para existirem, ou seja, são identificados os seguintes *core types*: Comida, Regras, Tabuleiro, Ranking, Cobra, Parede e Ponto. Para cada *core type* é criada uma interface de negócio, dando origem as seguintes interfaces: ControleComidaIF, ControleRegrasIF, ControleTabuleiroIF, ControleRankingIF, ControleCobraIF, ControleParedeIF e ControlePontoIF. Apesar de Jogo não ser um *core type*, surge a necessidade de acrescentar uma nova interface, JogoIF, a qual é responsável por controlar os diferentes tipos de jogos. O mesmo acontece com GeradorComida, que apesar de não ser um *core type*, surge a necessidade de uma interface para para controlar a geração de comidas no tabuleiro. Dessa forma, é criada a interface GeradorComidaIF. Adicionando essas interfaces ao Modelo de Tipo de Negócio, é gerado um novo diagrama, chamado de Diagrama de Responsabilidades de Interface apresentado na Figura 5.5.

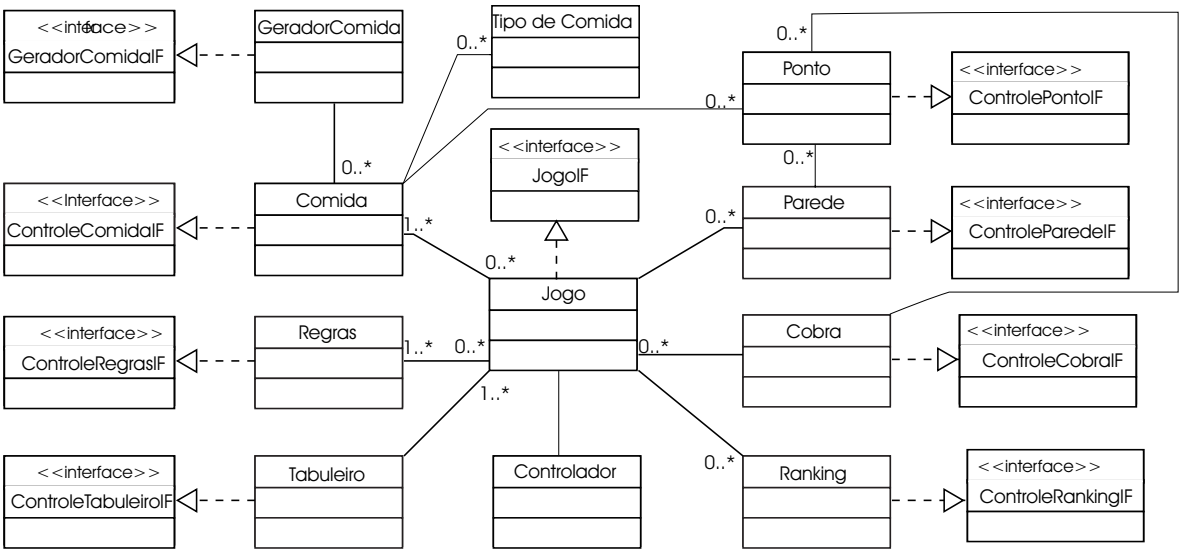

Figura 5.5: Modelo de Responsabilidade de Interfaces

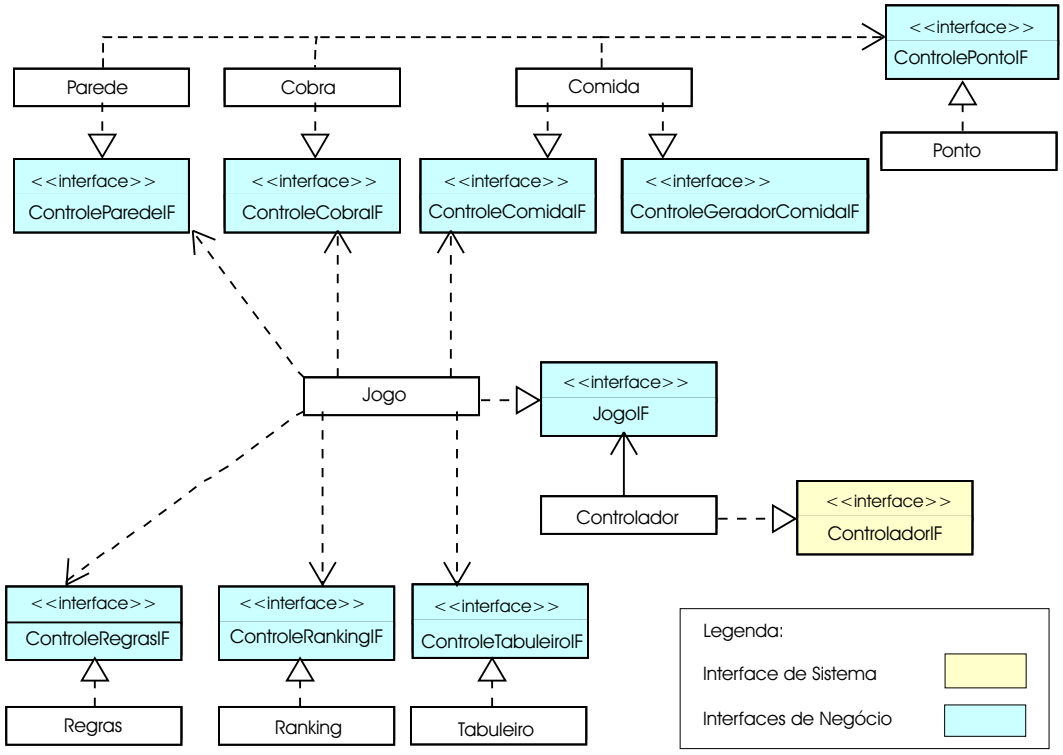

Figura 5.6: Arquitetura Inicial dos Componentes

Tendo identificado, até então, as interfaces de sistema e as interfaces de negócio, já se consegue obter uma Arquitetura Inicial dos Componentes, como pode ser observado na Figura 5.6.

Com o intuito de se entender melhor a finalidade de cada componente para que se possa dar continuidade ao estudo de caso, é fornecida uma breve introdução sobre o objetivo básico e as principais funções, de cada componente, no contexto da aplicação. A seguir, encontra-se a descrição de cada um dos componentes:

Cobra O componente Cobra é responsável pela criação de uma cobra normal ou de uma cobra mágica. A diferença entre as duas cobras é que a mágica pode ficar indestrutível por um certo período de tempo. O componente possui a interface *ControleCobraIF*, a qual fornece suas principais funcionalidades, que são: fazer a cobra se mover, e aumentar seu tamanho.

Comida O objetivo principal do componente Comida é criar diferentes tipos de comidas, que pode ser simples, especial ou estragada e, fazer com que essas comidas sejam geradas no tabuleiro. A diferença entre essas comidas é a pontuação obtida quando a cobra come uma delas. A comida simples vale 1 ponto, a especial vale 2 pontos e estragada vale -2 pontos. Existem duas interfaces responsáveis por oferecer as funcionalidades previstas pelo componente: *ConrtoleComidaIF* que controla a criação dos tipos de comidas que são criadas e, *ControleGeradorComidaIF* que controla a geração dessas comidas no tabuleiro. Para cada tipo de comida existe uma forma especial de se gerar comida no tabuleiro, por exemplo: a comida simples pode ser gerada em um intervalo de tempo determinado em qualquer lugar no tabuleiro, já a comida estragada é sempre gerada no final da cobra, como se fosse um bolo fecal produzido pela mesma.

Parede Ao componente Parede, cabe a criação de paredes, as quais servem de obstáculo para a cobra. Estas paredes podem aparecer no tabuleiro em posições e formas diferentes. Os obstáculos podem se apresentar de 2 formas: como 4 paredes, sendo cada uma delas paralela a um dos lados do tabuleiro e um pouco menor que o seu comprimento; ou como sendo um mini-labirinto. A cobra deve andar tendo que se desviar

desses obstáculos. A interface *ControleParedeIF* é responsável por gerar as paredes no tabuleiro, de maneira correta, de acordo com o tipo de obstáculo escolhido.

Ranking O componente Ranking gerencia os pontos adquiridos pelos jogadores, sendo responsável por retornar, ao fim de cada jogo, o ranking dos 5 maiores recordes. A interface *ControleRankingIF* é responsável por fornecer tais funcionalidades.

Tabuleiro O principal objetivo do componente Tabuleiro é gerar o tabuleiro por onde a cobra irá trafegar, de acordo com o tamanho fornecido. A interface que oferece esta funcionalidade é a *ControleTabuleiroIF*.

Regras Regras é um dos principais componentes do sistema. Ele é responsável por checar se a cobra bateu na parede, se a cobra bateu nela mesmo, se a cobra comeu alguma comida e, se o jogo chegou ao fim. Se alguma dessas ações acontecer, devem ser aplicadas as regras correspondentes ao jogo em questão. Por exemplo, se a cobra comer uma comida ao estar jogando o jogo Hungry Snake, deve ser somado 1 ponto na pontuação do jogador. Porém, se o mesmo acontecer durante a execução do jogo Magic Snake, antes de acrescentar um ponto ao placar do jogador, deve ser checado se a comida que a cobra comeu é do tipo especial ou normal, porque caso seja especial deverá se atribuir 2 pontos ao placar do jogador invés de 1. A interface *ControleRegrasIF* é responsável por fornecer essas funcionalidades.

Jogo O componente Jogo é o carro-chefe do sistema. Ele tem a finalidade de criar o jogo requisitado e, definir, para cada tipo de jogo, qual cobra, parede e tipos de comidas, devem ser criados. Também é responsável por fazer o jogo ser inicializado e continuar sendo executado até que algum fato determinante ocorra para que o mesmo seja finalizado. Para obter todas essas funcionalidades foi definida a interface JogoIF.

Controlador Controlador é o único componente de sistema, responsável por controlar a interação do cliente com o restante do sistema. Ele recebe as solicitações, de mais alto nível, do cliente e repassa as responsabilidades para os demais componentes. O cliente pode, através de ControladorIF, escolher o Jogo que deseja jogar, inicializar o jogo, mudar a direção da cobra, mudar o nível do jogo, ou seja, a velocidade com a qual a cobra anda e, obter os pontos adquiridos pelo jogador.

Ponto Ponto é um componente por criar e gerenciar as coordenadas dos pontos criados para a cobra, comidas e paredes.

Interação do Componente

Tendo definido o conjunto de componentes que deve se trabalhar, é hora de partir para uma nova fase, ou seja, decidir como deve ser a interação entre esses componentes de forma a fornecer a funcionalidade requerida por cada um deles. Para isto, são construídos diagramas de colaboração para cada operação da interface de sistema, que são as seguintes: *escolherJogo(String nome), rodaJogo(), mudarDirecao(int direcao), mudarVelocidade(int n), obterScore()*. À medida em que são construídos os diagramas, são descobertas algumas operações das interfaces de negócio. No diagrama de colaboração da operação *escolherJogo(String nome)*, nenhuma operação nova é criada, apenas são criadas instâncias de alguns componentes, como pode ser observado na Figura 5.7. Na construção do diagrama de colaboração da operação *mudarDirecao()*, representado na Figura 5.8, é criada a operação *setDirecao(int direcao)* na interface de negócio do componente Jogo. Da mesma forma, no diagrama de colaboração da operação *mudarVelocidade()*, representado na Figura 5.9, é criada a operação *setVelocidade(int velocidade)*, também para a interface de negócio do componente Jogo. Na Figura 5.10, encontra-se o diagrama da operação *obterScore()*, onde similarmente, é descoberta a operação getScore() para a interface de negócio do componente Jogo. Já no diagrama da operação *rodaJogo()*, ilustrado na Figura 5.11, são criadas as operações *getStatus()* e *run()* para a interface de negócio do componente Jogo e, as operações *testeCobraComeu(), testeCobraBateuParede(), testeCobraBateuNelaMesmo(), jogar()* e *testeFimJogo()* para a interface de negócio do componente Regras.

Especificação do Componente

Esta é a última etapa da fase de Modelagem. É aqui onde são definidos os Contratos de Uso e de Realização.

O Contrato de Uso é definido pela Especificação das Interfaces, a qual foi constituída dos seguinte ítens: Especificação de operações, que inclui parâmetros de entrada, parâmetros de saída, pré e pós-condições expressas em OCL e, os Modelos de Informação de cada componente. As especificações das operações *escolherJogo(), rodaJogo(), mudarDirecao(),*

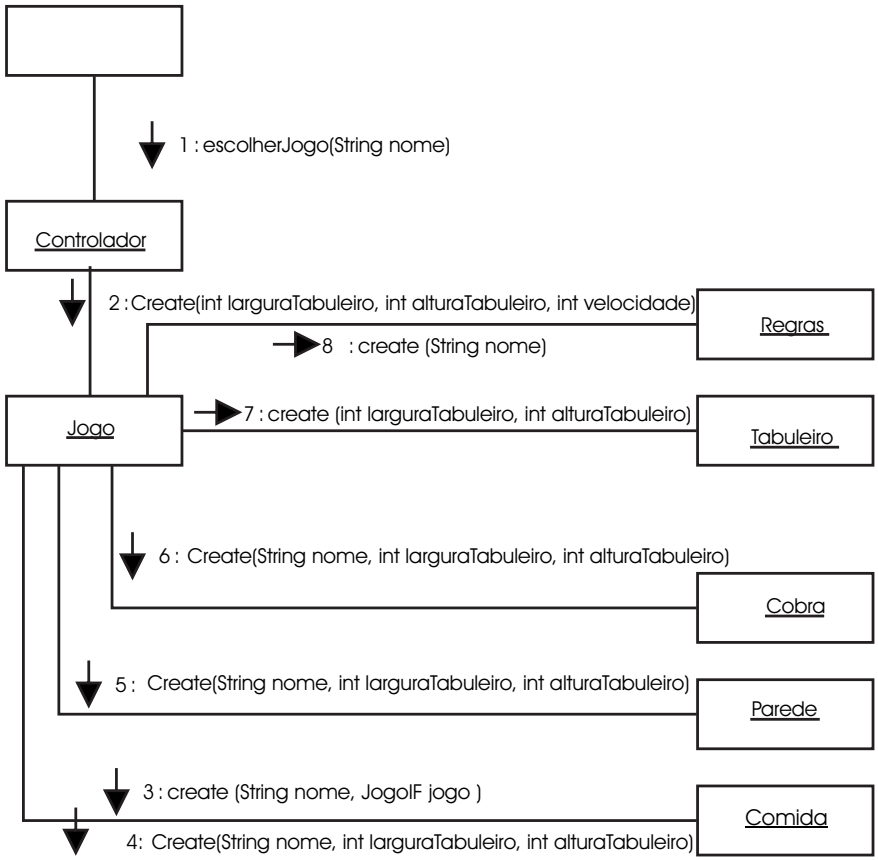

Figura 5.7: Diagrama de Colaboração da operação escolherJogo() da interface ControladorIF

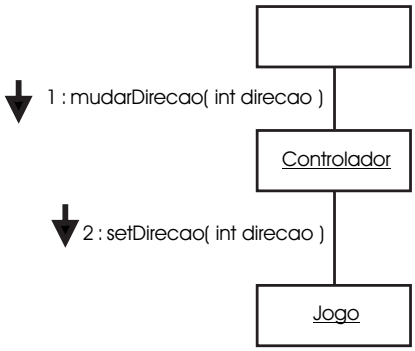

Figura 5.8: Diagrama de Colaboração da operação mudarDirecao() da interface ControladorIF

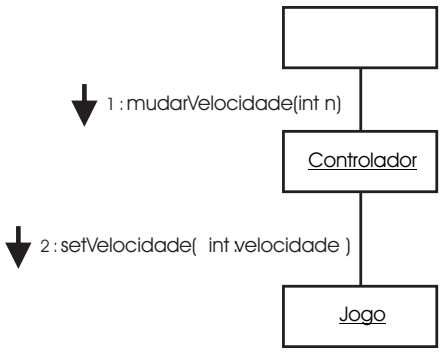

Figura 5.9: Diagrama de Colaboração da operação mudarVelocidade() da interface ControladorIF

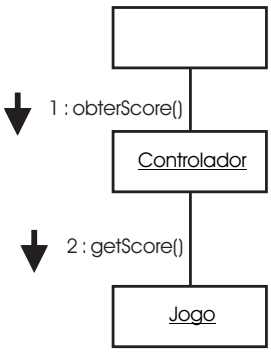

Figura 5.10: Diagrama de Colaboração da operação obterScore() da interface ControladorIF

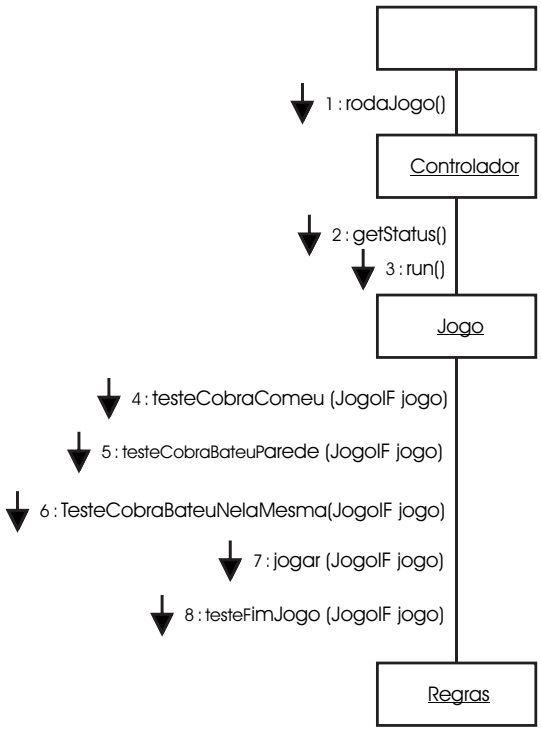

Figura 5.11: Diagrama de Colaboração da operação rodaJogo() da interface ControladorIF

mudarVelocidade() e *obterScore()* da interface *ControladorIF*, podem ser encontradas nas seguintes Figuras: 5.12, 5.13, 5.14, 5.15 e 5.16, respectivamente.

```
Operação:         public boolean escolherJogo(String nome);
Descrição:        permite escolher o jogo que se deseja jogar
Entradas:         nome: String  o nome do jogo que deseja se jogar
Saídas:
Pré-Condições:    o nome é uma string válida
Pós-Condições:    o jogo é escolhido com sucesso
OCL:              context Controlador :: escolherJogo(nome: String): Boolean
                  pre: nome = "MySnake" or nome  = "Gula" or nome  = "Gula2"
                       or nome  = "MagicSnake" or nome  = "HungrySnake"
                  post: result = true
```

Figura 5.12: Especificação da operação escolherJogo(String nome) do componente Controlador

```
Operação:         public void mudarDirecao( int direcao );
Descrição:        muda a direção que a cobra anda
Entradas:         direção: int - DIRECAO_CIMA = 38; DIRECAO_BAIXO = 40
                  DIRECAO_ESQUERDA = 37; DIRECAO_DIREITA = 39
Saídas:
Pré-Condições:    O jogo deve estar rodando e
                  a direção informada deve ser um valor válido.
Pós-Condições:    A cobra passa a se mover na nova
                  direção passada como parâmetro.
OCL:              context Controlador :: mudarDirecao(direcao:Integer )
                  pre: getStatus() = true and
                       ( (direcao  = 37) or (direcao  = 38) or
                       (direcao  = 39) or (direcao  = 40) )
                  Post: getJogo().getDirecao() = direcao
```

Figura 5.13: Especificação da operação rodaJogo() do componente Controlador

No Apêndice F, podem ser encontrados maiores detalhes sobre as especificações das principais operações, de cada componente, bem como sua estrutura interna.

O Modelo de Informação do componente Cobra está ilustrado na Figura 5.17. Neste modelo se encontram as operações que fazem parte do componente Cobra e que podem ser acessadas por qualquer outro componente, ou pelo sistema. Neste modelo, bem como nos outros, podem ser encontradas outras operações nas Interfaces de Negócio, além das identificadas anteriormente, durante a construção dos diagramas de colaboração. Essas operações vão surgindo à medida em que aparece a necessidade de se obter uma nova função, que seja necessária sua existência, para completar alguma funcionalidade. Por exemplo: as operações *crescer(), getCabeca()* e *moverCobra()* da Interface *ControleCobraIF*, precisam existir para que a cobra cresça ao comer uma determinada comida disposta no tabuleiro, apesar de não serem identificadas durante a elaboração dos Diagramas de Colaboração.

Os Modelos de Informação referente às demais interfaces: ControladorIF, Controle-

```
Operação:         public void mudarDirecao( int direcao );
Descrição:        muda a direção que a cobra anda
Entradas:         direção: int - DIRECAO_CIMA = 38; DIRECAO_BAIXO = 40
                  DIRECAO_ESQUERDA = 37; DIRECAO_DIREITA = 39
Saídas:
Pré-Condições:    O jogo deve estar rodando e
                  a direção informada deve ser um valor válido.
Pós-Condições:    A cobra passa a se mover na nova
                  direção passada como parâmetro.
OCL:              context Controlador :: mudarDirecao(direcao:Integer )
                  pre: getStatus() = true and
                       ( (direcao  = 37) or (direcao  = 38) or
                       (direcao  = 39) or (direcao  = 40) )
                  post: getJogo().getDirecao() = direcao
```

Figura 5.14: Especificação da operação mudarDirecao(int direcao) do componente Controlador

```
Operação:         public void mudarVelocidade(int n);
Descrição:        muda a velocidade que a cobra anda
Entradas:         n: int - valor da nova velocidade
Saídas:
Pré-Condições:    O jogo deve estar rodando
                  e a velocidade deve ser maior que zero.
Pós-Condições:    A cobra passa a se mover com
                  uma nova velocidade.
OCL:              context Controlador :: mudarVelocidade( n : Integer )
                  pre:  getStatus() = true and n > 0
                  post: getJogo().getVelocidade() = n
```

Figura 5.15: Especificação da operação mudarVelocidade(int n) do componente Controlador

```
Operação:         public int obterScore();
Descrição:        obtém a pontuação atual
Entradas:
Saídas:           número atual de pontos acumulados
Pré-Condições:    O jogo deve estar rodando
Pós-Condições:
OCL:              context Controlador :: obterScore()
                  Pre: getStatus() = true
```

Figura 5.16: Especificação da operação obterScore() do componente Controlador

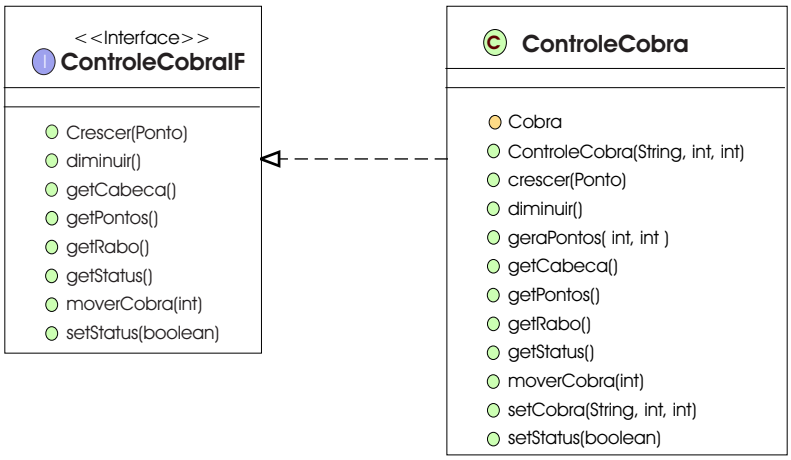

Figura 5.17: Modelo de Informação da interface do Componente Cobra

ComidaIF, ControleParedeIF, ControleTabuleiroIF, ControleRankingIF, ControleRegrasIF, ControlePontoIF e JogoIF, encontram-se em anexo no Apêndice B.

Para definição do Contrato de Realização são construídos diagramas de seqüência, a fim de representar as interações entre as classes que fazem parte dos componentes. Assim sendo, para cada cenário de cada Caso de Uso, deveria ser construído diagramas que representam o funcionamento interno de cada componente. Porém, visto que se tem 8 Casos de Uso e, em média, 4 cenários para cada Caso de Uso, iria se ter em torno de 32 diagramas para cada Caso de Uso. Como se tem 9 componentes, seria uma média de 288 diagramas de seqüência. Dessa forma, seria construído um número consideravelmente grande de diagramas, além do fato de existir uma grande similaridade entre eles. Sendo assim, para que não houvesse um trabalho demasiadamente grande e repetitivo, foram construídos diagramas de seqüência apenas para 2 dos 8 Casos de Uso existentes, ou seja, para os Casos de Uso Jogar Hungry Snake e Jogar Gula Gula, uma vez que os cenários referentes aos Casos de Uso Jogar Gula Gula 2, Jogar Magic Snake e Jogar My Snake são bastante parecidos. Neste estudo de caso, o número de diagramas de seqüência pôde ser reduzido sem maiores problemas, uma vez que o intuito da aplicação é demonstrar a aplicabilidade do método e os casos de uso são bastante parecidos. Contudo, para outros tipos de sistemas, a redução dos diagramas de seqüência que precisam ser feitos, pode acarretar limitações nos casos de teste que deveriam ser desenvolvidos.

Vale salientar que, do ponto de vista da aplicação, os Casos de Uso retratam cenários de uma forma ampla, representando o jogo inteiro. Para os componentes esta visão fica restrita a cenários menores, representando possíveis interações do jogo.

Como o foco do trabalho não é o teste individual de componentes e sim o teste da integração entre os componentes, são escolhidos alguns componentes para ilustrar o processo de teste dos componentes individualmente. Dentre os componentes existentes, são selecionados aqueles que possuem mais funcionalidades e se destacam na aplicação, que são: Jogo e Regras. Isto não quer dizer que os demais componentes não devam ser testados, mas que apenas esses dois encontram-se ilustrados no trabalho. Dessa forma, os artefatos construídos deste ponto em diante se referem a esses dois componentes.

