

Thânia Clair de Souza Vargas

Suporte à Edição de UML 2 no Ambiente SEA

Florianópolis - SC

2008

Thânia Clair de Souza Vargas

Suporte à Edição de UML 2 no Ambiente SEA

Orientador:
Ricardo Pereira e Silva

UNIVERSIDADE FEDERAL DE SANTA CATARINA
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA

Florianópolis - SC

2008

Trabalho de conclusão de curso submetido à Universidade Federal de Santa Catarina
como parte dos requisitos para obtenção do grau de Bacharel em Sistemas de Informação.

Prof^o. Dr. Ricardo Pereira e Silva
Orientador

Prof^a. Dra. Patrícia Vilain
Universidade Federal de Santa Catarina

Prof^o. Dr. Raul Sidnei Wazlawick
Universidade Federal de Santa Catarina

Dedico a Deus, minha família e ao meu namorado Caio.

Agradecimentos

Agradeço a Deus pela conclusão deste trabalho. Obrigado por dar-me forças, saúde e muita dedicação.

Aos meus pais, Angela e Noli. Obrigada por me amarem, obrigada por confiarem em mim, obrigada por me darem a oportunidade de existir e de tentar fazer a diferença.

A todos os meus irmãos por acreditarem em mim.

Ao Caio, que sempre esteve ao meu lado.

Ao meu tio Alberto, que me concedeu a oportunidade de estar aqui.

Ao Otávio, Ademir e Thiago, colegas que me deram algumas dicas neste trabalho.

A todos os meus parentes e amigos por torcerem por mim.

Ao professor Ricardo, pela dedicação, paciência e auxílio concedido a mim.

E finalmente, aos membros da banca examinadora os meus agradecimentos.

“Eu irei e cumprirei as ordens do Senhor, porque sei que o Senhor nunca dá ordens aos filhos dos homens sem antes preparar um caminho pelo qual suas ordens possam ser cumpridas”

O Livro de Mórmon, I Néfi 3:7

Resumo

O presente trabalho visa oferecer suporte à edição de UML 2 no ambiente de desenvolvimento de software SEA, tendo como resultado final uma nova versão escrita em linguagem Java do ambiente com a construção de novas ferramentas que reutilizam funcionalidades de edição supridas pelo framework OCEAN.

Frameworks orientados a objetos e modelagem UML são abordados de maneira a introduzir aspectos conceituais relacionados ao protótipo desenvolvido. Uma nova estrutura de especificação que usa técnicas atuais de modelagem UML 2 foi construída e incorporada ao ambiente SEA, agregando conceitos dos novos diagramas.

Palavras-chave: engenharia de software, reuso de software, frameworks orientados a objetos, ambientes de desenvolvimento, UML, ferramentas CASE.

Abstract

The current paper is intended to offer support to the UML 2 edition in the SEA software development environment, having as its final result a new version written in Java language of the environment with the building of new tools that reuse the edition features supplied for the OCEAN framework.

Object-oriented frameworks and UML modeling are approached in order to introduce the conceptual aspects related to the developed prototype. A new specification structure that uses the current techniques of UML 2 modeling was built and incorporated to the SEA environment, aggregating concepts of the new diagrams.

Keywords: software engineer, software reuse, object-oriented frameworks, development environments, UML, CASE tools.

Sumário

Lista de Figuras

Lista de Tabelas

1	Introdução	p. 18
1.1	Justificativa	p. 19
1.2	Objetivos	p. 19
1.2.1	Objetivo Geral	p. 20
1.2.2	Objetivos Específicos	p. 20
1.3	Metodologia	p. 20
1.4	Estrutura do documento	p. 21
2	Frameworks orientados a objetos	p. 22
2.1	Conceitos	p. 23
2.2	Características	p. 25
2.3	Classificação	p. 26
2.4	Utilização	p. 27
2.5	Vantagens	p. 27
2.6	Desvantagens	p. 28
2.7	Exemplos	p. 29
3	UML	p. 30
3.1	História	p. 30

3.2	Estrutura da especificação	p. 32
3.3	Organização dos Diagramas de UML 2	p. 33
3.4	Diagramas de UML 2	p. 34
3.4.1	Diagrama de Casos de Uso	p. 34
3.4.2	Diagrama de Classes	p. 35
3.4.3	Diagrama de Objetos	p. 36
3.4.4	Diagrama de Pacotes	p. 36
3.4.5	Diagrama de Estrutura Composta	p. 36
3.4.6	Diagrama de Componentes	p. 37
3.4.7	Diagrama de Utilização ou Implantação	p. 37
3.4.8	Diagrama de Seqüência	p. 38
3.4.9	Diagrama de Comunicação	p. 39
3.4.10	Diagrama de Máquina de Estados	p. 40
3.4.11	Diagrama de Atividades	p. 40
3.4.12	Diagrama de Visão Geral de Interação	p. 40
3.4.13	Diagrama de Temporização	p. 41
3.5	Comparação entre a primeira e a segunda versão de UML	p. 42
3.6	Ferramentas para modelagem UML	p. 43
3.6.1	Suporte à UML 1	p. 44
3.6.2	Suporte à UML 2	p. 47
3.6.3	Comparativo entre as ferramentas	p. 49
4	Framework OCEAN	p. 53
4.1	Características	p. 53
4.1.1	Caixa Cinza	p. 53
4.1.2	Flexibilidade	p. 54
4.1.3	Extensibilidade	p. 54

4.1.4	Suporte à edição gráfica	p. 55
4.1.5	Padrão <i>Model-View-Controller</i>	p. 56
4.2	Estrutura	p. 57
4.3	Problemas	p. 58
5	Ambiente SEA	p. 60
5.1	Características	p. 60
5.2	Classificação	p. 61
5.3	Estrutura	p. 62
5.4	Especificações	p. 63
5.5	Problemas	p. 64
6	Suporte à edição de UML 1 no Ambiente SEA	p. 65
6.1	Diagramas suportados	p. 65
6.2	Funcionalidades e ferramentas	p. 67
7	Suporte à edição de UML 2 no Ambiente SEA	p. 69
7.1	Novos documentos	p. 70
7.2	Nova especificação orientada a objetos	p. 72
7.3	Novos modelos e conceitos	p. 72
7.3.1	Diagrama de Objetos	p. 73
7.3.2	Diagrama de Pacotes	p. 77
7.3.3	Diagrama de Componentes	p. 84
7.3.4	Diagrama de Implantação	p. 87
7.3.5	Diagrama de Comunicação	p. 91
7.3.6	Diagrama de Casos de Uso	p. 94
7.3.7	Diagrama de Seqüência	p. 97
7.3.8	Diagrama de Estrutura Composta	p. 101

7.3.9	Diagrama de Máquina de Estados	p. 107
7.3.10	Diagrama de Atividades	p. 110
7.4	Suporte à Internacionalização e Localização	p. 121
7.4.1	Classes utilitárias	p. 122
7.4.2	Arquivos de tradução	p. 123
7.5	Modificações extras	p. 125
8	Conclusão	p. 126
8.1	Comparativo entre ferramentas CASE e o ambiente SEA	p. 127
8.2	Resultados Obtidos	p. 128
8.3	Avaliação	p. 129
8.4	Trabalhos Futuros	p. 130
8.5	Considerações Finais	p. 131
	Apêndice A – Cookbook para criação de documentos gráficos	p. 132
A.1	Passo 1: Criação da especificação de projeto	p. 132
A.2	Passo 2: Criação do modelo	p. 133
A.3	Passo 3: Criação dos conceitos	p. 135
A.4	Passo 4: Criação das classes gráficas	p. 137
A.4.1	Passo 4.1: Criação da classe Drawing	p. 137
A.4.2	Passo 4.2: Criação da classe DrawingView	p. 140
A.4.3	Passo 4.3: Criação da classe Window	p. 142
A.4.4	Passo 4.4: Criação da classe Editor	p. 144
A.5	Passo 5: Criação das figuras	p. 147
A.6	Passo 6: Edição da classe ReferenceManager	p. 150
	Anexo A – Artigo: A História de UML e seus diagramas	p. 152
A.1	Introdução	p. 152

A.2	História	p. 153
A.3	Estrutura da especificação	p. 154
A.4	Organização dos Diagramas de UML	p. 154
A.5	Diagramas de UML	p. 155
A.5.1	Diagrama de Casos de Uso	p. 156
A.5.2	Diagrama de Classes	p. 156
A.5.3	Diagrama de Objetos	p. 157
A.5.4	Diagrama de Pacotes	p. 157
A.5.5	Diagrama de Estrutura Composta	p. 157
A.5.6	Diagrama de Componentes	p. 157
A.5.7	Diagrama de Utilização ou Implantação	p. 158
A.5.8	Diagrama de Seqüência	p. 158
A.5.9	Diagrama de Comunicação	p. 158
A.5.10	Diagrama de Máquina de Estados	p. 158
A.5.11	Diagrama de Atividades	p. 159
A.5.12	Diagrama de Visão Geral de Interação	p. 159
A.5.13	Diagrama de Temporização	p. 159
A.6	Conclusão	p. 159
Anexo B – Código-fonte: Extensão do Ambiente SEA		p. 161
Referências Bibliográficas		p. 236

Lista de Figuras

2.1	Reuso de software.	p. 23
2.2	Estrutura de uma aplicação desenvolvida utilizando um framework.	p. 24
2.3	Princípio de Hollywood.	p. 26
3.1	Linha do Tempo de UML.	p. 31
3.2	Organização geral dos diagramas de UML 2.	p. 33
3.3	Diagramas estruturais.	p. 33
3.4	Diagramas de comportamento.	p. 34
3.5	Exemplo de diagrama de casos de uso.	p. 35
3.6	Exemplo de diagrama de classes.	p. 35
3.7	Exemplo de diagrama de objetos.	p. 36
3.8	Exemplo de diagrama de pacotes.	p. 36
3.9	Exemplo de diagrama de estrutura composta.	p. 37
3.10	Exemplo de diagrama de componentes.	p. 38
3.11	Exemplo de diagrama de utilização.	p. 38
3.12	Exemplo de diagrama de seqüência.	p. 39
3.13	Exemplo de diagrama de comunicação.	p. 39
3.14	Exemplo de diagrama de máquina de estados.	p. 40
3.15	Exemplo de diagrama de atividades.	p. 41
3.16	Exemplo de diagrama de visão geral de interação.	p. 41
3.17	Exemplo de diagrama de temporização.	p. 42
3.18	Gráfico de Ferramentas CASE.	p. 48
4.1	Superclasses do framework OCEAN que definem uma especificação.	p. 57

5.1	Estrutura do Ambiente SEA.	p. 62
6.1	Parte de um diagrama de classes modelado na versão atual do ambiente SEA.	p. 66
6.2	Estrutura de um diagrama de seqüência modelado na versão anterior a este trabalho do ambiente SEA.	p. 67
7.1	Camadas presentes do ambiente SEA.	p. 70
7.2	Diagrama de classes que exemplifica o processo de criação de um novo documento a ser adicionado no ambiente SEA.	p. 71
7.3	Diagrama de classes que expõe os modelos criados.	p. 73
7.4	Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de objetos.	p. 74
7.5	Exemplo de diagrama de objetos construído no ambiente SEA.	p. 77
7.6	Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de pacotes.	p. 78
7.7	Exemplo de diagrama de pacotes construído no ambiente SEA.	p. 83
7.8	Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de componentes.	p. 84
7.9	Exemplo de diagrama de componentes construído no ambiente SEA.	p. 87
7.10	Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de implantação.	p. 88
7.11	Exemplo de diagrama de implantação construído no ambiente SEA.	p. 91
7.12	Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de comunicação.	p. 92
7.13	Exemplo de diagrama de comunicação construído no ambiente SEA.	p. 94
7.14	Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de casos de uso.	p. 95
7.15	Exemplo de diagrama de casos de uso construído no ambiente SEA.	p. 97
7.16	Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de seqüência.	p. 98
7.17	Telas para a escolha do tipo do operador do fragmento combinado.	p. 101

7.18	Exemplo de diagrama de seqüência construído no ambiente SEA.	p. 102
7.19	Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de estrutura composta.	p. 102
7.20	Exemplo de diagrama de estrutura composta construído no ambiente SEA utilizando uso de colaboração.	p. 105
7.21	Exemplo de diagrama de estrutura composta construído no ambiente SEA utilizando colaboração.	p. 106
7.22	Exemplo de diagrama de estrutura composta construído no ambiente SEA utilizando classificador estruturado.	p. 106
7.23	Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de máquina de estados.	p. 107
7.24	Exemplo de diagrama de máquina de estados construído no ambiente SEA.	p. 111
7.25	Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de atividades.	p. 112
7.26	Barra de ferramentas customizada.	p. 116
7.27	Algumas das figuras que utilizaram a “ <i>PolygonFigure</i> ”.	p. 117
7.28	A figura “ <i>ConditionalNodeFigure</i> ”.	p. 118
7.29	As figuras “ <i>IfNodeFigure</i> ” e “ <i>IfElseNodeFigure</i> ”.	p. 119
7.30	Exemplo de diagrama de atividades construído no ambiente SEA utilizando partições.	p. 120
7.31	Exemplo de diagrama de atividades construído no ambiente SEA utilizando ferramentas específicas para modelagem de algoritmo.	p. 121
7.32	Arquivos de tradução criados no ambiente SEA.	p. 124
A.1	Organização geral dos diagramas de UML 2.	p. 155
A.2	Diagramas estruturais.	p. 155
A.3	Diagramas de comportamento.	p. 156

Lista de Tabelas

- 3.1 Comparação geral entre os diagramas da primeira e segunda versão de UML. p. 43
- 3.2 Quadro comparativo entre as ferramentas. p. 51
- 8.1 Quadro comparativo entre as ferramentas, incluindo o Ambiente SEA. p. 127

Lista de Listagens

A.1	Exemplo de implementação do método modeloLista()	p. 133
A.2	Exemplo de implementação do método initialize()	p. 134
A.3	Exemplo de implementação do método conceitoLista()	p. 134
A.4	Exemplo de implementação do método createEmptyDrawing()	p. 135
A.5	Sobrecarga de construtores em conceitos.	p. 136
A.6	Exemplo de implementação do método createDesiredFigureForConcept() .	p. 137
A.7	Exemplo de implementação do método createDesiredLineForConcept() .	p. 138
A.8	Exemplo de implementação do método startFigureFor()	p. 139
A.9	Exemplo de implementação do método stopFigureFor()	p. 140
A.10	Exemplo de implementação dos construtores da DrawingView	p. 140
A.11	Exemplo de sobrescrição do método updateConcepts()	p. 141
A.12	Exemplo de implementação do método getSelected()	p. 142
A.13	Exemplo de implementação do método show()	p. 142
A.14	Exemplo de implementação do método setDocument()	p. 142
A.15	Exemplo de implementação do método getDocument()	p. 143
A.16	Exemplo de implementação do construtor de uma classe Window	p. 143
A.17	Exemplo de implementação de construtores de um editor.	p. 144
A.18	Exemplo de implementação do método createDrawingViewObject()	p. 144
A.19	Exemplo de implementação do método createSpecificationDrawin()	p. 145
A.20	Exemplo de implementação do método createTools()	p. 146
A.21	Exemplo de sobrecarga de construtores em uma figura.	p. 147
A.22	Exemplo de implementação do método initialize()	p. 148

A.23	Exemplo de implementação do método relatedConceptClass() .	p. 148
A.24	Exemplo de implementação do método handles() .	p. 149
A.25	Exemplo de implementação do método redraw() .	p. 149
A.26	Exemplo de uso de decorações em linhas.	p. 150
A.27	Edição da classe ReferenceManager .	p. 150
B.1	Classe UML2SEASpecification.java	p. 161
B.2	Classe ActivityDiagram.java	p. 163
B.3	Classe CommunicationDiagram.java	p. 168
B.4	Classe ComponentDiagram.java	p. 170
B.5	Classe CompositeStructuredDiagram.java	p. 173
B.6	Classe DeploymentDiagram.java	p. 176
B.7	Classe ObjectDiagram.java	p. 179
B.8	Classe PackageDiagram.java	p. 181
B.9	Classe StateMachineDiagram.java	p. 184
B.10	Classe Component.java	p. 186
B.11	Classe Port.java	p. 189
B.12	Classe Interface.java	p. 190
B.13	Classe ComponentDiagramaDrawing.java	p. 192
B.14	Classe ComponentDiagramDrawingView.java	p. 197
B.15	Classe ComponentDiagramEditor.java	p. 198
B.16	Classe ComponentDiagramWindow.java	p. 209
B.17	Classe ComponentFigure.java	p. 211
B.18	Classe PortFigure.java	p. 222
B.19	Classe InterfaceFigure.java	p. 230

1 Introdução

O desenvolvimento de software está se tornando mais complexo ao longo do tempo, seja pelo aumento da importância da ¹TI dentro das organizações, seja pelo aumento da quantidade de domínios de problemas tratados. Acompanhar essa demanda de complexidade tem sido um grande desafio para a engenharia de software em termos da criação de metodologias e práticas que permitam ter maior controle sobre o desenvolvimento do software, seja na parte processual ou em sua parte estrutural.

Atualmente, há no mundo quase um trilhão de linhas de código escritas em pouco mais de quinhentas linguagens (FILHO, 2007). Adicionalmente, tem-se uma variedade de plataformas e utilitários de apoio que evoluem constantemente. Concomitante a isso, a pressão do mercado para reduzir o tempo de desenvolvimento dos produtos, exige dos desenvolvedores a habilidade de construir sistemas com qualidade e requer dos gerentes de projeto a capacidade de gestão que visa garantir tanto a entrega do produto quanto a satisfação do cliente. Nesse sentido, tem havido esforços para reutilizar artefatos desenvolvidos em projetos anteriores.

O conceito de reusabilidade no processo de desenvolvimento de software era pouco tratado antes da disseminação da orientação a objetos. Os softwares costumavam ser muito acoplados e pouco coesos, tornando a manutenção uma tarefa penosa. As funções estavam espalhadas por todo o código de maneira desordenada e, muitas vezes, repetida. O paradigma da orientação a objetos veio justamente solucionar os principais problemas dos outros paradigmas existentes.

As abordagens de frameworks orientados a objetos e desenvolvimento baseado em componentes promovem reuso em alto nível de granularidade (JOHNSON, 1997). Entretanto, a utilização dessas abordagens é prejudicada pela complexidade de desenvolvimento e uso (JOHNSON; FOOTE, 1988). Neste contexto de reuso de código e de projeto, apresenta-se um trabalho de conclusão de curso relacionado ao framework OCEAN e ao ambiente de desenvolvimento SEA, produzidos por Ricardo Pereira e Silva como parte de seu trabalho para obtenção do título de doutor. Propõe-se fornecer suporte à edição de UML 2 no ambiente SEA.

¹TI é a abreviação para “Tecnologia da Informação”. Pode ser definida como o conjunto de todas as atividades e soluções providas por recursos computacionais.

1.1 Justificativa

Como parte das atividades de desenvolvimento, um sistema precisa ser modelado como um conjunto de partes e de relações entre essas partes. Frequentemente a modelagem de software usa algum tipo de notação gráfica, apoiada pelo uso de ferramentas. A modelagem de software normalmente implica na construção de modelos gráficos que simbolizam os artefatos de software utilizados e seus interrelacionamentos. Uma forma comum de modelagem de programas procedurais (não orientados a objeto) é através de fluxogramas, enquanto que na modelagem de programas orientados a objeto, paradigma mais aceito atualmente, normalmente se utilizam a linguagem gráfica UML (*Unified Modeling Language* ou Linguagem de Modelagem Unificada). Na criação de sistemas com qualidade, é necessário a utilização de uma boa ferramenta para a modelagem de software. Neste âmbito, com o uso de modelagem de análise e projeto baseada em UML e padrão de notação orientada a objetos, pretende-se estender o ambiente SEA, que disponibiliza ferramentas de modelagem, a fim de fornecer suporte à versão atual de UML.

O framework OCEAN possui uma infraestrutura que suporta a construção de ambientes de desenvolvimento que manuseiam especificações de projeto. A fim de validar este framework, foi construído o ambiente SEA, que ainda não utiliza todas as técnicas de modelagem previstas em UML - atende a cinco destas técnicas, e a mais uma técnica adicional para a descrição do algoritmo dos métodos, que não é prevista em UML.

Sabendo que o conjunto de ferramentas de modelagem disponíveis no mercado que dão suporte à UML 2 ainda é escasso e há grande necessidade de utilização deste tipo de ferramenta em disciplinas acadêmicas de engenharia de software, o presente trabalho de conclusão de curso propõe uma nova versão em Java do ambiente SEA, mais robusta e consistente que a versão original a fim de dar apoio a segunda versão de UML. Para isso, foram inseridas novas técnicas e elementos sintáticos da modelagem UML. A nova versão possibilitará uma validação mais precisa e confiável do framework OCEAN, uma vez que será mais estável e abrangente.

1.2 Objetivos

Pode-se classificar os objetivos deste trabalho como gerais e específicos.

1.2.1 Objetivo Geral

O principal objetivo do trabalho é oferecer suporte à edição de UML 2 no ambiente SEA, tendo como resultado esperado uma nova versão em Java do ambiente com a construção de novas ferramentas que reutilizam funcionalidades de edição supridas pelo framework OCEAN. Neste sentido, foi criada uma especificação que use técnicas atuais de modelagem.

1.2.2 Objetivos Específicos

Dentre os principais objetivos específicos deste trabalho, destacam-se:

- Investigar e estudar sobre frameworks orientados a objetos;
- Estudar as características e o funcionamento do framework OCEAN;
- Estudar as características e o funcionamento do ambiente SEA;
- Estudar a versão atual de UML 2;
- Realizar um levantamento do estado da arte em termos de suporte à edição de UML 2;
- Fornecer suporte à UML 2 no ambiente SEA criando e melhorando modelos e conceitos.

1.3 Metodologia

A metodologia adotada para a extensão do ambiente SEA e para a elaboração teórica deste trabalho, ou seja, a sua estruturação, foi baseada em outras bibliografias que envolveram a construção de novas especificações no ambiente, como a obra de Ricardo Pereira e Silva (SILVA, 2000), Ademir Coelho (COELHO, 2007), Thiago Machado (MACHADO, 2007), João de Amorim (AMORIM, 2006) e Otavio Pereira (PEREIRA, 2004).

Devido à natureza do trabalho, a preocupação metodológica esteve mais voltada à construção de um instrumento de modelagem UML 2 sendo que o detalhamento teórico da forma em que este protótipo foi realizado teve importância secundária. Inicialmente é tratado o embasamento teórico, a fim de contextualizar o leitor, e posteriormente são passados dados detalhados sobre a construção do protótipo.

1.4 Estrutura do documento

Este trabalho de conclusão de curso está dividido em oito capítulos. No capítulo 2, são tratados os frameworks orientados a objetos, suas características, vantagens, desvantagens e exemplos. O capítulo 3 apresenta a linguagem UML e seus diagramas. No capítulo 4, dá-se uma visão sobre o framework OCEAN. O capítulo 5 introduzirá o ambiente SEA, suas características, sua estrutura, classificação e problemas. O capítulo 6 e 7 contém explicações sobre a forma como foi implementado o suporte à edição de UML 1 e 2, respectivamente. A conclusão do trabalho se encontra no capítulo 8, incluindo uma seção que detalha os resultados obtidos neste trabalho.

Ao fim do presente trabalho, encontra-se um apêndice que apresenta uma nova versão do *cookbook* para criação de documentos gráficos criado por João de Amorim (AMORIM, 2006), com novos comentários e dicas de implementação. Além disso, é possível observar nos anexos um artigo referente a este trabalho e o código-fonte implementado.

2 *Frameworks orientados a objetos*

A engenharia de software atua na aplicação de abordagens ao desenvolvimento e manutenção de software. Os principais objetivos da engenharia de software são a melhoria da qualidade do software e o aumento da produtividade da atividade de desenvolvimento de software. A reutilização de software (ver imagem 2.1), em contraposição ao desenvolvimento de todas as partes de um sistema, é um fator que pode levar ao aumento da qualidade e da produtividade da atividade de desenvolvimento de software. Este aspecto se baseia na perspectiva de que o reuso de ¹artefatos de software já desenvolvidos e depurados reduz o tempo de desenvolvimento, de testes e as possibilidades de introdução de erros na produção de novos artefatos.

A produção de artefatos de software reutilizáveis se coloca como um requisito para que ocorra reutilização de software. Os princípios que norteiam a modularidade, como a criação de módulos com interfaces pequenas, explícitas e em pequena quantidade, e o uso do princípio da ocultação de informação (MEYER, 1988), têm sido ao longo dos últimos anos o principal elemento de orientação para a produção de bibliotecas de artefatos reutilizáveis.

A reutilização de artefatos de software pode abranger tanto o nível de código como os níveis de análise e projeto. A reutilização de código consiste em reempregar diretamente trechos de código já desenvolvidos. Já a reutilização de análise e projeto consiste em reaproveitar concepções arquitetônicas de um artefato de software em outro artefato de software, não necessariamente com a utilização da mesma implementação (SILVA, 2000). O reuso de artefatos de software pode ser obtido através de objetos, rotinas, módulos, componentes, padrões ou frameworks.

Segundo a definição de WIRFS-BROCK R.; JOHNSON, um framework é:

“[...] um esqueleto de implementação de uma aplicação ou de um subsistema de aplicação, em um domínio de problema particular. É composto de classes abstratas e concretas e provê um modelo de interação ou colaboração entre as instâncias de classes definidas pelo framework. Um framework é utilizado através de configuração ou conexão de classes concretas e derivação de

¹Artefatos de software podem ser considerados como qualquer parte resultante de um processo de construção de software, tais como aplicações, frameworks ou componentes.

novas classes concretas a partir de classes abstratas do framework.” (WIRFS-BROCK R.; JOHNSON, 1990)

Já JOHNSON propõe a seguinte definição:

“Um framework é um projeto reusável de tudo ou de parte do sistema que é representado por uma coleção de classes abstratas e a maneira com que as instâncias interagem.” (JOHNSON, 1997)

Vale ressaltar que a diferença fundamental entre o reuso de um framework e a reutilização de rotinas ou classes de uma biblioteca, é que neste caso são usados artefatos de software isolados, cabendo ao desenvolvedor estabelecer sua interligação, e no caso do framework, é procedida a reutilização de um conjunto de classes interrelacionadas, cujo interrelacionamento é estabelecido no projeto do framework (SILVA, 2000 apud SCHEIDT, 2003).

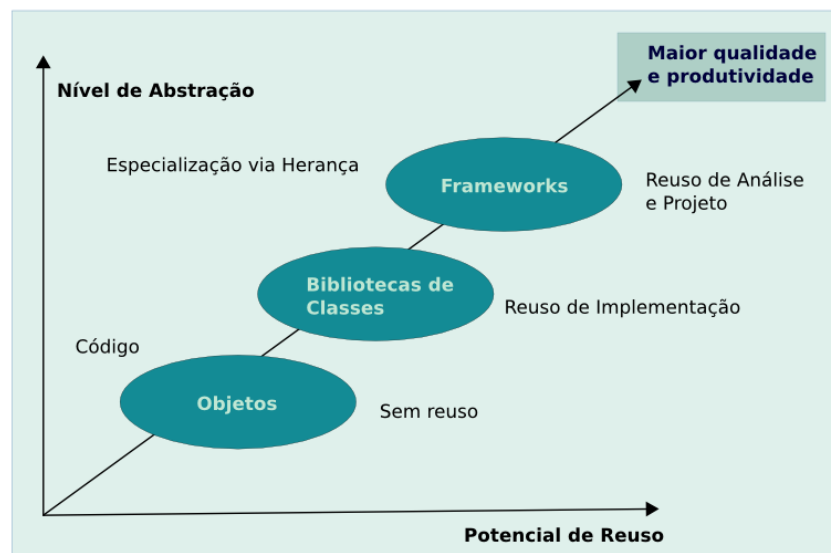


Figura 2.1: Reuso de software.

2.1 Conceitos

De modo geral, pode-se conceituar um framework como um artefato de software reutilizável que consiste em uma implementação incompleta para um conjunto de aplicações de um domínio. Sua estrutura deve ser adaptada para a geração de aplicativos específicos. Pode incluir programas de suporte, bibliotecas de código, linguagens de script e outros softwares para ajudar a desenvolver e juntar diferentes componentes de um projeto de software. É projetado com a intenção de facilitar o desenvolvimento de software, habilitando designers e programadores a

gastarem mais tempo determinando as exigências do software do que com detalhes tediosos de baixo nível do sistema.

O desenvolvimento de software através do paradigma orientado a objetos é impulsionado por características tais como: robustez, flexibilidade à mudança de requisitos e arquitetura, perspectiva de reutilização de código e projeto. Contudo, o modelo orientado a objetos por si só não garante todas essas características, é necessário o uso de técnicas e diretrizes que conduzam o desenvolvedor para atingir tais resultados, principalmente quando o tamanho e a complexidade das aplicações aumentam (WEBER; ROCHA, 1999) (ROCHA; MALDONADO; WEBER, 2001).

Os frameworks orientados a objetos (ver figura 2.2) representam uma categoria de software potencialmente capaz de promover reuso de alta ²granularidade (JOHNSON; FOOTE, 1988). Tratam-se de uma estrutura de classes interligadas, correspondendo a uma aplicação inacabada, para um conjunto de aplicações de um determinado domínio. A extensão dessa estrutura produzirá aplicações, proporcionando reuso.

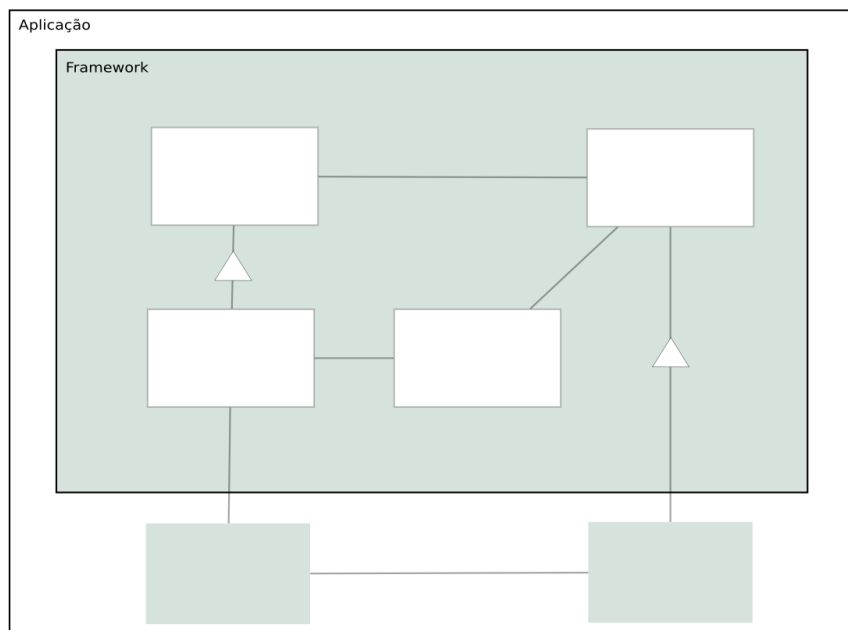


Figura 2.2: Estrutura de uma aplicação desenvolvida utilizando um framework.

No caso ideal, um framework deve conter todas as abstrações do domínio tratado, e uma aplicação gerada sob o framework não deveria conter abstrações que não estão presentes neste domínio, bastando ao usuário a tarefa de completar as informações específicas de sua aplicação. Geralmente é inviável o framework prover todas as abstrações, e por essa razão o framework

²Nível de detalhamento.

tende a absorver abstrações durante o seu ciclo de vida, caracterizando assim uma evolução iterativa (COELHO, 2007).

2.2 Características

Frameworks orientados a objetos são flexíveis. Esta característica reside em pontos específicos, os *hot spots*. Os *hot spots* são as partes da estrutura de classes de um framework que devem ser mantidas flexíveis, para possibilitar sua adaptação a diferentes aplicações do domínio. Contudo, frameworks também oferecem pontos fixos que definem a arquitetura geral de um sistema, os *frozen spots*. Nos *frozen spots*, os componentes básicos e seus relacionamentos permanecem fixos em todas as instanciações do framework de aplicação. Estes locais são os pontos de estabilidade do artefato de software.

Frameworks operam sob a inversão do fluxo de controle, conhecido como *princípio de Hollywood* (ver imagem 2.3). A responsabilidade de saber quais métodos serão chamados deve ser do framework e não da aplicação. Desta forma, as classes da aplicação esperam ser chamadas pelo framework durante o tempo de execução. Um conjunto de métodos pré-definidos (métodos *hooks*) é chamado pelo framework, e é neste ponto em que o comportamento específico da nova aplicação será inserido, através da herança e do polimorfismo. Os métodos *hooks*, que implementam os *hot spots*, formam, em conjunto com métodos *templates* (que implementam os *frozen spots*), um ponto de flexibilidade dos frameworks. Métodos *hooks* podem ser redefinidos pela aplicação e são utilizados pelos métodos *templates* que implementam de forma genérica um algoritmo.

Outra característica importante dos frameworks é a capacidade de generalização, que muitas vezes tornam o artefato difícil de compreender e utilizar. Isso acontece pois a fim de atender a um determinado domínio de problema ou permitir o desenvolvimento de um conjunto de aplicações diferentes, os frameworks necessitam abranger um grande conteúdo do mesmo.

Frameworks são complexos no que diz respeito ao seu desenvolvimento e uso. Em sua implementação, existe a necessidade de se produzir uma abstração de um domínio de aplicações adequada a diferentes aplicações. Em sua utilização, é necessário compreendê-lo. Isto implica em entender sua estrutura, o que geralmente requer tempo e esforço do usuário do framework.

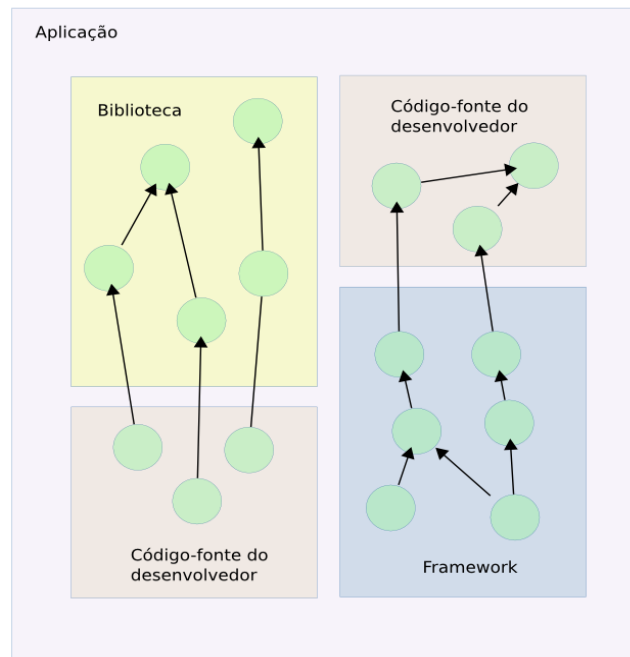


Figura 2.3: Princípio de Hollywood.

2.3 Classificação

Dentre as formas propostas na literatura para se categorizar frameworks, pode-se destacar a classificação quanto ao uso e finalidade (SILVA, 2000).

Quanto ao uso, os frameworks podem ser do tipo caixa-branca, caixa-preta ou caixa-cinza. Os frameworks do tipo caixa-branca é dirigido à arquitetura. Há a produção de subclasses das classes abstratas com funcionalidades necessárias à aplicação do usuário através de herança. Os frameworks do tipo caixa-preta são dirigidos a dados, favorecendo o reuso através de composição, com a instanciação das classes do framework para criação de novas classes. O usuário não é totalmente livre para criar ou alterar funcionalidades a partir da criação de classes, uma vez que o controle da aplicação é definido nas classes do framework e não nas classes do usuário. Os frameworks do tipo caixa-cinza são uma combinação das possibilidades anteriores. Apresentam inicialmente partes prontas, permitindo que sejam inseridas novas funcionalidades a partir da criação de classes. Este tipo de framework não somente fornece subclasses concretas, mas permite a criação de novas subclasses concretas.

Quanto à finalidade, os frameworks podem servir para a geração de aplicações completas, onde toda a estrutura de uma aplicação está no framework, ou para suportar a produção de unidades funcionais, cujo objetivo é desenvolver funcionalidades específicas, podendo prover apenas parte de uma aplicação (como interface, armazenamento de dados, entre outros).

2.4 Utilização

O principal objetivo de um framework é fornecer reuso na produção de diferentes aplicações, minimizando tempo e esforço requeridos. Procura ser uma descrição aproximada do domínio, construída a partir das informações até então disponíveis.

No início da utilização de um framework, ao desenvolver-se uma aplicação, é necessário que o desenvolvedor obtenha conhecimento em como usar o framework. O desenvolvedor precisa conhecer uma boa parte dos recursos que o framework disponibiliza a fim de evitar esforços desnecessários durante o desenvolvimento da aplicação.

A utilização ideal de frameworks consiste em completar ou alterar procedimentos e estruturas de dados neles presentes. Sob esta ótica, uma aplicação gerada sob um framework não deveria incluir classes que não fossem subclasses de classes do framework. Todavia, como um framework nunca é uma descrição completa de um domínio, é possível que a construção de aplicações por um framework leve à obtenção de novos conhecimentos do domínio tratado, indisponíveis durante a sua construção (SILVA, 2000).

2.5 Vantagens

Vale salientar que antes de se optar por utilizar um framework, devem ser entendidos seus pontos fortes e fracos, qual o alvo das aplicações a que ele se refere, seus componentes e estrutura, o seu processo de desenvolvimento, seus padrões de projeto fundamentais e suas técnicas de programação.

Um framework auxilia no desenvolvimento de uma aplicação de maneira oportuna e customizada às exigências do usuário. Esta aplicação se beneficia da robustez e da estabilidade conseguida através de um artefato maduro. Frameworks encapsulam implementações voláteis em interfaces estáveis provendo assim modularidade ao software produzido. Esta modularidade auxilia na melhoria da qualidade do produto, reduzindo o impacto causado por mudanças no projeto ou na implementação.

A utilização de frameworks em desenvolvimento de aplicações apresenta diversas vantagens, dentre elas podemos destacar:

1. *Reusabilidade*: promove reuso não somente de código, mas também de projeto. Gera melhoras substanciais na produtividade, assim como na qualidade, performance, robustez e na interoperabilidade das aplicações geradas.

2. *Generalidade*: principal característica que se busca quando se deseja construir um framework. Uma vez desprovido de generalidade, este não pode ser caracterizado como um framework, afinal servirá somente para a construção de uma única aplicação e fugindo da definição que prevê que um framework é utilizado para um domínio de aplicações.
3. *Extensibilidade*: permite que o domínio do framework seja aumentado ou refinado mais facilmente através da adição ou edição de métodos *hook* e *hot spots*.
4. *Modularidade*: facilita a manutenção e o entendimento do framework e das aplicações existentes. Além de ajudar a melhorar a qualidade de ambos, diminuindo o impacto de mudanças no projeto ou na implementação.
5. *Robustez*: mantém as funcionalidades mesmo com mudanças na estrutura interna ou externa. As aplicações geradas sobre o framework tornam-se mais confiáveis e robustas, uma vez que usam código e estrutura do framework, cuja estrutura e código já foram testados e depurados.

2.6 Desvantagens

Apesar das inúmeras vantagens, frameworks orientados a objetos apresentam alguns inconvenientes:

- *Complexidade no desenvolvimento*: o projetista de um framework necessita construir uma estrutura de classes capaz de generalizar os requisitos de um domínio e, ao mesmo tempo, permitir a adaptação às especificidades de cada aplicação.
- *Complexidade na utilização*: o usuário de um framework (desenvolvedor de aplicações) precisa entender a estrutura interna do framework antes de usá-lo no desenvolvimento de aplicações (BOSCH et al., 1999).

Assim, tanto o desenvolvedor do framework, quanto o desenvolvedor de aplicações, necessitam de informações a respeito de seus artefatos de software. O desenvolvedor do framework precisa de indicadores da conformidade de seu artefato com o domínio de aplicação pretendido, assim como, o desenvolvedor de aplicações precisa de indicadores da conformidade da sua aplicação desenvolvida sob um framework, em relação ao que foi projetado e pretendido no framework usado.

2.7 Exemplos

Frameworks podem apresentar inúmeras utilidades, abrangendo domínios de diversas áreas. Podemos enumerar alguns frameworks de grande aceitação na comunidade Java, apenas com fins ilustrativos:

- *Ant (build e deploy)*: framework amplamente divulgado pelo projeto Jakarta para automação de processos de construção, além de testes e distribuição;
- *Hibernate (persistência de dados)*: conhecido framework de persistência de dados que utiliza conceitos de banco de dados como mapeamento objeto-relacional (classes Java para tabelas de banco de dados);
- *Jasper Report (gerador de relatório)*: framework para geração de modo dinâmico de relatórios. Compatível com formatos XML, PDF e HTML;
- *Java Server Faces (Java EE)*: baseado em tecnologia de servlets e JSP (*Java Server Pages*), pode ser usado como uma opção ao Struts;
- *JUnit (testes)*: um dos frameworks mais usados em Java, incluído em IDEs (*Integrated Development Environment*) gratuitas e comerciais. É utilizado para testes unitários em geral;
- *Cactus (testes)*: framework específico para testes unitários de aplicações Java EE;
- *Log4J (log)*: amplamente usado e útil para geração de logs;
- *Jakarta commons-log (log)*: semelhante ao Log4J, sob o selo da Jakarta;
- *Prevaler (persistência de dados)*: outro famoso framework que prega uma JVM (*Java Virtual Machine*) invulnerável logicamente com uso de uma camada de prevalência de objetos;
- *Spring (POA)*: framework baseado em orientação a aspectos. Possibilidade de uso em conjuntos com outros frameworks MVC (padrão *Model-View-Controller*), como o Struts e JSF (*Java Server Faces*);
- *Struts (Java EE)*: um dos frameworks mais usados em ambientes corporativos para construção de aplicações web. Usa o modelo MVC e caracteriza-se por uma camada de controle com uso de Java EE e XML.

3 UML

Modelagem de software é a atividade de construir modelos que expliquem as características ou o comportamento de um software. Na construção de software, os modelos podem ser usados na identificação das características e funcionalidades que o software deverá fornecer (análise de requisitos), e no planejamento de sua construção. Frequentemente a modelagem de software usa algum tipo de notação gráfica, apoiada pelo uso de ferramentas.

A modelagem de software normalmente implica na construção de modelos gráficos que simbolizam os artefatos de software utilizados e os seus interrelacionamentos. Uma forma comum de modelagem de programas orientados a objeto é com a linguagem unificada UML.

A UML (*Unified Modeling Language*, ou “Linguagem de Modelagem Unificada”) é uma linguagem para especificação, documentação, visualização e desenvolvimento de sistemas orientados a objetos. Sintetiza os principais métodos existentes, sendo considerada uma das linguagens mais expressivas para modelagem de sistemas orientados a objetos. Por meio de seus diagramas é possível representar sistemas de softwares sob diversas perspectivas de visualização. Facilita a comunicação de todas as pessoas envolvidas no processo de desenvolvimento de um sistema - gerentes, coordenadores, analistas, desenvolvedores - por apresentar um vocabulário de fácil entendimento.

3.1 História

No início da utilização do paradigma de orientação a objetos, diversos métodos foram apresentados para a comunidade. Chegaram a mais de cinquenta entre os anos de 1989 a 1994, porém a maioria deles cometeu o erro de tentar estender os métodos estruturados da época. Com isso, os maiores prejudicados foram os usuários que não conseguiam encontrar uma maneira satisfatória de modelar seus sistemas. Foi a partir da década de 90 que começaram a surgir teorias que procuravam trabalhar de forma mais ativa com o paradigma da orientação a objetos. Diversos autores renomados contribuíram com publicações de seus respectivos métodos.

Por volta de 1993 existiam três métodos que mais cresciam no mercado, eram eles: *Booch'93* de *Grady Booch*, OMT-2 de *James Rumbaugh* e OOSE de *Ivar Jacobson*. Cada um deles possuía pontos fortes em algum aspecto. O OOSE possuía foco em casos de uso (*use cases*), OMT-2 se destacava na fase de análise de sistemas de informação e *Booch'93* era mais forte na fase de projeto. O sucesso desses métodos foi, principalmente, devido ao fato de não terem tentado estender os métodos já existentes. Seus métodos já convergiam de maneira independente, então seria mais produtivo continuar de forma conjunta (SAMPAIO, 2007).

Em outubro de 1994, começaram os esforços para unificação dos métodos. Já em outubro de 1995, *Booch* e *Rumbaugh* lançaram um rascunho do “Método Unificado” unindo o *Booch'93* e o OMT-2. Após isso, *Jacobson* se juntou a equipe do projeto e o “Método Unificado” passou a incorporar o OOSE. Em junho de 1996, *Booch*, *Rumbaugh* e *Jacobson* lançaram a primeira versão de uma nova linguagem de notação diagramática batizada de UML, que reuniu os três métodos (FOWLER, 2003). Posteriormente, foram lançadas novas versões as quais pode-se acompanhar através do gráfico na figura 3.1.

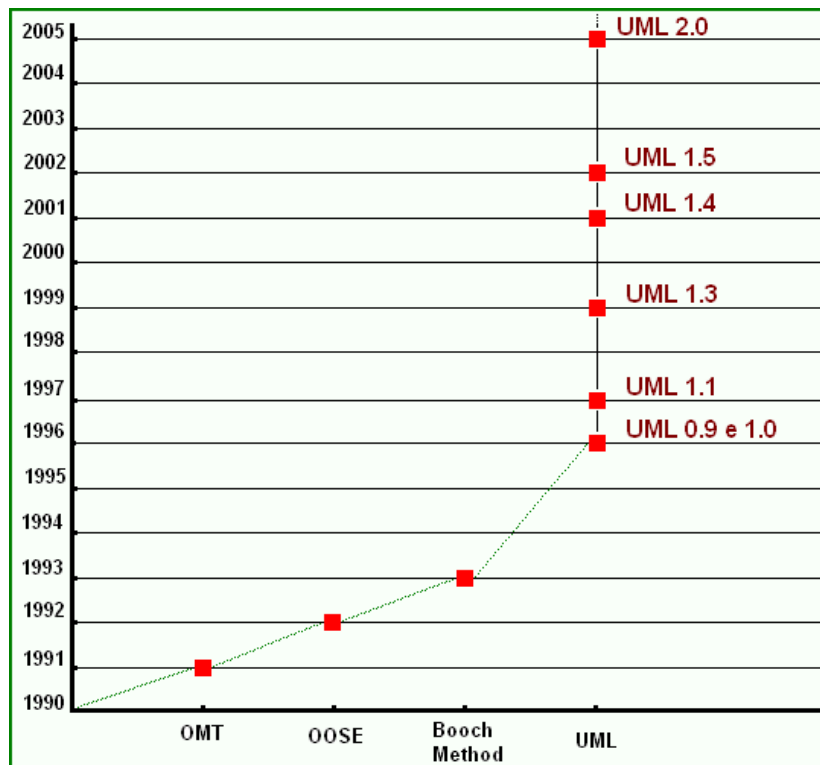


Figura 3.1: Linha do Tempo de UML.

A OMG (*Object Management Group*) lançou uma RFP (*Request for Proposals*) para que outras empresas pudessem contribuir com a evolução de UML, chegando à versão 1.1. Após alcançar esta versão, a OMG passou a adotá-la como padrão e a se responsabilizar (através da

RTF – *Revision Task Force*) pelas revisões. Atualmente, estas revisões são, de certa forma, controladas a não provocar uma grande mudança no escopo original. Ao observar as diferenças ocorridas com as versões atuais, é possível notar que de uma versão para a outra não há grande impacto. Esta é uma característica que facilita a disseminação mundial de UML.