Para todos os diagramas de seqüência desenvolvidos, existe um cenário inicial que é comum a todos, por isso, ele foi representado num diagrama a parte e se encontra ilustrado na

Figura 5.18. Neste Cenário Inicial, o componente Controlador cria uma instância do jogo

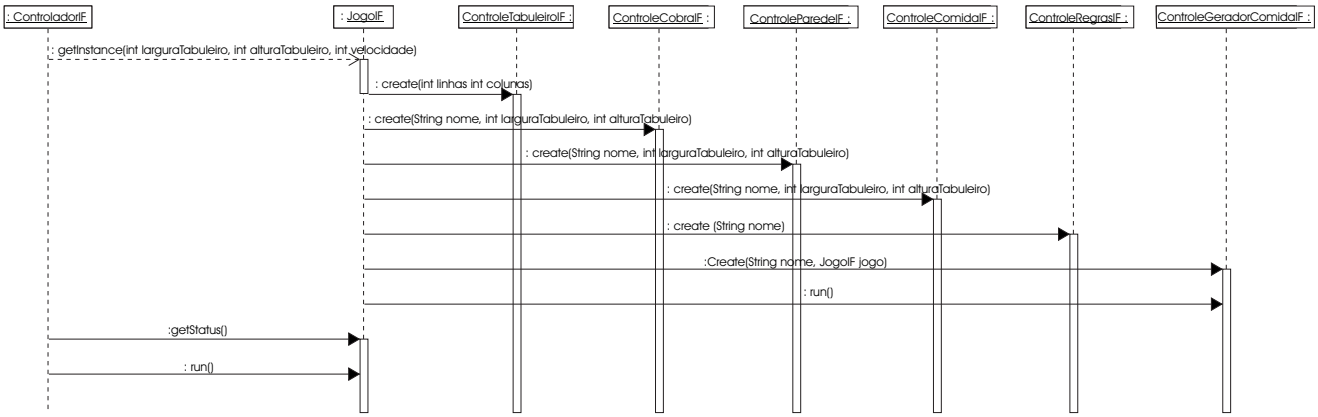

Figura 5.18: Cenário Inicial

que for escolhido. O componente Jogo, por sua vez, interage com os outros componentes, criando um tabuleiro, uma cobra, uma parede, um ou mais tipos de comida, dependendo do jogo em questão, um gerador de comidas e um conjunto de regras, as quais devem variar também, dependendo do tipo de jogo. Em seguida, o componente Jogo chama o método *run()* do componente Comida, para que seja iniciada a *thread* de geração de comidas. Logo mais, o componente Controlador verifica se o status do jogo é *true*, através da chamada do método *getStatus()* do componente Jogo e, chama o método *run()*, também do componente Jogo, para que ele possa ser inicializado. Estas duas últimas chamadas ficam sendo repetidas até que o status do jogo mude para *false* e o Jogo seja finalizado. Isto está representado nos outros diagramas, o quais representam diferentes cenários dos Casos de Uso. Para o diagrama ilustrado na Figura 5.19, tem-se o cenário onde o jogo termina por ter obtido a pontuação máxima, para o Caso do Uso Jogar Hungry Snake, referente ao Componente Jogo. Neste caso, após a chamada do método run() ao componente Jogo, visto no cenário inicial, a classe *JogoHungrySnake*, que faz parte do componente Jogo, chama o método *testeCobraComeu(JogoIF jogo)*, *testeCobraBateuParede(JogoIF jogo)* e *jogar(JogoIF jogo)* do componente Regras, para verificar se a cobra comeu alguma comida, se a cobra bateu na parede e para realizar mais uma jogada, respectivamente. Logo mais, o Componente Jogo interage com o componente Cobra ao chamar o método *moverCobra(int direcao)*, para fazer com que a cobra se movimente na direção indicada. Depois, o componente Jogo chama o método *testeFimJogo(JogoIF jogo)* do componente Regras, para verificar se o jogo atingiu uma

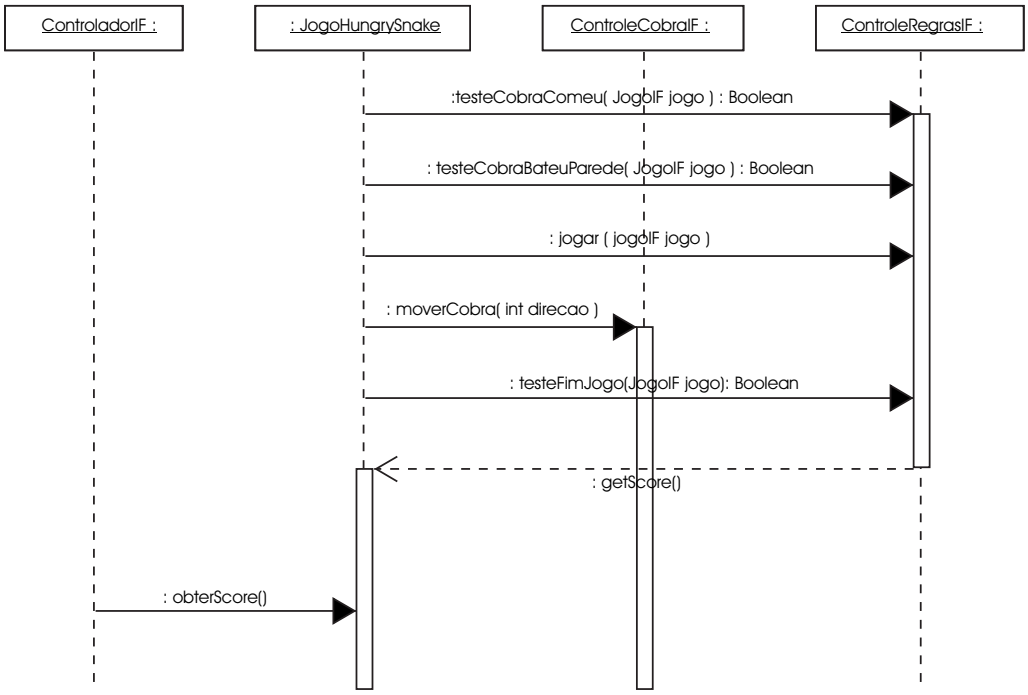

Figura 5.19: Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde o jogo termina por ter obtido a pontuação máxima, referente ao Componente Jogo

pontuação máxima. Para realizar esta verificação, o componente Regras precisa saber quantos pontos foram obtidos até então. Para isto, é feita uma chamada ao método *getScore()* do componente Jogo. Como sempre que o jogo é finalizado, o ranking é exibido, o componente Controlador chama o método *obterScore()* do componente Jogo, para saber a pontuação final obtida.

5.3.3 Fornecimento, Montagem e Testes

Neste ponto foi decidido reutilizar o código-fonte de uma aplicação já existente. Porém, foi preciso refatorá-la, pois a mesma não se encontrava componentizada. Sendo assim, o código foi reestruturado, de forma que os componentes foram todos gerados. O processo de teste se encontra detalhado no Capítulo 6.

5.4 Conclusões

Neste capítulo foi apresentado o desenvolvimento do estudo de caso realizado, ilustrando passo a passo cada etapa seguida: Requisitos, Modelagem, Fornecimento e Montagem. A etapa de teste, não foi detalhada neste capítulo, uma vez que nela encontra-se ilustrado todo o método de teste de integração proposto e, o qual se encontra detalhado no capítulo posterior.

Capítulo 6

Estudo de Caso - Aplicação do Método de Teste de Integração

O Método de Teste de Integração proposto é desenvolvido paralelamente ao desenvolvimento dos componentes. O mesmo, se encontra contextualizado dentro das seguintes etapas: Planejamento, Especificação, Construção, Execução e Análise dos Resultados. A seguir será ilustrado o que foi realizado em cada uma dessas etapas. A Seção 6.1 descreve a etapa de Planejamento; a Seção 6.2 corresponde a etapa de Especificação; a Seção 6.3 diz respeito a etapa de Construção, Execução e Análise dos Resultados; e na Seção 6.4 estão as conclusões.

6.1 Planejamento

O planejamento dos testes é realizado durante a definição dos requisitos. Para se planejar os testes é necessário se ter idéia da dimensão das tarefas de testes que se pretende realizar. Para isto são respondidas as seguintes perguntas:

Quem executará os testes? Por se tratar de uma aplicação de pequeno porte, a idéia é que o próprio desenvolvedor assuma o papel de testador.

Que partes serão testadas? O objetivo é testar apenas uma parte do sistema, já que testar tudo pode se tornar inviável do ponto de vista prático, não havendo recursos suficientes para tal. Dessa forma, é feita uma seleção de alguns casos de teste, de acordo com a técnica TOTEM descrita em [Briand and Labiche, 2001].

Quando serão executados os testes? Os testes devem ser realizados ao longo do processo de desenvolvimento, reduzindo assim, a probabilidade de encontrar erros apenas no final do desenvolvimento, o que aumentaria os custos do projeto. Porém, como o código já se encontrava implementado, havendo a necessidade apenas de refatorá-lo, os testes são realizados durante o processo de desenvolvimento restante, ou seja, durante a construção de toda a especificação.

Como os testes serão realizados? Os testes são realizados de acordo com o método de teste proposto neste trabalho, o qual é baseado na especificação, uma vez que é iniciado desde a fase de levantamento de requisitos.

Quanto devemos testar? Desde esse ponto, é decidido que as funcionalidades devem ser testadas de forma parcial, obedecendo um número viável de casos de teste, de acordo com a análise de risco, que é o próximo passo realizado.

6.1.1 Construção da Análise de Risco

Ela deve ter níveis suficientes para separar os casos de uso em grupos de tamanho razoável.

A Tabela 6.1 representa a Análise de Risco para os Casos de Uso do sistema.

Caso de Uso	Freqüência	Criticalidade	Grau de Risco	N ^{o} de Casos de Teste
Jogar Hungry Snake	Alta	Alta	Alta	3
Jogar Gula Gula	Alta	Alta	Alto	3
Jogar Gula Gula 2	Alta	Alta	Alto	3
Jogar Magic Snake	Alta	Alta	Alto	3
Jogar My Snake	Alta	Alta	Alto	3
Escolher Nível	Médio	Médio	Médio	2
Visualizar Ranking	Médio	Baixo	Médio	2

Tabela 6.1: Análise de Risco para cada Caso de Uso

Para separar os Casos de Uso em grupos de tamanho razoável, são definidos 3 níveis para se medir o grau de risco de cada Caso de Uso: Alto, Médio e Baixo. Para cada Caso de Uso, é analisada a freqüência e a criticalidade com que cada um pode ocorrer. O grau

de risco é definido através de uma análise da freqüência e da criticalidade. Por exemplo: para o primeiro Caso de Uso que é Jogar Hungry Snake, tem-se uma freqüência e uma criticalidade classificada como Alta, portanto, o grau de risco também é Alto. Já no Caso de Uso Visualizar Ranking, tem-se uma freqüência Média e uma criticalidade Baixa. Nesse caso, o maior nível é escolhido como grau de risco, ou seja, Médio. Uma vez classificados os graus de riscos, devem ser determinados os números de Casos de Teste para cada grau de risco. Como foram detectados três níveis de riscos, é determinado um certo número de Casos de Teste para cada nível. Assim sendo, para os Casos de Uso com grau de Risco Alto, é definido que três seja um número razoável de Casos de Teste a serem construídos. Da mesma forma, aqueles com grau de risco Médio têm dois Casos de Teste. E se tivesse algum com grau de risco baixo teria apenas um Caso de Teste, mas não é o caso.

6.2 Especificação

Esta etapa é realizada durante a fase de Modelagem do processo de desenvolvimento. Ela envolve 3 atividades principais que são: escolha da ordem de teste dos componentes e dos *stubs* necessários, geração e seleção dos casos de teste e, geração dos dados e oráculos de teste. A seguir, será demonstrado o que foi realizado em cada uma delas.

6.2.1 Escolha da ordem de teste dos componentes e dos *stubs* necessários

Esta etapa tem a finalidade de determinar quantos, quais e em que ordem os *stubs* devem ser construídos. Para isto, é construído o GTD do Snake Game, baseado nos Modelos de Informação e nos Diagramas de Colaboração criados na fase de Especificação do processo de desenvolvimento. Como pode ser observado na Figura 6.1, no GTD estão representados apenas 4 componentes: Controlador, Jogo, Regras e Cobra. Isto não quer dizer que os demais componentes não possuam integração com os outros, mas que no nível de abstração do GTD, eles não conseguiram ser representados.

Após construído o GTD, são aplicadas ao mesmo, algumas regras de normalização, onde os nodos que representam as classes: Jogo, Controlador, Cobra e Regras, são separados dos

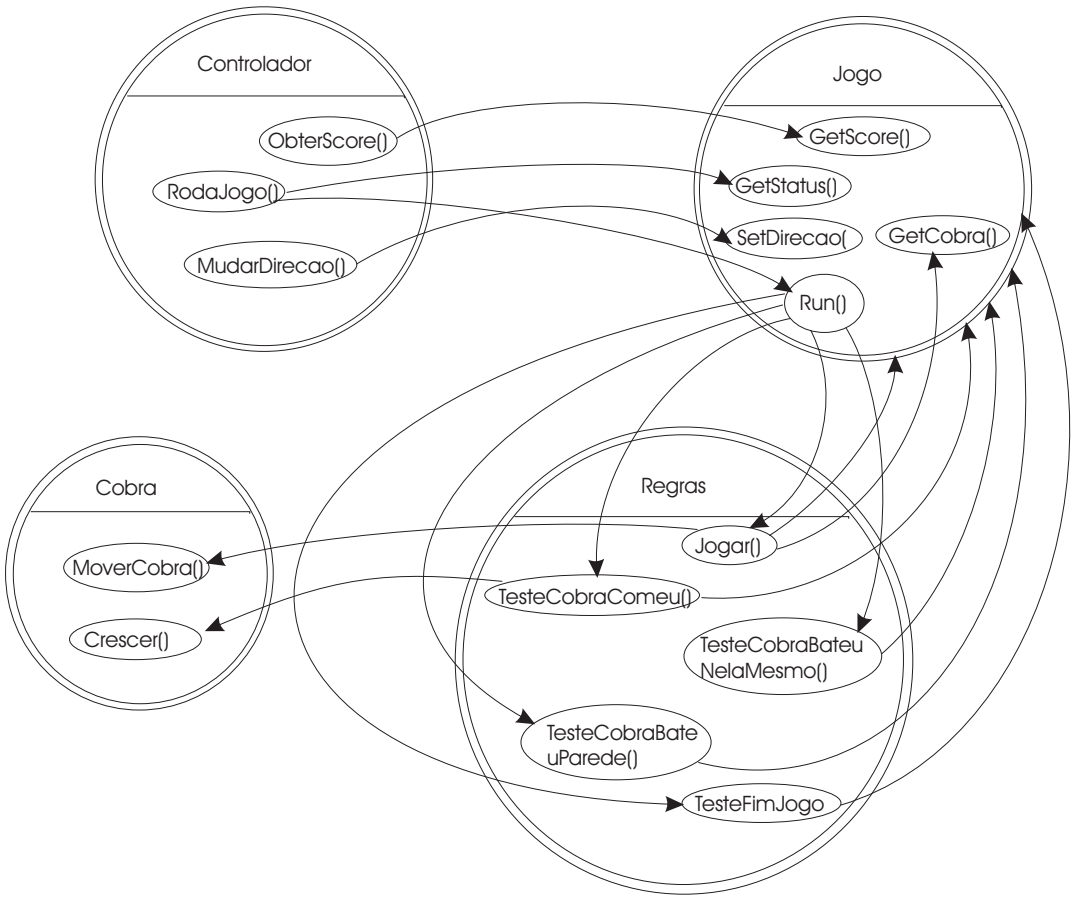

Figura 6.1: GTD do estudo de caso

nodos que representam os métodos, que são os nodos restantes. Após normalizado, o grafo apresentado na Figura 6.2, se encontra pronto se para aplicar o algoritmo demonstrado em [Traon et al., 1999].

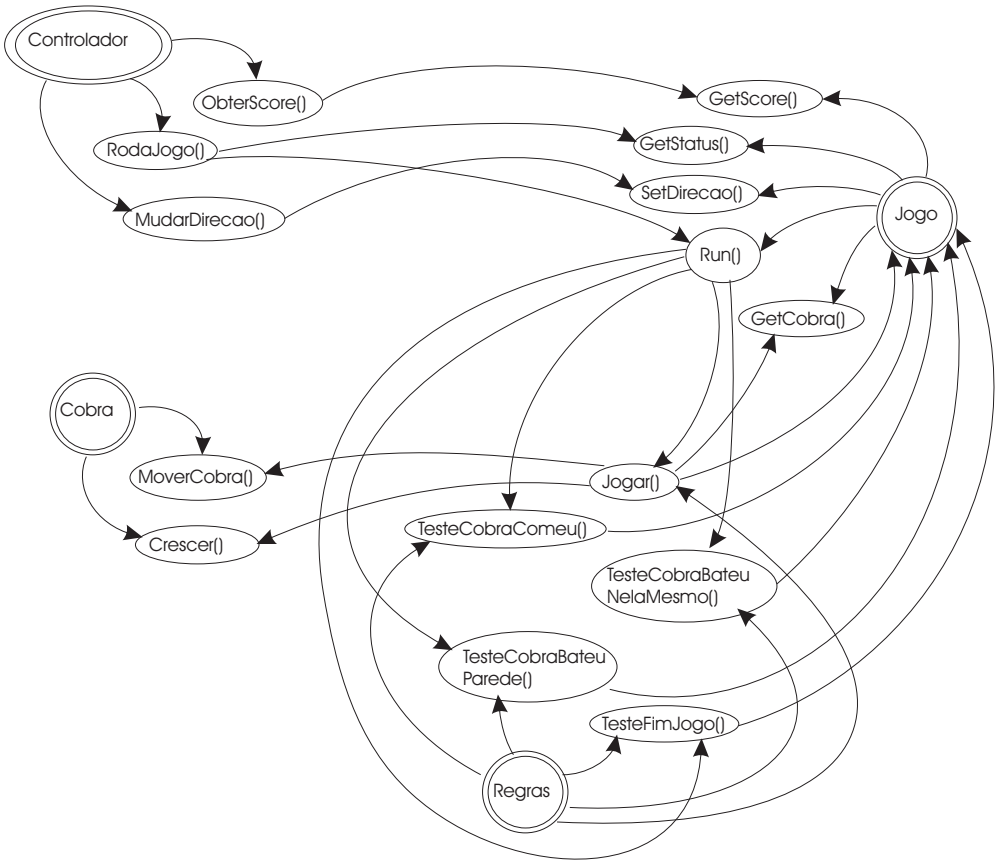

Figura 6.2: GTD normalizado

Uma vez aplicado o algoritmo, é obtido o grafo demonstrado na Figura 6.3, onde as ligações com linhas tracejadas e com setas vazadas indicam os arcos do tipo Frond, as ligações com linhas normais e setas pintadas representam os arcos do tipo Tree e, as ligações com linhas tracejadas e pontilhadas com setas pintadas representam os arcos do tipo Cross. Vale salientar que neste exemplo não é encontrado nenhum arco do tipo Forward.

Na primeira chamada ao algoritmo é identificado apenas um CFC não trivial, ou seja, apenas uma parte do grafo é cíclica, como pode se observar na Figura 6.4.

Encontrado o CFC, é determinado qual vértice possui o maior número de entradas e saídas de arcos do tipo Frond. Como apenas um arco do tipo Frond pode ser encontrado no GTD, então os vértices de número 6 e 8 são fortes candidatos. Sendo assim, o vértice de número 8 é escolhido aleatoriamente. Numa chamada recursiva do algoritmo no CFC

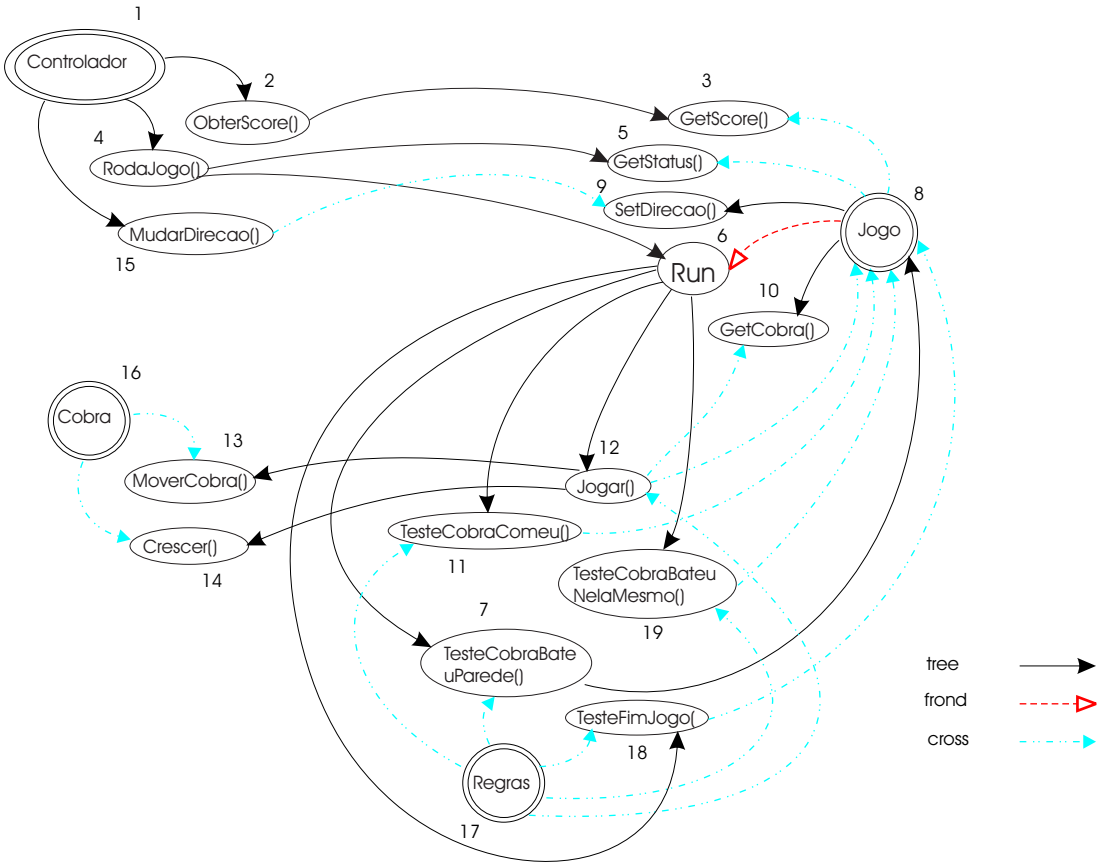

Figura 6.3: algoritmo aplicado ao GTD

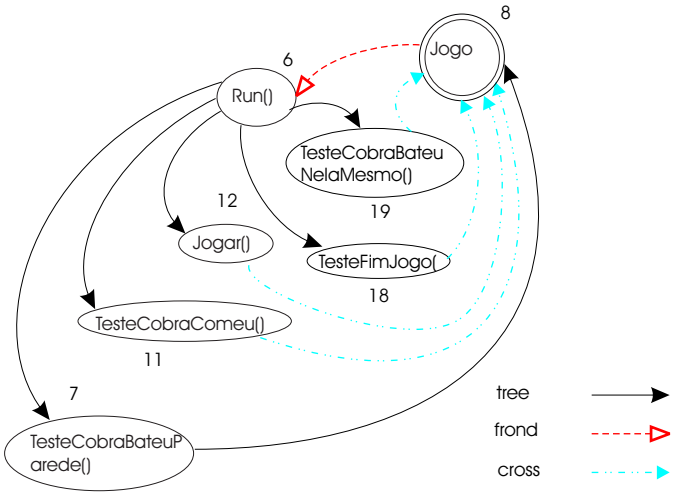

Figura 6.4: CFC do GTD

encontrado, a partir do vértice 8, os arcos 12 → 8, 11 → 8 e 7 → 8 são excluídos e o grafo se torna acíclico, como pode ser observado na Figura 6.5. Sendo assim, o algoritmo não precisa ser aplicado novamente, uma vez que não se tem mais nenhuma parte cíclica no grafo.

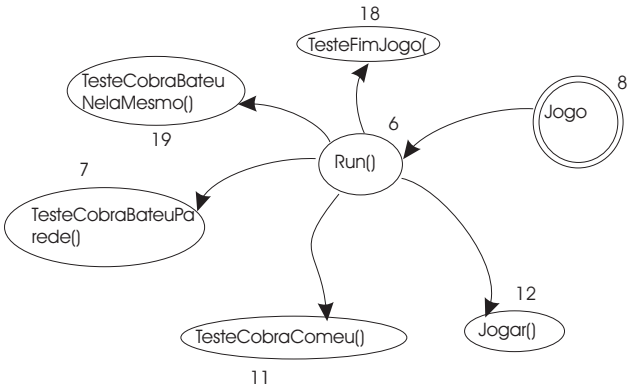

Figura 6.5: Algoritmo aplicado recursivamente ao CFC

Em termos de teste, de acordo com os grafos gerados, é fornecida a seguinte estratégia:

- 3 é testado,
- 5 é testado,
- 9 é testado,
- 10 é testado,
- 15 é testado usando 9,
- 2 é testado usando 3,
- 13 é testado,
- 14 é testado,
- 16 é testado usando 13 e 14,

Para o CFC 6,7,8,11,12,19,18, temos:

- 7 é testado usando *stub*(8),
- 12 é testado usando 13, 14, 10, e *stub*(8),

- 11 é testado usando *stub*(8),
- 19 é testado usando *stub*(8),
- 18 é testado usando *stub*(8),
- 6 é testado usando 7, 11, 12,
- 8 é testado usando 6.

Por fim:

- 17 é testado usando 18, 19, 12, 7, 11,
- 4 é testado usando 5 e 6,
- 1 é testado usando 4, 2, 15.

Como se pode observar, apenas um *stub* é detectado, o que deixa claro que, ao se obter uma ordem para testar os componentes, a necessidade de se construir *stubs* pode diminuir. E, como já se sabe que quanto menor o número de *stubs* construídos, menor são os custos relacionados com o teste de integração, isto significa que os custos do projeto são reduzidos, indicando que o algoritmo pode realizar um bom trabalho.

6.2.2 Geração e Seleção dos Casos de Teste

Tendo definido a ordem na qual os componentes devem ser testados e qual *stub* deve ser construído, chegou a hora de gerar os casos de teste para cada componente, para que a partir dai, os testes possam ser construídos e a ordem de integração possa ser seguida.

Obviamente, tendo que o GTD é aplicado a nível de método, a ordem de teste encontrada não se refere apenas a ordem dos componentes, mas sim a ordem dos métodos de cada componente. Isto significa que um método de um determinado componente pode ser testado, mesmo que tenha sido iniciado e ainda não finalizado os testes dos métodos de um outro componente. Isto complica um pouco o trabalho, uma vez que os casos de testes são gerados para cada componente e não para cada método de cada componente. Dessa forma, durante a construção dos casos de teste, é dada atenção especial a ordem de integração que deve ser

seguida, para que nenhum método que dependesse de outro, seja utilizado antes que o outro possa ser testado.

Para geração dos Casos de Teste é utilizada a técnica TOTEM, descrita em [Briand and Labiche, 2001] por Lionel Briand e Yvan Labiche. Seguindo esta técnica, a partir dos diagramas de seqüência produzidos, são geradas as Expressões Regulares para os Casos de Uso existentes. Como exemplo, pode se observar logo abaixo, a expressão regular, obtida através dos diagramas de seqüência especificados para o Caso de Uso Jogar Hungry Snake, com foco no componente Jogo. A expressão representa a soma dos 4 cenários existentes para o Caso de Uso Jogar Hungry Snake:

- o cenário alternativo onde a cobra passa por cima de uma comida.
- o cenário alternativo onde o jogo termina porque a cobra bateu na parede.
- o cenário alternativo onde o jogo termina porque a cobra bateu nela mesmo.
- o cenário principal onde o jogo termina por se obter o máximo de pontos.

(1) $testeCobraComeu_{ControleRegrasIF}$

+

(2): $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.getCobra_{JogoHungrySnake}.$
$moverCobra_{ControleCobraIF}.obterScore_{JogoHungrySnake}$

+

(3) $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$testecobraBateuNelaMesmo_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.getCobra_{JogoHungrySnake}.$
$moverCobra_{ControleCobraIF}.$
$obterScore_{JogoHungrySnake}$

+

(4) $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.$
$getCobra_{JogoGula}.moverCobra_{ControleCobraIF}.$
$testeFimJogo_{ControleRegrasIF}.$

$getScore_{JogoHungrySnake}$.$setSatus_{JogoHungrySnake}$.

$obterScore_{JogoHungrySnake}$

Vale salientar que as demais expressões referentes aos Casos de Uso Jogar Hungry Snake e Jogar Gula Gula, para os componentes Regras e Jogo, encontram-se em anexo no Apêndice E.

Para selecionar os Casos de Teste é utilizada a técnica de seleção de Casos de Teste usada no Cleanroom, a qual propõe a construção de um Modelo de Uso do sistema que representa todos os possíveis uso do sistema e suas probabilidades de ocorrência. Essas probabilidades são definidas usando uma função de distribuição de probabilidades. Os Modelos de Uso são gerados a partir das expressões regulares obtidas anteriormente. Como foram construídas 4 expressões regulares, são gerados quatro Modelos de Uso:

- Modelo de Uso para o Jogo Gula Gula referente ao componente Jogo, ilustrado na Figura 6.8;
- Modelo de Uso para o Jogo Gula Gula referente ao componente Regras, ilustrado na Figura 6.9;
- Modelo de Uso para o Jogo Hungry Snake referente ao componente Jogo, ilustrado na Figura 6.6;
- Modelo de Uso para o Jogo Hungry Snake referente ao componente Regras, ilustrado na Figura 6.7;

O Modelo de Uso do Caso de Uso Jogar Hungry Snake referente ao componente Jogo, que é gerado através da expressão regular apresentada mais acima. Neste caso, o vértice zero representa o estado inicial e vértice 6 o estado final. O caminho *0 1* retrata o cenário onde a cobra passa por cima de uma comida e, equivale ao primeiro termo da expressão regular; o caminho *0 1 2 3 4 5 6* representa o cenário onde o jogo termina porque a cobra bateu na parede e, equivale ao segundo termo da expressão regular; o caminho *0 1 2 7 3 4 5 6* representa o cenário onde o jogo termina porque a cobra bateu nela mesmo e, equivale ao terceiro termo da expressão regular; o caminho *0 1 2 3 4 5 8 9 6* retrata o cenário onde o jogo termina por se obter o máximo de pontos e, equivale ao quarto e último termo da expressão regular.

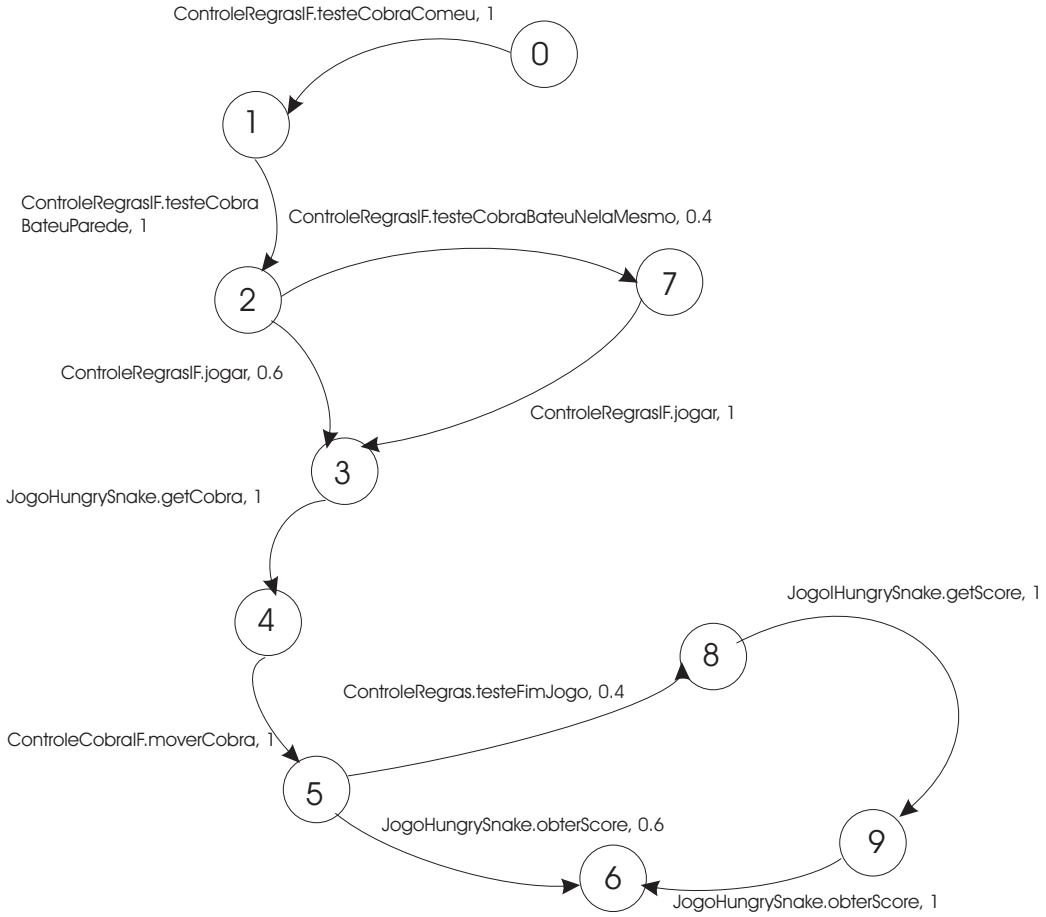

Figura 6.6: Modelo de Uso do Caso de Uso Jogar Hungry Snake referente ao componente Jogo

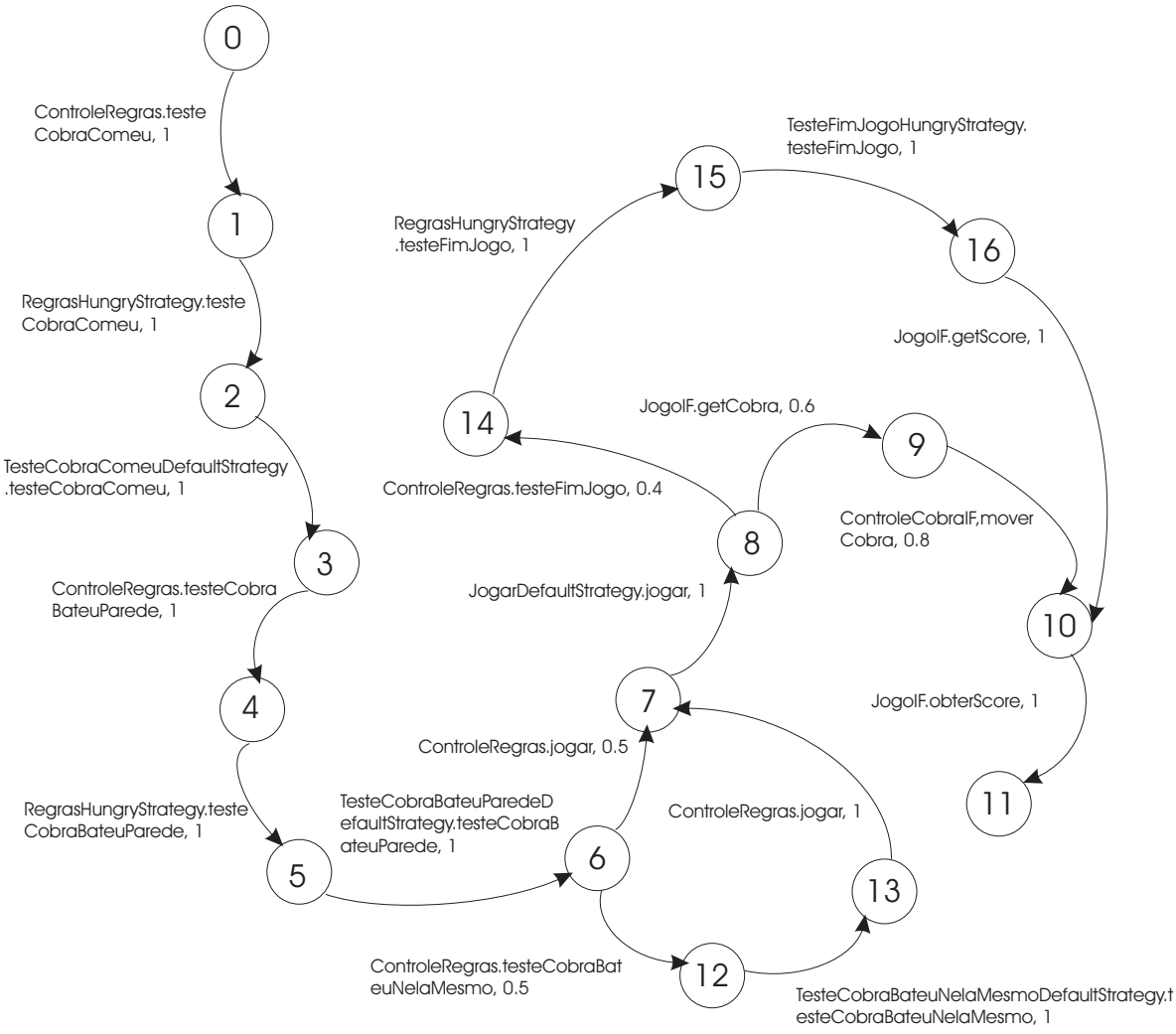

Figura 6.7: Modelo de Uso do Caso de Uso Jogar Hungry Snake referente ao componente Regras

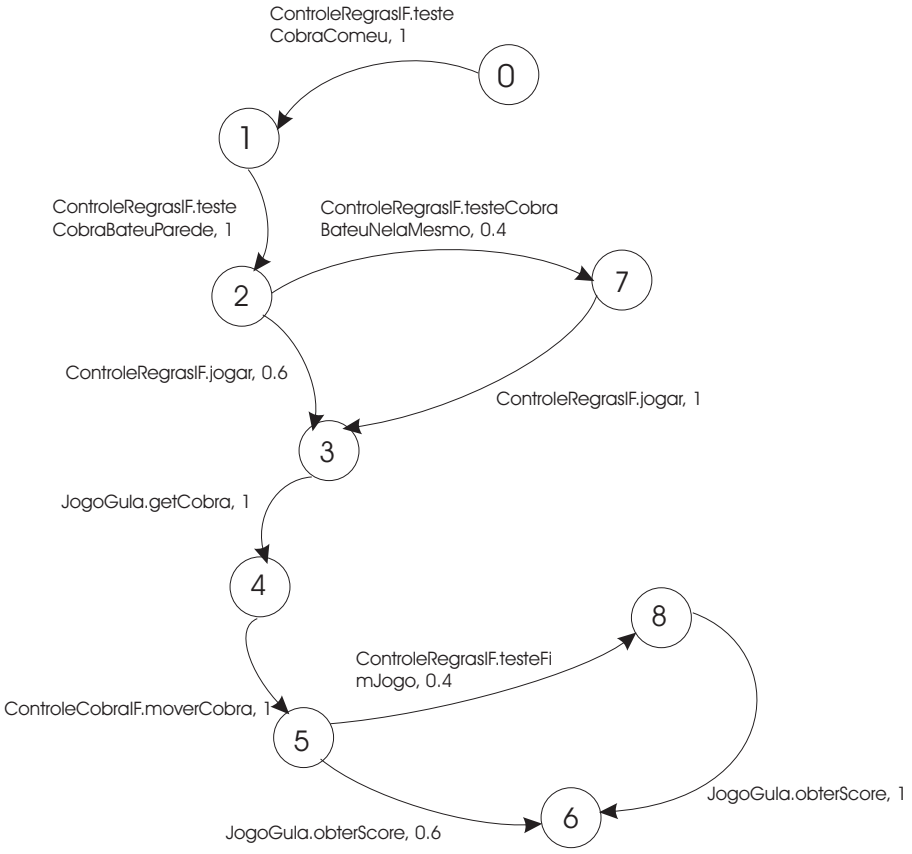

Figura 6.8: Modelo de Uso do Caso de Uso Jogar Gula Gula referente ao componente Jogo

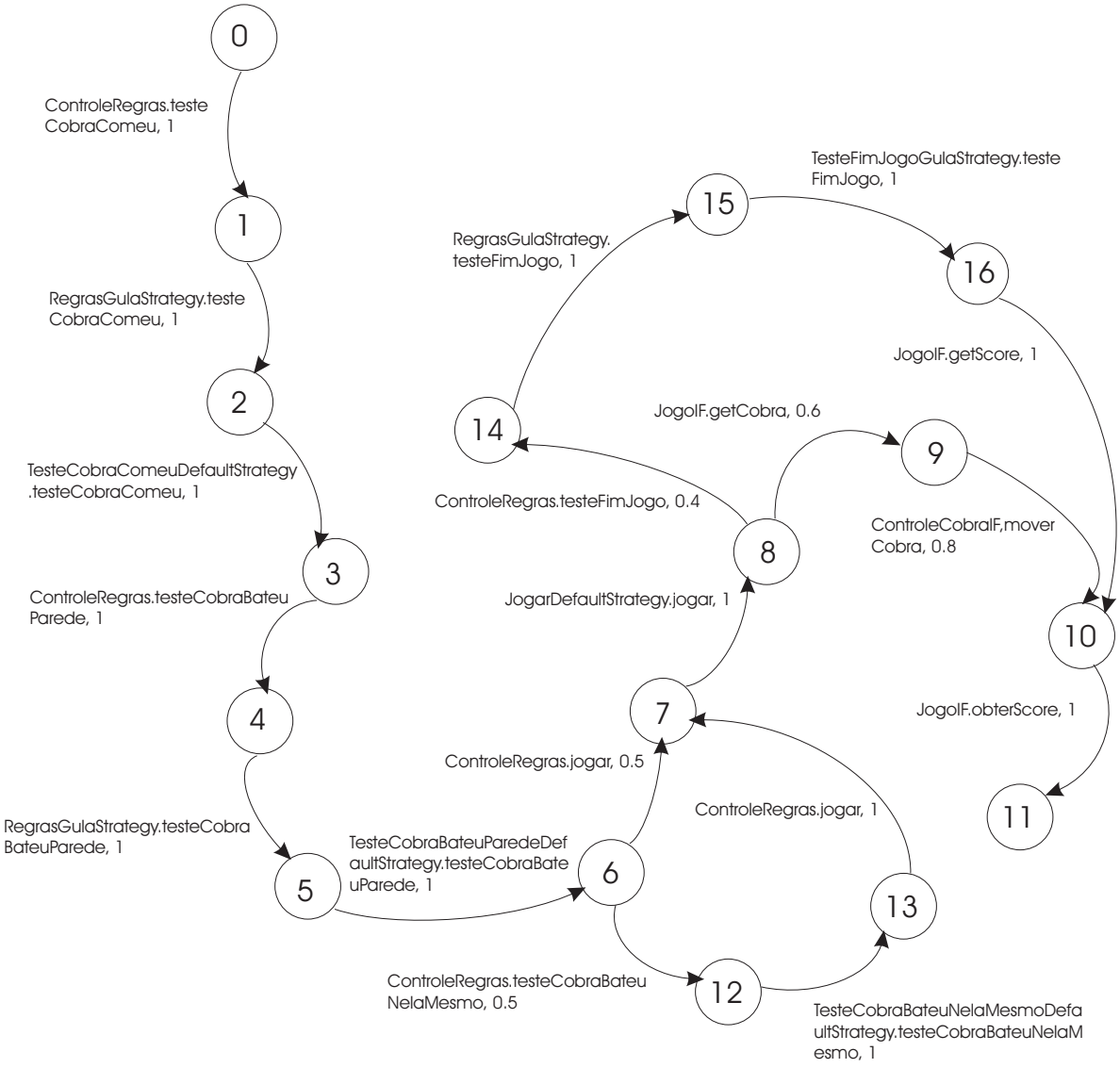

Figura 6.9: Modelo de Uso do Caso de Uso Jogar Gula Gula referente ao componente Regras

Para seleção dos Casos de Teste, é levada em consideração a Análise de Risco feita na fase de Planejamento. De acordo com esta análise, é definido um certo número de casos de teste a serem construídos para cada Caso de Uso especificado. Assim sendo, para o Caso de Uso Jogar Hungry Snake, são selecionados 3 Casos de Teste. De acordo com as probabilidades especificadas no Modelo de Uso, devem ser selecionados 3 caminhos de acordo com uma função de probabilidade. No caso, os seguintes caminhos são selecionados:

- *0 1 2 3 4 5 6*, onde o jogo Hungry Snake termina porque a cobra bate na parede;
- *0 1 2 3 4 5 8 9 6*, onde o jogo Hungry Snake termina por obter a pontuação máxima;
- *0 1 2 7 3 4 5 6*, onde o jogo Hungry Snake termina por a cobra bater nela mesmo.

Os casos de teste são selecionados da seguinte forma: começando pelo vértice 0, que representa o estado inicial, a única opção é ir para o vértice 1, seguindo da mesma forma para o vértice 2. No vértice 2, tem-se duas opções, ir para o vértice 3 ou para o vértice 7. De acordo com a função de distribuição de probabilidade, o vértice 3 foi escolhido, seguindo para o vértice 4 e 5. No vértice 5, mais uma vez, tem-se mais duas opções: seguir para o vértice 6 ou 8. Da mesma forma, é escolhido o vértice 6.

O segundo caso de teste é escolhido da mesma maneira, porém seguindo um caminho diferente. No vértice 5, ao invés de seguir para o vértice 6, segue-se para o vértice 8, uma vez que o vértice 6 já foi escolhido no primeiro caso de teste.

O terceiro caso de teste também segue o mesmo raciocínio. Porém, ao chegar no vértice 2, segue-se para o vértice 7, pois os caminhos seguindo pelo vértice 3 já foram seguidos nos casos de teste 1 e 2. Do vértice 7, segue-se para o vértice 3, 4 e 5. A partir dai pode-se ir para o vértice 8 ou 6, seguindo-se para o vértice 6 de acordo com a função de distribuição de probabilidade.

Seguindo o mesmo processo de seleção de casos de teste, são selecionados os seguinte caminhos:

para o Caso de Uso Jogar Hungry Snake referente ao Componente Regras:

- *0 1 2 3 4 5 6 7 8 9 10 11*, onde o jogo Hungry Snake termina porque a cobra bate na parede;

- *0 1 2 3 4 5 8 14 15 16 10 11*, onde o jogo Hungry Snake termina por obter a pontuação máxima;
- 0 *1 2 3 4 5 6 12 13 7 8 9 10 11*, onde o jogo Hungry Snake termina porque a cobra bate nela mesmo.

para o Caso de Uso Jogar Gula Gula referente ao Componente Jogo:

- *0 1 2 3 4 5 6*, onde o jogo Gula Gula termina porque a cobra bate na parede;
- *0 1 2 3 4 5 8 6*, onde o jogo Gula Gula termina por ter um número limite de comidas espalhada no tabuleiro;
- *0 1 2 7 3 4 5 6*, onde o jogo Hungry Snake termina por por a cobra bater nela mesmo.

para o Caso de Uso Jogar Gula Gula referente ao Componente Regras:

- *0 1 2 3 4 5 6 7 8 9 10 11*, onde o jogo Gula Gula termina porque a cobra bate na parede;
- *0 1 2 3 4 5 8 14 15 16 10 11*, onde o jogo Gula Gula termina por ter um número limite de comidas espalhada no tabuleiro.;
- *0 1 2 3 4 5 6 12 13 7 8 9 10 11*, onde o jogo Gula Gula termina porque a cobra bate nela mesmo.

Vale salientar que existe um cenário inicial, ilustrado na Figura 6.10, que é comum a todos os Modelos de Uso. Por isso ele encontra-se separado em um Modelo de Uso a parte, mas não pode ser descartado na seleção dos Casos de Teste. Na verdade, o caminho *1 2 3 4 5 6 7 8 9 10* apresentado no Modelo em questão, deve ser incluído antes de todos os outros caminhos selecionados como Caso de Teste.

6.2.3 Geração dos Dados e Oráculos de Teste

Uma vez definidos os caminhos de teste que devem ser percorridos, o próximo passo a seguir é gerar os dados de teste necessários para execução dos casos de teste. Para facilitar a geração desses dados são construídas algumas Tabelas de Decisão como proposto na técnica TOTEM.

Para cada expressão regular, que representa os cenários de uso do componente, é gerada uma Tabela de Decisão. Com o intuito de demonstrar como são gerados os oráculos, será

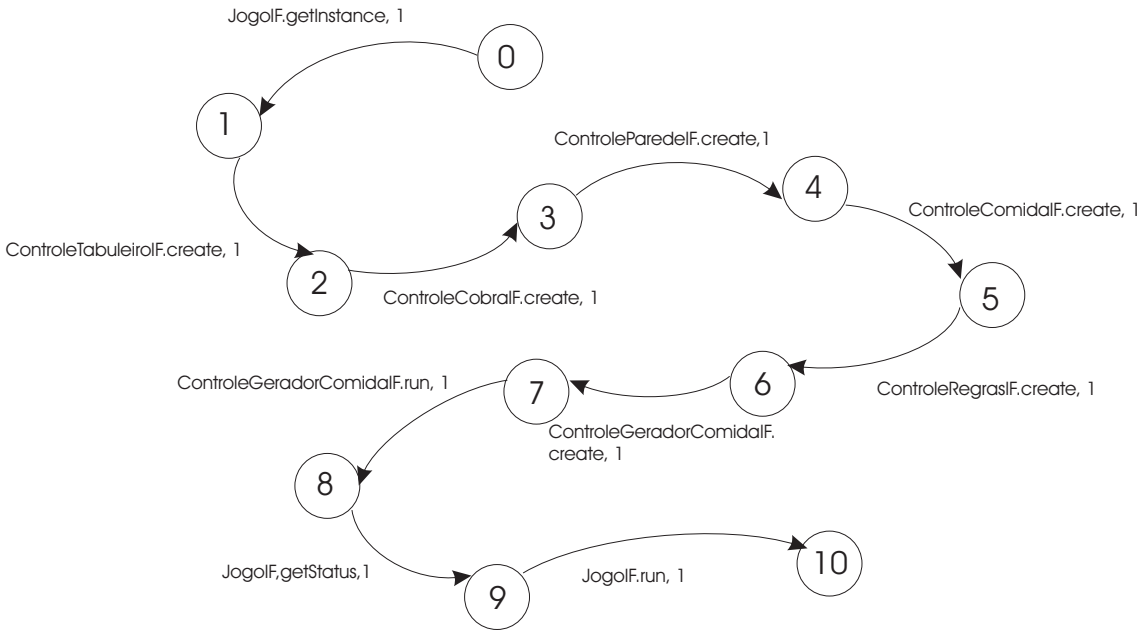

Figura 6.10: Cenário Inicial comum a todos os Modelos de Uso

utilizado o Caso de Uso Jogar Hungry Snake com as funcionalidades do componente Jogo. Como já visto anteriormente, o Caso de Uso Jogar Hungry Snake apresenta 4 cenários de uso possíveis, representados pela seguinte expressão regular:

(1) $testeCobraComeu_{ControleRegrasIF}$

+

(2): $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.getCobra_{JogoHungrySnake}.$
$moverCobra_{ControleCobraIF}.obterScore_{JogoHungrySnake}$

+

(3) $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$testecobraBateuNelaMesmo_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.getCobra_{JogoHungrySnake}.$
$moverCobra_{ControleCobraIF}.$
$obterScore_{JogoHungrySnake}$

+

(4) $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.$
$getCobra_{JogoGula}.moverCobra_{ControleCobraIF}.$

$testeFimJogo_{ControleRegrasIF}$.

$getScore_{JogoHungrySnake}.setSatus_{JogoHungrySnake}$.

$obterScore_{JogoHungrySnake}$

Para cada termo dessa expressão são identificadas suas condições de execução, e em seguida, expressas em OCL. A seguir, são mostradas as condições de execução, expressas em OCL, para cada termo definido acima para o Caso de Uso Jogar Hungry Snake. Para que o primeiro termo (1) possa ser executado é necessário que exista uma comida coincidindo com a cabeça da cobra. Então, para este termo, a condição de execução é a seguinte:

```
A : Jogo.getComidas()->exists(tc |
tc->includes(Jogo.getCobra().getCabeca()))
```

A expressão acima quer dizer que deve existir pelo menos uma comida cuja coordenada coincida com a cabeça da cobra.

O segundo termo (2) indica que a cobra bateu na parede e sua condição de execução é a seguinte:

```
B : Jogo.getParede().getPontos()->
                          includes(Jogo.getCobra().getCabeca())
```

A expressão B indica não pode existir nenhuma comida coincidindo com a cabeça da cobra e, que esta deve coincidir com algum ponto da parede.

O terceiro termo (3) indica que cobra bateu nela mesmo e é representado pela seguinte condição de execução:

```
C : Jogo.getCobra().getPontos()->
                  includes(Jogo.getCobra().getCabeca())
```

A expressão C quer dizer que para a cobra bater nela mesmo, a cabeça da cobra não pode coincidir com nenhuma comida e nem com nenhuma parede, mas deve existir algum ponto na cobra que coincida com a cabeça da mesma.

O quinto e último termo representa o cenário onde o jogo termina por ter obtido uma pontuação máxima. Sua condição de execução, é a seguinte:

```
D : not (Jogo.getComidas()->exists(tc |
    tc->includes(Jogo.getCobra().getCabeca()))) and not
    (Jogo.getParede().getPontos()->
    includes(Jogo.getCobra().getCabeca()))
    and Jogo.getScore() >= 15
```

Por fim, a última expressão, D, indica que a cabeça da cobra não pode coincidir com nenhuma comida e nem com nenhuma parede e, que os pontos deve ser o máximo permitido no jogo.

Após criadas as condições de execução de cada cenário do Caso de Uso, são definidas as mudanças de estado que podem ocorrer no componente com a execução de cada cenário, bem como, quais mensagens podem ser retornadas para o ator do Caso de Uso.

As mudanças de estado ocorridas após a execução de cada cenário do Caso de Uso, ou seja, de cada termo da expressão regular em questão, são expressas em OCL e encontram-se logo abaixo:

- para a expressão A, temos:

```
Jogo.getComidas()->size() =
    Jogo.getComidas()@pre->size() - 1 and
Jogo.getScore() = Jogo.getScore()@pre + 1 and
Jogo.getCobra()->size() = Jogo.getCobra()@pre->size() + 1
```

- para a expressão B, temos:

```
Jogo.getStatus() = false
```

- para a expressão C, temos:

```
Jogo.getStatus() = false
```

- para a expressão D, temos:

```
Jogo.getStatus() = false
```

O sistema exibe apenas uma mensagem, quando solicitado para exibir os pontos, que é a mensagem I, descrita abaixo, seguida dos pontos dos melhores jogadores:

I : Ranking dos jogadores 5 estrelas

As Figuras 6.11, 6.12, 6.13, 6.14 e 6.15 representam as Tabelas de Decisão criadas, informando as condições de realização do uso e as ações que serão tomadas pelo componente diante da ocorrência do uso. Essas tabelas ajudam na implementação dos testes.