É importante destacar que o ano de lançamento da UML 2.0 é questionável, uma vez que os documentos foram disponibilizados pela OMG gradualmente. Acredita-se que o marco do surgimento da UML 2.0 tenha sido no ano de 2005, pois foi somente nesta época em que a OMG disponibilizou oficialmente os documentos de especificação da superestrutura e infra-estrutura de UML, deixando evidente o conjunto de diagramas estabelecido.

3.2 Estrutura da especificação

A especificação de UML é composta por quatro documentos: infra-estrutura de UML (OMG, 2007a), superestrutura de UML (OMG, 2007b), *Object Constraint Language* (OCL) (OMG, 2006a) e intercâmbio de diagramas (OMG, 2006b).

- *Infra-estrutura de UML*: O conjunto de diagramas de UML é constituído por uma linguagem definida a partir de outra linguagem que define os elementos construtivos fundamentais. Esta linguagem que suporta a definição dos diagramas é apresentada no documento “Infra-estrutura de UML”.
- *Superestrutura de UML*: Documento que complementa o documento de infra-estrutura e que define os elementos da linguagem no nível do usuário.
- *Linguagem para Restrições de Objetos (OCL)*: Documento que apresenta a linguagem usada para descrever expressões em modelos UML, com pré-condições, pós-condições e invariantes.
- *Intercâmbio de diagramas de UML*: Apresenta uma extensão do meta-modelo voltado a informações gráficas. A extensão permite a geração de uma descrição no estilo XMI (*XML Metadata Interchange*) orientada a aspectos gráficos que, em conjunto com o XMI original, permite produzir representações portáteis de especificações UML.

3.3 Organização dos Diagramas de UML 2

A linguagem UML 2 é composta por treze diagramas, classificados em diagramas estruturais e diagramas de comportamento. A figura 3.2 apresenta a estrutura das categorias utilizando a notação de diagramas de classes (OMG, 2007a).

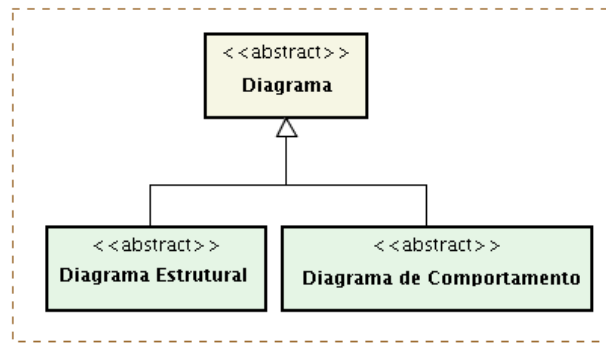


Figura 3.2: Organização geral dos diagramas de UML 2.

Os diagramas estruturais, ilustrados na imagem 3.3 conforme a especificação da OMG (OMG, 2007a), tratam o aspecto estrutural tanto do ponto de vista do sistema quanto das classes. Existem para visualizar, especificar, construir e documentar os aspectos estáticos de um sistema, ou seja, a representação de seu esqueleto e estruturas “relativamente estáveis”. Os aspectos estáticos de um sistema de software abrangem a existência e a colocação de itens como classes, interfaces, colaborações e componentes.

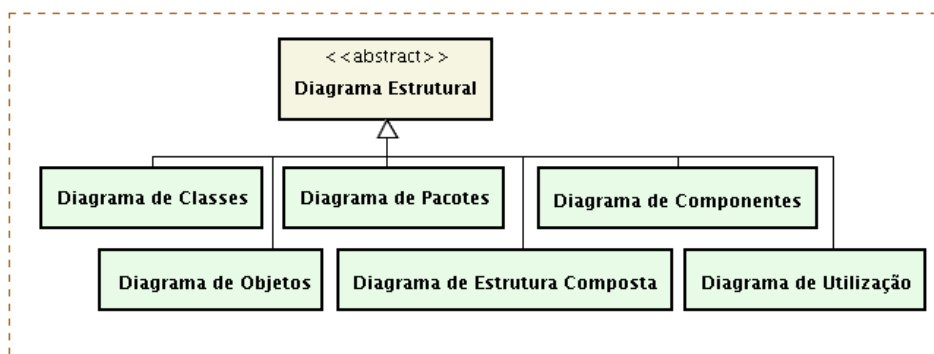


Figura 3.3: Diagramas estruturais.

Os diagramas de comportamento, ilustrados na imagem 3.4 conforme a especificação da OMG (OMG, 2007a), são voltados a descrever o sistema computacional modelado quando em execução, isto é, como a modelagem dinâmica do sistema. São usados para visualizar, especificar, construir e documentar os aspectos dinâmicos de um sistema que é a representação das

partes que “sofrem alterações”, como por exemplo, o fluxo de mensagens ao longo do tempo e a movimentação física de componentes em uma rede.

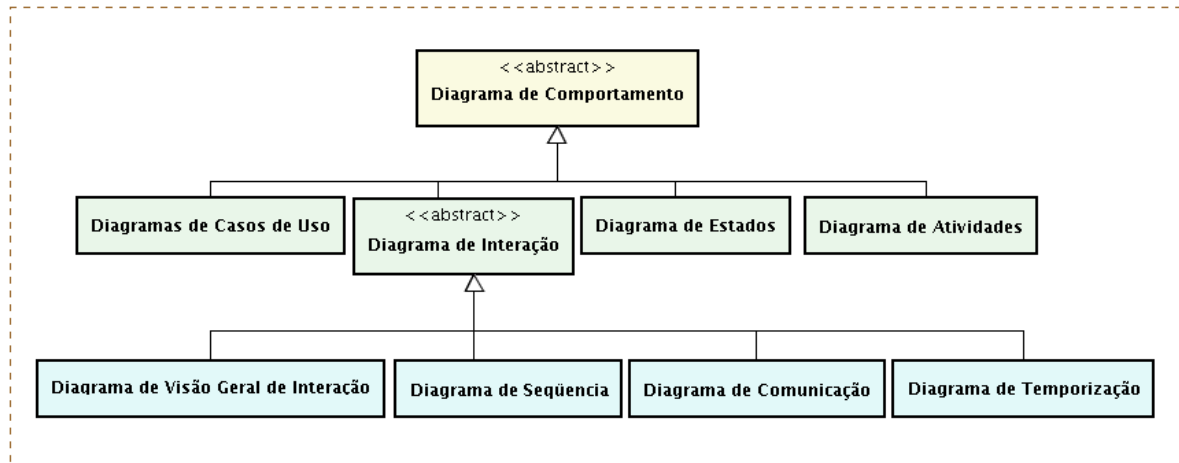


Figura 3.4: Diagramas de comportamento.

3.4 Diagramas de UML 2

Um diagrama é uma representação gráfica de um conjunto de elementos (classes, interfaces, colaborações, componentes, nós, etc) utilizados para visualizar o sistema sob diferentes perspectivas. A UML define um número de diagramas que permite dirigir o foco para aspectos diferentes do sistema de maneira independente. Se bem empregados, os diagramas facilitam a compreensão do sistema que está sendo desenvolvido.

Nas próximas seções, serão apresentados os treze diagramas que compõem a linguagem UML 2. Cada diagrama trará um exemplo de utilização, com objetivos meramente ilustrativos.

3.4.1 Diagrama de Casos de Uso

O diagrama de casos de uso (ver imagem 3.5) especifica um conjunto de funcionalidades, através do elemento sintático “casos de uso”, e os elementos externos que interagem com o sistema, através do elemento sintático “ator” (SILVA, 2007).

Além de casos de uso e atores, este diagrama contém relacionamentos de dependência (inclusão e extensão), generalização e associação, sendo basicamente usados para fazer a modelagem de visão estática do caso de uso do sistema. Essa visão proporciona suporte principalmente para o comportamento de um sistema, ou seja, os serviços externamente visíveis que o sistema

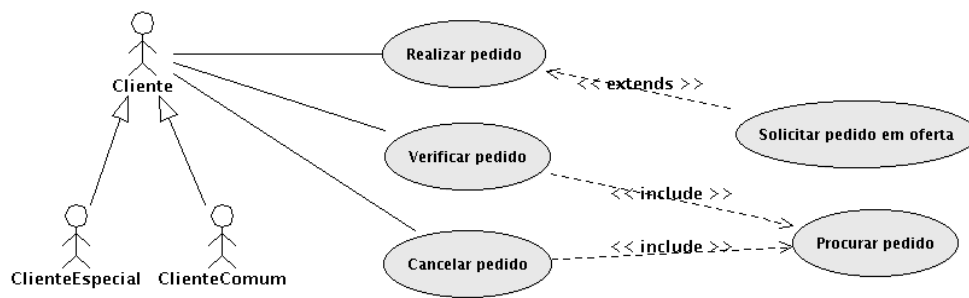


Figura 3.5: Exemplo de diagrama de casos de uso.

fornece no contexto de seu ambiente. Neste caso, os diagramas de caso de uso são usados para fazer a modelagem do contexto de um sistema e fazer a modelagem dos requisitos de um sistema.

3.4.2 Diagrama de Classes

Um diagrama de classes (ver imagem 3.6) é um modelo fundamental de uma especificação orientada a objetos. Produz a descrição mais próxima da estrutura do código de um programa, ou seja, mostra o conjunto de classes com seus atributos e métodos e os relacionamentos entre classes. Classes e relacionamentos constituem os elementos sintáticos básicos do diagrama de classes (SILVA, 2007).

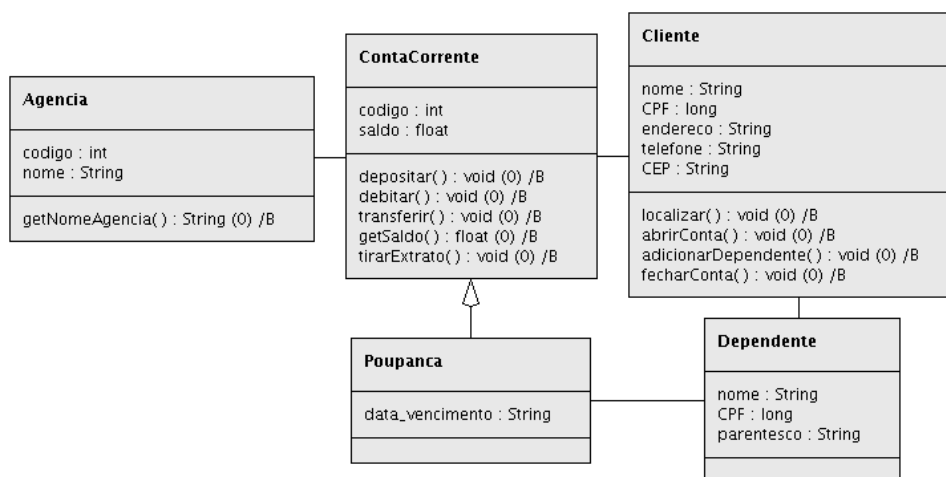


Figura 3.6: Exemplo de diagrama de classes.

O elemento sintático “classe” é representado por um retângulo dividido em três partes. A primeira divisão é utilizada para o nome da classe; na segunda divisão, coloca-se as informações de atributos; e a última divisão é utilizada para identificar os métodos.

3.4.3 Diagrama de Objetos

O diagrama de objetos (ver imagem 3.7) consiste em uma variação do diagrama de classes em que, em vez de classes, são representadas instâncias e ligações entre instâncias. A finalidade é descrever um conjunto de objetos e seus relacionamentos em um ponto no tempo.

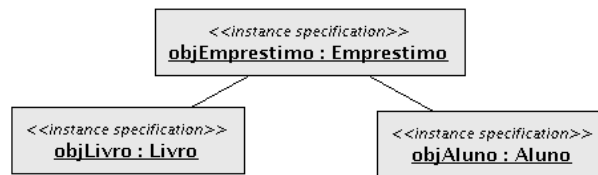


Figura 3.7: Exemplo de diagrama de objetos.

As instâncias e as ligações entre estas instâncias são usadas para fazer a modelagem da visão de projeto estática de um sistema a partir da perspectiva de instâncias reais ou prototípicas.

3.4.4 Diagrama de Pacotes

O pacote é um elemento sintático voltado a conter elementos sintáticos de uma especificação orientada a objetos. Esse elemento foi definido na primeira versão de UML para ser usado nos diagramas então existentes, como diagrama de classes, por exemplo. Na segunda versão da linguagem, foi introduzido um novo diagrama, o diagrama de pacotes (ver imagem 3.8), voltado a conter exclusivamente pacotes e relacionamentos entre pacotes (SILVA, 2007). Sua finalidade é tratar a modelagem estrutural do sistema dividindo o modelo em divisões lógicas e descrevendo as interações entre eles em alto nível.

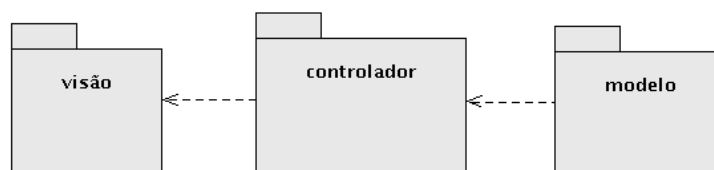


Figura 3.8: Exemplo de diagrama de pacotes.

3.4.5 Diagrama de Estrutura Composta

O diagrama de estrutura composta (ver imagem 3.9) fornece meios de definir a estrutura de um elemento e de focalizá-la no detalhe, na construção e em relacionamentos internos. É

um dos novos diagramas propostos na segunda versão de UML, voltado a detalhar elementos de modelagem estrutural, como classes, pacotes e componentes, descrevendo sua estrutura interna.

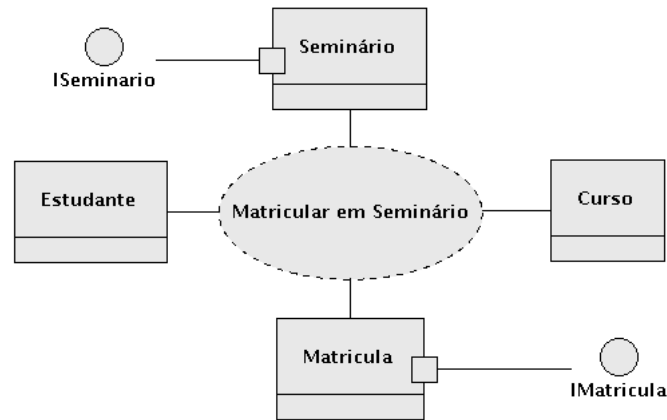


Figura 3.9: Exemplo de diagrama de estrutura composta.

O diagrama de estrutura composta introduz a noção de “porto”, um ponto de conexão do elemento modelado, a quem podem ser associadas interfaces. Também utiliza a noção de “colaboração”, que consiste em um conjunto de elementos interligados através de seus portos para a execução de uma funcionalidade específica – recurso útil para a modelagem de padrões de projeto (SILVA, 2007).

3.4.6 Diagrama de Componentes

O diagrama de componentes (ver imagem 3.10) é um dos dois diagramas de UML voltados a modelar software baseado em componentes. Tem por finalidade indicar os componentes do software e seus relacionamentos. Este diagrama mostra os artefatos de que os componentes são feitos, como arquivos de código fonte, bibliotecas de programação ou tabelas de bancos de dados. As interfaces é que possibilitam as associações entre os componentes.

3.4.7 Diagrama de Utilização ou Implantação

O diagrama de utilização (ver imagem 3.11), também denominado diagrama de implantação, consiste na organização do conjunto de elementos de um sistema para a sua execução. O principal elemento deste diagrama é o nodo, que representa um recurso computacional. Podem ser representados em um diagrama tanto os nodos como instâncias de nodos.

O diagrama de implantação é útil em projetos onde há muita interdependência entre artefatos de hardware e software.

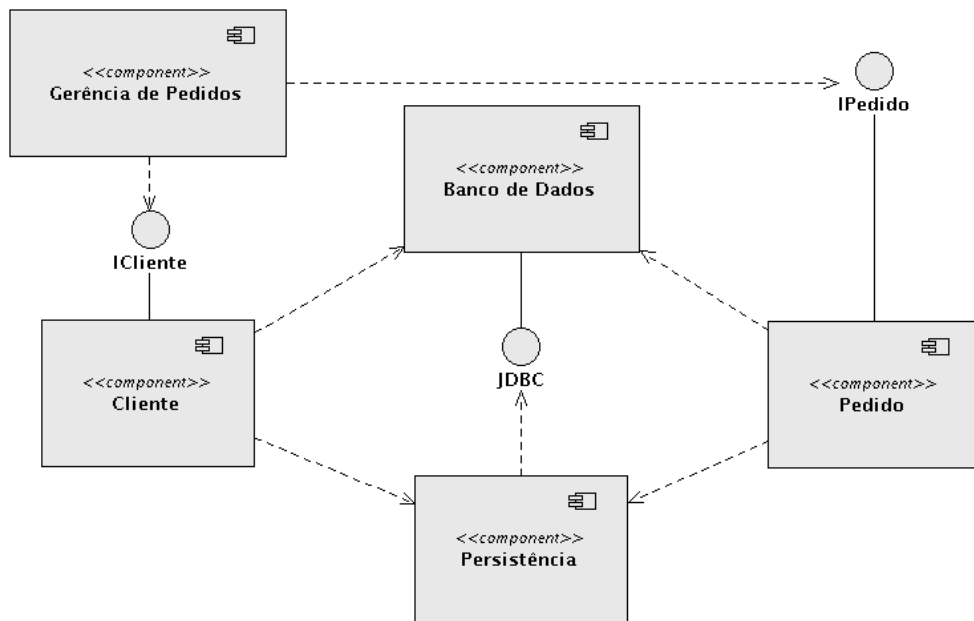


Figura 3.10: Exemplo de diagrama de componentes.

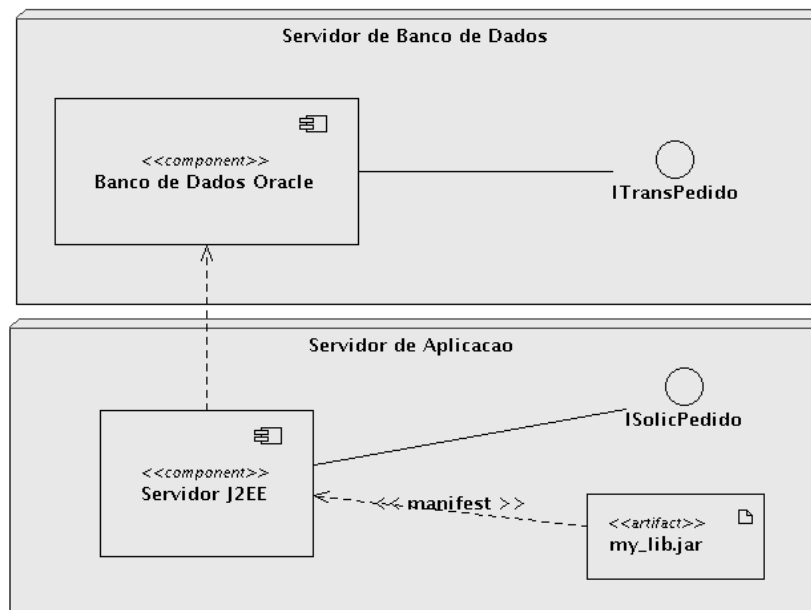


Figura 3.11: Exemplo de diagrama de utilização.

3.4.8 Diagrama de Seqüência

O diagrama de seqüência mostra a troca de mensagens entre diversos objetos, em uma situação específica e delimitada no tempo. Coloca ênfase especial na ordem e nos momentos nos quais mensagens para os objetos são enviadas.

Em diagramas de seqüência (ver imagem 3.12), objetos são representados através de linhas

verticais tracejadas (denominadas como linha de existência), com o nome do objeto no topo. O eixo do tempo é também vertical, aumentando para baixo, de modo que as mensagens são enviadas de um objeto para outro na forma de setas com a operação e os nomes dos parâmetros.

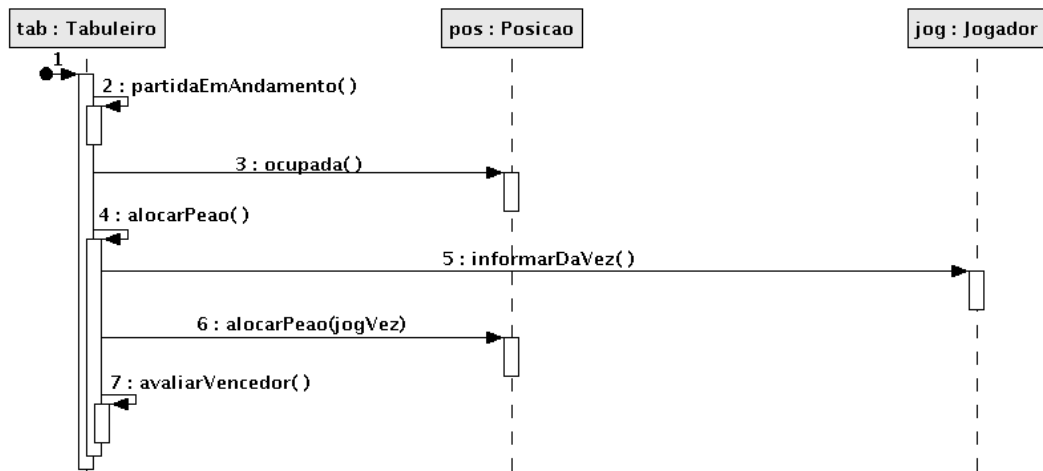


Figura 3.12: Exemplo de diagrama de seqüência.

3.4.9 Diagrama de Comunicação

Os elementos de um sistema trabalham em conjunto para cumprir os objetivos do sistema e uma linguagem de modelagem precisa poder representar esta característica. O diagrama de comunicação (ver imagem 3.13) é voltado a descrever objetos interagindo e seus principais elementos sintáticos são “objeto” e “mensagem”. Corresponde a um formato alternativo para descrever interação entre objetos. Ao contrário do diagrama de seqüência, o tempo não é modelado explicitamente, uma vez que a ordem das mensagens é definida através de enumeração.

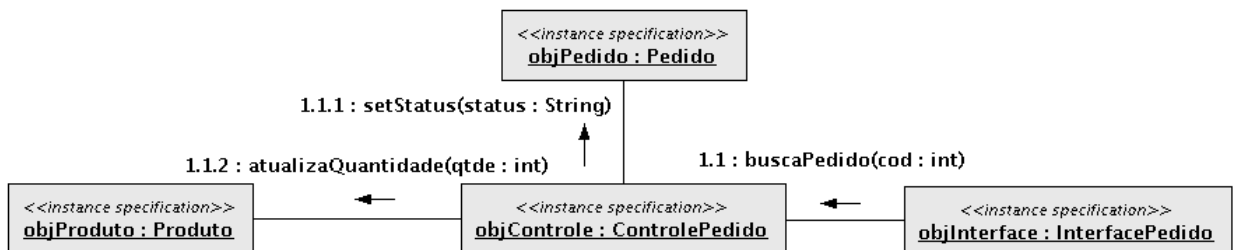


Figura 3.13: Exemplo de diagrama de comunicação.

Vale ressaltar que tanto o diagrama de comunicação como o diagrama de seqüência são diagramas de interação.

3.4.10 Diagrama de Máquina de Estados

O diagrama de máquina de estados (ver imagem 3.14) tem como elementos principais o estado, que modela uma situação em que o elemento modelado pode estar ao longo de sua existência, e a transição, que leva o elemento modelado de um estado para o outro.

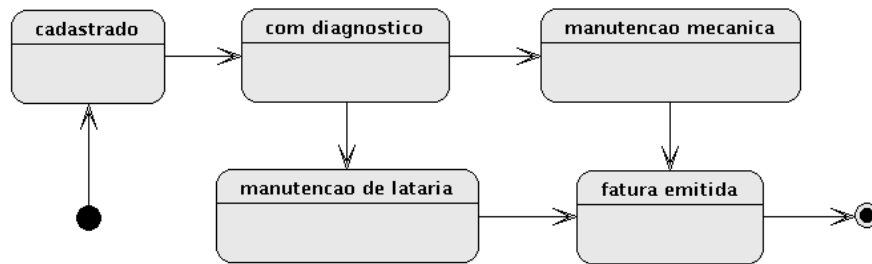


Figura 3.14: Exemplo de diagrama de máquina de estados.