Versões	Condições					Ações	
						Mensagem	Mudança de Estado
	A	B	C	D	E		
1	Sim	Não	Não	Não	Não		Sim
2	Não	Sim	Não	Não	Não		Sim
3	Não	Não	Sim	Não	Não		Sim
4	Não	Não	Não	Sim	Não		Sim
5	Não	Não	Não	Não	Sim		Sim

Figura 6.11: Tabela de Decisão do jogo Hungry Snake referente ao componente Jogo

Versões	Condições					Ações	
						Mensagem	Mudança de Estado
	A	B	C	D	E		
1	Sim	Não	Não	Não	Não		Sim
2	Não	Sim	Não	Não	Não		Sim
3	Não	Não	Sim	Não	Não		Sim
4	Não	Não	Não	Sim	Não		Sim
5	Não	Não	Não	Não	Sim		Sim

Figura 6.12: Tabela de Decisão do jogo Hungry Snake referente ao componente Regras

Uma vez criadas as Tabelas de Decisão, resta gerar os dados de teste para que se possa partir para implementação dos mesmos. Como o estudo de caso trata de uma aplicação que não possui muita interação com o usuário, ou seja, não possui muitos dados de entrada que

Versões	Condições				Ações	
					Mensagem	Mudança de Estado
	A	B	C	D		
1	Sim	Não	Não	Não		Sim
2	Não	Sim	Não	Não		Sim
3	Não	Não	Sim	Não		Sim
4	Não	Não	Não	Sim		Sim

Figura 6.13: Tabela de Decisão do jogo Gula Gula referente ao componente Jogo

Versões	Condições				Ações	
					Mensagem	Mudança de Estado
	A	B	C	D		
1	Sim	Não	Não	Não		Sim
2	Não	Sim	Não	Não		Sim
3	Não	Não	Sim	Não		Sim
4	Não	Não	Não	Sim		Sim

Figura 6.14: Tabela de Decisão do jogo Gula Gula referente ao componente Regras

Versões	Condições	Ações	
		Mensagem	Mudança de Estado
	A		
1	Sim	I	Sim

Figura 6.15: Tabela de Decisão do Caso de Uso Obter Ranking

possam ter uma importância significativa durante os testes, os dados são selecionados de forma aleatória para cada situação. Para os casos de teste referentes ao Caso de Uso Jogar Hungry Snake, com relação ao componente Jogo, tem-se os seguintes dados para os diversos casos de teste:

Caso de Teste onde a cobra bateu na parede:

- a cobra anda para baixo e bate na parede;
- a cobra muda a direção para a direita e depois bate na parede;
- a cobra muda a direção para esquerda e depois bate na parede.

Caso de Teste onde o jogo termina por obter a pontuação máxima:

- a cobra anda para baixo, muda a direção para a direita ao chegar próximo a parede e segue andando ao redor do tabuleiro, encostada na parede, onde existem comidas, até atingir os pontos máximos;
- a cobra anda para baixo e vira à direita comendo algumas comidas dispostas no tabuleiro, atingindo a pontuação máxima;
- a cobra anda para baixo e vira à direita comendo algumas comidas dispostas no tabuleiro, atingindo a pontuação máxima;

Caso de Teste onde a cobra bateu nela mesmo:

- a cobra anda come algumas comidas para aumentar seu tamanho e vira para direita, para cima e para esquerda, batendo nela mesmo.
- a cobra anda come algumas comidas para aumentar seu tamanho e vira para esquerda, para cima e para direita, batendo nela mesmo.

Já para os Casos de Teste referentes ao Caso de Uso Jogar Gula Gula com relação ao componente Jogo, tem-se os seguintes dados para os diversos Casos de Teste:

Caso de Teste onde a cobra bateu na parede:

- a cobra anda para baixo e bate na parede;
- a cobra muda a direção para a direita e depois bate na parede;
- a cobra muda a direção para esquerda e depois bate na parede.

Caso de Teste onde o jogo termina por ter um número limite de comidas espalhada no tabuleiro:

- o jogo chega a não ter nenhuma comida no tabuleiro
- o jogo fica com 20 comidas no tabuleiro e a cobra não come nenhuma.

Caso de Teste onde a cobra bateu nela mesmo:

- a cobra anda come algumas comidas para aumentar seu tamanho e vira para direita, para cima e para esquerda, batendo nela mesmo.
- a cobra anda come algumas comidas para aumentar seu tamanho e vira para esquerda, para cima e para direita, batendo nela mesmo.

6.3 Construção, Execução e Análise dos Resultados

A construção dos testes é realizada logo que os Dados de Teste são selecionados, ou seja, logo após a modelagem dos componentes.

Os testes são implementados com base nos Casos de Teste elaborados, nos dados de teste selecionados e nas Tabelas de Decisão, seguindo também a ordem de integração obtida através do GTD.

Para implementação dos testes foi utilizada a ferramenta Eclipse 2.1. Já para a execução e análise dos resultados foi utilizado o JUnit.

Os Casos de Teste para o Caso de Uso Jogar Hungry Snake, estão implementados nas seguintes classes: TestePontuacaoMaximaHungry, TesteBateuParedeHungry, TesteBateuNelaMesmaHungry e TesteBateuNelaMesmaHungry2. Já para o Caso de Uso Jogar Gula Gula os Casos de Teste encontram-se implementados nas classes: TesteFimJogoGula,

TesteBateuParedeGula, TesteBateuNelaMesmaGula e TesteBateuNelaMesmaGula2. Para garantir a ordem de integração fornecida pelo GTD, foi necessário construir alguns testes, antes e durante a implementação dos Casos de Teste elaborados. Estes testes se encontram nas seguintes classes: CobraTest, TesteRodaJogo, TesteJogar, TesteIntegracao, TesteCobraComeu, JogoTest e stubJogo.

Os testes da aplicação foram implementados obedecendo a ordem de integração anteriormente prevista, que consiste em testar, seqüencialmente, as seguintes operações e componentes:

- getScore(),
- getStatus(),
- setDirecao(),
- getCobra(),
- mudarDirecao() usando setDirecao(),
- obterScore() usando getScore(),
- moverCobra(),
- crescer(),
- componente Cobra usando moverCobra() e crescer(),
- testeCobraBateuParede() usando *stub*(Jogo),
- jogar() usando moverCobra(), crescer(), getCobra(), e *stub*(Jogo),
- testeCobraComeu() é testado usando *stub*(Jogo),
- testeCobraBateuNelaMesmo() é testado usando *stub*(Jogo),
- testeFimJogo() é testado usando *stub*(Jogo),
- run() é testado usando testeCobraBateuParede(), testeCobraComeu(), jogar(), testeCobraBateuNelaMesmo() e testeFimJogo()

- Jogo é testado usando run().
- Regras é testado usando testeCobraBateuParede(), testeCobraComeu(), jogar(), testeCobraBateuNelaMesmo() e testeFimJogo()
- rodaJogo() é testado usando getStatus() e run(),
- Controlador é testado usando rodaJogo(), obterScore(), mudaDirecao().

Para exemplificar como são construídos os testes, foi usado o Caso de Teste Jogar Hungry Snake no cenário onde o jogo termina por a cobra ter batido na parede. Este teste é realizado na classe TesteBateuParedeHungry, que aos poucos será detalhada.

Para implementar este caso de teste, é necessário utilizar o método *testeCobraBateuParede()* do componente Regras, usando o *stub(Jogo)*, como recomenda a ordem de integração obtida. Porém, seguindo esta ordem, antes de testar este método, devem ser testados os seguintes métodos: *getScore(), getStatus(), setDirecao(), getCobra(), mudarDirecao()*, *obterScore(), moverCobra(), crescer()*, e o componente Cobra, nesta seqüência.

Uma vez testados os métodos necessários, retoma-se o caso de teste que se deseja testar: Jogar Hungry Snake no cenário onde o jogo termina por a cobra ter batido na parede. Para este cenário, foram selecionados anteriormente 3 dados de teste:

- a cobra anda para baixo e bate na parede: este caso encontra-se implementado no método testHungryBateuParede1 no Código 3;
- a cobra muda a direção para a direita e depois bate na parede: este caso encontra-se implementado no método testHungryBateuParede2 no Código 4;
- a cobra muda a direção para esquerda e depois bate na parede: este caso encontra-se implementado no método testHungryBateuParede3 no Código 5.

Para exemplificar a implementação do caso de teste em questão, alguns métodos são analisados. As referências a objetos usados nos métodos, bem como algumas constantes, são mostradas no Código 1.

```
// referência para o \emph{stub} de Jogo
public stubJogo stubJogo1;
```

```
// constantes para identificar a direção
protected static final int DIRECAO_BAIXO = 40;
protected static final int DIRECAO_CIMA = 38;
protected static final int DIRECAO_ESQUERDA = 37;
protected static final int DIRECAO_DIREITA = 39;
```

Código 1: referências a objetos e algumas constantes da classe *TesteBateuParedeHungry*

Os objetos necessários aos testes são instanciados no método *setUp()*, mostrado no Código 2. Neste caso, na linha 02, é criada uma instância do jogo Hungry Snake. Como o componente Jogo ainda não foi testado, e seguindo a ordem de integração obtida, neste momento é criado um *stub* de Jogo.

```
01      protected void setUp() {
02          stubJogo1 = stubJogo.getInstance("Hungry");
03      }
```

Código 2: Método *setUp()* da classe *TesteBateuParedeHungry*

Para o primeiro dado de teste selecionado, onde a cobra anda para baixo e bate na parede, é criado o método *testHungryBateuParede1()*. Nele são criados 3 atributos: *altura*, *existeComida*, *existeParede*, nas linhas 06, 07 e 08, respectivamente.

O primeiro atributo, serve para saber a altura da metade do tabuleiro, para forçar a situação onde a cobra deve andar até que bata na parede, uma vez que a cobra inicia o jogo sempre na metade do tabuleiro. O segundo atributo, *existeComida*, recebe o valor false a princípio. Ele serve para testar se existe alguma comida coincidindo com a cabeça da cobra, no momento em que ela bate na parede. O terceiro atributo, *existeParede*, serve para testar se existe alguma parte da parede coincidindo com a cabeça da cobra no momento em que ela bate na parede.

Ainda no Código 3, na linha 13, antes de começar os testes, é verificado se o jogo já está com o status *true* antes da cobra bater na parede. Na linha 16, é chamado o método

moverCobra(), que já foi testado anteriormente, para que a cobra se movimente uma casa para baixo. Este método é chamado repetidas vezes, até que a cobra bata na parede. Neste momento é testado se não existe nenhum ponto da cobra coincidindo com uma comida e se existe algum ponto da cobra coincidindo com a parede. Estes testes são feitos nas linhas 40 e 43, respectivamente. Da linha 44 à 47, é testado se o método *testeCobraBateuParede(JogoIF jogo)* está retornando o valor *true*, como esperado. Na linha 48, é verificado se após a cobra bater na parede o status do jogo torna-se *false*.

```
01      /**
02       *  Método que testa a situação onde a cobra anda
03       *  para baixo e bate na parede.
04       */
05      public void testHungryBateuParede1() {
06          int altura = stubJogo1.getTabuleiro().getLinhas() / 2;
07          boolean existeComida = false;
08          boolean existeParede = false;
09          // faz a cobra andar até bater na parede
10          for (int i = 1; i <= altura; i++) {
11
12              if (i == altura) {
13                  assertEquals(true, stubJogo1.getStatus());
14              }
15
16              stubJogo1.getCobra().moverCobra(DIRECAO_BAIXO);
17
18              if (i == altura - 1) {
19
20                  for (int j = 0; i < stubJogo1.getComidas().
21                                              size(); j++) {
22                      if ((stubJogo1
23                          .getComida(i)
24                          .getPontos()
```

```
25                          .contains((stubJogo1.getCobra()).
26                                  getCabeca()))) {
27                          existeComida = true;
28                      }
29                  }
30
31                  if ((stubJogo1
32                      .getParede()
33                      .getPontos()
34                      .contains((stubJogo1.getCobra()).
35                              getCabeca()))) {
36                      existeParede = true;
37                  }
38                  //testa se nao existe nenhuma comida
39                  //coincidindo com a cabeca da cobra
40                  assertFalse(existeComida);
41                  //testa se a cabeca da cobra coincide
42                  //com um ponto da parede
43                  assertTrue(existeParede);
44                  assertEquals(
45                      true,
46                      stubJogo1.getRegras().
47                      testeCobraBateuParede(stubJogo1));
48                  assertEquals(false, stubJogo1.getStatus());
49              }
50          }
51      }
```

Código 3: Método *testHungryBateuParede1()* da classe *TesteBateuParedeHungry*

O segundo método de teste *testHungryBateuParede2()*, apresentado no Código 4, repre-

senta a implementação do caso de teste para o segundo dado de teste selecionado, ou seja, implementa basicamente o mesmo procedimento do método anterior detalhado no Código 3, porém, com uma pequena diferença, nas linhas 10 e 17. Na linha 10, a direção da cobra é setada para a direita, e na linha 17, a cobra se move na direção direita, e não para baixo como no teste anterior, até bater na parede. Na linha 38 é verificado se não existe nenhum ponto da cobra coincidindo com uma comida. Já na linha 39 é verificado se existe algum ponto da cobra coincidindo com a parede. E da linha 40 à 43 , é testado se o método *testeCobraBateuParede(JogoIF jogo)* está retornando o valor *true*, como esperado. Na linha 44, é verificado se após a cobra bater na parede o status do jogo torna-se *false*.

```
01      /**
02       * Método que testa se a cobra mudou
03       * a direção para direita e bateu na parede.
04       */
05      public void testHungryBateuParede2() {
06          int altura = stubJogo1.getTabuleiro().getLinhas() / 2;
07          boolean existeComida = false;
08          boolean existeParede = false;
09
10          stubJogo1.setDirecao( DIRECAO_DIREITA );
11
12          for (int i = 1; i <= altura; i++) {
13              if (i == altura) {
14                  assertEquals(true, stubJogo1.getStatus());
15              }
16
17              stubJogo1.getCobra().moverCobra(DIRECAO_DIREITA);
18
19              if (i == altura - 1) {
20
21                  for (int j = 0; i < stubJogo1.getComidas().
22                                          size(); j++) {
```

```
23                      if ((stubJogo1
24                          .getComida(i)
25                          .getPontos()
26                          .contains((stubJogo1.getCobra()).
27                                  getCabeca()))) {
28                          existeComida = true;
29                      }
30                  }
31                  if ((stubJogo1
32                      .getParede()
33                      .getPontos()
34                      .contains((stubJogo1.getCobra()).
35                              getCabeca()))) {
36                      existeParede = true;
37                  }
38                  assertFalse(existeComida);
39                  assertTrue(existeParede);
40                  assertEquals(
41                      true,
42                      stubJogo1.getRegras().
43                      testeCobraBateuParede(stubJogo1));
44                  assertEquals(false, stubJogo1.getStatus());
45              }
46
47          }
48      }
```

Código 4: Método *testHungryBateuParede2()* da classe *TesteBateuParedeHungry*

A implementação do terceiro dado de teste, também é bem parecida com a do segundo, diferenciando nas linhas: 10 e 18, onde a direção da cobra ao invés de ser setada para a direita, é setada para a esquerda na linha 10, fazendo com que a cobra se 139, mova para

a esquerda, na linha 18. Na linha 41 é verificado se não existe nenhum ponto da cobra coincidindo com uma comida. Já na linha 43 é verificado se existe algum ponto da cobra coincidindo com a parede. E da linha 45 à 48 , é testado se o método *testeCobraBateuParede(JogoIF jogo)* está retornando o valor *true*, como esperado. Na linha 50, é verificado se após a cobra bater na parede o status do jogo torna-se *false*.

```
01      /**
02       * Método que testa se a cobra mudou a direção
03       * para esquerda e bateu na parede.
04       */
05      public void testHungryBateuParede3() {
06          int altura = stubJogo1.getTabuleiro().getLinhas() / 2;
07          boolean existeComida = false;
08          boolean existeParede = false;
09
10          stubJogo1.setDirecao( DIRECAO_ESQUERDA );
11
12          for (int i = 1; i <= altura; i++) {
13
14              if (i == altura) {
15                  assertEquals(true, stubJogo1.getStatus());
16              }
17
18              stubJogo1.getCobra().moverCobra(DIRECAO_ESQUERDA);
19
20              if (i == altura - 1) {
21
22                  for (int j = 0; i < stubJogo1.getComidas().
23                                              size(); j++) {
24                      if ((stubJogo1
25                              .getComida(i)
26                              .getPontos()
```

```
27                          .contains((stubJogo1.getCobra()).
28                                  getCabeca()))) {
29                          existeComida = true;
30                      }
31                  }
32
33                  if ((stubJogo1
34                      .getParede()
35                      .getPontos()
36                      .contains((stubJogo1.getCobra()).
37                              getCabeca()))) {
38                      existeParede = true;
39                  }
40
41                  assertFalse(existeComida);
42
43                  assertTrue(existeParede);
44
45                  assertEquals(
46                      true,
47                      stubJogo1.getRegras().
48                      testeCobraBateuParede(stubJogo1));
49
50                  assertEquals(false, stubJogo1.getStatus());
51              }
52  }
53      }
```

Código 5: Método *testHungryBateuParede3()* da classe *TesteBateuParedeHungry*

A análise dos resultados é feita de acordo com os dados contidos nas Tabelas de decisões. Após executar determinado teste, os resultados são comparados com os dados existentes na

tabela. Por exemplo, para o Caso de teste referente ao Caso de Uso jogar Hungry Snake, onde o jogo termina por a cobra ter batido na parede, o qual corresponde ao termo (2) da expressão regular apresentada anteriormente, e à condição de execução B, temos uma mudança de estado expressa em OCL, a qual diz que após realizado esse teste o *status* do jogo deve ser *false*. Assim, caso o status seja *true* é porque deve ter algum erro no sistema ou na elaboração do teste.

Como o sistema em estudo não depende muito de entradas externas, os testes foram um pouco trabalhosos de serem construídos porque uma determinada situação deveria ser forçada a acontecer, situação na qual, na execução normal do sistema, ela aconteceria naturalmente sem nenhuma interferência externa. Porém, os testes foram todos executados com sucesso.

6.4 Conclusões

Neste capítulo foi apresentado, como foi aplicado o método de teste de integração proposto no estudo de caso realizado, ilustrando passo a passo cada etapa seguida. Algumas observações e comentários foram ressaltados mediante a aplicação do método de uma forma geral, o que gerou a obtenção de uma série de conclusões, relatando algumas vantagens e dificuldades encontradas na aplicação do método. A seguir, estão apresentados alguns desses comentários.

Inicialmente, para aplicar o método proposto, é necessário que os componentes já tenham sido testados anteriormente. Como o sistema foi desenvolvido desde o início, com reuso apenas do código-fonte, foi preciso testar os componentes de forma individual. Caso contrário, assumiria-se que os componentes já foram sidos anteriormente testados. Para realizar o teste dos componentes individuais foi aplicado o método de teste de componentes, apresentado em [Farias, 2003]. Isto demandou muito tempo, uma vez que a aplicação deste método ainda requer um esforço muito braçal. Contudo, este problema está sendo resolvido, pois está sendo implementada uma ferramenta, por um aluno do grupo de teste da UFCG, em [Barbosa, 2003], que pretende automatizar tanto o método de teste individual de componentes quanto o método de teste de integração proposto neste trabalho. A ferramenta está em andamento e fará parte da tese de mestrado do aluno, devendo estar pronta em breve.

Durante a aplicação do método, algumas dificuldades também foram encontradas, como por exemplo: a construção do GTD desprende muito esforço para ser construído manualmente, uma vez que é preciso identificar todos os relacionamentos existentes entre os método de cada componente. Como o GTD é gerado a partir dos Modelos de Informação e dos Diagramas de Colaboração, este processo poderia ser automatizado, diminuindo o esforço desprendido para elaboração do mesmo, bem como possíveis inconsistências que possam vir a existir, devido à não identificação de alguma ligação, já que são muitos detalhes para serem identificados visualmente. Da mesma forma, se dá a aplicação do algoritmo ao grafo. Trata de outro processo trabalhoso e que pode ser facilmente automatizado, já que o algoritmo se encontra bem definido. Espera-se que esses problemas também sejam resolvidos com a ferramenta que está em construção.

Uma outra dificuldade encontrada foi durante a implementação dos testes. Os casos de teste dos componentes são gerados e implementados seguindo a seqüência de integração obtida pelo método proposto. Porém, esta seqüência é dada a nível de métodos, e os casos de teste, bem como os dados de teste obtidos, são amplos demais para se ter dados de teste para um determinado método. De certa forma, o teste desses métodos poderiam ser feitos a nível de teste de classe, ou teste de unidade. Como este tipo de teste não foi considerado no foco deste trabalho, não foram gerados casos de teste e dados de teste para esses métodos. Foram realizados alguns testes baseados apenas na experiência do desenvolvedor. Isto é um ponto que deve ser melhorado no método.

Do ponto vista de execução dos testes, além de pequenos problemas de implementação errada dos testes, mas que foram rapidamente resolvidos, não houveram maiores problemas. Os testes foram executados e nenhuma falha foi encontrada. Isto provavelmente deve-se ao fato da haver uma certa qualidade na especificação do componente, trazendo uma maior facilidade em transformar especificação OCL em código Java, o que facilita a produção de um código mais correto.

Algumas vantagens sobre a utilização do método também foram ressaltadas, como por exemplo: apesar do método ainda se encontrar muito braçal, ele possui um grande potencial para automação, visto que:

- a especificação dos componentes e do sistema como um todo é fornecida através de uma linguagem formal de especificação(OCL);

- o GTD é gerado através de artefatos padronizados em UML;
- é utilizado um algoritmo bem definido para encontrar a ordem de integração.

O tempo desprendido na elaboração do estudo caso, devido a não automatização do método, foi compensado até certo ponto, pelo uso de especificações UML já produzidas durante a construção do sistema.

Para obter melhores chances de sucesso com a utilização do método proposto é interessante que a aplicação seja desenvolvida utilizando o processo de desenvolvimento sugerido, uma vez que as etapas do método de teste ocorrem em paralelo às etapas do processo de desenvolvimento e já estão contextualizadas no mesmo. Também é interessante que os componentes sejam testados utilizando o método proposto em [Farias, 2003], por já fornecer os resultados requeridos utilizados no método aqui proposto.

Capítulo 7

Considerações Finais

O objetivo principal deste trabalho foi criar um método prático de teste de integração para SBCs. Na Seção 7.1 são destacados os passos realizados para atingir o objetivo do trabalho, bem como os resultados alcançados com a finalização do trabalho. A Seção 7.2 apresenta as contribuições Por fim, na Seção 7.3, foram destacados possíveis trabalhos futuros.

7.1 Resultados Alcançados

Com o intuito de contribuir para a melhoria do desenvolvimento dos SBCs e, como um método de teste para componentes individuais, já havia sido criado como parte de um trabalho produzido pelo grupo de teste da UFCG, resolveu-se dar continuidade ao trabalho já existente e estender os testes criando um método que verificasse a integração entre esses componentes, de forma a se obter um processo de teste completo do ponto de vista de componentes.

Era esperado que o método fornecesse potencial para automação, uma vez que torna-se praticamente inviável, aplicar um método de teste sem um suporte ferramental. Para facilitar esta automação, era desejável que os componentes utilizados pelo método possuíssem uma descrição formal dos serviços providos.

Como observado em outros trabalhos, a gerência dos relacionamentos existentes entre os componentes não é uma tarefa trivial, por isso era desejado que o método, de alguma forma, identificasse e representasse os relacionamentos existentes entre os componentes, de maneira tal, que fosse possível gerenciá-los sem maiores problemas. Além disso, também foi notada, por outros autores, a necessidade de atrelar a atividade de teste ao processo de

desenvolvimento, com a finalidade de aproveitar artefatos já produzidos e de evitar que problemas existentes na especificação do software fossem propagados para etapas posteriores do desenvolvimento.

Por fim, não menos importante, era desejável que fosse realizado um estudo de caso para facilitar a compreensão do método e demonstrar a sua aplicação.

7.2 Contribuições

A principal contribuição deste trabalho foi permitir a verificação efetiva das funcionalidades dos componentes dentro de uma determinada aplicação.

O método proposto visou melhorar algumas dificuldades encontradas nos trabalhos que abordavam teste de integração, trazendo, portanto, algumas contribuições significativas que se encontram relatadas a seguir.

Geralmente, os componentes são fornecidos através de descrições informais junto com a assinatura dos métodos, o que dificulta a automação dos testes. Assim sendo, um dos pontos de relevância do trabalho foi fazer com que os componentes fossem especificados em OCL, descrevendo formalmente os serviços por ele provido e facilitando uma possível automação do método. Além disso a geração do GTD também pode ser facilmente automatizada, uma vez que o algoritmo utilizado já encontra-se bem definido. O método proposto foi baseado em técnicas de teste funcional, o que tornou-se bastante interessante por permitir a realização dos testes sem o código-fonte dos componentes, uma vez que os mesmos, normalmente, não se encontram disponíveis.

A falta de contextualização do processo de teste dentro de um processo de desenvolvimento foi outro problema encontrado em muitas das técnicas já existentes. Por isso, uma outra contribuição do trabalho é que o método de teste proposto foi contextualizado dentro de um processo de desenvolvimento, como defendido em [Jeff, 1995] e [Briand and Labiche, 2001], permitindo que a atividade de teste pudesse ser iniciada cedo, facilitando a detecção e remoção das falhas o quanto antes e, possibilitando a redução dos recursos envolvidos no projeto. O método de teste proposto também pode ser utilizado em conjunto com outro processo de desenvolvimento, desde que este forneça as características necessárias para utilização do método. A princípio nenhum impacto é causado pela utilização de um processo de

desenvolvimento diferente do utilizado neste trabalho. Contudo, seria interessante realizar outros estudos de caso para verificar a eficiência do mesmo.

Uma outra contribuição é que o método deu continuidade aos esforços que vêm sendo desenvolvidos no grupo de pesquisa de testes da UFCG, no tocante ao suporte metodológico e ferramental ao teste de sistemas baseados em componentes.

Por fim, não menos importante, o trabalho contribui com um estudo de caso o que ilustra a aplicabilidade do método na prática, tendo uma visão inicial da sua viabilidade no ponto de vista prático.

7.3 Trabalhos Futuros

Após finalizar este trabalho foi feita uma análise sobre possíveis propostas de melhorias e continuidade do mesmo. Um proposta interessante seria melhorar o método em termos de sua especificação. De uma forma geral, para aplicar o método, primeiro são definidas as dependências existentes entre os componentes, depois é definida a ordem de integração e finalmente os testes são desenvolvidos. O desenvolvimento dos testes é feito a partir dos casos de teste gerados, obedecendo a ordem de integração definida. Esta ordem é definida a nível de métodos, os quais representam os serviços fornecidos pelos componentes. Então, ao tentar se implementar os casos de teste, que abrangem um contexto mais geral, às vezes é preciso parar para testar uma determinada operação de um determinado componente, de acordo com a ordem de integração. Portanto, a maneira de implementar os testes juntamente com a ordem de integração definida pode ser melhor definida.

Outro trabalho interessante seria o desenvolvimento de ferramentas de suporte ao método. O desenvolvimento de testes de forma manual, não é uma prática utilizada pela comunidade de engenharia de software. Testes muito trabalhosos tornam-se insatisfatórios. Por isso, a existência de um suporte ferramental é de grande importância para viabilizar a aplicação do método de forma prática.

Complementar o método em termos de sua abrangência também é um trabalho importante. O método faz uso de uma análise de risco para definir a quantidade de casos de teste que devem ser elaborados para cada caso de uso. Entretanto, uma análise para avaliar o quanto do sistema foi testado também seria um trabalho de grande contribuição. Seria in-

teressante estender o método, acrescentando ao mesmo alguma forma de medir o grau de cobertura dos testes realizados no sistema.

Existe ainda a necessidade de avaliar o uso do método em situações em que o componente já foi pré-desenvolvido (COTS). Dessa forma, um outro trabalho interessante seria investigar quais impactos seriam causados no método no uso de tais componentes.

Bibliografia

[Barbosa, 2003] Barbosa, D. L. (2003). Automação de métodos e técnicas para teste funcional de componentes. Proposta de dissertação de mestrado - COPIN, UFCG.

[Beizer, 1999] Beizer, B. (1999). *Black-Box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons.

[Beydeda and Gruhn, 2001] Beydeda, S. and Gruhn, V. (2001). An integrated testing technique for component-based software. In *AICCSA ACS/IEEE International Conference on Computer Systems and Applications*.

[Briand and Labiche, 2001] Briand, L. and Labiche, Y. (2001). A UML-based approach to system testing. In *UML'2001*, volume 2185 of *LNCS*, pages 60–70.

[Cheesman and Daniels, 2001] Cheesman, J. and Daniels, J. (2001). *UML Components: A Simple Process for Specifying Component-Based Software*. Addison-Wesley.

[Dellarocas, 1997] Dellarocas, C. (1997). Software component interconnection should be treated as a distinct design problem. In *WISR Proceedings of the 8th Annual Workshop on Software Reuse*.

[Domingues, 2002] Domingues, A. L. S. (2002). Avaliação de critérios e ferramentas de teste para programas oo. Master's thesis, Instituto de Ciências Matemáticas e de Computação, USP.

[Farias, 2003] Farias, C. M. (2003). Um método de teste funcional para verificação de componentes. Master's thesis, Pós-Graduação em Informática, Universidade Federal de Campina Grande.

[Hanh et al., 2001] Hanh, V. L., Akif, K., Traon, Y. L., and Jézéquel, J. (2001). Selecting an efficient OO integration testing strategy: An experimental comparison of actual strategies. *LNCS*, 2072.

[Harrold et al., 1999] Harrold, M., Liang, D., and Sinha, S. (1999). An approach to analyzing and testing component-based systems. In *First International ICSE Workshop Testing Distributed Component-Based Systems*.

[Harrold and Soffa, 1991] Harrold, M. J. and Soffa, M. L. (1991). Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65.

[Hartmann et al., 2000] Hartmann, J., Imoberdorf, C., and Meisinger, M. (2000). UML-based integration testing. *ACM Transactions on Software Engineering and Methodology*, 8(11):60–70.

[Ian, 2003] Ian, S. (2003). *Engenharia de Software*. Addison-Wesley.

[Jeff, 1995] Jeff, Z. J. (1995). Integrating testing with the software development process.

[Jin and Offutt, 1998] Jin, Z. and Offutt, A. J. (1998). Coupling-based criteria for integration testing. *Software Testing, Verification & Reliability*, 8(3):133–154.

[Kim and Carlson, 2001] Kim, J. and Carlson, C. R. (2001). The role of design components in test plan generation. In *Third International Conference*.

[Larman, 1999] Larman, C. (1999). *Applying UML and Patterns - An Introduction to Object-Oriented Analysis and Design*. Prentice Hall, Inc.

[Martins et al., 2001] Martins, E., Toyota, C. M., and Yanagawa, R. L. (2001). Constructing self-testable software components. In *The International Conference on Dependable Systems and Networks (DSN'01)*.

[McGregor and Sykes, 2001] McGregor, J. D. and Sykes, D. A. (2001). *A Practical Guide to Testing Object-Oriented Software*. Addison-Wesley.

[Prowell et al., 1999] Prowell, S. J., TRammell, C. J., Linger, R. C., and Poore, J. H. (1999). *Cleanroom Software Engineering: Technology and Process*. The SEI Series in Software Engineering. Addison-Wesley.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY.

[Traon et al., 1999] Traon, Y. L., Jéron, T., Jézéquel, J., and Morel, P. (1999). Efficient strategies for integration and regression testing of oo systems. In *IEEE Software - 10th International Symposium on Software Reliability Engineering*.

[Traon et al., 2000] Traon, Y. L., Jéron, T., Jézéquel, J., and Morel, P. (2000). Efficient object-oriented integration and regression testing. *IEEE Transactions on Reability*, 49(1).

[Tsai et al., 2001] Tsai, W. T., Bai, X., Paul, R., Shao, W., and Agarwal, V. (2001). End-to-end integration testing design. In *IEEE Software - 25th Annual International Computer Software and Applications Conference (COMPSAC'01)*.

[Vieira et al., 2001] Vieira, M. E. R., Dias, M. S., and Richardson, D. J. (2001). Describing dependencies in component access points. In *4th ICSE Workshop on Component-Based Software Engineering*.

[Wu et al., 2003] Wu, Y., Chen, M., and Offutt, J. (2003). Uml-based integration testing for component-based software. In *The 2nd International Conference on COTS-Based Software Systems(ICCBSS)*.

[Wu et al., 2001] Wu, Y., Chen, M., and Pan, D. (2001). Techniques for testing component-based software. In *Seventh International Conference on Engineering of Complex Computer Systems*.

Apêndice A

Casos de Uso

Sub-Caso de Uso Nº:	**1**
Nome:	**Controlar a Direção**
Ator:	**Usuário**
Objetivo:	O usuário controla a direção na qual a cobra se move.
Cenário Principal:	
	1. Usuário solicita mudança de direção 2. A cobra se move na direção escolhida

Tabela A.1: Caso de Uso Controlar a Direção

Sub-Caso de Uso Nº: 2

Nome: Cobra Come

Ator: Usuário

Objetivo: A cobra passa por cima de uma comida, comendo a mesma e aumentando seu tamanho.

Cenário Principal:

1. A Cobra passa por cima de uma comida.
2. A cobra aumenta de tamanho.

Tabela A.2: Caso de Uso Cobra Come

Caso de Uso Nº: 1

Nome: Jogar Hungry Snake

Ator: Usuário

Objetivo: Jogar o Hungry Snake onde quanto mais comidas a cobra comer, maior o seu score. Após a cobra ter eliminado uma comida, outra comida é exibida no tabuleiro e a cobra aumenta de tamanho.

Cenário Principal:

1. O sistema cria o jogo apenas com o tabuleiro, uma comida e a cobra.
2. O jogo é iniciado.
3. Sub-Caso de Uso 1
4. Sub-Caso de Uso 2
5. O score atinge a pontuação igual a 15.
6. O jogo termina.
7. O score é exibido.

Extensões:

5. A cobra bate nela mesma.
6. O jogo termina.
7. O score é exibido.

5. A cobra bate na parede.
6. O jogo termina.
7. O score é exibido.

Tabela A.3: Caso de Uso Jogar Hungry Snake

Caso de Uso Nº: 2

Nome: Jogar Gula Gula

Ator: Usuário

Objetivo: Jogar o Gula Gula onde quanto mais comidas a cobra comer maior o seu score. No tabuleiro são criadas 4 paredes como obstáculos. A comida é exibida aleatoriamente no tabuleiro em tempos aleatórios. O jogo termina se todas as comidas forem eliminadas ou se um determinado número de comidas estiverem ao mesmo tempo no tabuleiro ou se a cobra bater em uma das paredes ou nela mesmo.

Cenário Principal:

1. O sistema cria o jogo com o tabuleiro, 4 paredes, a cobra e uma comida.
2. O jogo é iniciado.
3. Sub-Caso de Uso 1
4. Sub-Caso de Uso 2
5. A cobra come a última comida do tabuleiro.
6. O jogo termina.
7. O score é exibido.

Extensões:

5. A cobra bate nela mesma.
6. O jogo termina.
7. O score é exibido.

5. A cobra bate na parede.
6. O jogo termina.
7. O score é exibido.

5. Aparecem 20 comidas no tabuleiro ao mesmo tempo.
6. O jogo termina.
7. O score é exibido.

Tabela A.4: Caso de Uso Jogar Gula Gula

Caso de Uso Nº: 3

Nome: Jogar Gula Gula 2

Ator: Usuário

Objetivo: Jogar o Gula Gula 2 onde quanto mais comidas a cobra comer maior o seu score. No tabuleiro são criadas 2 paredes como obstáculos. A comida é exibida aleatoriamente no tabuleiro em tempos aleatórios. O jogo termina se todas as comidas forem eliminadas ou se um determinado número de comidas estiverem ao mesmo tempo no tabuleiro ou se a cobra bater em uma das paredes ou nela mesmo.

Cenário Principal:

1. O sistema cria o jogo com o tabuleiro, 2 paredes, a cobra e uma comida.
2. O jogo é iniciado.
3. Sub-Caso de Uso 1
4. Sub-Caso de Uso 2
5. A cobra come a última comida do tabuleiro.
6. O jogo termina.
7. O score é exibido.

Extensões:

5. A cobra bate nela mesma.
6. O jogo termina.
7. O score é exibido.

5. A cobra bate na parede.
6. O jogo termina.
7. O score é exibido.

5. Aparecem 20 comidas no tabuleiro ao mesmo tempo.
6. O jogo termina.
7. O score é exibido.

Tabela A.5: Caso de Uso Jogar Gula Gula 2

Caso de Uso Nº: 4

Nome: Jogar Magic Snake

Ator: Usuário

Objetivo: Jogar o Magic Snake onde o score é contado de acordo com o tamanho da cobra, levando-se em consideração que o jogo termina em um determinado tempo.

Cenário Principal:

1. O sistema cria o jogo com o tabuleiro, 4 paredes, a cobra e uma comida.
2. O jogo é iniciado.
3. Sub-Caso de Uso 1.
4. O jogo termina por ter se passado um determinado período de tempo.

Extensões:

4. A cobra passa por cima de uma comida normal ganhando 1 ponto no jogo.
5. A cobra aumenta o seu tamanho.
6. A cobra aumenta sua velocidade, caso esteja abaixo da velocidade limite.
7. Volta ao passo 4.

4. A cobra come uma comida especial ganhando 2 pontos no jogo.
5. A cobra aumenta o seu tamanho.
6. A cobra aumenta sua velocidade, caso esteja abaixo da velocidade limite.
7. Volta ao passo 4.

4. A cobra bate nela mesma.
5. O jogo termina.
6. O score é exibido.

4. A cobra bate nela mesma, mas está indestrutível.
5. A cobra atravessa a si mesma.
6. Volta ao passo 4.

4. A cobra bate na parede.
5. O jogo termina.
6. O score é exibido.

4. A cobra bate na parede, mas está indestrutível.
5. A cobra volta no sentido oposto.
6. Volta ao passo 4.

4. A cobra come 5 comidas do mesmo tipo e fica indestrutível por algum tempo.
5. A cobra produz fezes aleatoriamente.
6. Volta ao passo 4.

4. A cobra come um bolo fecal.
5. O jogo termina.
6. O score é exibido.

Caso de Uso Nº: 5

Nome: Jogar My Snake

Ator: Usuário

Objetivo: Jogar o My Snake onde o score é contado de acordo com o tamanho da cobra. O jogo só termina quando a cobra morre, batendo na parede ou nela mesma.

Cenário Principal:

1. O sistema cria o jogo com o tabuleiro, 4 paredes, a cobra e uma comida.
2. O jogo é iniciado.
3. Sub-Caso de Uso 1.
4. A cobra bate nela mesma.
5. O jogo termina.
6. O score é exibido.

Extensões:

5. A cobra passa por cima de uma comida normal ganhando 1 ponto no jogo.
6. A cobra aumenta o seu tamanho.
7. A cobra aumenta sua velocidade, caso esteja abaixo da velocidade limite.
8. Volta ao passo 4.

5. A cobra come uma comida estragada perdendo 2 pontos no jogo.
6. Um bloco de obstáculo é criado atrás da cobra.
7. A cobra aumenta sua velocidade, caso esteja abaixo da velocidade limite.
8. Volta ao passo 4.

5. A cobra come uma comida especial ganhando 2 pontos no jogo.
6. A cobra aumenta o seu tamanho.
7. A cobra aumenta sua velocidade, caso esteja abaixo da velocidade limite.
8. Volta ao passo 4.

5. A cobra bate na parede.
6. O jogo termina.
7. O score é exibido.

Tabela A.7: Caso de Uso Jogar My Snake

Caso de Uso Nº: 6

Nome: Escolher nível

Ator: Usuário

Objetivo: Escolher o nível do jogo que se deseja, podendo ser verme, coral e sucuri (velocidade).

Cenário Principal:

1. O usuário escolhe o nível do jogo que deseja jogar.

Tabela A.8: Caso de Uso Escolher Nível

Caso de Uso Nº: 7

Nome: Visualizar Scores

Ator: Usuário

Objetivo: Ver o histórico dos pontos marcados.

Cenário Principal:

1. O usuário deseja ver o score do jogo.
2. Sistema exibe o histórico dos scores acumulados.

Tabela A.9: Caso de Uso Visualizar Scores

Caso de Uso Nº: 8

Nome: Escolher Jogo

Ator: Usuário

Objetivo: Escolher qual dos jogos existentes se deseja jogar.

Cenário Principal:

1. O usuário escolhe qual jogo se deseja jogar.
2. O sistema armazena informações sobre o jogo escolhido.

Tabela A.10: Caso de Uso Escolher Jogo

Apêndice B

Modelos de Informação

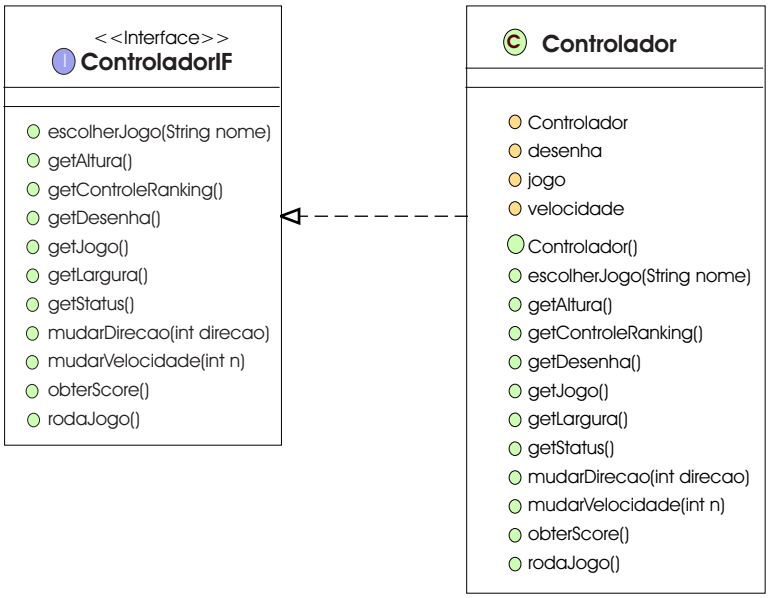

Figura B.1: Modelo de Informação da Interface do Componente Controlador

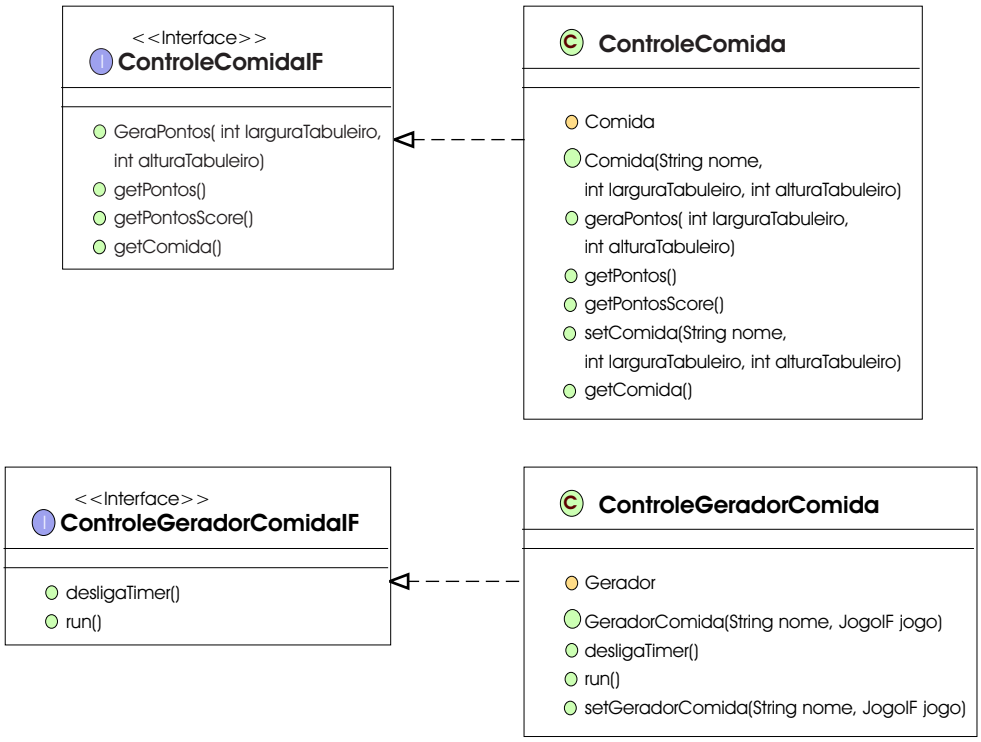

Figura B.2: Modelo de Informação da interface do Componente Comida

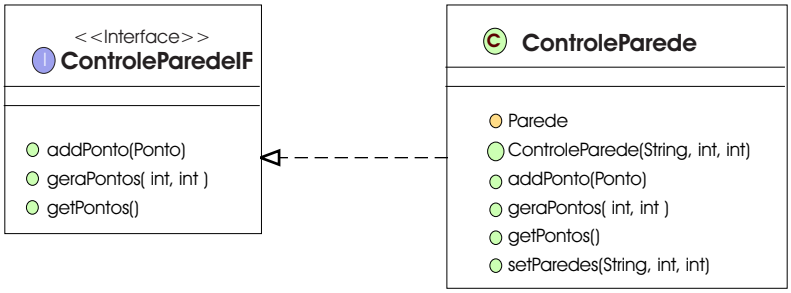

Figura B.3: Modelo de Informação da interface do Componente Parede

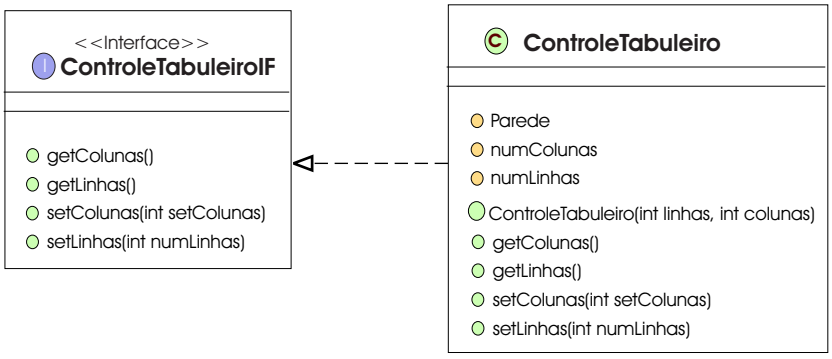

Figura B.4: Modelo de Informação da interface do Componente Tabuleiro

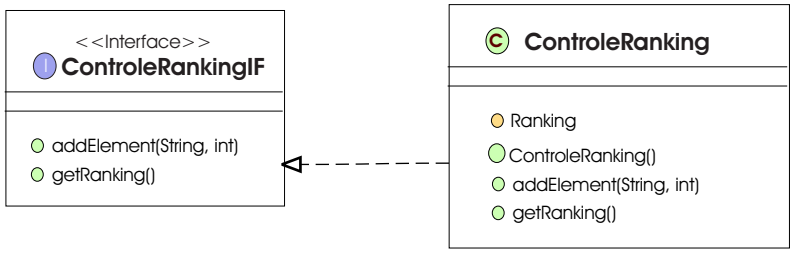

Figura B.5: Modelo de Informação da interface do Componente Ranking

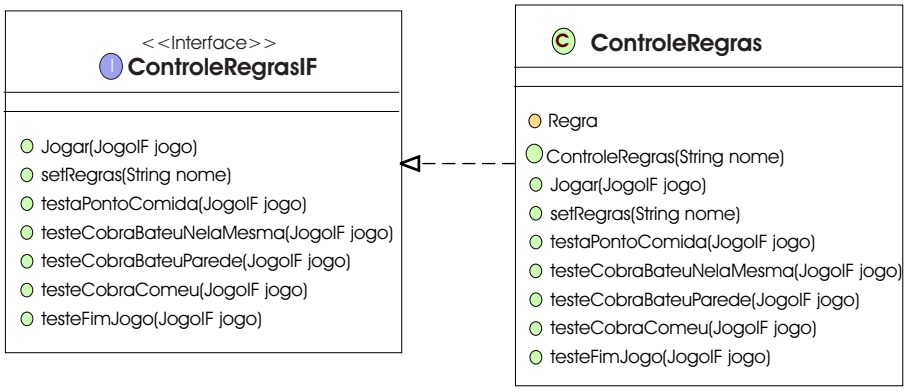

Figura B.6: Modelo de Informação da interface do Componente Regras

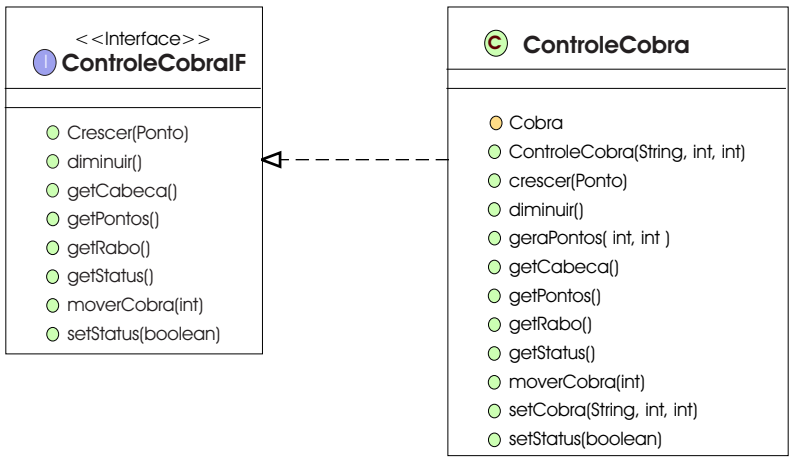

Figura B.7: Modelo de Informação da interface do Componente Cobra

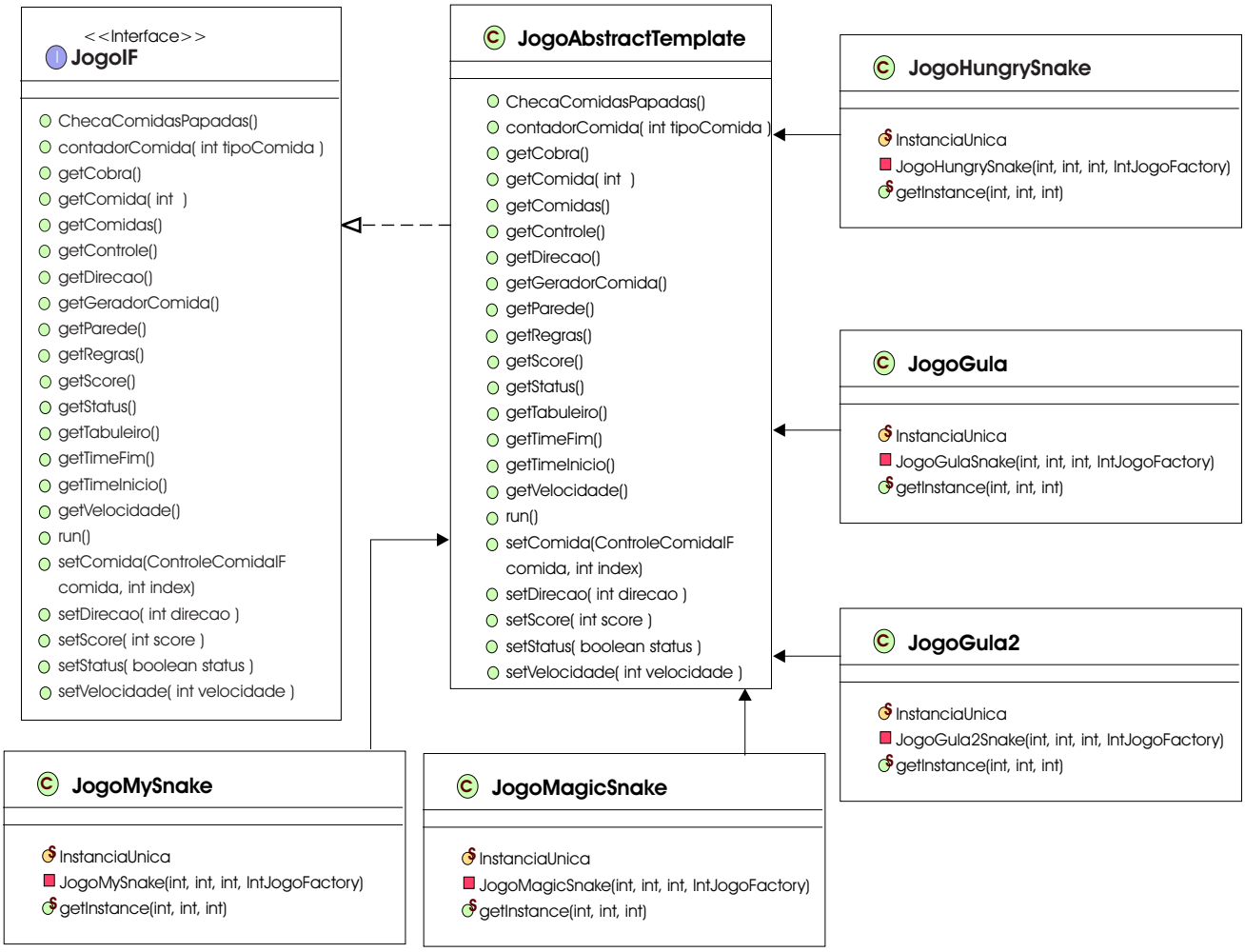

Figura B.8: Modelo de Informação da interface do Componente Jogo

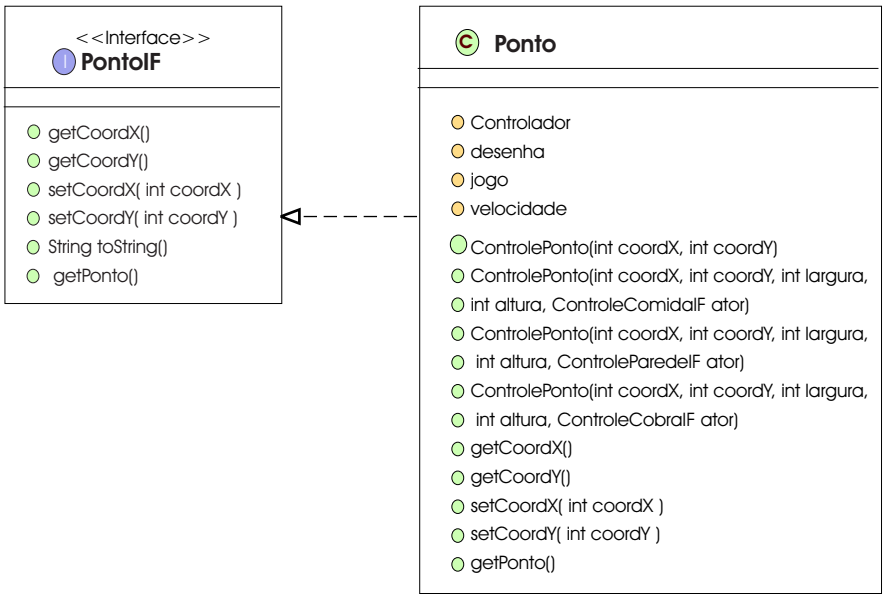

Figura B.9: Modelo de Informação da interface do Componente Ponto

Apêndice C

Diagramas de Seqüência referente ao Componente Jogo

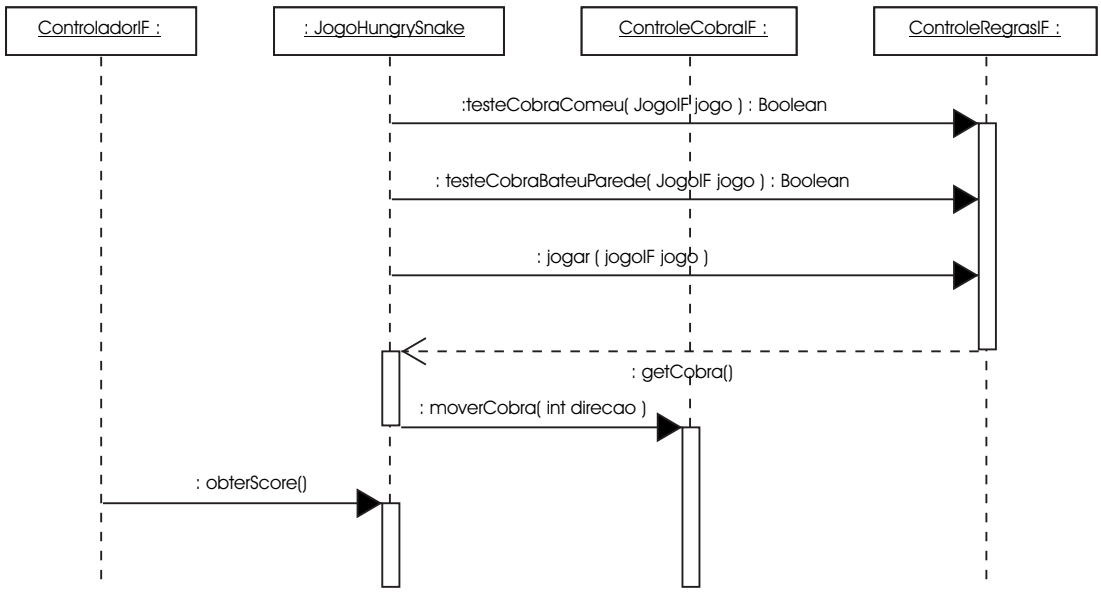

Figura C.1: Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, onde o jogo termina por a cobra ter batido na parede, referente ao Componente Jogo