O diagrama de máquina de estados vê os objetos como máquinas de estados ou autômatos finitos que poderão estar em um estado pertencente a uma lista de estados finita e que poderão mudar o seu estado através de um estímulo pertencente a um conjunto finito de estímulos.

3.4.11 Diagrama de Atividades

O diagrama de atividades (ver imagem 3.15) representa a execução das ações e as transições que são acionadas pela conclusão de outras ações ou atividades.

Uma atividade pode ser descrita como um conjunto de ações e um conjunto de atividades. A diferença básica entre os dois conceitos que descrevem comportamento é que a ação é atômica, não admitindo particionamento, o que não se aplica a atividade, que pode ser detalhada em atividades e ações (SILVA, 2007).

3.4.12 Diagrama de Visão Geral de Interação

O diagrama de visão geral de interação (ver imagem 3.16) é uma variação do diagrama de atividades, proposto na segunda versão de UML.

Os elementos sintáticos deste diagrama são os mesmos do diagrama de atividades, exceto os nodos que são substituídos por interações. As interações que fazem parte do diagrama de visão geral de interação podem ser referências a diagramas de interação existentes na especificação tratada.

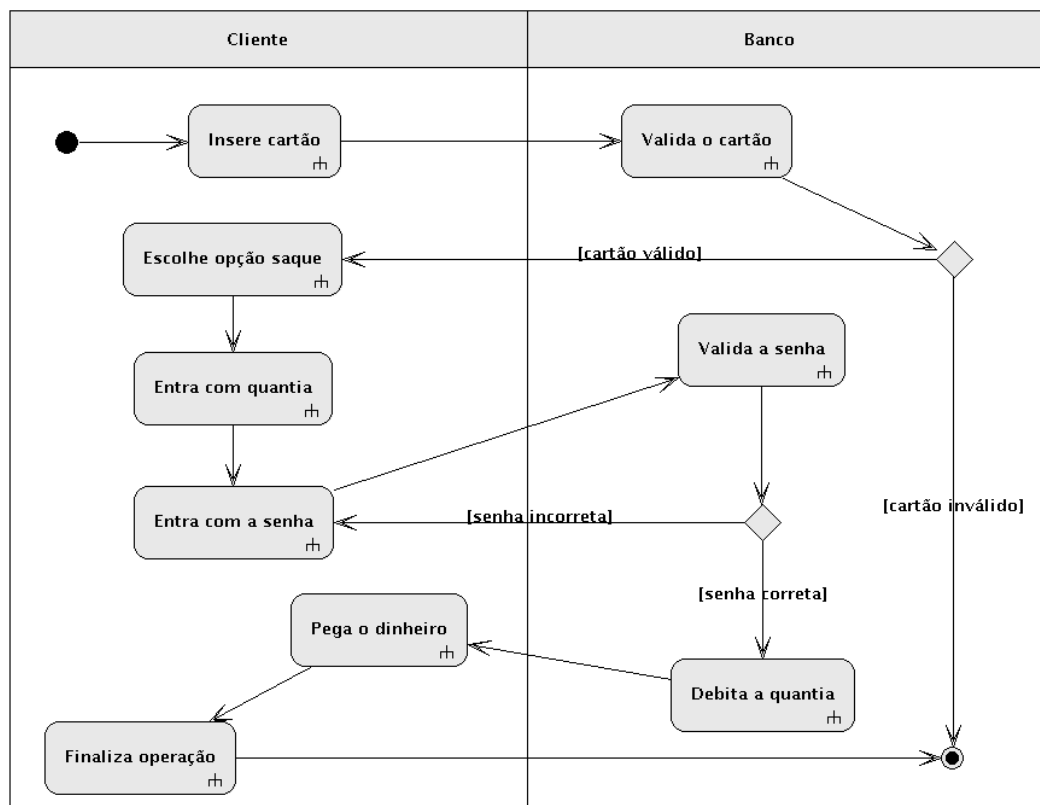


Figura 3.15: Exemplo de diagrama de atividades.

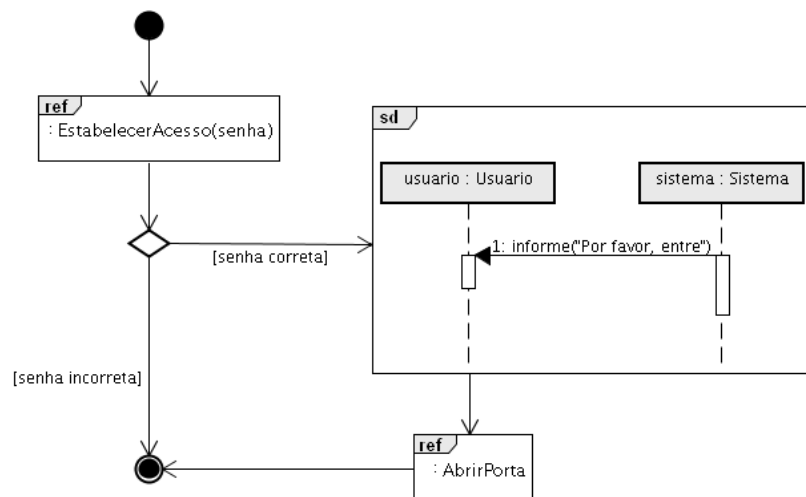


Figura 3.16: Exemplo de diagrama de visão geral de interação.

3.4.13 Diagrama de Temporização

O diagrama de temporização (ver imagem 3.17) consiste na modelagem de restrições temporais do sistema. É um diagrama introduzido na segunda versão de UML, classificado como diagrama de interação. Este diagrama modela interação e evolução de estados.

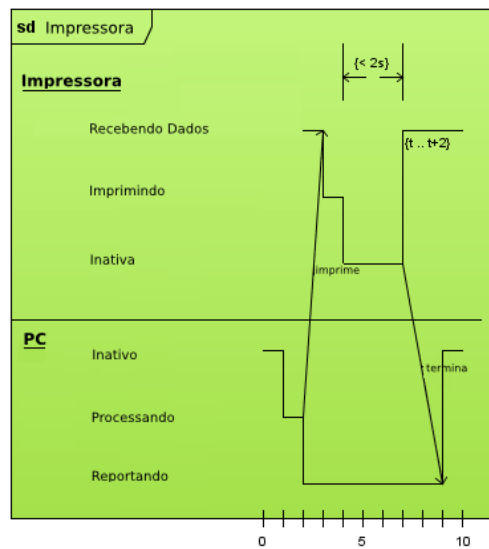


Figura 3.17: Exemplo de diagrama de temporização.

3.5 Comparação entre a primeira e a segunda versão de UML

Algumas mudanças importantes podem ser observadas na nova versão de UML. A quantidade de diagramas aumentou de nove para treze, e dois diagramas da versão 1 tiveram seus nomes alterados na versão 2. O quadro 3.1 apresenta uma comparação geral entre os conjuntos de diagramas da primeira e da segunda versão de UML.

Entre as alterações mais significativas, observa-se que o diagrama de componentes, através de recursos sintáticos introduzidos com o diagrama de estrutura composta, passa a poder descrever a estrutura interna de um componente. A impossibilidade de fazê-lo na primeira versão de UML tornava esse diagrama semanticamente pobre e com uma fraca ligação com os outros elementos de uma especificação orientada a objetos (SILVA, 2007).

Na nova versão do diagrama de seqüência, foi inserido o conceito de agrupamento de mensagens, com o elemento sintático “fragmento combinado”, possibilitando uma melhora na representação do envio de mensagens com condição e repetição. Além disso, também foi introduzido o uso de interação, que permite referenciar um comportamento descrito em outros diagramas de seqüência.

No diagrama de máquina de estados, foi introduzido o estado submáquina, com formato similar ao de um estado. O estado submáquina consiste em uma referência a uma máquina de estados modelada em outro diagrama.

O diagrama de atividades deixou de ser uma especialização do diagrama de *statechart* (SILVA, 2007), o que favoreceu a modificação de alguns elementos no diagrama, como troca

Tabela 3.1: Comparação geral entre os diagramas da primeira e segunda versão de UML.

Modelagem	Diagrama	UML 1	UML 2
Estrutural	Diagrama de Classes	V	V
	Diagrama de Objetos	V	V
	Diagrama de Pacotes	X	V
	Diagrama de Estrutura Composta	X	V
	Diagrama de Componentes	V	V
	Diagrama de Utilização	V	V
Dinâmica	Diagrama de Casos de Uso	V	V
	Diagrama de Seqüência	V	V
	Diagrama de Comunicação	X	V
	Diagrama de Colaboração *	V	X
	Diagrama de Máquina de Estados	X	V
	Diagrama de Statechart **	V	X
	Diagrama de Atividades	V	V
	Diagrama de Visão Geral de Interação	X	V
	Diagrama de Temporização	X	V

LEGENDA:

- V Existência do diagrama
- X Ausência do diagrama
- O Diagrama novo em UML 2
- * Denominado **diagrama de comunicação** em UML 2.
- ** Denominado **diagrama de máquina de estados** em UML 2.

de transições pelos fluxos de controle (idênticos em questão de sintaxe). Também foi inserido o conceito de agrupamento na modelagem de uma atividade.

Como pode-se observar nas melhorias realizadas na nova versão de UML, nota-se a preocupação com o suporte a referências de um diagrama a outro diagrama. Esse recurso é de grande utilidade, uma vez que contribui para que seja estabelecido um vínculo entre diagramas de uma especificação.

3.6 Ferramentas para modelagem UML

As ferramentas de modelagem surgiram a fim de facilitar a criação dos modelos e as mais avançadas permitem a geração de uma parte do código-fonte do software a partir dos modelos criados. Inúmeras ferramentas estão disponíveis no mercado, algumas *open source* e gratuitas, outras proprietárias.

Realizou-se um levantamento de algumas ferramentas de modelagem existentes, classificando-as inicialmente de acordo com a versão de suporte à UML e sua propriedade. Na pesquisa realizada, destaca-se a constatação da existência de cerca de apenas seis ferramentas não proprietárias com suporte a UML 2. O gráfico 3.18 resume a pesquisa quantitativamente.

As próximas seções mostram as principais características das ferramentas encontradas.

3.6.1 Suporte à UML 1

Nesta seção serão apresentadas ferramentas proprietárias e não proprietárias que fornecem suporte apenas à UML 1.

Proprietárias

- *AgileJ StructureViews*: Voltada a aplicações Java, constrói engenharia reversa customizada para diagrama de classes.
- *Cadifra UML Editor*: Editor de diagramas UML para a plataforma Windows.
- *eRequirements*: Ferramenta de gerência de requisitos para web.
- *GatherSpace*: Gerência de requisitos e casos de uso online.
- *Gliffy*: Solução para modelagem em UML baseado na web.
- *IBM Rational Requisite Pro*: Produto integrado de fácil utilização para gerenciamento de requisitos e de referência de utilização que promove melhor comunicação, aprimora o trabalho em equipe e reduz o risco do projeto. Inclui ferramentas de gerenciamento de requisitos, de modelagem dos negócios e de modelagem de dados.
- *MacA&D*: Gerência de requisitos e UML para Mac OS.
- *MasterCraft*: Conjunto de ferramentas da *Tata Consultancy Service Limited* que suporta análise orientada a objetos e projeto usando UML para desenvolvimento baseado em MDA (*Model-driven architecture*).
- *Metamil*: Ferramenta UML para C++, C# e Java. Roda sobre as plataformas Windows e Linux.
- *Microsoft Visio Pro*: Ferramenta de modelagem UML da empresa *Microsoft*.
- *MyEclipse*: Uma IDE baseada no Eclipse. Sua edição profissional inclui soluções UML.
- *OmniGraffle*: Ferramenta UML para Mac OS X.
- *OptimalJ*: Ambiente de desenvolvimento dirigido ao modelo para Java.
- *Rational Rose*: Pertencente a empresa *Rational Software*. Foi vendida para IBM em 2003.

- *SDMetrics*: Ferramenta que checa regras de projeto e medidas de qualidade de modelos UML.
- *Select Architect*: Plataforma UML/PDA para Microsoft Windows, rodando em um repositório escalável que integra com Eclipse e VS.NET.
- *SmartDraw*: Ferramenta para modelagem de inúmeros diagramas, inclusive os de UML. É para a plataforma Windows.
- *Use Case Studio*: Ferramenta para modelagem de casos de uso da empresa *Rewritten Software*. Gratuita para uso educacional.
- *Visustin*: Ferramenta para engenharia reversa de diagrama de atividades e fluxuogramas.
- *WinA&D*: Ferramenta para gerência de requisitos e UML. É para a plataforma Windows.
- *Yalips*: Ferramenta de modelagem UML que suporta *brainstorming* e gerência de projeto *gant*.

Não Proprietárias

- *ArgoUml*: Ferramenta de modelagem UML baseada em Java. Possui licença ¹BSD.
- *Astade*: Ferramenta de modelagem UML de plataforma independente baseado em wxWidgets.
- *ATL*: Uma ferramenta ²QVT que pode transformar modelos UML dentro de outros modelos. Disponível através do projeto Eclipse GMT (*Generative Modeling Tools*).
- *Dia*: Uma ferramenta para criação de diagramas da ³GTK+/GNOME que suporta UML. É licenciada pela ⁴GNU GPL.
- *Fujaba*: Acrônimo para “*From UML to Java and back*”. Permite comportamento de modelagem usando diagramas.

¹BSD é uma licença de código aberto utilizada inicialmente em sistemas *Berkley Software Distribution*. Atualmente vários outros sistemas são distribuídos sob esta licença.

²QVT significa “*Queries/Views/Transformations*”. É um padrão para transformação de modelos definida pela OMG (*Object Management Group*)

³GTK+ é um toolkit multi-plataforma para a criação de interfaces gráficas. GNOME é um ambiente gráfico desktop livre e gratuito para sistemas UNIX.

⁴O acrônimo GNU significa “*GNU is Not Unix*”. É um projeto que tem como objetivo criar um sistema operacional totalmente livre. GNU GPL significa GNU General Public Licence. Esta é a designação da licença para software livre no âmbito do projeto GNU.

- *JUDE ou Java and UML Developer Environment*: Uma das ferramentas ⁵*free* para UML mais poderosas disponíveis atualmente. Apresenta características que não são encontradas nas outras ferramentas *free*, como adicionar métodos no diagrama de sequência e a alteração se refletir no diagrama de classes. Sua performance impressiona, principalmente tratando-se de uma ferramenta 100% Java e ⁶*Swing*, dismistificando que *Swing* é lento.
- *Metric View Evolution*: Uma ferramenta para métricas baseada em análise de qualidade e melhor compreensão de modelos UML.
- *MonoUML*: Baseado na última versão do Mono, GTK+ e ExpertCoder.
- *NetBeans*: Parte da IDE NetBeans 6.0 Enterprise apresenta suporte à UML.
- *StarUML*: Uma plataforma UML para Microsoft Windows, licenciada sobre um versão modificada de GNU GPL, sendo a maior parte escrita em Delphi.
- *Taylor*: Arquitetura voltada ao modelo *on Rails*. É licenciada sobre ⁷GNU LGPL.
- *Topcased*: Editores de modelo *open source*, com ferramentas de verificação formal, transformação e linguagens de modelagem.
- *Umbrello UML Modeller*: Ferramenta parte do KDE. Permite criar diagramas de UML e outros sistemas em um formato padrão. É *open source*.
- *PalmOS*: Ferramenta UML para PalmOS.
- *UMLGraph*: Ferramenta *open source* que permite especificação declarativa e modelagem de diagramas de classes e de seqüência de UML.
- *UMLet*: Ferramenta UML baseada em Java e licenciada sobre a GNU GPL.
- *Use Case Maker*: Ferramenta de gerência de *use cases*. É licenciada sobre a GNU GPL.
- *Violet UML Editor*: Editor de UML baseado em Java. É integrado ao Eclipse e é licenciado sobre GNU GPL.

⁵Free é o mesmo que “livre” em português. Um software livre é qualquer programa de computador que pode ser usado copiado, estudado, modificado e redistribuído sem nenhuma restrição.

⁶Swing é uma API Java para interfaces gráficas.

⁷GNU LGPL significa *GNU Lesser General Public Licence*. É a licença de software livre publicada pela *Free Software Foundation*.

3.6.2 Suporte à UML 2

Nesta seção serão apresentadas algumas ferramentas proprietárias e não proprietárias que fornecem suporte a todos os diagramas de UML 2.

Proprietárias

- *Altova UModel*: Editor de UML que suporta a versão 2.1, capaz de exportar para ⁸XMI.
- *Apollo for Eclipse*: Suporta UML 2.0 e Java 5. Integra com a IDE Eclipse.
- *ARTiSAN Studio*: Suporta UML 2.0 e ⁹SysML.
- *Blueprint Software Modeler*: Um ambiente de modelagem de software integrado com suporte a modelagem UML 2.1 e meta-modelagem. É baseado no Eclipse.
- *Borlang Together*: Ferramenta de modelagem UML, integrada com o Eclipse e com Microsoft VS.NET 2005. Suporta UML 2.0, ¹⁰MDA, ¹¹OCL e ¹²MOF.
- *ConceptDraw 7*: Ferramenta de modelagem para Windows e Mac, suporta UML 2.0.
- *MagicDraw UML*: Ferramenta para UML 2.0 que suporta engenharia reversa e suporta muitos produtos para MDA. Integra com muitas IDEs, incluindo Eclipse e NetBeans. Suporta SysML.
- *Objecteering*: Provê completa cobertura de desenvolvimento dirigido ao modelo. Há uma versão gratuita disponível.
- *Poseidon for UML*: Ferramenta de modelagem de sistemas da empresa alemã *Gentleware AG*. É uma evolução da ferramenta *open source* ArgoUML que com mais de 350.000 instalações está entre as ferramentas de modelagem mais conhecidas. Seu principal foco está na facilidade de uso que a torna simples de aprender a usar.
- *Rational Software Architect*: Ferramenta para UML 2 baseada no Eclipse pela *Rational Division* da IBM.

⁸XMI significa *XML Metadata Interchange*. É um padrão da OMG para troca de informações baseado em XML.

⁹SysML significa *Systems Modeling Language*. É uma linguagem de modelagem de domínio específico para engenharia de sistemas. Suporta especificação, análise, projeto, verificação e validação de sistemas.

¹⁰MDA significa *Model-driven architecture*. É uma abordagem de projeto de software lançado pela OMG em 2001.

¹¹OCL significa *Object Constraint Language*. É uma linguagem declarativa que descreve regras que se aplicam a modelos UML desenvolvido na IBM e agora é parte do padrão UML.

¹²MOF significa *Managed Object Format*. É a linguagem usada para descrever classes CIM (*Common Information Model*).

- *Sparx Enterprise Architect*: Ferramenta da empresa Sybase. Suporta UML 2.0, modelagem de dados e modelagem de processos de negócio.
- *Telelogic Rhapsody*: Suporta UML 2.1 e SysML.
- *TextUML Toolkit*: Ferramenta para a criação de modelos de UML 2.1, usando uma notação textual.
- *Visual Paradigm for UML*: Suporta UML 2.1, modelagem de dados e modelagem de negócios.

Não Proprietárias

- *BOUML*: Ferramenta UML 2 multi-plataforma. Apresenta alta performance. É escrita em C++ e é licenciada sobre GNU GPL.
- *Eclipse*: Acoplado ao EMF (*Eclipse Modeling Framework*), fornece suporte à UML 2.
- *Gaphor*: Ambiente de modelagem UML 2 GTK+/GNOME, escrito em Python.
- *Omondo*: Plugin para Eclipse 3.2. Implementa UML 2.1 e usa JDK 5.
- *Papyrus*: Ferramenta UML 2 *open source* baseada no Eclipse e licenciada sobre a ¹³EPL.
- *Xholon*: Ferramenta *open source* que transforma, simula e executa modelos de UML 2.

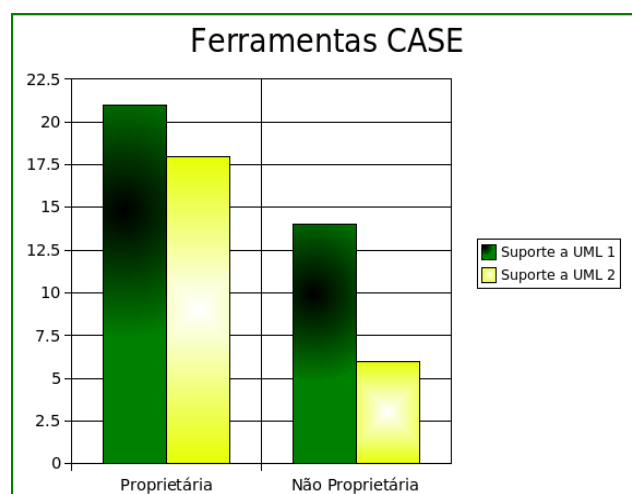


Figura 3.18: Gráfico de Ferramentas CASE.

¹³EPL significa *Eclipse Public Licence*. É uma licença de código aberto usada pelos softwares da *Eclipse Foundation*.

3.6.3 Comparativo entre as ferramentas

Com o objetivo de conhecer de modo mais aprofundado o ¹⁴estado da arte das ferramentas CASE atuais e de entender alguns conceitos teóricos tratados neste trabalho, criou-se um quadro comparativo (ver tabela 8.1) que resume algumas das principais funcionalidades esperadas de uma boa ferramenta CASE. A fim de realizar o comparativo, foram selecionadas oito ferramentas CASE dentre as citadas nas seções anteriores, sendo estas julgadas as mais populares. São elas:

- ArgoUML (versão 0.24);
- Umbrello UML Modeller (versão 1.5.8);
- NetBeans (versão 6.0);
- JUDE (versão 1.4);
- IBM Rational Software Architect (versão 7.0);
- Sparx Enterprise Architect (versão 7.0);
- Visual Paradigm for UML (versão 6.0).

É importante salientar que, nesta pesquisa, utilizou-se a versão *Community* da ferramenta JUDE, sendo esta gratuita e mais comumente empregada. Em oposto a isto, utilizou-se a versão *Professional* da ferramenta Visual Paradigm, sendo esta proprietária e não gratuita. Buscou-se trabalhar com ferramentas diversificadas de maneira a enriquecer a pesquisa e, para cumprir este quesito, selecionou-se quatro ferramentas gratuitas e três ferramentas pagas.

As ferramentas citadas foram analisadas de acordo com alguns critérios de avaliação. Estes critérios foram selecionados através das funcionalidades gerais de uma ferramenta CASE e de um levantamento das especificações técnicas. São eles:

1. *Plataforma Windows*: Ter esta característica indica que a ferramenta CASE roda em plataforma Windows 9x/NT/ME/2000/XP;
2. *Plataforma Linux*: Indica que a ferramenta CASE roda em plataforma Linux;

¹⁴O estado da arte é o nível mais alto de desenvolvimento, seja de um aparelho, de uma técnica ou de uma área científica, alcançado em um tempo definido. O estado da arte indica o ponto em que o produto em questão deixa de ser um projeto técnico para se tornar uma obra-prima.

3. *Dicionário unificado de dados*: Mostra que a ferramenta CASE possui um repositório central de dados integrado para o armazenamento de vários projetos;
4. *Compartilhamento do repositório*: Esta característica dá a ferramenta a possibilidade de vários analistas compartilharem um mesmo repositório;
5. *Permissões e grupos de usuários*: Torna possível definir diferentes níveis de acesso a usuários ou grupos de usuários da ferramenta;
6. *Prototipação de telas*: Esta característica possibilita os usuários da ferramenta projetarem a interface do sistema em diagramas específicos para protótipos;
7. *Processos de negócio*: Propriedade que permite a criação de diagramas como IDEF0, IDEF3, IDF1X e possibilita mecanismos de referência cruzada entre processos e sistemas;
8. *Geração de código*: Ter esta propriedade garante que a ferramenta pode gerar código a partir de diagramas de classes e diagramas de seqüência.
9. *Engenharia reversa*: Indica que a ferramenta CASE possibilita a engenharia reversa, processo de análise de software que identifica os componentes de um sistema e seus interrelacionamentos, e que cria representações do sistema em outra forma ou em um nível de abstração mais elevado (PEREIRA, 2004 apud CROSS, 1990).
10. *Proprietária*: Característica que indica se a ferramenta é de propriedade de alguém ou de alguma empresa, instituição, organização, entre outros (qualquer pessoa jurídica ou física). Esta característica em geral está presente em ferramentas CASE que são vendidas almejando-se lucro.
11. *Livre*: A ferramenta CASE pode ser usada, copiada, estudada, modificada e redistribuída sem nenhuma restrição pelas empresas e usuários.
12. *Gratuita*: A ferramenta não envolve custo. Entretanto, empresas e usuários podem explorar a ferramenta comercialmente através do serviço envolvido (principalmente suporte).
13. *UML 1*: Apresentar esta propriedade significa que a ferramenta CASE oferece suporte a todos os diagramas da versão 1 de UML.
14. *UML 2*: Apresentar esta propriedade significa que a ferramenta CASE oferece suporte a todos os diagramas da versão 2 de UML.