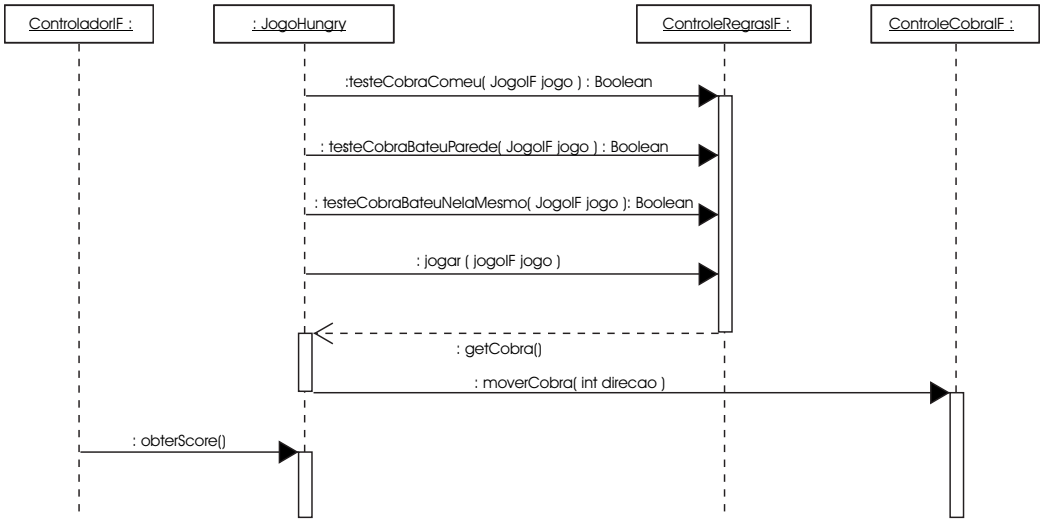

Figura C.2: Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde a cobra bate nela mesmo, referente ao Componente Jogo

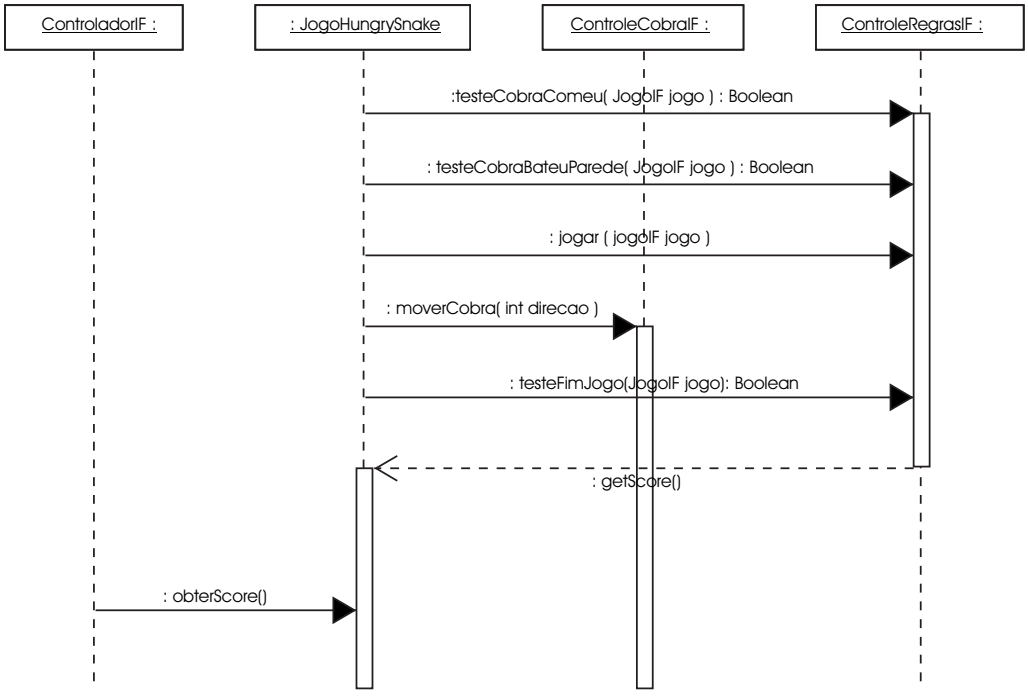

Figura C.3: Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde o jogo termina por ter obtido a pontuação máxima, referente ao Componente Jogo

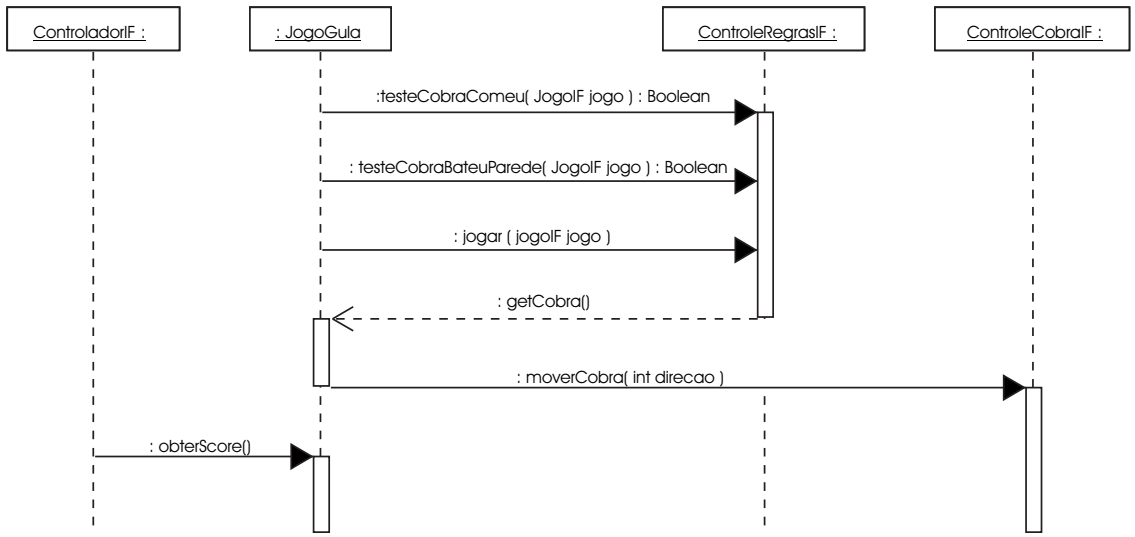

Figura C.4: Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina por a cobra ter batido na parede, referente ao Componente Jogo

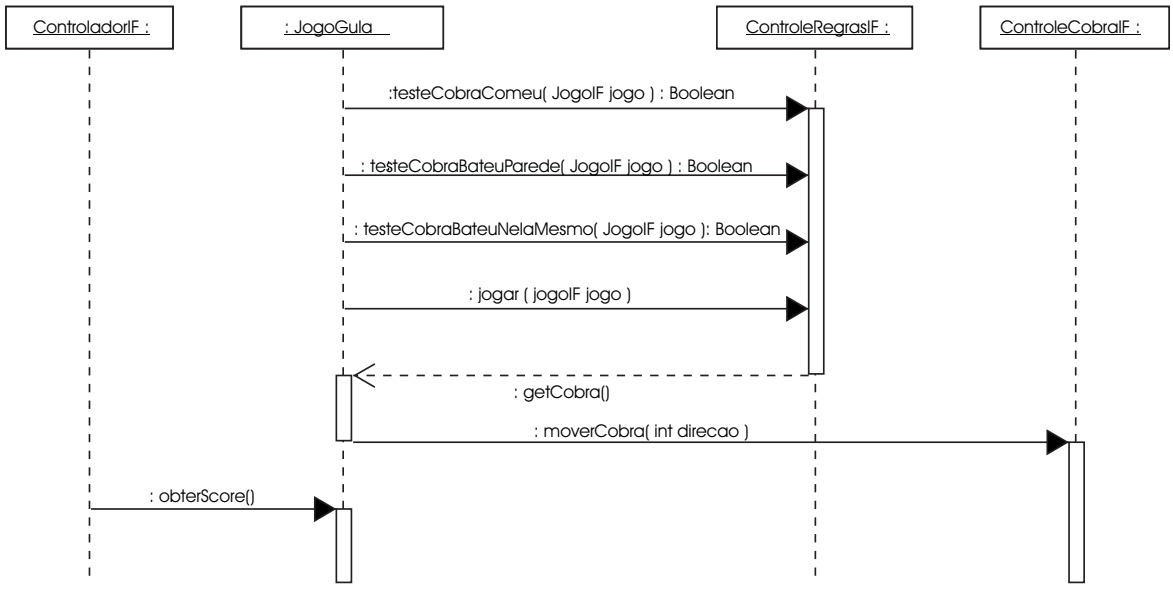

Figura C.5: Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina porque a cobra bateu nela mesmo, referente ao Componente Jogo

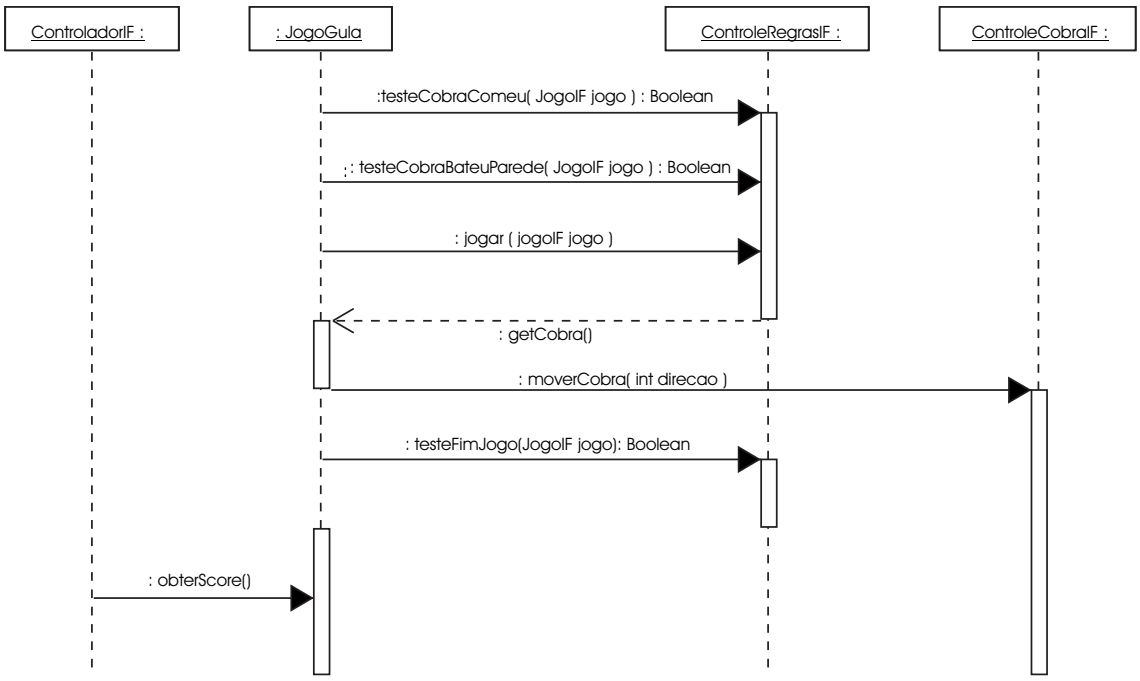

Figura C.6: Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina por ter obtido a pontuação máxima, referente ao Componente Jogo

Apêndice D

Diagramas de Seqüência referente ao Componente Regras

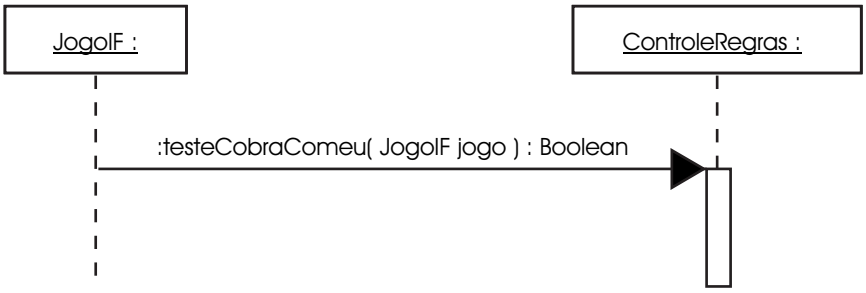

Figura D.1: Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde a cobra come uma comida, referente ao Componente Regras

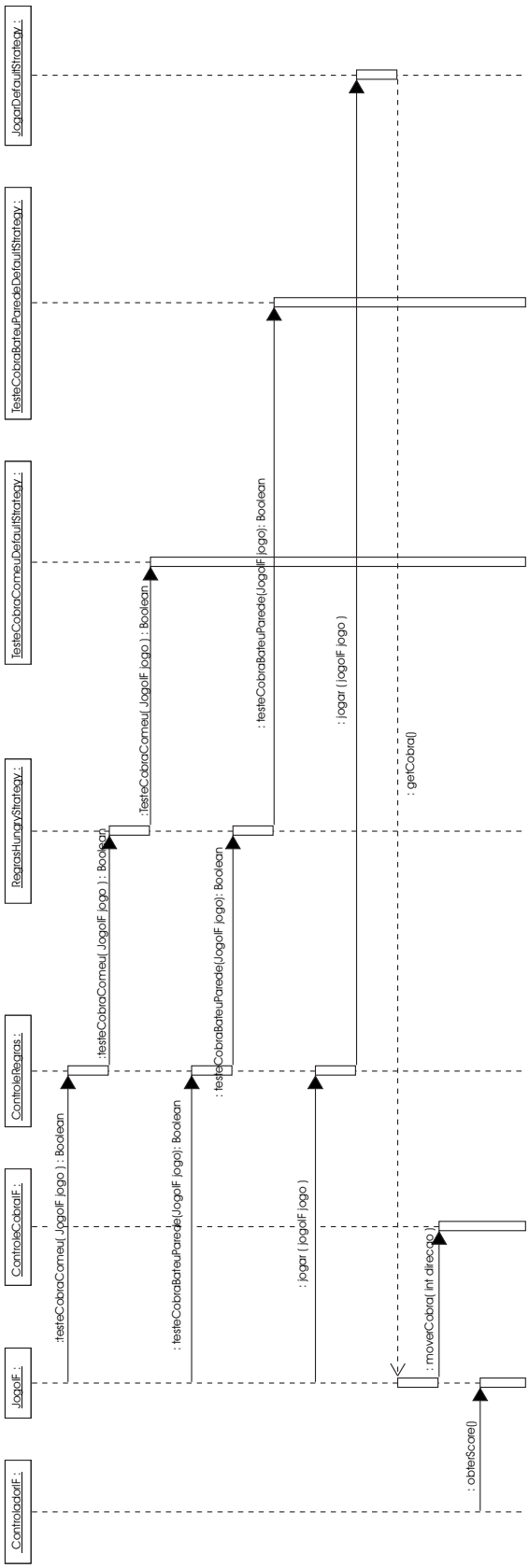

Figura D.2: Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde o jogo termina por a cobra ter batido na parede, referente ao Componente Regras

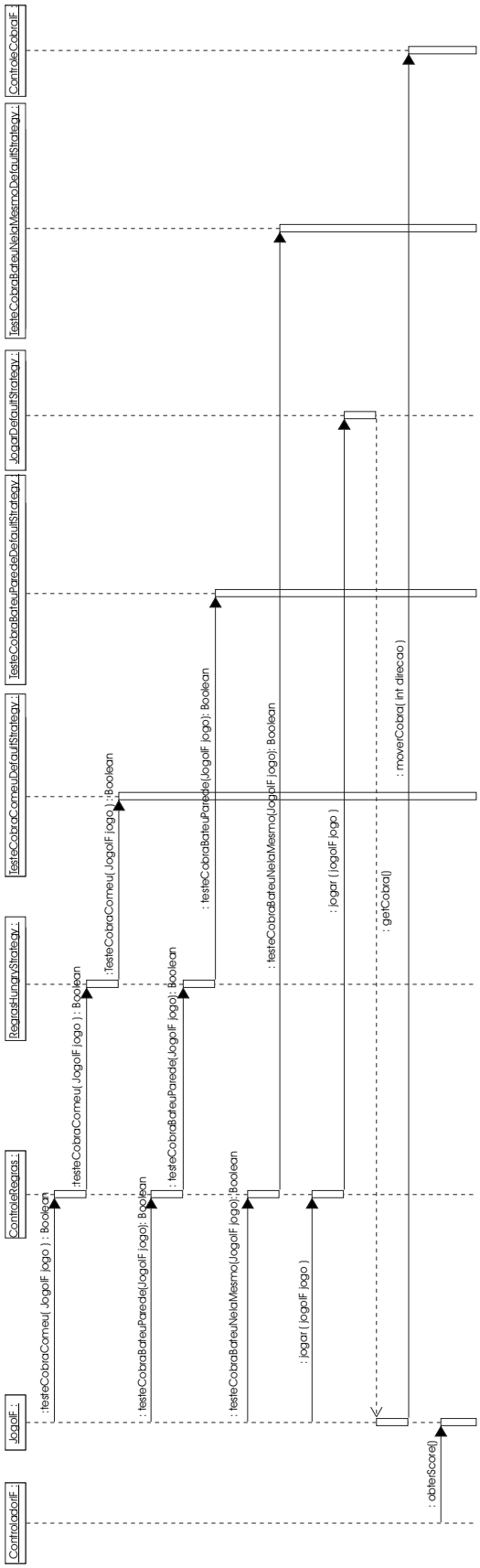

Figura D.3: Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde o jogo termina porque a cobra bateu nela mesmo, referente ao Componente Regras

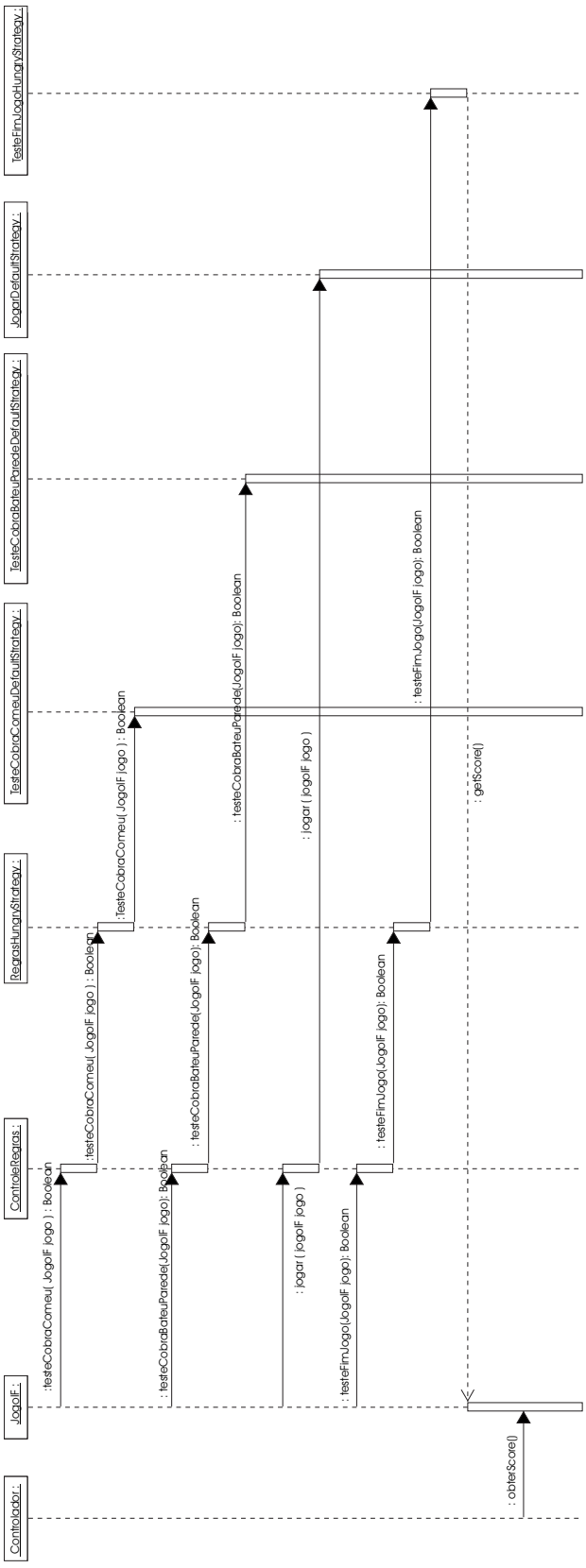

Figura D.4: Diagrama de Seqüência do Caso de Uso Jogar Hungry Snake, no cenário onde o jogo termina por ter obtido a pontuação máxima, referente ao Componente Regras

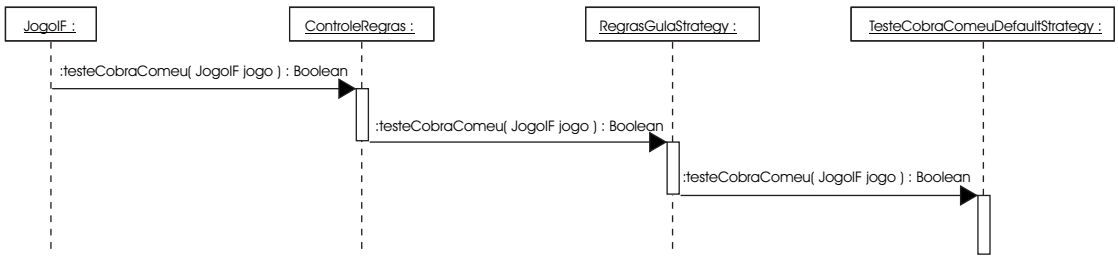

Figura D.5: Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde a cobra come uma comida, referente ao Componente Regras

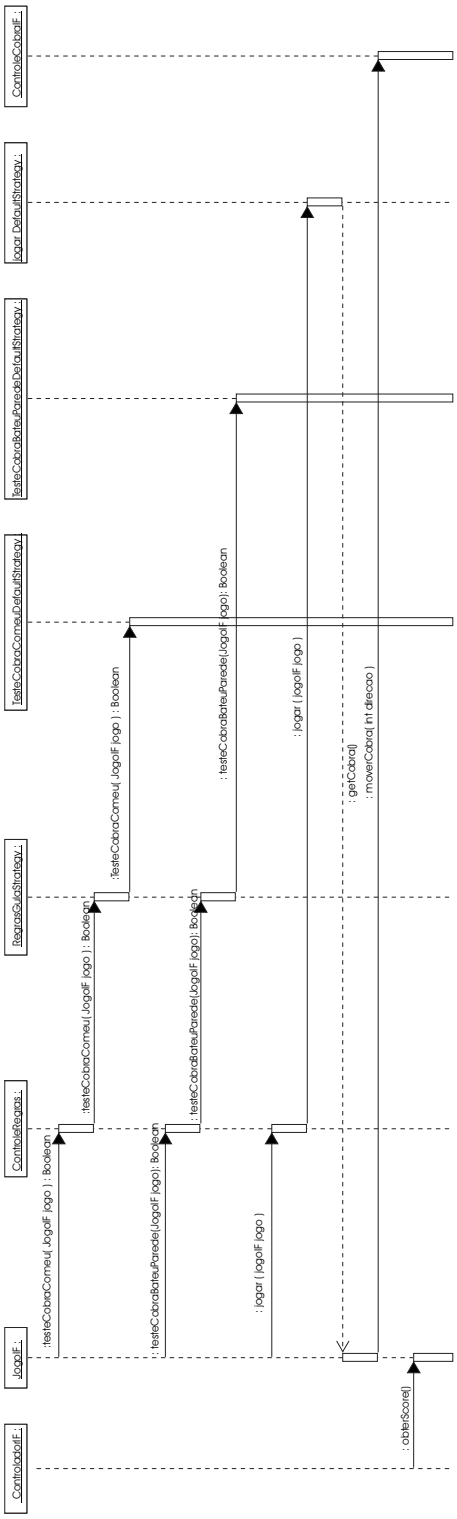

Figura D.6: Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina por a cobra ter batido na parede, referente ao Componente Regras