15. *Armazenamento XMI*: Indica que a ferramenta CASE fornece troca de informações entre projetos baseando-se em XML. Mais especificamente, facilita a troca de metadados entre outras ferramentas de modelagem baseadas neste padrão da OMG.
16. *Controle de Versões*: Garante que o usuário da ferramenta CASE possa manter os diagramas em controle de versões, podendo também realizar verificações das alterações entre modelos de maneira visual.
17. *Relatórios*: Característica que indica que a ferramenta gera relatórios em formatos HTML, DOC ou PDF.

Tabela 3.2: Quadro comparativo entre as ferramentas.

CARACTERÍSTICAS		FERRAMENTAS CASE						
		Argo	Umbrello	NetBeans	JUDE	Rational	EA	VP
1	Plataforma Windows	✓	✗	✓	✓	✓	✓	✓
2	Plataforma Linux	✓	✓	✓	✓	✓	✗	✓
3	Dicionário unificado de dados	✗	✗	✗	✗	✓	✓	✓
4	Compartilhamento do repositório	✗	✗	✓	✗	✓	✓	✓
5	Permissões e grupos de usuários	✗	✗	✗	✗	✓	✓	✓
6	Prototipação de telas	✗	✗	✗	✗	✗	✓	✓
7	Processos de negócio	✗	✗	✗	✗	✓	✓	✓
8	Geração de código	✓	✓	✓	✗	✓	✓	✓
9	Engenharia reversa	✓	✗	✓	✗	✓	✓	✓
10	Proprietária	✗	✗	✗	✗	✓	✓	✓
11	Livre	✓	✓	✓	✓	✗	✗	✗
12	Gratuita	✓	✓	✓	✓	✗	✗	✗
13	UML 1	✓	✓	✓	✓	✓	✓	✓
14	UML 2	✗	✗	✗	✗	✓	✓	✓
15	Armazenamento XMI	✓	✓	✓	✗	✓	✓	✓
16	Controle de Versões	✗	✗	✓	✗	✓	✓	✓
17	Relatórios	✗	✗	✓	✗	✓	✓	✓

LEGENDA:

✓ Presença da característica

✗ Ausência da característica

Analisando-se a tabela 8.1, podemos eleger a ferramenta NetBeans como a mais vigorosa dentre as ferramentas gratuitas e a Visual Paradigm dentre as ferramentas pagas. Observa-se que o NetBeans apresenta características que poucas ferramentas CASE gratuitas possuem, estando praticamente comparável aos recursos oferecidos pela ferramenta Visual Paradigm. Além disso,

ela é a única presente neste comparativo que oferece suporte a alguns dos diagramas de UML 2 (essa característica não foi assinalada no quadro já que não engloba todos os diagramas de UML 2). Por outro lado, o Netbeans não é uma ferramenta leve e para se obter o máximo desta ferramenta é necessário uma configuração de hardware adequada que exige recursos computacionais nem sempre acessíveis ao usuário. Já a ferramenta Visual Paradigm, é ligeiramente mais leve que o NetBeans e apresenta características atraentes que contribuem muito na modelagem de software. Todavia é uma ferramenta proprietária e paga, sendo que a *suite* (o conjunto de ferramentas atreladas ao desenvolvimento de software disponível pela empresa) custa cerca de US\$ 2,400.00 (valor obtido em dezembro de 2007) que inclui um ano gratuito de manutenção.

Por conseguinte, atenta-se para a inviabilidade do uso destas ferramentas em laboratórios de universidades públicas, em incubadoras e em empresas de pequeno porte, uma vez que nem sempre possuem grandes recursos computacionais e financeiros para suportar as limitações apresentadas por estas ferramentas.

4 *Framework OCEAN*

O OCEAN é um framework orientado a objetos escrito originalmente em linguagem Small-Talk. Foi produzido por Ricardo Pereira e Silva em sua tese de doutorado (SILVA, 2000). Em meados de 2005, este framework passou por uma reengenharia onde foi reestruturado e re-codificado em linguagem Java por Ademir Coelho (COELHO, 2007), Thiago Machado (MACHADO, 2007) e João de Amorim (AMORIM, 2006). Novas características foram adicionadas e uma versão em Java do framework foi gerada.

O domínio do OCEAN atende à produção de diferentes ambientes de desenvolvimento de software baseado em notação de alto nível. O OCEAN possibilita que tais ambientes manipulem diferentes estruturas de especificação de projeto e apresentem diferentes funcionalidades. Cada tipo de especificação no framework OCEAN define quais os tipos de modelos válidos para aquela especificação, e cada modelo define quais os tipos de conceitos tratados por cada modelo (COELHO, 2007).

4.1 Características

Podemos destacar algumas características no framework OCEAN. Nesta seção, algumas delas são discutidas.

4.1.1 Caixa Cinza

O OCEAN é um framework do tipo caixa cinza (COELHO, 2007), uma vez que fornece funcionalidades através de subclasses concretas e permite que novas funcionalidades sejam adicionadas a partir da criação de novas subclasses.

Um ambiente desenvolvido a partir do framework OCEAN possui como única classe específica uma subclasse concreta de *EnvironmentManager*, que é uma classe abstrata. Todas as demais classes que compõem o ambiente poderão ser reutilizadas no desenvolvimento de outros ambientes. Através de uma subclasse de *EnvironmentManager* são definidas características es-

pecíficas de um ambiente, tais como, o tipo de especificações que serão tratadas, o mecanismo de visualização e edição associado a cada modelo e conceito que o mesmo trata, o mecanismo de armazenamento de especificações que utiliza, e quais as ferramentas utilizáveis para manipular especificações (AMORIM, 2006).

4.1.2 Flexibilidade

O framework OCEAN foi criado inicialmente a fim de se construir o ambiente SEA (apresentado em detalhes no capítulo 5), tendo como motivação a necessidade de construir um ambiente de software que suportasse o desenvolvimento e o uso de frameworks e componentes, e que fosse dotado de flexibilidade. Desta forma, o desenvolvimento do framework foi norteado pela busca da flexibilidade, caracterizada como requisito não funcional, a fim de permitir que o ambiente se moldasse às necessidades encontradas durante o seu ciclo de vida.

A análise dos ambientes existentes quando comparada com os requisitos desejáveis ao framework possibilitou generalizar o domínio destes ambientes, identificando características de flexibilidade ao framework OCEAN. Sua estrutura lhe confere flexibilidade para suportar o desenvolvimento de ambientes diferentes, como suporte à criação de estruturas de documentos, suporte à edição semântica de especificações (criação e remoção de modelos e conceitos; alteração, transferência e fusão de conceitos) e suporte à composição de ações de edição complexas através de ferramentas (por exemplo, a ferramenta de inserção de padrões de projeto em um artefato de software em desenvolvimento).

4.1.3 Extensibilidade

Um framework é um artefato de software que, diferente de uma aplicação, não precisa corresponder a um artefato executável. Sua finalidade é prover uma estrutura extensível, que generalize um domínio de aplicações, a ser utilizada no desenvolvimento de diferentes aplicações.

Especificações de projeto de artefatos do software são documentos estruturados, que descrevem visões distintas de projeto. O framework OCEAN suporta a definição de diferentes estruturas de documentos, e para isso, provê uma definição genérica de estrutura de documento e também funcionalidades genéricas para visualização e armazenamento de dados. Sendo assim, podemos caracterizar o OCEAN como um framework extensível.

Desenvolver uma especificação consiste em criar, modificar ou remover elementos de especificação, isto é, modelos e conceitos. Basicamente a alteração de um modelo consiste em incluir e remover conceitos. A modificação de um conceito consiste na alteração de suas

informações, podendo acarretar na inclusão e remoção de conceitos envolvidos em associações de sustentação ou referência (COELHO, 2007).

O framework OCEAN suporta edição semântica de especificações onde modificações sobre algum elemento de especificação refletem em modificações em sua representação visual. A edição semântica faz com que as alterações sejam procedidas de maneira coerente, possibilitando o estabelecimento de restrições às ações de edição.

Ferramentas no contexto do framework OCEAN são estruturas funcionais que reutilizam processos de edição semântica fornecidos pelo framework. Um ambiente construído sob o framework OCEAN agrega um gerente de ferramentas. Estas ferramentas passam a ser acessíveis através de menus, podendo ou não serem selecionadas em um contexto específico durante a operação do ambiente, além de poderem executar uma tarefa auxiliar durante a atuação de outra ferramenta (COELHO, 2007).

4.1.4 Suporte à edição gráfica

Conforme comentado anteriormente, o OCEAN foi produzido originalmente em linguagem Smalltalk. Sabendo que este foi construído para atender à criação de ambientes de desenvolvimento de software, era necessário prover suporte à edição gráfica, um meio pelo qual pudessem ser contruídos os artefatos de software. Para fornecer esta característica, utilizou-se o HotDraw (BRANT, 1999), framework escrito em Smalltalk que auxilia no desenvolvimento de editores gráficos.

Durante a reengenharia do framework OCEAN para a linguagem Java, igualmente se ofereceu suporte gráfico. Entretanto, para isto, utilizou-se a versão Java do HotDraw, o framework JHotDraw, embutido no OCEAN por João de Amorim em seu trabalho de conclusão de curso (AMORIM, 2006).

JHotDraw

O JHotDraw é um framework que auxilia no desenho técnico e gráfico estruturado de aplicações em linguagem Java. É voltado para o desenvolvimento de editores gráficos bidimensionais com semântica associada aos elementos gráficos. Este framework é uma versão produzida por Thomas Eggenschwiler e Erich Gamma para Java do framework HotDraw, que foi desenvolvido em SmallTalk por Kent Beck e Ward Cunningham (KAISER, 2007).

O JHotDraw define um esqueleto básico para um editor baseado em ¹GUI com ferramentas

dentro de paletas de ferramentas, com suporte a diferentes visões, figuras gráficas definidas pelo usuário, e com suporte para salvar, ler, e imprimir desenhos. Além disso, pode ser adaptado usando herança e combinando-se componentes.

Atrelado ao fato de ser pobre em termos de documentação, o JHotDraw não é um framework de fácil utilização, principalmente no que se refere à criação de figuras complexas. Ele traz, em conjunto com seu código-fonte, exemplos de aplicações que utilizam editores gráficos. Entretanto, suas poucas demonstrações apresentam apenas figuras simples e, em geral, são necessárias figuras compostas no contexto deste trabalho, com restrições de movimentação, ação, seleção e outros aspectos peculiares. Sendo assim, a implementação de aplicações que usufruem do JHotDraw se dá por tentativa e erro: inserindo trechos de código, testando e observando se o resultado obtido era o esperado. Poucos manuais sobre o framework JHotDraw são encontrados na internet e são ainda mais raros os tutoriais que exemplificam a codificação de situações específicas, como por exemplo, fazer uma determinada figura agregar outra figura.

Por outro lado, optar por não agregar o JHotDraw ao framework OCEAN faria com que os usuários do OCEAN consumissem muito mais tempo na implementação da parte gráfica, uma vez que o conjunto de classes padrão para a criação de elementos gráficos da linguagem Java não é tão alto nível como o conjunto de classes do JHotDraw e não oferece apoio à associação de semântica aos elementos gráficos. A ausência do JHotDraw tiraria o foco principal de um usuário do framework OCEAN: a construção de especificações de projeto.

Conclui-se que o JHotDraw favorece bastante o framework OCEAN nas tarefas gráficas, entretanto, oferece muito pouco auxílio aos usuários no aprendizado de utilização. Certamente, este é o principal aspecto que torna o JHotDraw pouco aceito pela comunidade Java.

4.1.5 Padrão *Model-View-Controller*

O padrão *Model-View-Controller* (MVC) é uma estratégia de desenvolvimento de software, onde a aplicação é separada e analisada em três partes distintas: o modelo (*model*), a visão (*view*) e o controlador (*controller*). O modelo corresponde à estrutura de armazenamento e tratamento dos dados específicos da aplicação. A visão é a representação visual destes dados. O controlador tem a responsabilidade de gerenciar os dispositivos de entrada de dados, recebendo os eventos da entrada e traduzindo em serviços para o modelo ou para a visão. O padrão MVC é muito visto em aplicações para web, onde a visão é geralmente a página HTML, e o código que gera os dados dinâmicos para dentro do HTML é o controlador. O modelo é representado

¹É a sigla para *Graphical User Interface*, em português: “Interface gráfica do usuário”.

pelo conteúdo de fato, geralmente armazenado em bancos de dados ou arquivos XML.

Uma das características mais importantes do padrão MVC é a separação da estrutura da informação (dados) de sua representação visual. Isto permite que se possa apresentar visualmente de diversas formas (visão) diferentes um mesmo conjunto de dados (modelo).

A estrutura dos documentos definida através do framework OCEAN se baseia no padrão MVC. Assim, documentos possuem a definição de sua estrutura em separado de sua apresentação visual. O modelo é representado pelas especificações e elementos de especificações. O framework gráfico JHotDraw (AMORIM, 2006), utilizado para criar a interface no OCEAN, também segue o padrão MVC.

4.2 Estrutura

O framework OCEAN apresenta um conjunto de superclasses abstratas que definem a estrutura das especificações que podem ser manipuladas por ambientes desenvolvidos sob este framework (ver figura 4.1). Essa estrutura de classes suporta a criação de estruturas de especificação distintas, através da definição de subclasses concretas. Uma especificação (uma instância de subclasse de *Specification*) agrega elementos de especificação (instâncias de subclasses de *ConceptualModel* e *Concept*), podendo estes serem conceitos ou modelos. Uma especificação registra associações de sustentação e referência entre pares de elementos de especificação, isto é, um elemento de especificação pode estar associado a outros elementos de especificação (AMORIM, 2006).

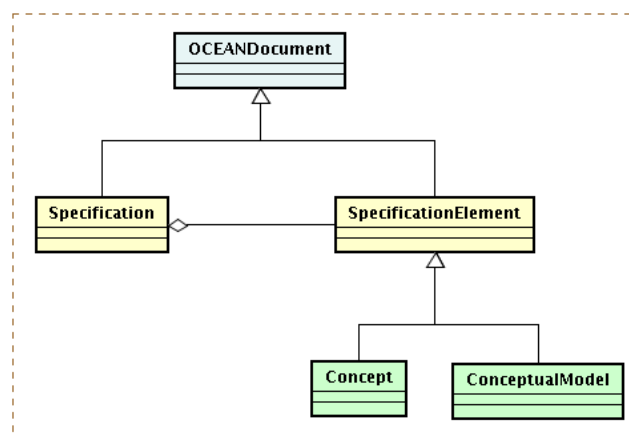


Figura 4.1: Superclasses do framework OCEAN que definem uma especificação.

4.3 Problemas

Como todo framework, o OCEAN apresenta alguns problemas específicos. Dentre eles, podemos destacar:

- Documentação de código-fonte incompleta;
- Documentação e código-fonte escritos de maneira mesclada, ora escrito em língua portuguesa, ora em língua inglesa. Desta forma, a compreensão do framework como um todo fica perturbada, uma vez que este não utiliza, nem mesmo impõe, nenhum padrão convencional de documentação e codificação;
- Ausência de manual de utilização do framework, e conseqüentemenete, falta de material de apoio que apresente noções essenciais para boas práticas de uso do framework;
- Em virtude de o framework ter sido originalmente produzido em linguagem Smalltalk, a indexação de listas é feita a partir de 1 (um) - e não de índice 0 (zero), como é o padrão da linguagem Java. Este aspecto dá margens à geração de erros de captura e inserção de elemento;
- Uso de recursos da linguagem Java depreciados e arcaicos, incitando os usuários do framework a reutilizar estes mesmos recursos em toda a aplicação final. Este aspecto se deve em parte ao software de tradução automática utilizado para a conversão de Smalltalk para Java e em parte devido ao fato de a reengenharia do framework ter-se estendido por vários anos, nos quais novas versões da linguagem Java foram lançadas. Assim, estes mecanismos arcaicos geram uma porção de alertas (*warnings*) na compilação, provocando uma execução não tão eficiente quanto esta poderia ser se utilizasse os recursos mais atuais disponíveis;
- Uso de uma versão desatualizada do framework JHotDraw (versão 5.3), instigando o usuário a adicionar características importantes que não estão incluídas na versão utilizada porém estão presentes na nova versão do JHotDraw (7.0). É importante destacar que quando João de Amorim (AMORIM, 2006) realizou seu trabalho de conclusão de curso em 2006, com o objetivo de embutir o framework JHotDraw no framework OCEAN, a versão mais atual era a versão 5.3. Além disso, é preciso levar em consideração que a atualização do JHotDraw não seria uma tarefa trivial, já que a versão atual não é compatível com a versão anterior.

Vale ressaltar que os problemas apontados dificultaram a utilização do framework. Com o objetivo de tornar o OCEAN maduro suficiente para ser usado em muitas outras aplicações, seria necessário uma ²refatoração (*refactoring*) do código existente escrito em Java, de forma a corrigir os problemas apresentados.

²Refatoração ou *refactoring* é o processo de modificar um sistema de software para melhorar a estrutura interna do código sem alterar seu comportamento externo.

5 *Ambiente SEA*

O ambiente SEA, assim como o framework OCEAN, foi produzido em linguagem Small-Talk durante a tese de doutorado de Ricardo Pereira e Silva (SILVA, 2000). Foi desenvolvido com o objetivo de estender as classes do framework OCEAN, prevendo o desenvolvimento e uso de artefatos de software reutilizáveis. Posteriormente, este ambiente de desenvolvimento de software adquiriu uma nova estrutura e recodificação em linguagem Java, uma vez que sofreu uma reengenharia por Ademir Coelho (COELHO, 2007) e Thiago Machado (MACHADO, 2007), em conjunto com o framework OCEAN.

Com o objetivo de promover o reuso no desenvolvimento de software, o ambiente SEA possibilita a utilização integrada de abordagens de desenvolvimento orientado a componentes e desenvolvimento baseado em frameworks, incluindo o uso de padrões de projeto. Suporta o desenvolvimento de frameworks, componentes e aplicações como estruturas baseadas no paradigma de orientação a objetos. O desenvolvimento de software no ambiente SEA consiste em produzir uma especificação de projeto deste artefato utilizando UML, havendo a possibilidade de converter essa especificação em código.

5.1 **Características**

O ambiente SEA proporciona especificação de framework, algo não contemplado na maioria das metodologias de desenvolvimento orientadas a objetos. A maior deficiência dessas metodologias em relação aos frameworks, diz respeito à impossibilidade de documentação das partes flexíveis do framework e a não ligação semântica entre o framework e as aplicações desenvolvidas a partir dele (MACHADO, 2007).

Outra característica do ambiente SEA é que este introduziu propriedades de redefinibilidade e essencialidade na especificação do framework. A redefinibilidade registra explicitamente se uma classe poderá ou não ser redefinida com subclasses nas aplicações desenvolvidas a partir do framework. Essencialidade de classe pode ser definida somente para as classes redefiníveis

e estabelece que qualquer aplicação desenvolvida a partir do framework em questão deverá usar essa classe ou estender uma classe como subclasse dessa classe no framework. Os métodos também recebem uma rotulação em nível de definição de flexibilidade nesse ambiente, podendo ser explicitamente classificados como base, abstratos ou *templates* (SILVA, 2000).

O ambiente SEA é constituído por um conjunto de ferramentas que lêem e alteram uma especificação de artefato de software da base de dados de especificações e que podem ser classificadas como ferramentas de edição, análise e transformação (MACHADO, 2007). Além disso, o ambiente foi concebido e implementado com a capacidade de inserir ou remover as ferramentas de forma flexível. Ele gerencia uma lista de ferramentas e fornece ao usuário a possibilidade de realizar alterações nessa lista. Quando uma ferramenta é inserida no ambiente, começa a fazer parte dessa lista e assim pode ser chamada pelo ambiente.

5.2 Classificação

Cada prática da engenharia de software deve ser apoiada por ferramentas adequadas a fim de garantir a visibilidade e produtividade da execução das tarefas. Portanto, ter ferramentas adequadas é tão importante quanto ter pessoas capacitadas e procedimentos de trabalhos bem definidos.

Uma ferramenta CASE (*Computer-Aided Software Engineering*, Engenharia de Software Auxiliada por Computador) é uma classificação que abrange toda ferramenta baseada em computadores que auxilia atividades de engenharia de software, desde análise de requisitos e modelagem até programação e testes. A partir deste âmbito, podemos classificar o ambiente SEA como uma ferramenta CASE, uma vez que este suporta a edição de UML e esta é uma área da engenharia de software.

Não há um padrão definido para a categorização das ferramentas CASE, no entanto, os termos abaixo são os que melhor o identificam.

- *Front End ou Upper CASE*: apóiam as etapas iniciais de criação dos sistemas como as fases de planejamento, análise e projeto do programa ou aplicação.
- *Back End ou Lower CASE*: dão apoio à parte física, isto é, a codificação, testes e manutenção da aplicação.
- *I-CASE ou Integrated CASE*: classifica os produtos que cobrem todo o ciclo de vida do software, desde os requisitos do sistema até o controle final da qualidade.

De acordo com este tipo de categorização, podemos classificar o ambiente SEA como uma ferramenta *Upper CASE*.

5.3 Estrutura

A estrutura do ambiente SEA é suportada pelo framework OCEAN e pelo framework JHot-Draw. O ambiente possui um repositório de especificações que possibilita o compartilhamento com diversas ferramentas. O gerente de ambiente é responsável pela interface com o usuário e possibilita o uso de ferramentas para manipular as especificações contidas no seu repositório. O gerente de armazenamento é responsável pelo intercâmbio entre o repositório de especificações e os dispositivos de armazenamento, possibilitando que especificações sejam gravadas ou carregadas (COELHO, 2007). A figura 5.1 mostra como é a estrutura do ambiente SEA.

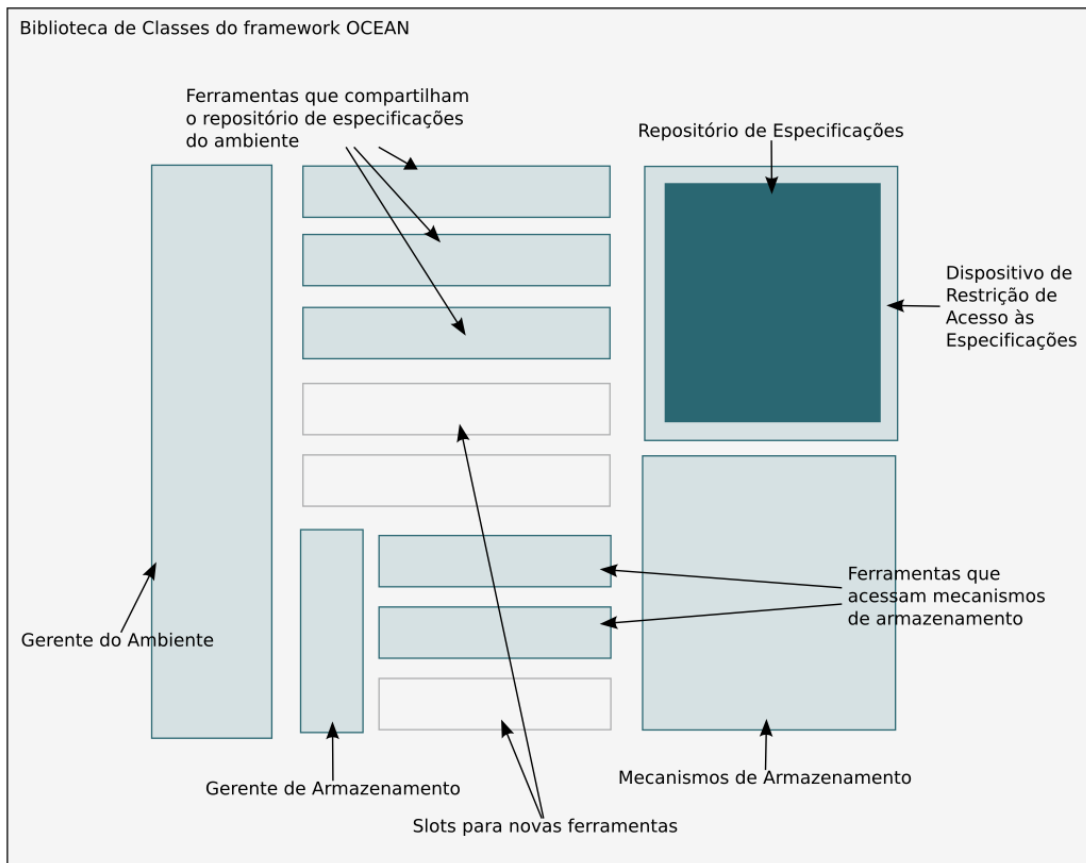


Figura 5.1: Estrutura do Ambiente SEA.

5.4 Especificações

De acordo com a abordagem adotada no framework OCEAN (SILVA, 2007), uma especificação agrega elementos de especificação, que podem ser conceitos ou modelos. Um elemento de especificação pode estar associado a outros elementos de especificação. Produzir uma estrutura de especificação consiste em definir um conjunto de tipos de modelo (subclasses concretas de *ConceptualModel*), um conjunto de tipos de conceito (subclasses concretas de *Concept*) e uma rede de associações entre os elementos destes conjuntos.

Dentre as especificações suportadas no ambiente de desenvolvimento de software SEA, podemos destacar:

- *Orientada a Objetos (OO)*: consiste em uma estrutura de classes que pode corresponder à especificação de projeto de um framework, de uma aplicação ou de um componente. No contexto do ambiente SEA, uma especificação OO é utilizada para designar uma especificação baseada no paradigma de orientação a objetos e produzida a partir do uso de notação de alto nível, no caso a UML. Esta especificação foi recodificada e reestruturada nos trabalhos de Ademir Coelho (COELHO, 2007) e Thiago Machado (MACHADO, 2007).
- *Cookbook ativo*: oferece suporte ao uso de frameworks através do fornecimento de instruções que auxiliam na tarefa de construir uma aplicação a partir do framework. *Cookbooks* ativos são mais que meros manuais, são hiperdocumentos que possuem link ativos que permitem ações de edição automatizada. Esta especificação também está presente na versão mais recente do ambiente SEA em linguagem Java.
- *Interface de componente*: relaciona os métodos fornecidos, os métodos requeridos e a associação a cada canal da interface. Sua estrutura é produzida relacionando assinaturas de métodos e canais, e definindo a ligação entre eles.
- *Geração Semi-Automatizada de Adaptação para Componentes*: automatização da análise de compatibilidade estrutural e comportamental entre componentes, durante o processo de especificação de projeto. Modela-se uma arquitetura de componentes, para que se possa visualizar suas conexões, e com isto pode-se fazer as análises necessárias para garantir a compatibilidade e o perfeito funcionamento desta arquitetura. Caso haja alguma incompatibilidade, possíveis soluções são propostas, ficando a cargo do desenvolvedor escolher, dentre elas, a melhor solução (SARTORI, 2005) (CUNHA, 2005). Esta especificação está presente apenas na versão em linguagem Smalltalk do ambiente SEA.

- *Workflow e CMM*: oferece gerenciamento durante o desenvolvimento das aplicações. Possibilita a modelagem e gerenciamento de processos de acordo com os requisitos. Este gerenciamento é feito através da implementação de workflow e da adaptação de requisitos de CMM (SCHEIDT, 2003). Esta especificação está presente apenas na versão em linguagem Smalltalk do ambiente SEA.
- *Documentos Textuais Estruturados*: fornece a criação, edição e verificação de consistência de documentos textuais estruturados, tais como registro de inspeção, especificação de requisitos, planos de testes, relatório de aplicação dos testes e registro de verificação da consistência de documentos. Estes documentos são incorporáveis a projetos em conjunto com outros documentos, como diagramas UML. Transforma a produção manual de documentos em um processo estruturado através de uma estrutura de documentos textuais dentro do ambiente SEA e do tratamento organizado e estruturado destes documentos, o que leva ao registro mais efetivo de informações do processo de desenvolvimento de software (BONFIM, 2004). Esta especificação está presente apenas na versão em Smalltalk do ambiente SEA.

5.5 Problemas

O ambiente SEA apresenta alguns problemas específicos, sendo a maioria deles causados por inconvenientes presentes no framework OCEAN. Dentre eles, podemos destacar:

- Assim como o framework OCEAN, apresenta documentação de código-fonte incompleta;
- Apresenta documentação e código-fonte escritos de maneira mesclada, ora escrito em língua portuguesa, ora em língua inglesa;
- Algumas ações geram mensagens de erros que são mostradas repetidamente em janelas de confirmação, quando se deveria apresentar ao usuário o erro apenas uma única vez;
- Uso de recursos da linguagem Java depreciados e arcaicos, uma vez que o OCEAN assim o faz;
- Dificuldade para depurar, já que o controle de execução oscila entre o SEA e o OCEAN.

6 Suporte à edição de UML 1 no Ambiente SEA

Nem todas as técnicas de modelagem da primeira versão de UML e nem todos os elementos sintáticos previstos nas técnicas de UML foram incorporados aos editores do ambiente SEA, o que resultou em uma diminuição da expressividade original. Todavia, foram introduzidas as seguintes extensões à linguagem no ambiente SEA (SILVA, 2000):

- Representação de conceitos do domínio de frameworks, não representáveis nas técnicas de UML;
- Agregação de recursos de modelagem a UML, como a possibilidade de estabelecimento de restrições semânticas aos elementos da especificação;
- Possibilidade de descrever o algoritmo dos métodos na especificação de projeto;
- Representação da ligação semântica entre elementos de uma especificação e entre especificações distintas;
- Possibilidade de especificações serem tratadas como hiperdocumentos, de modo que os elementos de uma especificação possuam links associados que apontem para outros documentos.

6.1 Diagramas suportados

Conforme citado anteriormente, o ambiente SEA não utilizou todas as técnicas de modelagem em UML na versão anterior a este trabalho, atendo-se a cinco destas técnicas (estando três destas incompletas), e a mais uma técnica adicional para a descrição do algoritmo dos métodos, que não é prevista em UML. Os diagramas suportados até a versão do ambiente SEA anterior a este trabalho são:

1. *Diagrama de Casos de Uso*: Suporte à criação de casos de uso, atores e relacionamentos de associação. Apresenta uma janela específica para a edição dos conceitos de ator e caso de uso. Os relacionamentos de extensão e inclusão ainda não estavam implementados.
2. *Diagrama de Classes*: Suporte à criação de classes e seus relacionamentos. Apresenta uma janela específica para a edição dos conceitos referentes a métodos e atributos de classe. As funcionalidades básicas desse diagrama foram supridas, observe o exemplo da figura 6.1.
3. *Diagrama de Seqüência*: Suporte à criação de objetos e mensagens. É um dos diagramas mais completos desta versão Java do ambiente SEA, ver exemplo na figura 6.2. Agrega uma janela específica para a edição dos conceitos de objeto e mensagem. Apresenta alguns problemas na inserção e seqüenciamento de mensagens e figuras incompletas.
4. *Diagrama de Atividades*: Apenas com a estrutura base e o conceito de atividade criados.
5. *Diagrama de Transição de Estados*: Apenas com a estrutura base e o conceito de estado criados. Apresenta uma janela específica para a edição de estado.
6. *Diagrama de Corpo de Método*: Concebido por Ricardo Pereira e Silva (SILVA, 2000), oferece suporte à modelagem de algoritmo de métodos de classe. Não é prevista em UML.

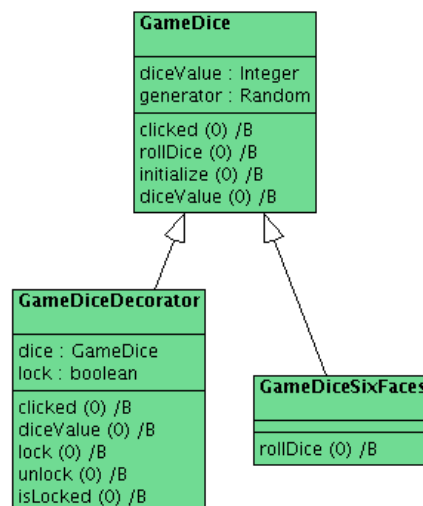


Figura 6.1: Parte de um diagrama de classes modelado na versão atual do ambiente SEA.

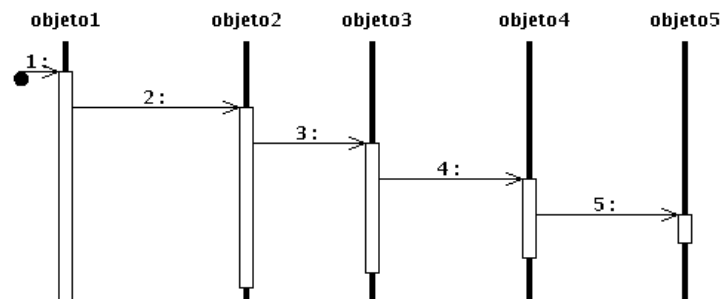


Figura 6.2: Estrutura de um diagrama de seqüência modelado na versão anterior a este trabalho do ambiente SEA.

6.2 Funcionalidades e ferramentas

Com o objetivo de suportar a produção de especificações orientadas a objetos, o ambiente SEA apresenta um conjunto de funcionalidades, que podem estar incorporadas às ferramentas (COELHO, 2007). Seguem descrições destas funcionalidades:

- *Consistência de especificações OO*: uma especificação OO no ambiente SEA apresenta semântica definida e uma gramática de atributos. Isto possibilita a verificação de consistência de especificações OO, bem como a conversão de uma especificação em outras linguagens.
- *Alteração de relações semânticas*: uma especificação contém conceitos e relações semânticas registradas, segundo a abordagem de definição de estruturas de especificação adotada no framework OCEAN.
- *Reuso de conceitos por cópia e colagem*: cópia e colagem são um dos recursos de edição semântica do ambiente SEA que usa área de transferência do ambiente, possibilitando o reuso de estruturas de conceito já criadas.
- *Criação automática de métodos de acesso aos atributos*: o ambiente SEA possui ferramentas para criação automática de métodos de acesso aos atributos, capazes de criar toda a estrutura de um método, isto é, assinatura e corpo.
- *Suporte à composição do diagrama de corpo de métodos*: o ambiente SEA possui uma ferramenta que varre os diagramas de seqüência e os diagramas de transição de estados e gera comandos *external*, que fornece apoio à construção de diagramas de corpo de método.

- *Suporte para alteração de frameworks*: o ambiente SEA possui uma ferramenta de transferência de partes de especificação para o framework. Esta ferramenta automatiza o procedimento de alterar o framework para suportar novas aplicações.
- *Suporte a ¹padrões de projeto*: o desenvolvimento no ambiente SEA corresponde à criação de uma especificação de projeto para posterior geração de código. O ambiente SEA possui uma ferramenta de inserção semi-automática de padrões, que permite selecionar uma especificação de padrão de projeto na biblioteca de padrões do ambiente, e inseri-la em uma especificação em desenvolvimento (AMORIM, 2006).
- *Suporte a engenharia reversa e geração de código*: o ambiente SEA dispõe de uma ferramenta com a capacidade de gerar diagrama de classes e código executável (PEREIRA, 2004).

¹Padrões de projeto são estruturas de classe que correspondem a soluções para problemas de projeto.

7 Suporte à edição de UML 2 no Ambiente SEA

Além de trazer diagramas inéditos, a linguagem UML 2 conduziu modificações estruturais importantes com relação a sua arquitetura, causando um notável refinamento e aumento de qualidade na generalidade de alguns dos diagramas. É evidente que as ferramentas CASE disponíveis no mercado estão buscando fornecer suporte a estas melhorias, entretanto, nota-se que pouquíssimas destas ferramentas não têm custo e/ou estão disponíveis para uso acadêmico, conforme pôde-se acompanhar no capítulo 3.

Ao realizar a reengenharia do framework OCEAN, Ademir Coelho (COELHO, 2007) teve como foco principal a implementação do núcleo do framework em linguagem Java. Já João Amorim (AMORIM, 2006) voltou-se à adaptação do framework JHotDraw ao framework OCEAN. Sendo assim, observa-se que dar suporte a diagramas no ambiente SEA foi uma atividade secundária para estes desenvolvedores pioneiros, visto que o objetivo de se criarem diagramas, no contexto de seus trabalhos, era validar o framework a fim de que futuros desenvolvedores pudessem usufruir do framework de maneira adequada.

Diante do exposto, poucos diagramas foram suportados e alguns ficaram incompletos como o diagrama de atividades e o diagrama de casos de uso. Detalhes estéticos não foram observados, assim como o framework OCEAN e o JHotDraw não puderam ser explorados de forma aprofundada.

Neste sentido, o presente trabalho consiste em dar suporte às mudanças de UML 2 no ambiente SEA (ver imagem 7.1), tornando-o mais aplicável e robusto para utilização. Integraram-se novos diagramas após criada uma nova especificação orientada objetos que estende a especificação atual. Conseqüentemente, foi possível validar o framework OCEAN de maneira mais expressiva.

As próximas seções deste capítulo descreverão detalhadamente o processo de construção da extensão do ambiente SEA por meio de diagramas e imagens que exemplificam diagramas possíveis de serem criados.

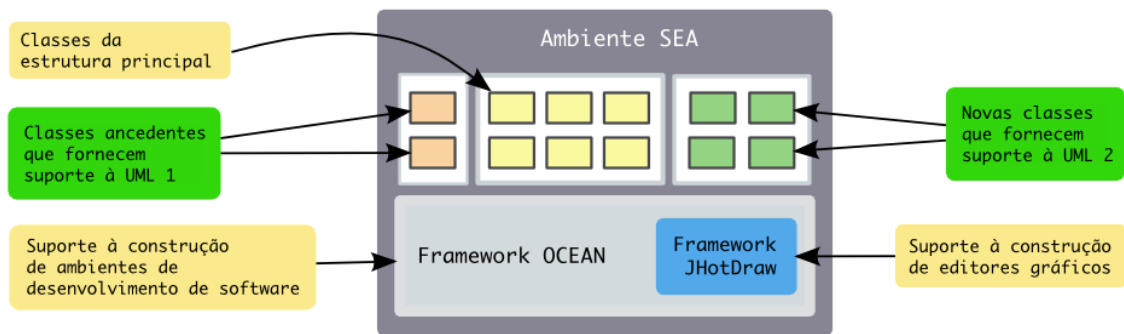


Figura 7.1: Camadas presentes do ambiente SEA.

7.1 Novos documentos

O desenvolvimento e a utilização de frameworks são tarefas complexas e, por isso, uma boa documentação é imprescindível (AMORIM, 2006). Tal documentação deve conter informações sobre o domínio tratado pelo framework, sua estrutura e seu funcionamento. A notação UML de um framework é uma valiosa fonte de informação sobre o mesmo. Existe ainda a forma de documentação mais elementar, que é a disponibilização do código fonte aos usuários.

AMORIM adotou a estratégia de criação de um ¹*cookbook* a fim de documentar os passos a serem realizados para a criação um novo documento com editores gráficos sob o framework OCEAN, que sub-utiliza o framework JHotDraw para a edição gráfica. O objetivo buscado era que cada novo desenvolvimento de um documento seguisse este roteiro para criá-lo de maneira mais rápida, cobrindo todas as características básicas necessárias para o correto funcionamento. Assim, este *cookbook* foi estudado e utilizado para a criação de novos diagramas no presente trabalho, uma vez que estes necessitam de editores gráficos.

Analisando o roteiro elaborado por AMORIM de forma detalhada, pode-se sintetizar os passos a serem realizados na criação de um documento com editor gráfico da seguinte forma:

1. Criar um modelo, uma subclasse de *ConceptualModel*;
2. Criar um *Drawing*, uma subclasse de *SpecificationDrawing*;
3. Criar um *DrawingView*, uma subclasse de *SpecificationDrawingView*;
4. Criar um *Window*, uma subclasse de *EditorAuxWindow* e implementação da interface

¹Cookbooks são conjuntos de receitas textuais para a utilização de um framework. Nele são descritos passos de como se desenvolver uma aplicação naquele framework, da maneira mais rápida e tranqüila possível, tal como uma receita de bolo. Sua principal vantagem é a capacidade de responder a questões chave minimizando o tempo gasto para produzir aplicações.

IComunicacao;

5. Criar um *Editor*, uma subclasse de *SpecificationEditor*;
6. Criar conceitos (de acordo com a necessidade do documento), subclasses de *Concept*;
7. Criar figuras (de acordo com a necessidade do documento), subclasses de *SpecificationCompositeFigure* ou *SpecificationLineFigure*.

O diagrama de classes presente na imagem 7.2 exibe a arquitetura resultante da utilização do framework OCEAN para a construção de um novo documento gráfico no ambiente SEA, seguindo os passos do *cookbook* apresentado por AMORIM.

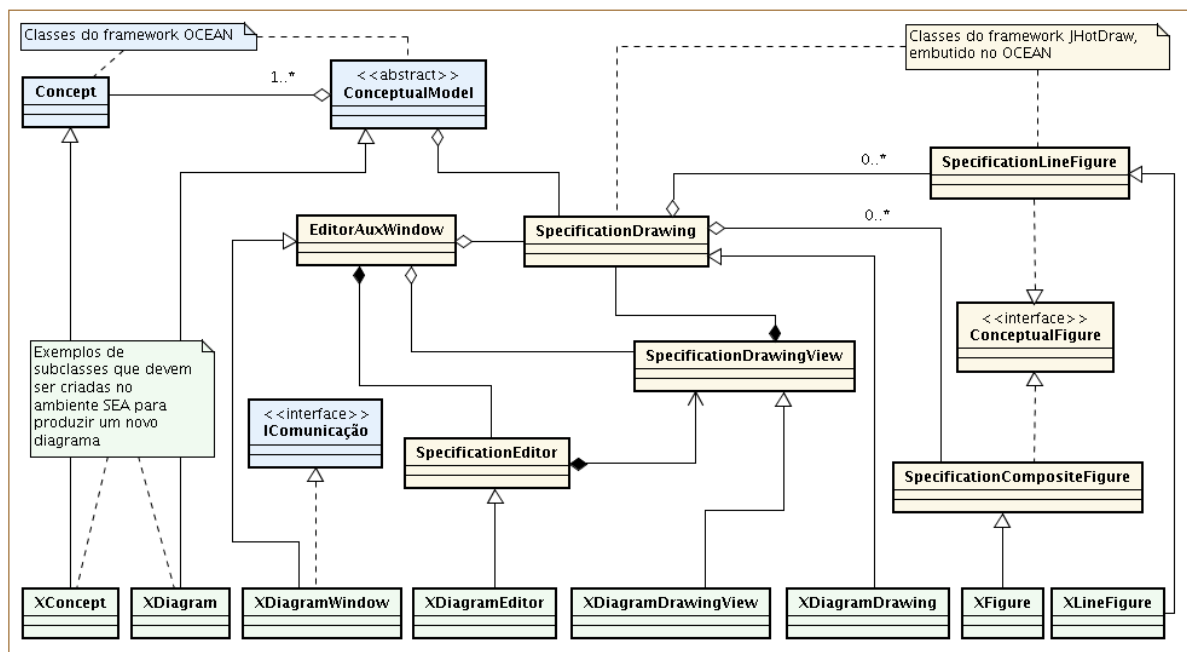


Figura 7.2: Diagrama de classes que exemplifica o processo de criação de um novo documento a ser adicionado no ambiente SEA.

Comprovando a grande utilidade da documentação, oito documentos foram criados neste trabalho seguindo os passos do roteiro acima e outros dois foram modificados a fim de se acrescentar novas características visuais e semânticas. Convém salientar que, no contexto deste trabalho, cada documento representa um diagrama de UML.

No apêndice A deste trabalho, encontra-se uma nova versão do *cookbook* concebido por João de Amorim, contendo novos comentários e particularidades de implementação presentes neste trabalho.

7.2 Nova especificação orientada a objetos

A expressão “especificação orientada a objetos” é utilizada neste trabalho para designar uma especificação de artefato de software, baseada no paradigma de orientação a objetos e produzida a partir do uso de notação de alto nível, como UML.

Em sua versão anterior a este trabalho, o ambiente SEA continha a especificação orientada a objetos: *SEASpecification*, que fornecia apoio a alguns diagramas de UML 1. Com o objetivo de dar suporte a UML 2, estendeu-se esta especificação criando-se uma nova especificação: *UML2SEASpecification*, que herda todas as características de *SEASpecification*. Inicialmente, a idéia era criar uma nova especificação e não especializar a já existente. Entretanto, analisando que muitas funcionalidades eram reusáveis, optou-se por herdar conceitos e modelos existentes, sobrecrevendo as características que foram alteradas na segunda versão de UML.

7.3 Novos modelos e conceitos

Um modelo em uma especificação orientada a objetos possui uma estrutura de informações (classes, atributos, métodos) e uma ou mais representações visuais desta estrutura (representação gráfica e textual). A expressão “modelo conceitual” e a expressão “modelo” se referem à estrutura de informações. No ambiente SEA, existem algumas subclasses concretas de *Conceptual-Model* associadas a tipos de modelo. Um modelo de uma especificação é uma instância de uma dessas subclasses. No âmbito de modelagem no SEA, qualquer diagrama de UML pode ser um tipo de modelo. Na imagem 7.3 pode-se observar os modelos criados para a especificação orientada a objetos UML 2.

A expressão “conceito” é usada neste trabalho para denominar as unidades de informação do domínio de modelagem tratado. No caso da modelagem baseada no paradigma de orientação a objetos como tratado no ambiente SEA, constituem tipos de conceito: classe, atributo, mensagem, caso de uso, ator, herança, entre outros. No ambiente SEA, existem algumas subclasses concretas de *Concept* para tipos de conceito anteriormente construídos. Modelos referenciam instâncias destas classes.

Como o ambiente SEA, na versão anterior a este trabalho, apenas fornecia suporte a alguns diagramas de UML 1, muitos modelos (diagramas) e conceitos de UML 1 e 2 ficaram ausentes. Entretanto, no decorrer deste trabalho, estes modelos e conceitos foram construídos e são apontados e detalhados nas próximas subseções.

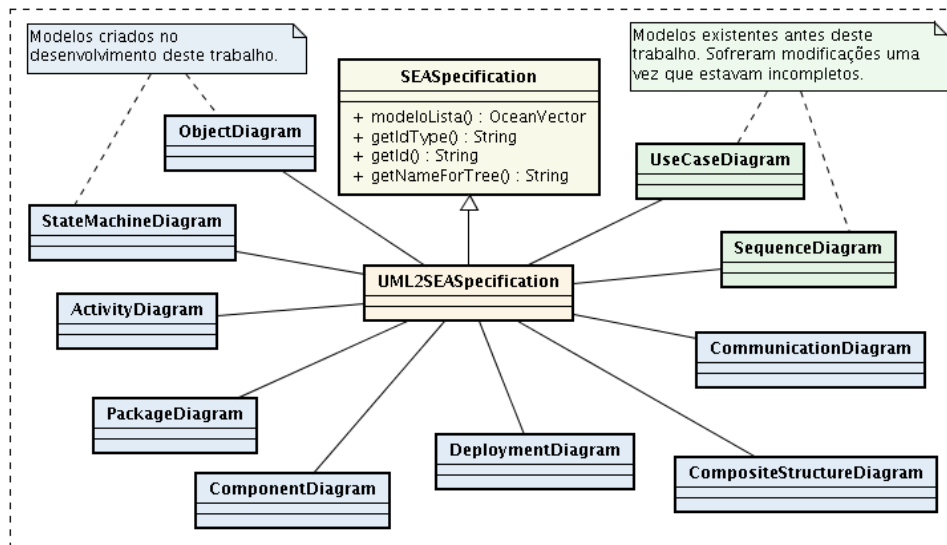


Figura 7.3: Diagrama de classes que expõe os modelos criados.