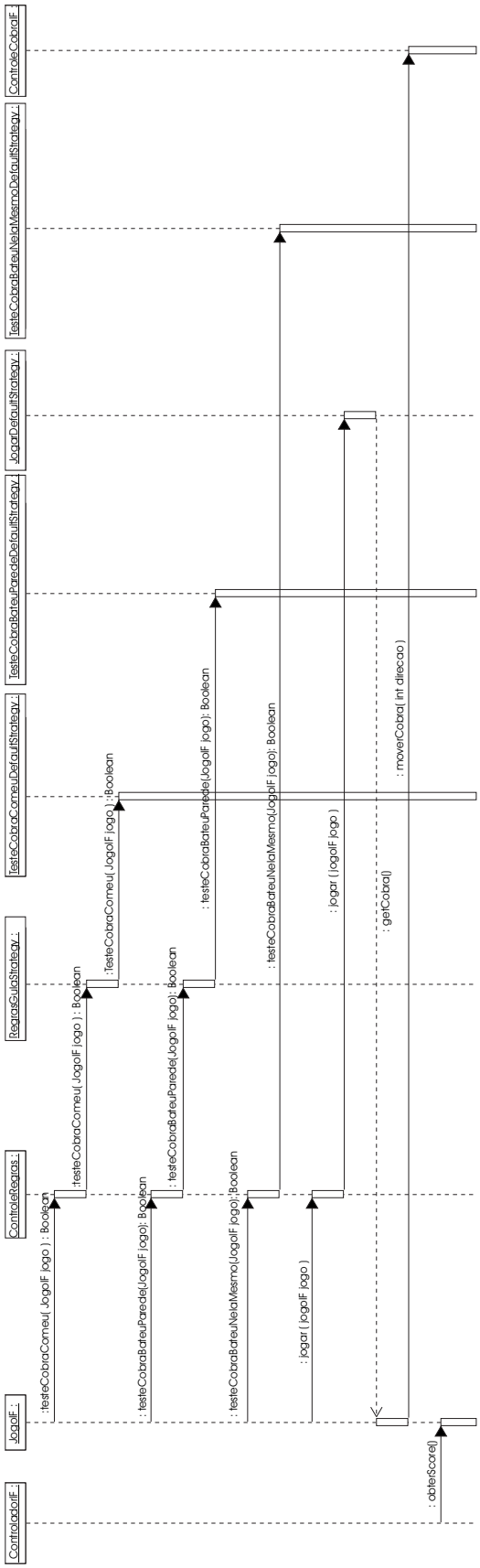

Figura D.7: Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina porque a cobra bateu nela mesmo, referente ao Componente Regras

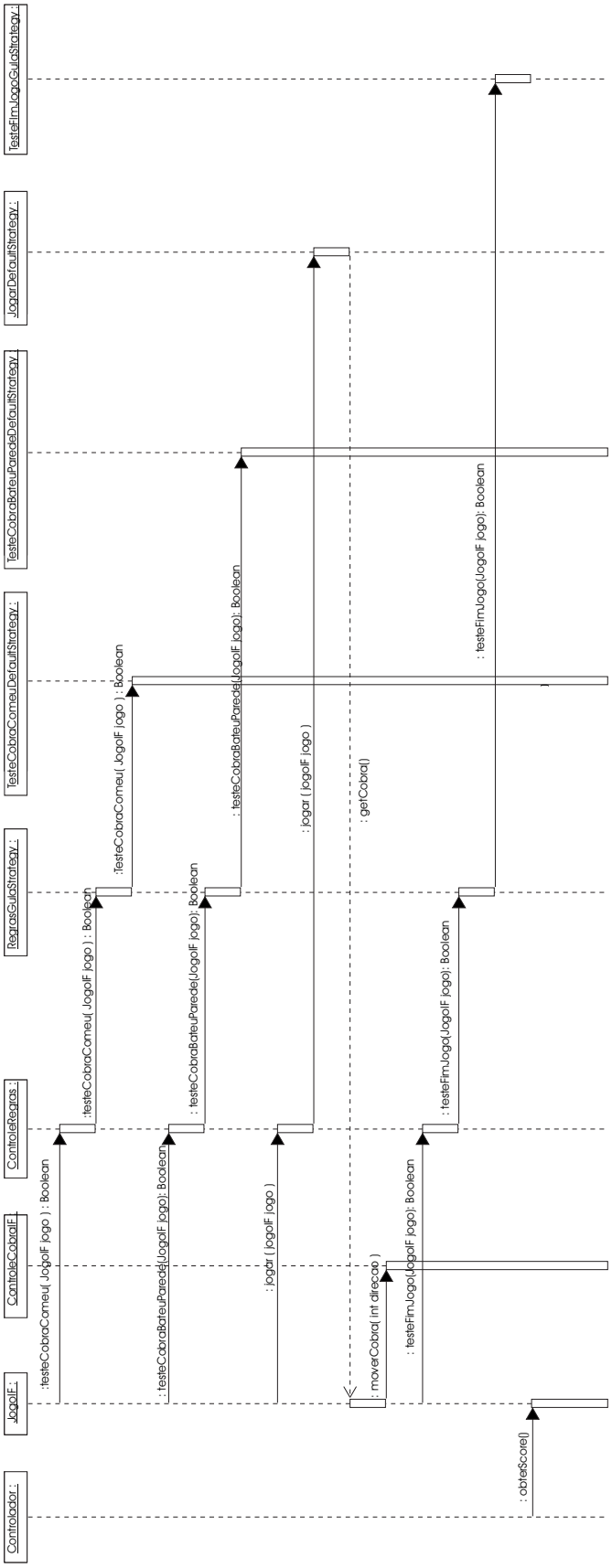

Figura D.8: Diagrama de Seqüência do Caso de Uso Jogar Gula Gula, no cenário onde o jogo termina por ter obtido a pontuação máxima, referente ao Componente Regras

Apêndice E

Expressões Regulares e OCL's

Expressão Regular para o Caso de Uso Jogar Hungry Snake, referente ao componente Jogo:

(1) $testeCobraComeu_{ControleRegrasIF}$

+

(2): $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.getCobra_{JogoHungrySnake}.$
$moverCobra_{ControleCobraIF}.obterScore_{JogoHungrySnake}$

+

(3) $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$testecobraBateuNelaMesmo_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.getCobra_{JogoHungrySnake}.$
$moverCobra_{ControleCobraIF}.$
$obterScore_{JogoHungrySnake}$

+

(4) $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.$
$getCobra_{JogoGula}.moverCobra_{ControleCobraIF}.$
$testeFimJogo_{ControleRegrasIF}.$
$getScore_{JogoHungrySnake}.setSatus_{JogoHungrySnake}.$
$obterScore_{JogoHungrySnake}$

Condições de execução para cada termo da expressão regular, expressa OCL:

```
A : Jogo.getComidas()->exists(tc |
            tc->includes(Jogo.getCobra().getCabeca()))


B : Jogo.getParede().getPontos()->
            includes(Jogo.getCobra().getCabeca())


C : Jogo.getCobra().getPontos()->
            includes(Jogo.getCobra().getCabeca())


D : not (Jogo.getComidas()->exists(tc |
            tc->includes(Jogo.getCobra().getCabeca())))
            and not (Jogo.getParede().getPontos()->
            includes(Jogo.getCobra().getCabeca()))
            and Jogo.getScore() >= 15
```

Expressão Regular para o Caso de Uso Jogar Gula Gula, referente ao componente Jogo:

(1) $testeCobraComeu_{ControleRegrasIF}$

+

(2): $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.getCobra_{JogoGula}.$
$moverCobra_{ControleCobraIF}.obterScore_{JogoGula}$

+

(3) $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$
$testecobraBateuNelaMesmo_{ControleRegrasIF}.$
$jogar_{ControleRegrasIF}.getCobra_{JogoGula}.$
$moverCobra_{ControleCobraIF}.$

$obterScore_{JogoGula}$

+

(4) $testeCobraComeu_{ControleRegrasIF}.testeCobraBateuParede_{ControleRegrasIF}.$

$jogar_{ControleRegrasIF}.$

$getCobra_{JogoGula}.moverCobra_{ControleCobraIF}.$

$testeFimJogo_{ControleRegrasIF}.$

$setSatus_{JogoGula}.$

$obterScore_{JogoGula}$

Condições de execução para cada termo da expressão regular, expressa OCL:

```
A : Jogo.getComidas()->exists(tc |
          tc->includes(Jogo.getCobra().getCabeca()))


B : Jogo.getParede().getPontos()->
          includes(Jogo.getCobra().getCabeca())


C : Jogo.getCobra().getPontos()->
          includes(Jogo.getCobra().getCabeca())


D : not ((jogo.getComidas()->forAll(tc | tc->size = 0) and
          jogo.getStatus() = true)
          or jogo.getComidas()->forAll(tc | tc->size = 20))
```

Expressão Regular para o Caso de Uso Jogar HungrySnake, referente ao componente Regras:

(1) $testeCobraComeu_{ControleRegras}.testeCobraComeu_{RegrasHungryStrategy}.$
$testeCobraComeu_{TesteCobraComeuDefaultStrategy}$ +
(2) $testeCobraComeu_{ControleRegras}.testeCobraComeu_{RegrasHungryStrategy}.$
$testeCobraComeu_{TesteCobraComeuDefaultStrategy}.testeCobraBateuParede_{ControleRegras}.$
$testeCobraBateuParede_{RegrasHungryStrategy}.$
$testeCobraBateuParede_{TesteCobraBateuParedeDefaultStrategy}.$
$jogar_{ControleRegras}.jogar_{JogarDefaultStrategy}.getCobra_{JogoIF}.$
$moverCobra_{ControleCobraIF}.obterScore_{JogoIF}$
+
(3)$testeCobraComeu_{ControleRegras}.testeCobraComeu_{RegrasHungryStrategy}.$
$testeCobraComeu_{TesteCobraComeuDefaultStrategy}.testeCobraBateuParede_{ControleRegras}.$
$testeCobraBateuParede_{RegrasHungryStrategy}.$
$testeCobraBateuParede_{TesteCobraBateuParedeDefaultStrategy}.$
$testecobraBateuNelaMesmo_{ControleRegras}.$
$testecobraBateuNelaMesmo_{TestecobraBateuNelaMesmoDefaultStrategy}.$
$jogar_{ControleRegras}.jogar_{JogarDefaultStrategy}.getCobra_{JogoIF}.$
$moverCobra_{ControleCobraIF}.$
$obterScore_{JogoIF}$
+
(4) $testeCobraComeu_{ControleRegras}.testeCobraComeu_{RegrasHungryStrategy}.$
$testeCobraComeu_{TesteCobraComeuDefaultStrategy}.testeCobraBateuParede_{ControleRegras}.$
$testeCobraBateuParede_{RegrasHungryStrategy}.$
$testeCobraBateuParede_{TesteCobraBateuParedeDefaultStrategy}.$
$jogar_{ControleRegras}.jogar_{JogarDefaultStrategy}.$
$testeFimJogoJogo_{RegrasHungryStrategy}.testeFimJogo_{TesteFimJogoHungryStrategy}.$
$getScore_{JogoIF}.$
$setSatus_{JogoIF}.$
$obterScore_{JogoIF}$

Condições de execução para cada termo da expressão regular, expressa OCL:

```
A : Jogo.getComidas()->exists(tc |
            tc->includes(Jogo.getCobra().getCabeca()))


B : Jogo.getParede().getPontos()->
            includes(Jogo.getCobra().getCabeca())


C : Jogo.getCobra().getPontos()->
            includes(Jogo.getCobra().getCabeca())


D : not (Jogo.getComidas()->exists(tc |
            tc->includes(Jogo.getCobra().getCabeca())))
            and not (Jogo.getParede().getPontos()->
            includes(Jogo.getCobra().getCabeca()))
            and Jogo.getScore() >= 15
```

Expressão Regular para o Caso de Uso Jogar Gula Gula, referente ao componente Regras:

(1) $testeCobraComeu_{ControleRegras}.testeCobraComeu_{RegrasGulaStrategy}.$
$testeCobraComeu_{TesteCobraComeuDefaultStrategy}$ +
(2) $testeCobraComeu_{ControleRegras}.testeCobraComeu_{RegrasGulaStrategy}.$
$testeCobraComeu_{TesteCobraComeuDefaultStrategy}.testeCobraBateuParede_{ControleRegras}.$
$testeCobraBateuParede_{RegrasGulaStrategy}.$
$testeCobraBateuParede_{TesteCobraBateuParedeDefaultStrategy}.$
$jogar_{ControleRegras}.jogar_{JogarDefaultStrategy}.getCobra_{JogoIF}.$
$moverCobra_{ControleCobraIF}.obterScore_{JogoIF}$
+
(3)$testeCobraComeu_{ControleRegras}.testeCobraComeu_{RegrasGulaStrategy}.$
$testeCobraComeu_{TesteCobraComeuDefaultStrategy}.testeCobraBateuParede_{ControleRegras}.$
$testeCobraBateuParede_{RegrasGulaStrategy}.$
$testeCobraBateuParede_{TesteCobraBateuParedeDefaultStrategy}.$
$testecobraBateuNelaMesmo_{ControleRegras}.$
$testecobraBateuNelaMesmo_{TestecobraBateuNelaMesmoDefaultStrategy}.$
$jogar_{ControleRegras}.jogar_{JogarDefaultStrategy}.getCobra_{JogoIF}.$
$moverCobra_{ControleCobraIF}.$
$obterScore_{JogoIF}$
+
(4) $testeCobraComeu_{ControleRegras}.testeCobraComeu_{RegrasGulaStrategy}.$
$testeCobraComeu_{TesteCobraComeuDefaultStrategy}.testeCobraBateuParede_{ControleRegras}.$
$testeCobraBateuParede_{RegrasGulaStrategy}.$
$testeCobraBateuParede_{TesteCobraBateuParedeDefaultStrategy}.$
$jogar_{ControleRegras}.jogar_{JogarDefaultStrategy}.$
$testeFimJogoJogo_{RegrasGulaStrategy}.testeFimJogo_{TesteFimJogoGulaStrategy}.$
$getScore_{JogoIF}.$
$setSatus_{JogoIF}.$
$obterScore_{JogoIF}$

Condições de execução para cada termo da expressão regular, expressa OCL:

```
A : Jogo.getComidas()->exists(tc |
```

```
            tc->includes(Jogo.getCobra().getCabeca()))

B : Jogo.getParede().getPontos()->
            includes(Jogo.getCobra().getCabeca())

C : Jogo.getCobra().getPontos()->
            includes(Jogo.getCobra().getCabeca())

D : not ((jogo.getComidas()->forAll(tc | tc->size = 0)
            and jogo.getStatus() = true)
            or jogo.getComidas()->forAll(tc | tc->size = 20))
```

Apêndice F

Especificações das Operações das Interfaces

```
Operação:          public boolean escolherJogo(String nome);
Descrição:         permite escolher o jogo que se deseja jogar
Entradas:          nome: String  o nome do jogo que deseja se jogar
Saídas:
Pré-Condições:     o nome é uma string válida
Pós-Condições:     o jogo é escolhido com sucesso
OCL:               context Controlador :: escolherJogo(nome: String): Boolean
                   pre: nome = "MySnake" or nome  = "Gula" or nome  = "Gula2”
                        or nome  = "MagicSnake" or nome  = "HungrySnake”
                   post: result = true
```

Figura F.1: Especificação da operação escolherJogo(String nome) do componente Controlador

```
Operação:          public void mudarDirecao( int direcao );
Descrição:         muda a direção que a cobra anda
Entradas:          direção: int - DIRECAO_CIMA = 38; DIRECAO_BAIXO = 40
                   DIRECAO_ESQUERDA = 37; DIRECAO_DIREITA = 39
Saídas:
Pré-Condições:     O jogo deve estar rodando e
                   a direção informada deve ser um valor válido.
Pós-Condições:     A cobra passa a se mover na nova
                   direção passada como parâmetro.
OCL:               context Controlador :: mudarDirecao(direcao:Integer )
                   pre: getStatus() = true and
                        ( (direcao  = 37) or (direcao  = 38) or
                        (direcao  = 39) or (direcao  = 40) )
                   Post: getJogo().getDirecao() = direcao
```

Figura F.2: Especificação da operação rodaJogo() do componente Controlador

```
Operação:          public void mudarDirecao( int direcao );
Descrição:         muda a direção que a cobra anda
Entradas:          direção: int - DIRECAO_CIMA = 38; DIRECAO_BAIXO = 40
                   DIRECAO_ESQUERDA = 37; DIRECAO_DIREITA = 39
Saídas:
Pré-Condições:     O jogo deve estar rodando e
                   a direção informada deve ser um valor válido.
Pós-Condições:     A cobra passa a se mover na nova
                   direção passada como parâmetro.
OCL:               context Controlador :: mudarDirecao(direcao:Integer )
                   pre: getStatus() = true and
                        ( (direcao  = 37) or (direcao  = 38) or
                        (direcao  = 39) or (direcao  = 40) )
                   post: getJogo().getDirecao() = direcao
```

Figura F.3: Especificação da operação mudarDirecao(int direcao) do componente Controlador

```
Operação:          public void mudarVelocidade(int n);
Descrição:         muda a velocidade que a cobra anda
Entradas:          n: int - valor da nova velocidade
Saídas:
Pré-Condições:     O jogo deve estar rodando
                   e a velocidade deve ser maior que zero.
Pós-Condições:     A cobra passa a se mover com
                   uma nova velocidade.
OCL:               context Controlador :: mudarVelocidade( n : Integer )
                   pre:  getStatus() = true and n > 0
                   post: getJogo().getVelocidade() = n
```

Figura F.4: Especificação da operação mudarVelocidade(int n) do componente Controlador

```
Operação:          public int obterScore();
Descrição:         obtém a pontuação atual
Entradas:
Saídas:            número atual de pontos acumulados
Pré-Condições:     O jogo deve estar rodando
Pós-Condições:
OCL:               context Controlador :: obterScore()
                   Pre: getStatus() = true
```

Figura F.5: Especificação da operação obterScore() do componente Controlador

```
Operação:          public void geraPontos( int larguraTabuleiro, int
                    alturaTabuleiro );
Descrição:         gera um Ponto aleatório válido no Tabuleiro para a
                    Comida
Pré-Condições:     os parâmetros passados devem ser maior que zero
Entradas:          larguraTabuleiro: int  largura do tabuleiro
                   alturaTabuleiro: int  altura do tabuleiro
Saídas:
Pós-Condições:     mais um ponto deve ser gerado no tabuleiro para uma
                    comida
OCL:               context Comida :: geraPontos( larguraTabuleiro: Integer,
                    alturaTabuleiro: Integer)
                   pre:  larguraTabuleiro > 0 and alturaTabuleiro > 0
                   post: self.getPontos()->size() =
                         self.getPontos()@pre->size() + 1
```

Figura F.6: Especificação da operação geraPontos(int larguraTabuleiro, int alturaTabuleiro) do componente Comida

```
Operação:          public int getPontosScore();
Descrição:         Método que retorna o número de pontos que vale uma
                    comida para o ranking
Entradas:
Saídas:            número de pontos que vale uma Comida para o ranking
Pré-Condições:
Pós-Condições:     os pontos do ranking que vale cada comida só pode ser
                   1, 2 ou -2, para comida simples, especial ou estragada
OCL:               context Comida :: getPontosScore(): Integer
                   post: if getComida().oclIsTypeOf(Comida) then
                            result = 1
                         else
                          if getComida().oclIsTypeOf(ComidaEspecial)then
                              result = 2
                          else
                           if getComida().oclIsTypeOf(ComidaEstragada)then
                               result = -2
                          endif
                         Endif
```

Figura F.7: Especificação da operação getPontosScore() do componente Comida

```
Operação:         public Ponto getRabo();
Descrição:        Método que retorna o Ponto correspondente ao Rabo de
                   uma Cobra
Entradas:
Saídas:           retorna o ponto correspondente ao rabo da cobra
Pré-Condições:
Pós-Condições:    o rabo deve ser um ponto válido da cobra.
OCL:              context Cobra :: getRabo(): Ponto
                  post: self.getPontos()->includes(result)
```

Figura F.8: Especificação da operação getRabo() do componente Cobra

```
Operação:         public Ponto getCabeca();
Descrição:        Método que retorna o Ponto correspondente ao Cabeça de
                   uma Cobra.
Entradas:
Saídas:           retorna o ponto correspondente à cabeça da cobra
Pré-Condições:
Pós-Condições:    a cabeça deve ser um ponto válido da cobra.
OCL:              context Cobra:: getCabeca(): Ponto
                  post: self.getPontos()->includes(result)
```

Figura F.9: Especificação da operação getCabeca() do componente Cobra

```
Operação:         public void moverCobra( int direcao );
Descrição:        faz a cobra se mover na direção indicada
Entradas:         direcao: int  a direção que a cobra deve se mover
Saídas:
Pré-Condições:    a direção passada como parâmetro deve ser igual a 37,
                   38, 39 ou 40
Pós-Condições:    se a direção for para:
                   esquerda(37)- a coordenada y continua a mesma e a x diminui 1
                   cima(38)    - a coordenada x continua a mesma e a y diminui 1
                   direita(39) - a coordenada y continua a mesma e a x aumenta 1
                   baixo(40)   - a coordenada x continua a mesma e a y aumenta 1
OCL:              context Cobra:: moverCobra( direcao: Integer )
                  pre: (direcao  = 37) or (direcao  = 38) or (direcao  =
                        39) or (direcao  = 40)
                  post: if direcao = 37 then
                          getCabeca.getCoordX() = getCabeca.getCoordX()@pre - 1
                          and getCabeca.getCoordY() = getCabeca.getCoordY()@pre
                        else
                          if direcao = 38 then
                            getCabeca.getCoordY() = getCabeca.getCoordY()@pre-1
                            and getCabeca.getCoordX() = getCabeca.getCoordX()@pre
                          else
                            if direcao = 39 then
                             getCabeca.getCoordX() = getCabeca.getCoordX()@pre + 1
                             and getCabeca.getCoordY() = getCabeca.getCoordY()@pre
                            else
                              if direcao = 40 then
                               getCabeca.getCoordY() = getCabeca.getCoordY()@pre + 1
                               and getCabeca.getCoordX() = getCabeca.getCoordX()@pre
                              endif
                            endif
                          endif
                        endif
```

Figura F.10: Especificação da operação moverCobra(int direcao) do componente Cobra

```
Operação:         public void crescer( Ponto crescer );
Descrição:        faz a cobra aumentar ser tamanho
Entradas:         crescer: Ponto - o ponto que a cobra deve crescer
Saídas:
Pré-Condições:
Pós-Condições:    a cobra tem que ter seu tamanho aumentado.
OCL:              context Cobra:: crescer( Ponto crescer )
                  post: getPontos() = getPontos()@pre->including(crescer)
```

Figura F.11: Especificação da operação crescer(Ponto crescer) do componente Cobra

```
Operação:         public void geraPontos( int larguraTabuleiro, int alturaTabuleiro );
Descrição:        gera o conjunto de Pontos da Parede
Entradas:         larguraTabuleiro: int  largura do tabuleiro
                  alturaTabuleiro: int  altura do tabuleiro
Saídas:
Pré-Condições:    os parâmetros passados devem ser maior que zero.
Pós-Condições:    o número de pontos que constitui a parede deve ser igual
                  a duas vezes largura do tabuleiro
                  (para constituir a parte inferior e superior)
                  mais duas vezes a sua altura (para constituir as partes laterais),
                  retirando os pontos em comum.
OCL:              context Parede :: geraPontos( larguraTabuleiro: int,
                          AlturaTabuleiro: int )
                  pre:  larguraTabuleiro > 0 and alturaTabuleiro > 0
                  post: self.getPontos()->size() = (alturaTabuleiro * 2 +
                        larguraTabuleiro  * 2) - 4
```

Figura F.12: Especificação da operação geraPontos(int larguraTabuleiro, int alturaTabuleiro) do componente Parede

```
Operação:                 public void addPonto( Ponto umPonto ) ;
Descrição:                adiciona uma ponto a parede
Entradas:                 crescer: Ponto  ponto que deve ser adicionado à parede
Saídas:
Pré-Condições:
Pós-Condições:
OCL:                      context Parede :: addPonto( Ponto umPonto )
                          post: getPontos() = getPontos()@pre->including(umPonto)
```

Figura F.13: Especificação da operação addPonto(Ponto umPonto) do componente Parede

```
Operação:         public int getLinhas();
Descrição:        retorna o número de Linhas de um Tabuleiro
Entradas:
Saídas:           int - número de Linhas de um Tabuleiro
Pré-Condições:
Pós-Condições:    o retorno da função deve ser igual ao valor do atributo numLinhas
OCL:              context Tabuleiro :: getLinhas(): Integer
                  post: result = numlinhas
```

Figura F.14: Especificação da operação getLinhas() do componente Tabuleiro

```
Operação:         public int getColunas();
Descrição:        retorna o número de Colunas de um Tabuleiro
Entradas:
Saídas:           int - número de Colunas de um Tabuleiro
Pré-Condições:
Pós-Condições:    o retorno da função deve ser igual ao valor do atributo numColunas
OCL:              context Tabuleiro :: getColunas(): Integer
                  post: result = numColunas
```

Figura F.15: Especificação da operação getColunas() do componente Tabuleiro

```
Operação:         public void setLinhas( int numLinhas );
Descrição:        Modifica o valor do numero de Linhas de um Tabuleiro.
Entradas:         numLinhas: int  novo número de linhas do tabuleiro
Saídas:
Pré-Condições:
Pós-Condições:    o número de linhas atual deve ser o que foi passado como parâmetro
OCL:              context Tabuleiro :: setLinhas( numLinhas: Integer )
                  post: self.numLinhas = numLinhas
```

Figura F.16: Especificação da operação setLinhas(int numLinhas) do componente Tabuleiro

```
Operação:         public void setColunas( int numColunas );
Descrição:        Modifica o valor do numero de Colunas de um Tabuleiro.
Entradas:         numColunas: int  novo número de colunas do tabuleiro
Saídas:
Pré-Condições:
Pós-Condições:    o número de colunas atual deve ser o que foi passado como parâmetro
OCL:              context Tabuleiro :: setColunas( numColunas: Integer )
                  post: self.numColunas = numColunas
```

Figura F.17: Especificação da operação setColunas(int numColunas) do componente Tabuleiro

```
Operação:         public void addElement(String nome, int score);
Descrição:        adiciona um novo elemento ao ranking
                  contendo um nome a pontuação.
Entradas:         nome: String  o nome do jogador
                  score: int  os pontos do jogador
Saídas:
Pré-Condições:    os pontos do jogador, que é passado como parâmetro,
                  Deve ser maior que zero
```