7.3.1 Diagrama de Objetos

Com o objetivo de fornecer suporte ao diagrama de objetos de UML 2, um novo modelo foi adicionado ao ambiente SEA: *ObjectDiagram*. Este modelo inicializa e agrega uma lista de conceitos que envolvem este diagrama, como por exemplo: *Instance specification* e *Association*. Um objeto de *ObjectDiagram* cria um objeto *ObjectDiagramDrawing*, que por sua vez cria as figuras necessárias ao diagrama como *InstanceSpecificationFigure* e *AssociationLine-Connection*.

Uma vez que o modelo *ObjectDiagram* foi criado, este precisa ser vinculado a uma janela, no caso, um objeto da classe *ObjectDiagramWindow*, a fim de estar disponível para utilização via *menu*. Através da classe *ReferenceManager* fazemos esta associação por meio de um *hash*, que guarda um modelo como chave e uma janela como valor. Esta janela cria um editor - instância de *ObjectDiagramEditor*. O objeto editor cria as ferramentas necessárias à confecção do diagrama associando-as com as figuras e conceitos respectivos. De modo geral, este editor é utilizado para se criar ações de interface.

É possível observar a arquitetura completa das classes implementadas para a criação do diagrama de objetos no ambiente SEA através do diagrama de classes na imagem 7.4. Neste diagrama, encontram-se todos os conceitos criados e utilizados, as figuras e todas as classes que compõem a estrutura básica do modelo.

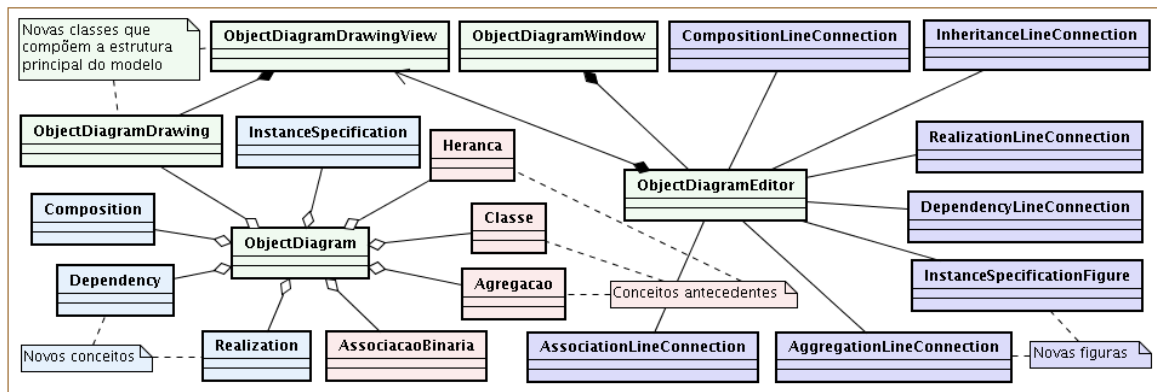


Figura 7.4: Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de objetos.

Conceitos antecedentes

O diagrama de objetos utiliza alguns conceitos já existentes em UML 1 e já suportados pelo ambiente SEA. Diferente da forma como a extensão foi implementada, as classes anteriores estão em língua portuguesa. Assim, representados por classes em linguagem Java no ambiente SEA, os conceitos são listados a seguir:

- *Association*: É uma conexão entre classes, um relacionamento que descreve uma série de ligações, onde a ligação é definida como a semântica entre as duplas de objetos ligados. Este conceito é representado pela classe “AssociacaoBinaria” no ambiente SEA.
- *Aggregation*: É uma forma especializada de *Association* na qual um todo é relacionado com suas partes. Esta relação também é conhecida como relação de conteúdo. Este conceito é representado pela classe “Agregacao” no ambiente SEA.
- *Class*: Representa um conjunto de objetos com características afins. Define o comportamento dos objetos através de operações ou métodos, e quais estados ele é capaz de manter, através de atributos. Este conceito é representado pela classe “Classe” no ambiente SEA.
- *Generalization*: É a capacidade de se criar superclasses que encapsulam estrutura e comportamento comuns a várias classes. Este conceito é representado pela classe “Heranca” no ambiente SEA.

Conceitos novos

O diagrama de objetos traz alguns conceitos novos de UML 2. Outros são conceitos existentes na primeira versão de UML, contudo ainda não havia sido implementados no ambiente

SEA. Estes conceitos novos no ambiente SEA são listados a seguir. Estão identificados em inglês, em conformidade com os elementos sintáticos apresentados nas especificações OMG (OMG, 2006a) (OMG, 2007b). No ambiente, são representados por classes codificadas em linguagem Java.

- *Instance specification*: Uma especificação de instância é um elemento do modelo que representa um objeto em um sistema modelado. Especifica a existência de uma entidade em um sistema modelado e descreve a entidade parcialmente ou completamente. Este conceito é representado pela classe “*InstanceSpecification*” no ambiente SEA.
- *Composition*: É uma agregação onde uma classe está contida e constitui em outra. Se o objeto da classe que contém for destruído, as classes da agregação de composição serão destruídas juntamente, já que as mesmas fazem parte de outra. Este conceito é representado pela classe “*Composition*” no ambiente SEA.
- *Dependency*: É o relacionamento que indica a ocorrência de um relacionamento semântico entre dois ou mais elementos do modelo, onde uma classe é dependente de algumas operações de outra classe, mas não tem uma dependência estrutural interna com esta classe que fornece as operações. Este conceito é representado pela classe “*Dependency*” no ambiente SEA.
- *Realization*: É o relacionamento entre uma interface e o elemento que a implementa. Este conceito é representado pela classe “*Realization*” no ambiente SEA.

Figuras novas

Construiu-se uma figura para cada conceito, sendo que cinco destas figuras são representadas por linhas (classes filhas da classe *SpecificationLineFigure* que pertence ao JHotDraw). Apesar de não estar mapeado na imagem 7.4 por falta de espaço, vale ressaltar que uma figura possui uma referência para o conceito que está representando. No editor do modelo isto sempre é deixado explícito ao se criar a figura.

Primeiramente foi criada a classe *AssociationLineConnection*. Ela representa uma linha simples que não apresenta decorações, como setas, tanto no início como no fim. Uma vez concluída a implementação da linha de conexão, bastou adicioná-la na classe *drawing* do modelo e implementar os construtores referentes aos conceitos que podem se relacionar através de uma associação. Em outras palavras, se queremos fazer com que dois objetos se relacionem através de uma associação, basta incluir no conceito “*AssociacaoBinaria*” um construtor que recebe duas instâncias da classe *InstanceSpecification*.

Todas as outras figuras que utilizam linhas simples puderam estender a classe *AssociationLineConnection*, uma vez que ela já implementou os métodos necessários para uma linha rudimentar. Apenas era necessário inserir decorações ou alterar a forma de desenhar a linha, afinal algumas figuras precisavam ser tracejadas. Nas figuras *AggregationLineConnection* e *CompositionLineConnection* colocou-se uma decoração já implementada na versão anterior do ambiente SEA: uma instância de *AggregationDecoration*. Ao criar essa instância era necessário passar pelo construtor a string “agregacao”, a fim de se optar por uma decoração com um losango sem preenchimento, ou “composicao”, de maneira a escolher uma decoração com um losango preenchido em preto. Na figura *DependencyLineConnection*, foi necessário sobrescrever o método *drawLine*, já que essa figura necessitou de um desenho de linha particular por ser tracejada. Uma seta aberta também foi inserida nesta figura como decoração e, assim, a figura *RealizationLineConnection* pode estendê-la, somente alterando a decoração final para uma seta fechada, característica da realização.

Por fim, tem-se a figura *InstanceSpecificationFigure* que representa o conceito *InstanceSpecification*. Como é uma figura composta, ela estende *SpecificationCompositeFigure*, classe do framework JHotDraw. Toda figura composta possui uma figura de representação, que pode ser qualquer instância de *Figure*. Essa figura de representação é a figura de fundo da figura resultante. Se queremos criar uma figura quadrada (uma classe, por exemplo), então devemos setar uma *RectangleFigure* na figura de representação; se queremos criar uma figura arredondada (uma colaboração, por exemplo), então devemos setar uma *EllipseFigure*. Para a figura *InstanceSpecificationFigure*, setou-se um retângulo para englobar todas as figuras internas. Dentro deste retângulo, foi necessário criar duas instâncias de figuras que dão suporte à composição de outras figuras internas (instâncias da classe *GraphicalCompositeFigure*). Na primeira figura, incluiu-se duas figuras de texto: uma figura que representa o estereótipo, outra que representa o nome da classe seguido do nome da instância - separados por dois pontos e com sublinhado. A segunda figura não é criada por *default*, apenas se o usuário adiciona atributos e seus valores. Cada requisição do usuário para criação de um novo atributo no objeto cria uma nova figura de texto (uma instância de *TextFigure*) dentro da figura de agregação. O resultado final é um retângulo dividido por uma linha que separa a definição do objeto de seus atributos.

Exemplo de uso

Um exemplo de diagrama de objetos executável na nova versão do ambiente SEA pode ser visualizado na imagem 7.5. Os objetos criados neste diagrama representam figuras geométricas e seus relacionamentos.

Se o usuário desejar adicionar um novo atributo ao objeto, basta clicar com o botão direito na figura do conceito *InstanceSpecification* e, fornecer o nome do atributo e seu respectivo valor. O nome da classe que o objeto referencia e o nome do objeto também são possíveis de serem alterados através do botão direito do mouse.

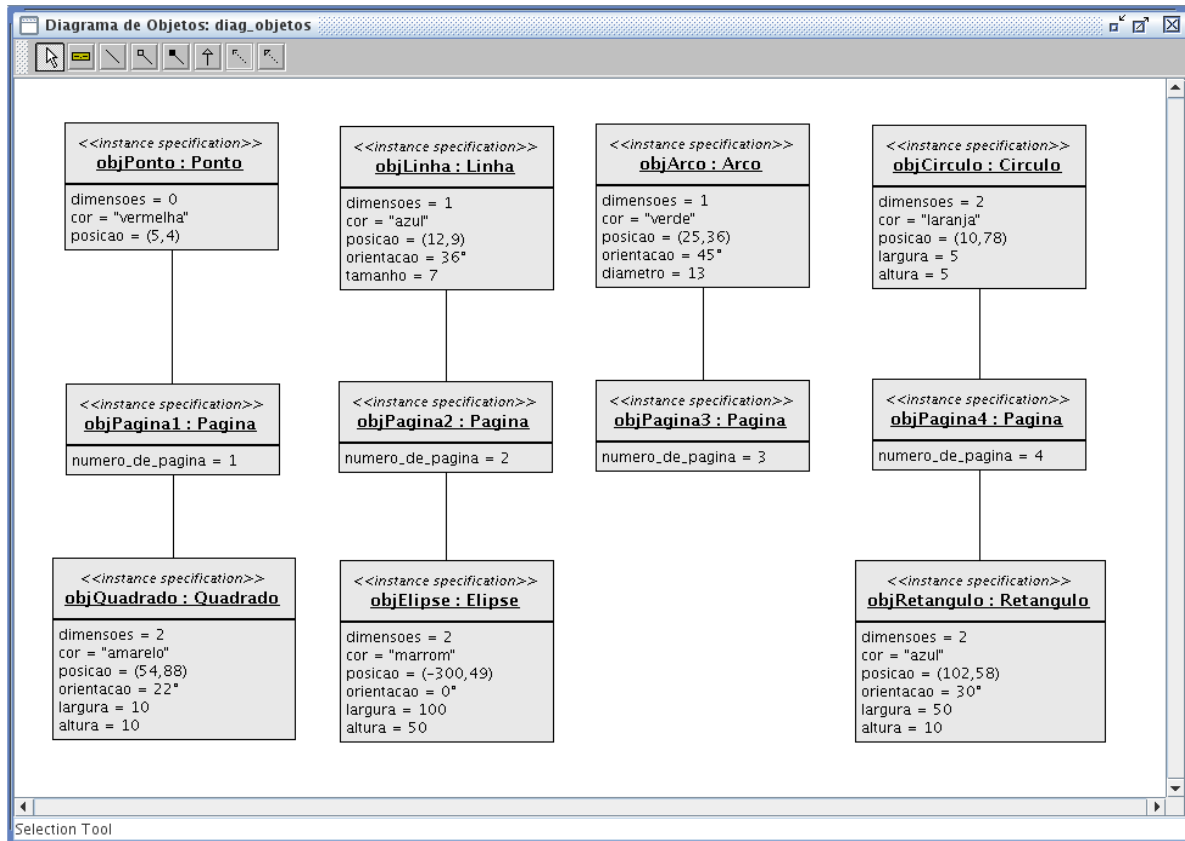


Figura 7.5: Exemplo de diagrama de objetos construído no ambiente SEA.

7.3.2 Diagrama de Pacotes

Na segunda versão de UML, o diagrama de pacotes ganhou particularidade e, desta maneira, a implementação de seu modelo pôde ser realizada e adicionada ao ambiente SEA. É possível visualizar a arquitetura completa das classes implementadas para a criação do diagrama de pacotes através do diagrama de classes na imagem 7.6. Neste diagrama exibem-se todos os conceitos criados e utilizados, as figuras e todas as classes que compõem a estrutura básica do modelo.

O diagrama de pacotes pode conter pacotes que agregam classes e estas podem estar relacionadas entre si de diversas maneiras. Pacotes também se relacionam entre si através de novos conceitos de dependências específicas comentados adiante.

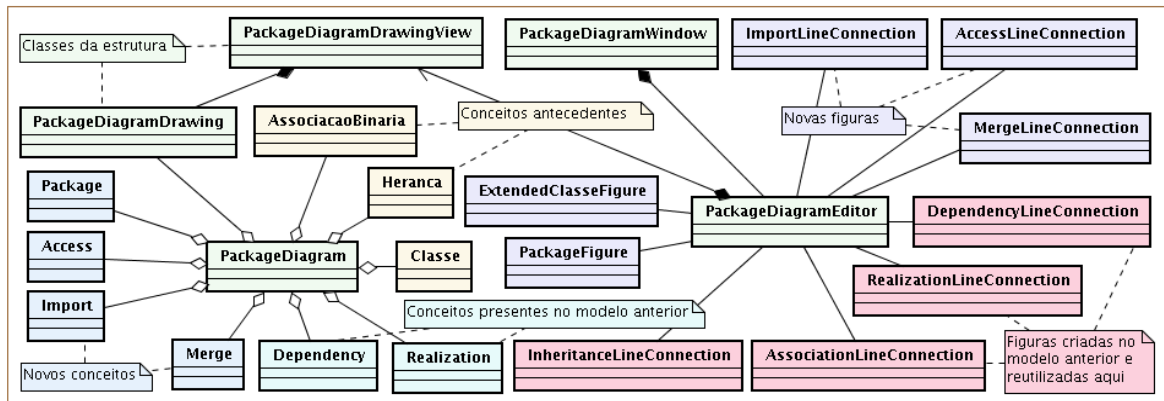


Figura 7.6: Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de pacotes.

Conceitos antecedentes

Assim como o diagrama de objetos, o diagrama de pacotes também reusa alguns conceitos existentes na versão anterior do ambiente SEA. É importante observar que as classes criadas, referentes a estes conceitos, foram nomeadas em língua portuguesa, diferente da forma da extensão realizada. Também foi possível reusar alguns conceitos pertencentes ao modelo anterior, o diagrama de objetos. Estes conceitos reutilizados são listados a seguir:

- *Association*: Este conceito é representado pela classe “AssociaçãoBinaria”.
- *Class*: Este conceito é representado pela classe nomeada “Classe”.
- *Generalization*: Este conceito é representado pela classe “Heranca”.
- *Dependency*: Este conceito foi criado no modelo de diagrama de objetos e apenas foi reutilizado neste modelo. É representado pela classe “Dependency”.
- *Realization*: Este conceito foi criado no modelo de diagrama de objetos e apenas foi reutilizado neste modelo. É representado pela classe “Realization”.

Conceitos novos

O diagrama de pacotes formalizou o elemento sintático “pacote” e forneceu tipos inéditos de relacionamentos para se representar os possíveis vínculos entre pacotes. Os novos conceitos são indicados e detalhados abaixo:

- *Package*: Um pacote é um agrupador de elementos de especificação voltado a auxiliar o desafio da administração de complexidade em especificações de projeto. Pode ser usado

para agrupar um diagrama, um conjunto de diagramas ou um subconjunto dos elementos de um diagrama, como classes. Este conceito é representado neste modelo pela classe “*Package*”.

- *Import*: A importação de pacote estabelece que o elemento importador incorpore o conteúdo do pacote importado, como se fosse definido nele. Existem duas modalidades de importação, em função da visibilidade do conteúdo importado: importação e acesso. A importação, propriamente, se refere a uma incorporação de elementos com visibilidade pública. Isto é, quem observar o conteúdo do elemento importador verá tanto os elementos definidos nele quanto os elementos importados, como se tivessem sido definidos todos nesse elemento (SILVA, 2007). Este conceito é representado no modelo de diagrama de pacotes pela classe “*Import*”.
- *Access*: O conceito de acesso é uma modalidade de importação com visibilidade privada. Em outras palavras, apenas o elemento importador tem visibilidade do conteúdo do pacote importado. Um observador que olhe o elemento importador não terá visibilidade do conteúdo importado através de uma relação de acesso (SILVA, 2007). Este conceito é representado no modelo de diagrama de pacotes pela classe “*Access*”.
- *Merge*: A fusão de pacote é um tipo de relacionamento que envolve um par de pacotes. Um dos pacotes fornece conteúdo a ser fundido com conteúdo de outro pacotes. O resultado da fusão é diferente para elementos que se repetem nos pacotes envolvidos e para os que não se repetem. Para os elementos comuns, há uma combinação, similar a uma relação de generalização, em que o resultado é uma fusão do conteúdo original com o conteúdo vindo do outro pacote (SILVA, 2007). Este conceito é representado no modelo de diagrama de pacotes pela classe “*Merge*”.

Figuras novas

Algumas das figuras que estão presentes neste diagrama e que representam conceitos de relacionamentos já haviam sido implementadas ou no modelo anterior ou na versão anterior do ambiente. Apenas era necessário reutilizá-las referenciado-as na construção do modelo. Restou criar linhas de conexão aos conceitos de importação, acesso e fusão, e a figura que representa o elemento sintático “pacote”.

As figuras que representam os conceitos “*Import*”, “*Access*” e “*Merge*” foram denominadas “*ImportLineConnection*”, “*AccessLineConnection*” e “*MergeLineConnection*”, respectivamente. Todas estas classes puderam herdar as propriedades e as operações da classe “*Depen-*

dencyLineConnection”, uma vez que graficamente e semanticamente são muito semelhantes. Foi suficiente sobrescrever o método *drawLine* a fim de adicionar ao meio da linha tracejada o estereótipo “<< import >>”, “<< access >>” ou “<< merge >>”, que varia de acordo com figura sendo criada.

A figura que representa o conceito “pacote” teve a implementação extremamente custosa, pois fugia de todos os padrões de figuras já implementados. Ela é composta por dois retângulos: o primeiro com largura e altura fixa, que simboliza a “aba” do pacote; o segundo possui largura e altura que variam de acordo com o conteúdo do pacote, agregando apenas uma figura de texto por padrão, que guarda o nome do pacote. O pacote é redimensionável e é possível agregar classes em seu conteúdo.

Criou-se uma extensão da figura “*ClasseFigure*”, nomeada “*ExtendedClassFigure*”. Esta figura redefine algumas propriedades gráficas como a cor e o tamanho default de margem do retângulo que compõe a figura. Isto foi necessário uma vez que a figura antiga não seguia os padrões que as novas figuras estavam sendo desenvolvidas. Criou-se uma classe estendida com o intuito de preservar a figura antiga, tão utilizada por outras classes.

Redimensionamento

Antes da versão do ambiente SEA desenvolvida no decorrer deste trabalho, as figuras não eram redimensionáveis. Esta característica não estava inerente pois não era imprescindível nos modelos até então construídos e porque o framework JHotDraw somente trazia esta característica por padrão em figuras simples. Em UML 2, muitos elementos sintáticos trazem a possibilidade de agregar outros elementos sintáticos em seu interior e, para isto, a característica de redimensionamento se tornou essencial.

O JHotDraw traz duas classes para que um nova figura sendo criada possa estender a fim de ganhar propriedades e funcionalidades gráficas: *SpecificationCompositeFigure*, usada quando se deseja conceber uma figura fechada que pode trazer outras figuras em seu interior, ou *SpecificationLineFigure*, usada quando se deseja criar algum tipo de linha. Ambas as classes trazem consigo o método *handles* que cria os pontos de seleção na figura para realização de edição e movimentação. Entretanto, estes pontos de seleção deveriam possibilitar por padrão o redimensionamento. O fato é que quando se tem uma figura com muitas outras figuras internas (figuras de texto, figuras de separação, ícones), o framework JHotDraw desconhece qual destas figuras ele deve redimensionar e em que proporção, e por esta razão o redimensionamento não se torna automático nestes casos.

Diante do exposto, o desafio inicial era tornar a figura que representa o elemento sintático “pacote” redimensionável, de maneira que posteriormente pudesse ser adicionada a característica de agregar figuras que representam o conceito “classe” em seu interior. Esta foi uma dentre as tarefas mais custosas deste trabalho, pois como não haviam tutorias e exemplos que proovessem esta característica, a implementação se deu por tentativa e erro, assim como tantas outras tarefas gráficas deste trabalho.

A solução adotada foi sobrescrever o método *handles*, que retorna um vetor de pontos de seleção. Cada ponto de seleção adicionado no vetor é uma instância de *Handle*. Estas instâncias, que por *default* eram adquiridas através de métodos estáticos da classe *BoxHandleKit*, foram redefinidas criando-se classes internas. As instâncias dessas classes internas foram passadas como argumentos nos métodos *addElement* da classe *Vector*. A classe *Handle* herda de *LocatorHandle*, que implementa a interface *AbstractHandle*. Necessitou-se sobrescrever o método *invokeStep* de *AbstractHandle*. Neste método, realizou-se o redimensionamento através da chamada do método *displayBox(Point, Point)* da figura interna que se desejava redimensionar. No caso da figura de pacote, a figura interna que desejava-se redimensionar era o retângulo maior que guarda o nome do pacote - não era necessário redimensionar a “aba” do pacote, o retângulo menor sem conteúdo. É importante observar que o método *invokeStep* é por *default* sobrescrito pelo JHotDraw redimensionando a figura mais externa (a figura de fundo, a que guarda o conteúdo) e por isso o redimensionamento não era sucedido. Era necessário especificar qual figura interna da figura mais externa desejava-se redimensionar.

Convém lembrar que este mecanismo foi implementado em todas as figuras dos modelos criados que necessitaram de redimensionamento. Poderá ser utilizado também em novas figuras complexas que venham a ser criadas em outros trabalhos no ambiente SEA.

Agregação de Figuras

Quando se trata de agregação de figuras neste trabalho, se refere a elementos sintáticos agregando outros elementos sintáticos, uma vez que um elemento sintático é representado sintaticamente e semanticamente por um conceito e graficamente por uma figura - uma classe filha de classes do framework JHotDraw.

Sabe-se que os pacotes, presentes no diagrama de pacotes, podem agregar classes em seu interior. Assim como o redimensionamento, esta não é uma característica inerente ao framework OCEAN e, até o presente trabalho, não havia sido necessário que elementos sintáticos englobassem outros. A implementação desta característica também foi uma das tarefas mais penosas deste trabalho, uma vez que não haviam exemplos de implementação.