Figura F.18: Especificação da operação (String nome, int score) do componente Ranking

```
Operação:         public void setRegras(String nome);
Descrição:        cria a regra do jogo em questão
Entradas:         nome: String  nome que referencia o jogo para o qual
                  devem ser criadas as regras
Saídas:
Pré-Condições:    o nome passado como parâmetro deve ser um dos nomes do domínio:
                  "RegrasGula", "RegrasGula2", "RegrasHungry", "RegrasMagic" ou
                  "RegrasMySnake"

Pós-Condições:
OCL:              context Regras :: setRegras( nome: String )
                  pre: nome = "RegrasGula" or nome  = "RegrasGula2" or
                       nome  = "RegrasHungry" or nome  = "RegrasMagic" or
                       Nome  = "RegrasMySnake"
```

Figura F.19: Especificação da operação setRegras(String nome) do componente Regras

```
Operação:         public boolean testeCobraComeu( JogoIF jogo ) ;
Descrição:        testa se a cobra comeu uma comida
Entradas:         jogo: JogoIF  o jogo que está rodando
Saídas:           true  caso a cobra tenha comido uma comida
                  false  caso acobra não tenha comido uma comida
Pré-Condições:    o jogo deve ser rodando
Pós-Condições:    se a cabeça da cobra estiver no mesmo ponto de uma comida retorna true
OCL:              context Regras :: testeCobraComeu( jogo: JogoIF ): Boolean
                  pre:  jogo.getStatus() = true
                  post: if (jogo.getComidas()->exists
                        (tc | tc->includes(jogo.getCobra().getCabeca()))) then
                          result = true
                        else
                          result = false
                        endif
```

Figura F.20: Especificação da operação testeCobraComeu(JogoIF jogo) do componente Regras

```
Operação:         public boolean testeCobraBateuParede( JogoIF jogo ) ;
Descrição:        testa se a cobra bateu na parede
Entradas:         jogo: JogoIF  o jogo que está rodando
Saídas:           true -  caso a cobra tenha batido na parede e não esteja invencível
                          se o jogo for MagicSnake
                  false - caso a cobra não tenha batido na parede ou tenha batido
                          quando o jogo for MagicSnake e a cobra estiver invencível
Pré-Condições:    o jogo passado como parâmetro deve ser do tipo de um dos cinco jogos
                   existentes
Pós-Condições:    se o jogo for do tipo MagicSnake e a cobra estiver invencível ao bater
                   na parede, a mesma irá retornar na direção contrária e a operação
                   deverá retornar false. Caso a cobra não tenha batido na parede a
                   operação deverá retornar false. Caso o jogo não seja MagicSnake e a
                   cobra bata na parede, deverá ser retornado true.
OCL:              context Regras :: testeCobraBateuParede( jogo: JogoIF ): Boolean
                  pre: jogo.oclIsTypeOf(JogoHungrySnake) or jogo.oclIsTypeOf(JogoGula)
                       or jogo.oclIsTypeOf(JogoGula2) or
                       jogo.oclIsTypeOf(JogoMagicSnake) or jogo.oclIsTypeOf(JogoMySnake)
                  post: if (jogo.getParede().getPontos()
                                ->includes(jogo.getCobra().getCabeca())) then
                           if (jogo.oclIsTypeOf(JogoMagicSnake) and
                                    (jogo.getCobra().getStatus() = true)) then
                              result = false and
                              if jogo.getDirecao()@pre = 40 then
                                 jogo.getDirecao() = 38
                              else
                                 if jogo.getDirecao()@pre = 38 then
                                    jogo.getDirecao() = 40
                                 else
                                   if jogo.getDirecao()@pre = 37 then
                                      jogo.getDirecao() = 39
                                   Else
                                     if jogo.getDirecao()@pre = 39 then
                                        jogo.getDirecao() = 37
                                     endif
                                   endif
                                 endif
                              endif
                           else
                             result = true
                           endif
                         else
                            result = false
                       endif
```

Figura F.21: Especificação da operação testeCobraBateuParede(JogoIF jogo) do componente Regras

```
Operação:         public boolean testeCobraBateuNelaMesma( JogoIF jogo );
Descrição:        testa se a cobra passou por cima dela mesma
Entradas:         jogo: JogoIF  o jogo que está rodando
Saídas:           true - caso a cobra tenha batido nela mesmo e não esteja invencível
                         se o jogo for MagicSnake
                  false - caso a cobra não tenha batido nela mesmo ou tenha batido
                          quando o jogo for MagicSnake e a cobra estiver invencível
Pré-Condições:    o jogo passado como parâmetro deve ser do tipo de um dos cinco jogos
                   existentes
Pós-Condições:    se o jogo for do tipo MagicSnake e a cobra estiver invencível ao
                   passar por cima dela mesmo, o jogo não acabará e a operação deverá
                   retornar false. Caso a cobra não tenha passado por cima dela mesmo a
                   operação deverá retornar false. Caso o jogo não seja MagicSnake e a
                   cobra passe por cima dela mesmo, deverá ser retornado true.
OCL:              context Regras :: testeCobraBateuNelaMesma ( jogo: JogoIF ): Boolean
                  pre: jogo.oclIsTypeOf(JogoHungrySnake) or jogo.oclIsTypeOf(JogoGula)
                       or jogo.oclIsTypeOf(JogoGula2)
                       or jogo.oclIsTypeOf(JogoMagicSnake) or
                       jogo.oclIsTypeOf(JogoMySnake)
                  post: if (jogo.oclIsTypeOf(JogoMagicSnake) and
                            jogo.getCobra().getStatus() = false) or (not
                            jogo.oclIsTypeOf(JogoMagicSnake) ) then
                           if (jogo.getCobra().getPontos()
                               ->includes(jogo.getCobra().getCabeca())) then
                                result = true
                           else
                                result = false
                           endif
                        endif
```

Figura F.22: Especificação da operação testeCobraBateuNelaMesma(JogoIF jogo) do componente Regras

```
Operação:          public void jogar( JogoIF jogo ) ;
Descrição:         realiza uma jogada, ouseja, faz a cobra se movimentar
Entradas:          jogo: JogoIF  o jogo que está rodando
Saídas:
Pré-Condições:     o jogo passado como parâmetro deve ser do tipo de um dos cinco jogos
                    existentes
Pós-Condições:     se a direção atual for:
                    esquerda(37)- a coordenada y continua a mesma e a x diminui 1
                    cima(38)    - a coordenada x continua a mesma e a y diminui 1
                    direita(39) - a coordenada y continua a mesma e a x aumenta 1
                    baixo(40)   - a coordenada x continua a mesma e a y aumenta 1

OCL:               context Regras :: jogar ( jogo: JogoIF )
                   pre: jogo.oclIsTypeOf(JogoHungrySnake) or jogo.oclIsTypeOf(JogoGula)
                        or jogo.oclIsTypeOf(JogoGula2)
                        or jogo.oclIsTypeOf(JogoMagicSnake)
                        or jogo.oclIsTypeOf(JogoMySnake)
                   post: if jogo.getDirecao() = 37 then
                           jogo.getCobra().getCabeca.getCoordX() =
                           jogo.getCobra().getCabeca.getCoordX()@pre  1 and
                           jogo.getCobra().getCabeca.getCoordY() =
                           jogo.getCobra().getCabeca.getCoordY()@pre
                         else
                           if jogo.getDirecao() = 38 then
                              jogo.getCobra().getCabeca.getCoordY() =
                              jogo.getCobra().getCabeca.getCoordY()@pre  1 and
                              Jogo.getCobra().getCabeca.getCoordX() =
                              jogo.getCobra().getCabeca.getCoordX()@pre
                            else
                              if jogo.getDirecao() = 39 then
                                 jogo.getCobra().getCabeca.getCoordX() =
                                 jogo.getCobra().getCabeca.getCoordX()@pre + 1
                                 jogo.getCobra().getCabeca.getCoordY() =
                                 jogo.getCobra().getCabeca.getCoordY()@pre
                               else
                                 if jogo.getDirecao() = 40 then
                                    jogo.getCobra().getCabeca.getCoordY() =
                                    jogo.getCobra().getCabeca.getCoordY()@pre + 1
                                    jogo.getCobra().getCabeca.getCoordX() =
                                    jogo.getCobra().getCabeca.getCoordX()@pre
                                 endif
                               endif
                             endif
                           endif
```

Figura F.23: Especificação da operação jogar(JogoIF jogo) do componente Regras

```
Operação:         public boolean testeFimJogo( JogoIF jogo ): Boolean ;
Descrição:        testa se o jogo chegou ao fim por ter violado alguma regra
Entradas:         jogo: JogoIF  o jogo que está rodando
Saídas:           true -  caso o jogo termine
                  false - caso a o jogo não termine
Pré-Condições:    o jogo passado como parâmetro deve ser do tipo de um dos cinco jogos
                   existentes
Pós-Condições:    se o jogo for:
                  Hungry Snake: o jogo deve terminar ao atingir 15 pontos
                  Gula Gula ou Gula Gula2: o jogo deve terminar quando existir nenhuma ou 20
                                           comidas no tabuleiro
                  Magic Snake: o jogo deve terminar se passar de um determinado tempo
                               Caso o jogo termine deve se retornar true, caso não termine
                               deve se retornar false
OCL:              context Regras :: testeFimJogo ( jogo: JogoIF ): Boolean
                  pre: jogo.oclIsTypeOf(JogoHungrySnake) or jogo.oclIsTypeOf(JogoGula) or
                       jogo.oclIsTypeOf(JogoGula2)or jogo.oclIsTypeOf(JogoMagicSnake) or
                       jogo.oclIsTypeOf(JogoMySnake)
                  post: if jogo.oclIsTypeOf(JogoHungrySnake) then
                           if jogo.getScore() = 15 then
                              jogo.getStatus() = false
                              result = true
                           else
                              result = false
                           endif
                        else
                           if jogo.oclIsTypeOf(JogoGula) or jogo.oclIsTypeOf(JogoGula2) then
                              if (jogo.getComidas()->forAll(tc | tc->size = 0) and
                                jogo.getStatus() = true) or jogo.getComidas()->forAll(tc | tc->size =
                                20) then
                                 jogo.getStatus() = false
                                 result = true
                              else
                                 result = false
                              Endif
                           else
                             if jogo.oclIsTypeOf(JogoMagicSnake) then
                                if ( jogo.getTimeFim() - jogo.getTimeInicio() ) > 180000 then
                                   jogo.getStatus() = false
                                   result = true
                                else
                                   result = false
                                endif
                             else
                               if jogo.oclIsTypeOf(JogoMySnake) then
                               result = false
                             endif
                           endif
                        endif
```

Figura F.24: Especificação da operação testeFimJogo(JogoIF jogo) do componente Regras

```
Operação:         public void setDirecao( int direcao );
Descrição:        altera a direção que a cobra anda
Entradas:         direcao: int  a nova direção que que a cobra deve se mover
Saídas:
Pré-Condições:    o jogo deve estar rodando e a direção deve ser uma direção válida
Pós-Condições:    a direção passada como parâmetro deve ser a nova direção da cobra
OCL:              context Jogo :: setDirecao( direcao : Integer)
                  pre: getStatus() = true and
                       ( (direcao  = 37) or (direcao  = 38) or
                       (direcao  = 39) or (direcao  = 40) )
                  post: self.direcao = direcao
```

Figura F.25: Especificação da operação setDirecao(int direcao) do componente Jogo

```
Operação:         public void setScore( int score ) ;
Descrição:        altera a pontuação
Entradas:         score: int  nova pontuação
Saídas:
Pré-Condições:    o jogo deve estar rodando
Pós-Condições:    a pontuação passada como parâmetro deve ser a nova pontuação
OCL:              context Jogo :: setScore( score : Integer)
                  pre: self.getStatus() = true
                  Post: self.score = score
```

Figura F.26: Especificação da operação setScore(int score) do componente Jogo

```
Operação:         public void setStatus( boolean status) ;
Descrição:        altera o status do jogo
Entradas:         status: boolean  o novo status do jogo
Saídas:
Pré-Condições:
Pós-Condições:    o status passado como parâmetro deve ser o novo status da cobra
OCL:              context Jogo ::  setStatus( status : Boolean)
                  Post: self.status = status
```

Figura F.27: Especificação da operação setStatus(boolean status) do componente Jogo

```
Operação:         public void setVelocidade(  int velocidade ) ;
Descrição:        altera a velocidade da cobra
Entradas:         velocidade: int  a nova velocidade da cobra
Saídas:
Pré-Condições:    a velocidade deve ser maior que zero
Pós-Condições:    a velocidade passada como parâmetro deve ser a nova velocidade da cobra
OCL:              context Jogo :: setVelocidade( velocidade : Integer )
                  pre: velocidade > 0
                  Post: self.velocidade = velocidade
```

Figura F.28: Especificação da operação setVelocidade(int velocidade) do componente Jogo

```
Operação:         public void run() ;
Descrição:        faz o jogo rodar até que algo aconteça para que o jogo termine
Entradas:
Saídas:
Pré-Condições:
Pós-Condições:    caso a cobra bata na parede, nela mesma ou o jogo termine por outro motivo,
                  o status do jogo deve mudar para false. Caso o jogo não termine, a cobra deverá
                  se mover.
OCL:              context Jogo :: run() : void
                  post: if getRegras().testeCobraBateuParede( self )=true or
                        getRegras().testeCobraBateuNelaMesma( self) =true or
                        getRegras().testeFimJogo( self ) = true  then
                            self.getStatus() = false
                        else
                            (self.getStatus() = true and (
                            (self.getCobra().getCabeca().getCoordX() <>
                            self. GetCobra().getCabeca().getCoordX()@pre) or
                            (self.getCobra().getCabeca().getCoordY() <>
                            self. getCobra().getCabeca().getCoordY()@pre) ) )
                        endif
```

Figura F.29: Especificação da operação run() do componente Jogo

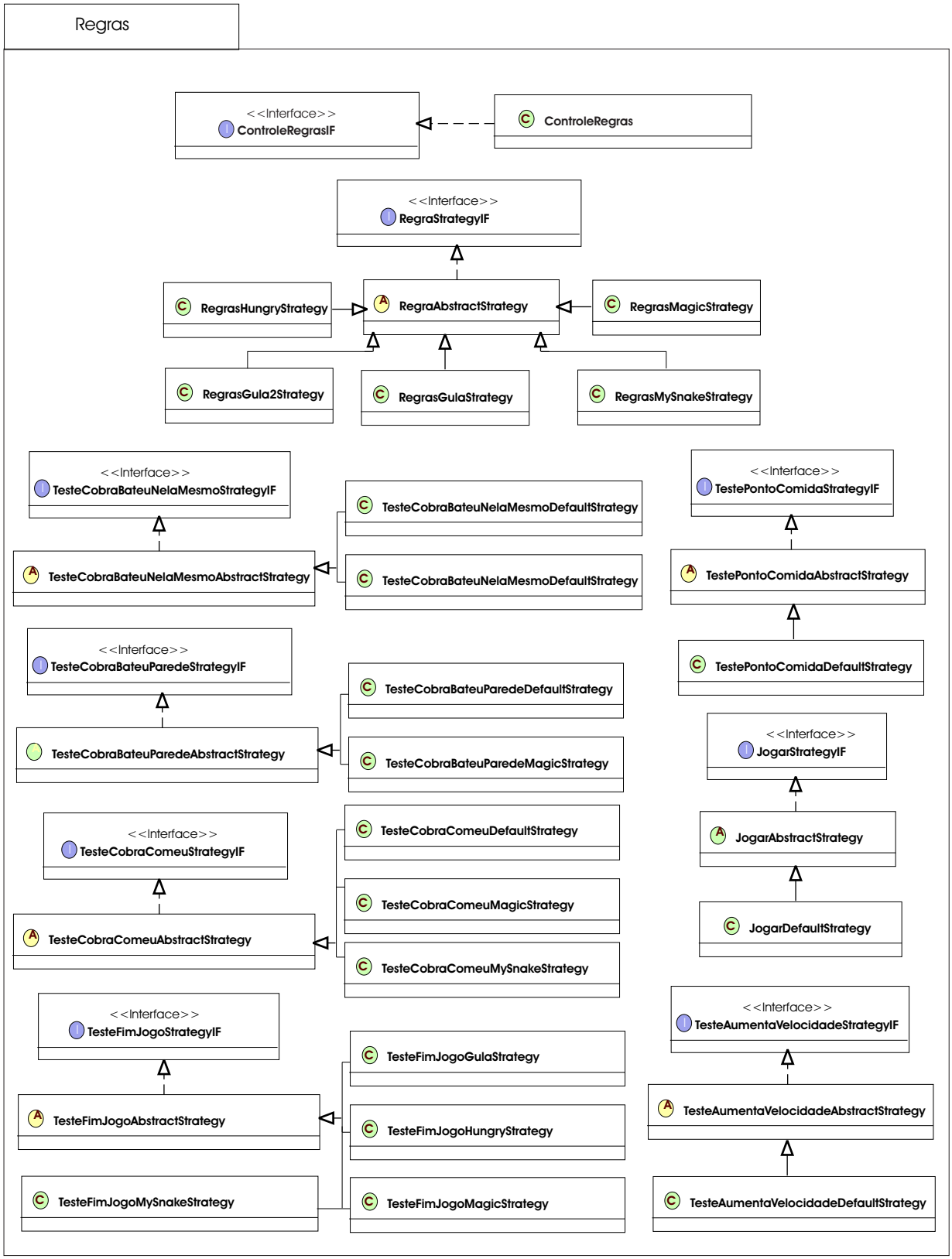

Figura F.30: Estrutura Interna do Componente Regras

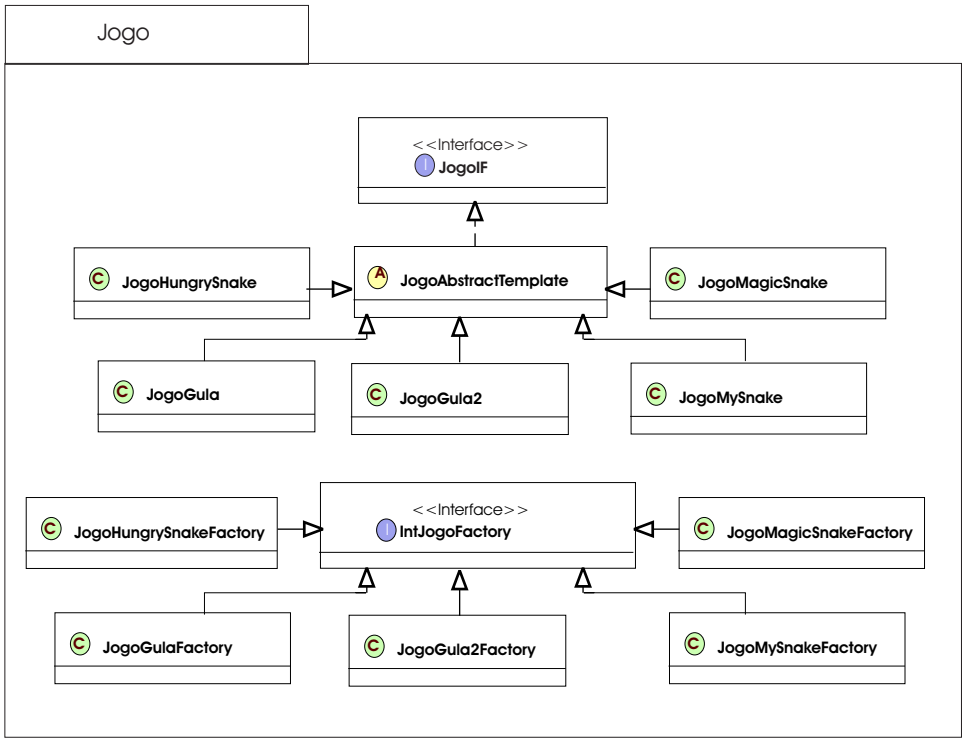

Figura F.31: Estrutura Interna do Componente Jogo

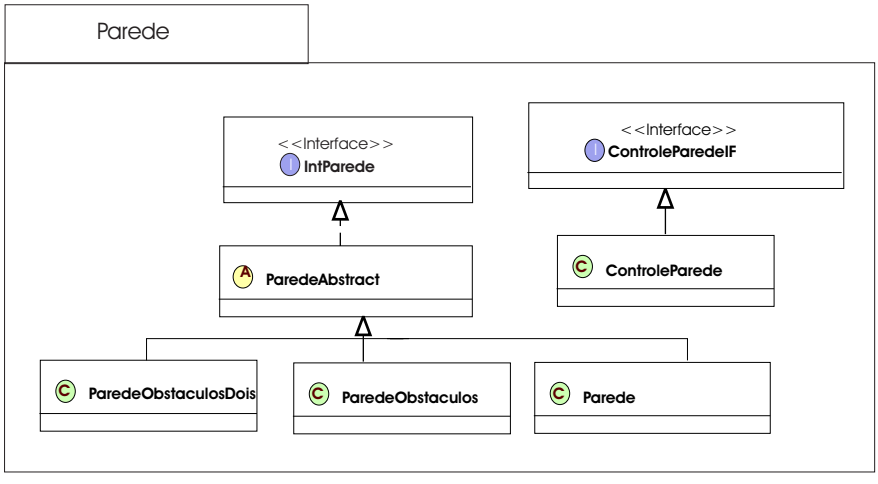

Figura F.32: Estrutura Interna do Componente Parede

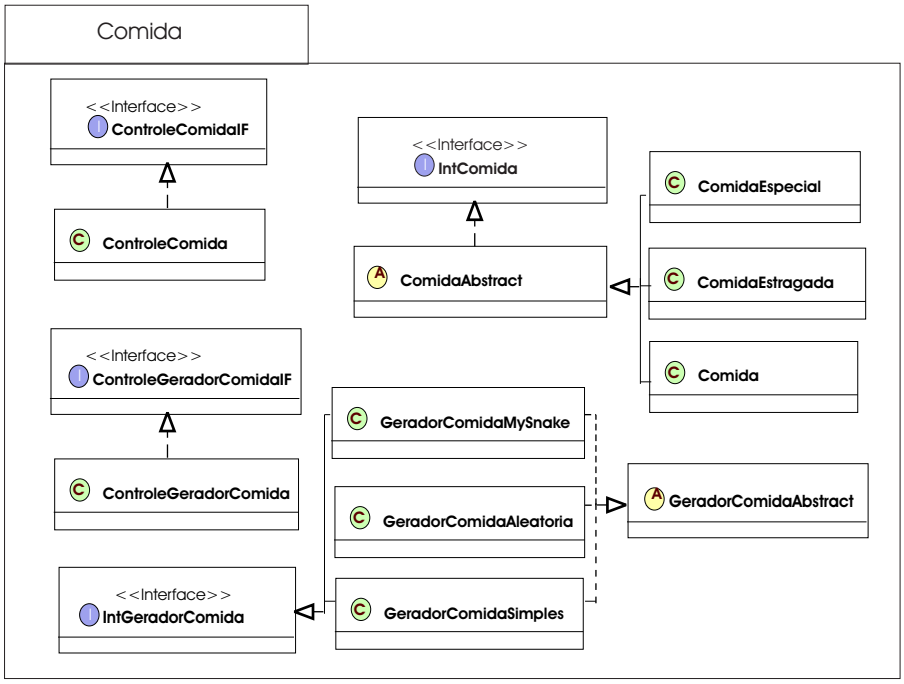

Figura F.33: Estrutura Interna do Componente Comida

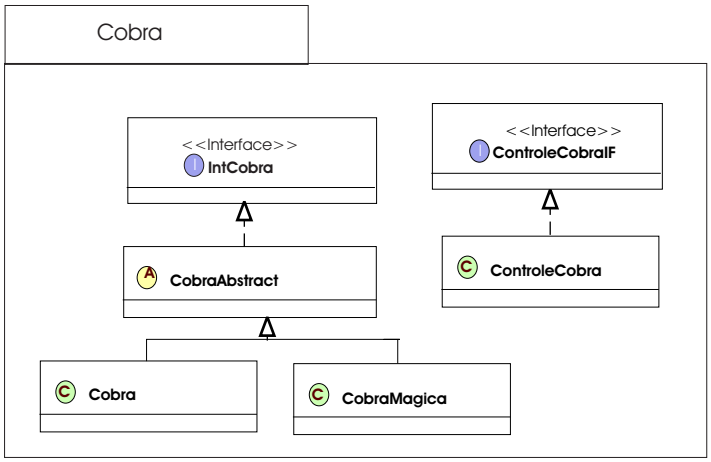

Figura F.34: Estrutura Interna do Componente Cobra

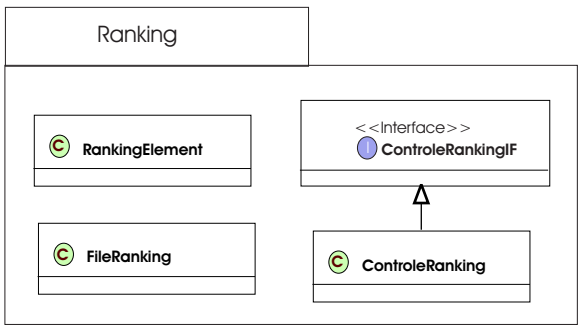

Figura F.35: Estrutura Interna do Componente Ranking

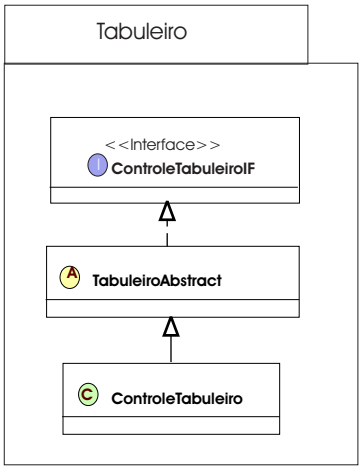

Figura F.36: Estrutura Interna do Componente Tabuleiro

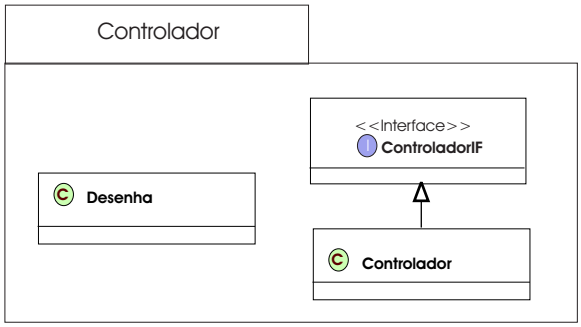

Figura F.37: Estrutura Interna do Componente Controlador

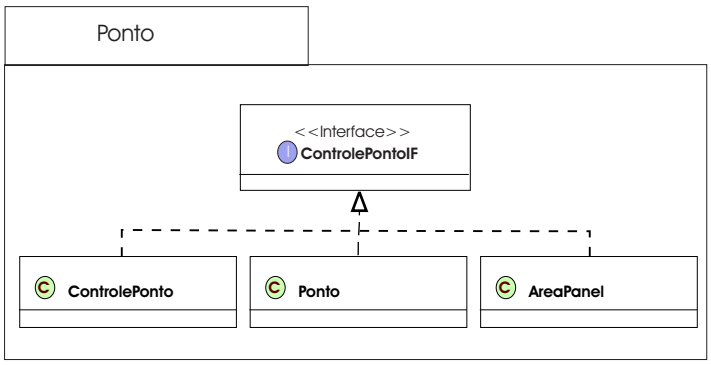

Figura F.38: Estrutura Interna do Componente Ponto