A solução adotada para prover esta funcionalidade consistiu em implementar restrições de movimentação em figuras agregadas e vínculos entre as instâncias dos conceitos associados. A idéia era que a partir do momento em que uma determinada figura fosse arrastada totalmente acima de outra, que fosse acionado um mecanismo que se retringisse a movimentação da figura interna para se movimentar apenas nos limites da figura mais externa. Além disso, a figura interna deveria se movimentar quando a figura externa se movimentasse.

O algoritmo para agregação de figuras consistiu em alguns passos, resumidamente:

1. No conceito agregador, adicionou-se como atributo uma lista dos conceitos a serem agregados. Por exemplo, no caso do conceito “pacote”, adicionou-se uma lista de conceitos “classe”, que permanece nula no caso de não haver agregação;
2. Na figura agregadora, adicionou-se como atributo uma lista das figuras agregadas. No caso da figura que representa o conceito “pacote”, adicionou-se uma lista de figuras “*ClasseFigure*”, que permanece nula no caso de não haver agregação;
3. Na figura agregadora, sobrescreveu-se o método *moveBy* chamando-se o método da superclasse e, em seguida, varrendo-se cada item da lista de figuras agregadas a fim fazê-las também se moverem para o mesmo ponto da figura agregadora;
4. Na figura a ser agregada, adicionou-se uma referencia à figura agregadora;
5. Na figura a ser agregada, sobrescreveu-se o método *moveBy* a fim de restringir a movimentação dentro dos limites da figura agregadora. O acionamento do mecanismo de restrição de movimentação se dá quando existe uma determinada figura em sua totalidade em cima de outra. No caso de um pacote, o mecanismo é acionado quando uma classe está em sua totalidade acima do pacote (especificamente, acima do retângulo maior pertencente ao pacote). Neste momento, além de se aplicarem as restrições de movimentação, realizam-se também as associações semânticas entre os conceitos e o conceito pacote passa a agregar a instância do conceito de classe que está acima dele. Além disso, a figura agregada ganha uma referência da figura que a agregou. Enquanto esta referência é nula, a movimentação ocorre sem restrições.

Estes passos são a base do algoritmo utilizado nas agregações de figuras deste trabalho. Convém destacar que o algoritmo pôde ser aplicado em figuras pertencentes aos modelos do diagrama de pacotes, diagrama de componentes, diagrama de implantação, diagrama de estrutura composta, diagrama de máquina de estados e diagrama de atividades. Essencialmente, foi criado para o diagrama de estrutura composta, onde muitos dos elementos sintáticos agregam

outros elementos sintáticos, como os classificadores estruturados e colaborações que serão detalhados mais adiante. Entretanto, outros diagramas também puderam utilizá-lo. Exemplos de diagramas que apresentam esta característica podem ser vistos nas seções de “exemplo de uso”.

Exemplo de uso

Um exemplo de diagrama de pacotes possível de ser modelado na nova versão do ambiente SEA pode ser observado na imagem 7.7. Nela é possível conferir a utilização dos novos relacionamentos entre pacotes deste diagrama, como o acesso, importação e fusão. O exemplo também mostra o uso de agregação de classes em pacotes. A fim de que um pacote englobe uma classe, é necessário que o usuário crie uma classe externa e jogue-a por completo dentro do pacote. O algoritmo que realiza esta agregação seta a classe englobada na lista de conceitos “Classe” que um pacote contém. Além disso, a figura da classe fica restrita a se mover apenas no interior do pacote e uma vez que o pacotes se move, esta assim também o faz.

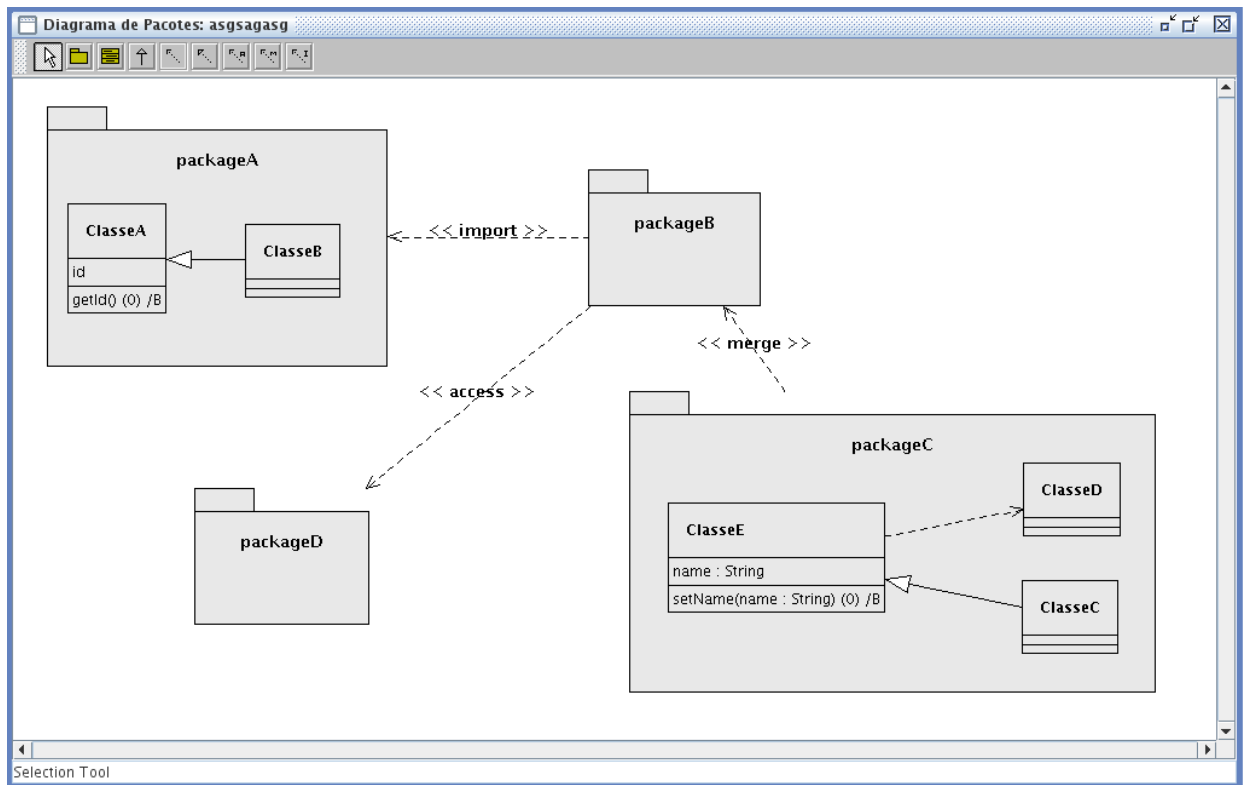


Figura 7.7: Exemplo de diagrama de pacotes construído no ambiente SEA.

Caso o usuário desejar alterar o nome do pacote, basta clicar com o botão direito em cima da figura e solicitar a edição. Uma caixa de diálogo será apresentada para que se possa alterar o nome anterior e após a confirmação a edição será imediata.

7.3.3 Diagrama de Componentes

O diagrama de componentes descreve os componentes de software e suas dependências entre si. Pode ser utilizado para modelar os componentes do código-fonte e do código executável do software; para destacar a função de cada módulo para facilitar a sua reutilização; e para auxiliar no processo de engenharia reversa, por meio da organização dos módulos do sistema e seus relacionamentos.

Em UML 2, a figura do elemento sintático “componente” foi ligeiramente alterada e o conceito de “porto” foi introduzido. As classes criadas e modificadas para a implementação do diagrama de componentes podem ser visualizadas na imagem 7.8. Pode-se observar também que alguns conceitos e figuras do modelo anterior, o diagrama de pacotes, puderam ser reutilizados.

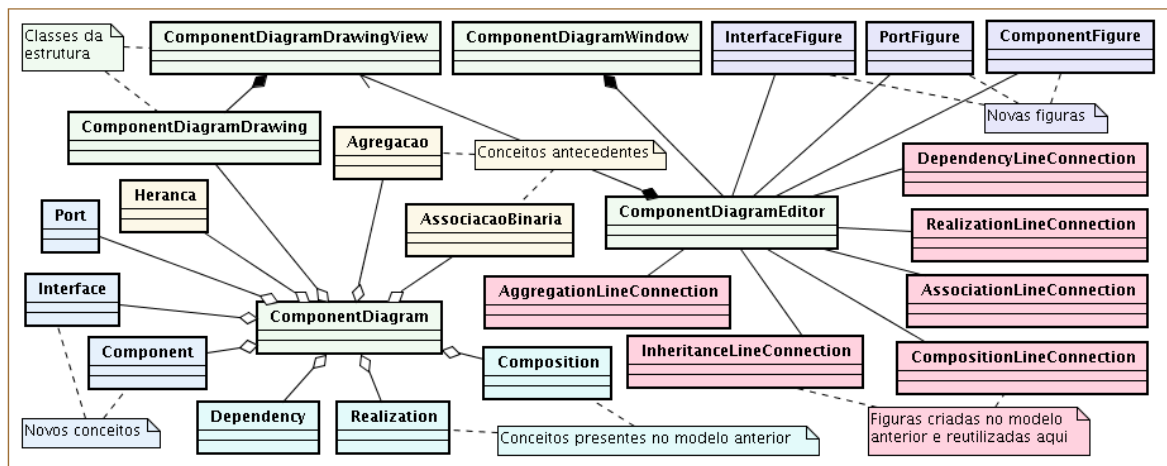


Figura 7.8: Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de componentes.

Conceitos antecedentes

O diagrama de componentes pôde utilizar alguns conceitos presentes na versão anterior do ambiente SEA e outros implementados no diagrama de pacotes. Estes conceitos reutilizados são indicados a seguir:

- *Association*: Este conceito é representado pela classe “AssociacaoBinaria”.
- *Aggregation*: Este conceito é representado pela classe “Agregacao”.
- *Composition*: Este conceito é representado pela classe “Composition”.

- *Class*: Este conceito é representado pela classe nomeada “Classe”.
- *Generalization*: Este conceito é representado pela classe “Heranca”.
- *Dependency*: Este conceito é representado pela classe “*Dependency*”.
- *Realization*: Este conceito é representado pela classe “*Realization*”.

Conceitos novos

Três conceitos foram criados na implementação do diagrama de componentes. São apresentados abaixo:

- *Component*: Um componente é definido como “uma unidade de composição com interfaces contratualmente especificadas e dependências de contexto explícitas”, sendo que componentes podem ser duplicados e são sujeitos a participar de composições com terceiros (SILVA, 2000 apud SZYPERSKI, 1997). Este conceito é representado no modelo de diagrama de componentes do ambiente SEA pela classe “*Component*”.
- *Interface*: Uma interface é a fronteira que define a forma de comunicação entre duas entidades. Ela pode ser entendida como uma abstração que estabelece a forma de interação da entidade com o mundo exterior, através da separação dos métodos de comunicação externa dos detalhes internos da operação, permitindo que esta entidade seja modificada sem afetar as entidades externas que interagem com ela. Componentes de software utilizam interfaces padronizadas para criar uma camada de abstração que facilite a reutilização e a manutenção do software (AYOAMA, 2002). O conceito de interface é representado no modelo de diagrama de componentes do ambiente SEA pela classe “*Interface*”.
- *Port*: Um porto representa a fronteira entre uma classe, componente ou pacote e seu meio externo. Corresponde a um ponto de conexão entre o elemento que o possui e seu exterior, bem como uma ligação entre a fronteira e sua estrutura interna (SILVA, 2007). Este conceito é representado no modelo de diagrama de componentes do ambiente SEA pela classe “*Port*”.

Figuras novas

Foi necessário criar apenas três figuras para este diagrama, todas as outras puderam ser reutilizadas. A primeira figura criada foi a que representa o conceito de componente. Esta figura, denominada “*ComponentFigure*”, teve a implementação demasiadamente árdua pois apresenta

uma pequena imagem localizada no topo de sua direita, representando um micro-componente, com a notação de UML 1. Sendo assim, era preciso de alguma classe do framework JHotDraw que manipulasse linhas a fim de desenhar esta pequena imagem. Após muitas investigações no código-fonte do framework JHotDraw, encontrou-se a classe “*PolyLineFigure*”. Com ela foi possível desenhar o ícone linha a linha, de maneira que ele ficasse flutuando sempre à direita e ao topo da figura do componente, independente do tamanho do conteúdo do componente. Através do método *addPoint(x, y)* desta classe conseguiu-se desenhar os mini-retângulos um acima do outro sem que estes se interpolassem entre si. Além deste ícone, a figura de componente agrega em seu interior uma figura de texto que representa o estereótipo e outra que representa o nome do componente.

A segunda figura implementada no diagrama de componentes foi a “*InterfaceFigure*” que representa o conceito de “interface”. Ela é composta por uma elipse com tamanho fixo e por uma figura de texto que guarda o nome da interface. Foi necessário deixar a figura de fundo (representada pela classe “*GraphicalCompositeFigure*” do framework JHotDraw) transparente, pois quando a interface é agregada a outros conceitos esta deve ficar com a cor de fundo destes.

Por fim, implementou-se a figura que representa o conceito de porto. Esta figura teve a codificação trivial se comparada às demais. Apenas criou-se uma “*GraphicalCompositeFigure*” vazia com poucos pixels de margem, utilizando o método *setInsets()*. Após construída, implementou-se a funcionalidade de um porto poder ser agregado a um componente, utilizando-se um algoritmo parecido com a agregação de classes em pacotes.

Exemplo de uso

Na imagem 7.9 pode-se observar um exemplo de uso do diagrama de componentes implementado no ambiente SEA. O exemplo se refere a um sistema de reservas que necessita interação entre vários componentes de software.

A edição de componentes e interfaces se dá através do clique do botão direito nestas figuras. Em um componente, é possível editar o nome e o estereótipo. Já na interface é possível alterar apenas o nome.

Componentes também suportam a agregação de portos. Para agregar um porto, basta que o usuário solte um porto já criado em qualquer aresta do componente. Este irá agregar o porto automaticamente. Quando o componente se movimentar, o porto também se movimentará. É possível mudar a posição do porto arrastando-o através das arestas do componente.

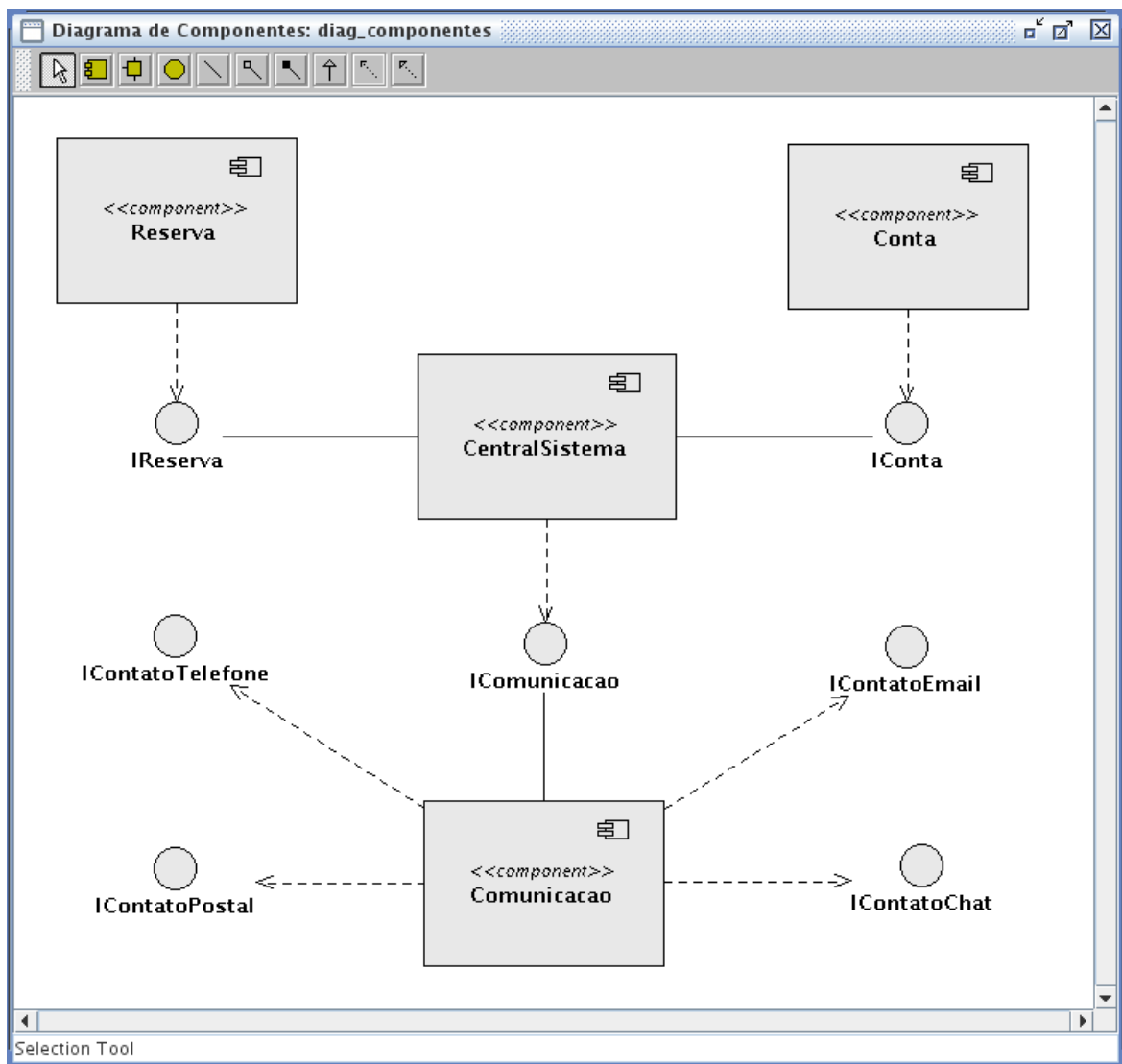


Figura 7.9: Exemplo de diagrama de componentes construído no ambiente SEA.

7.3.4 Diagrama de Implantação

Os diagramas de implantação são diagramas que mostram a configuração de nós de processamento em tempo de execução e os componentes que neles existem. São importantes para visualizar, especificar e documentar sistemas embutidos, cliente/servidor, distribuídos e para gerenciar sistemas por engenharia de produção e engenharia reversa.

Este tipo de diagrama é empregado para a modelagem da visão estática de implantação de um sistema. Essa visão direciona primariamente a distribuição, entrega e instalação das partes que formam o sistema físico. Na maior parte, isso envolve a modelagem da topologia do hardware em que o sistema é executado.

O diagrama de implantação foi introduzido na segunda versão de UML. As classes produ-

zidas para a implementação deste modelo podem ser visualizadas na figura 7.10. É importante salientar que todos os conceitos e figuras do diagrama de componentes puderam ser reusados neste diagrama, entretanto, não foram colocados explicitamente no diagrama da imagem 7.10 a fim de simplificá-lo e torná-lo mais legível.

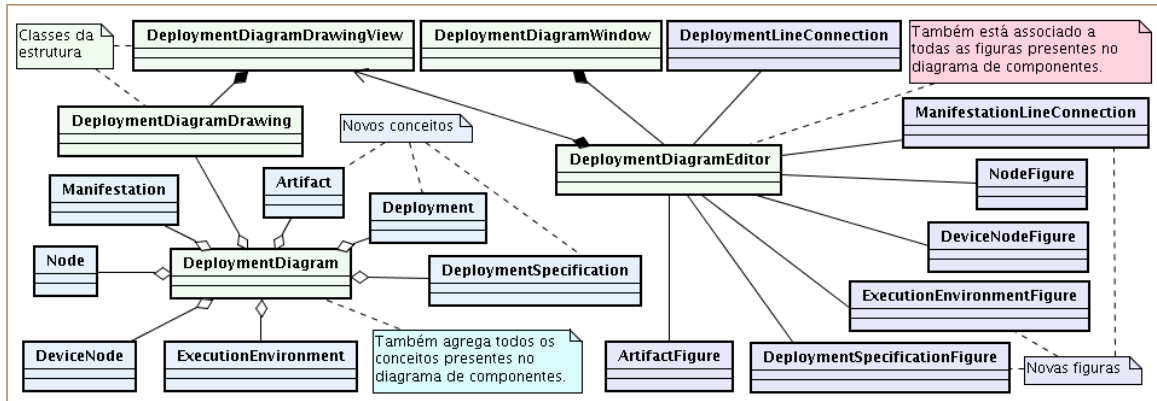


Figura 7.10: Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de implantação.

Conceitos antecedentes

O diagrama de implantação também pôde utilizar alguns conceitos presentes na versão anterior do ambiente SEA e todos os conceitos pertencentes ao diagrama de componentes, conforme comentado anteriormente. Todos os conceitos reusados neste diagrama são apontados a seguir:

- *Association*: Este conceito é representado pela classe “AssociacaoBinaria”.
- *Aggregation*: Este conceito é representado pela classe “Agregacao”.
- *Composition*: Este conceito é representado pela classe “Composition”.
- *Generalization*: Este conceito é representado pela classe “Heranca”.
- *Dependency*: Este conceito é representado pela classe “Dependency”.
- *Realization*: Este conceito é representado pela classe “Realization”.
- *Component*: Este conceito é representado pela classe “Component”.
- *Interface*: Este conceito é representado pela classe “Interface”.
- *Port*: Este conceito é representado pela classe “Port”.

Conceitos novos

Tratando-se de um diagrama inédito em UML 2, novos conceitos foram concebidos. Estes conceitos são listados e detalhados abaixo, de acordo com a superestrutura da OMG (OMG, 2007b).

- *Node*: Um nodo representa um recurso computacional, que pode ser de software ou de hardware, que é parte de um sistema computacional. Este conceito é representado neste modelo pela classe “*Node*”.
- *Device node*: É uma especialização do conceito “*Node*”. Este conceito é representado neste modelo pela classe “*DeviceNode*”.
- *Execution environment*: É uma especialização do conceito “*Node*”. Este conceito é representado neste modelo pela classe “*ExecutionEnvironment*”.
- *Deployment specification*: Uma especificação de implantação corresponde a um artefato que define como outro artefato (ou conjunto de artefatos) é implantado em um nodo, especificando para uma situação específica de uso a configuração deste artefato. Este conceito é representado neste modelo pela classe “*DeploymentSpecification*”.
- *Artifact*: Um artefato representa um elemento físico que corresponde a uma fração de informação de um sistema computacional, que pode ser implantado em um nodo. Pode referir-se, por exemplo, a um arquivo de configuração, a um arquivo executável, a um documento, como um email. Este conceito é representado neste modelo pela classe “*Artifact*”.
- *Manifestation*: A manifestação é uma relação que envolve um artefato e um elemento de modelagem (ou conjunto de elementos) e estabelece que o artefato corresponde a uma implementação correta deste elemento. Este conceito é representado neste modelo pela classe “*Manifestation*”.
- *Deployment*: A implantação de um componente ou artefato em um nodo pode ser representada com estes elementos dentro do nodo ou através de uma relação de implantação. Esta relação é representada neste modelo pela classe “*Deployment*”.

Figuras novas

Ao todo foram sete novas figuras implementadas neste diagrama. Sabendo que o JHotDraw não dá suporte a imagens 3D, a figura que mais custou a ser desenvolvida foi a “*NodeFigure*”,

que representa o conceito “*Node*” através de um cubo. O desafio era conseguir que uma “*RectangleFigure*” pudesse obter bordas tridimensionais, característica que não estava presente por padrão no framework. Dado o problema, depois de diversas tentativas mal sucedidas, criou-se uma nova classe, “*CubeFigure*”, que estende “*RectangleFigure*” e sobrescreve o método *drawBackground*. Externamente ao retângulo principal, este novo método desenha linha por linha outros dois retângulos inclinados, liga-os e preenche-os da mesma cor que o retângulo principal, formando assim um cubo. Desta forma, bastava que a classe “*NodeFigure*” agregasse esta figura de maneira que esta fosse sua figura de representação, isto é, sua figura de fundo padrão. Após isto, foi necessário inserir uma figura de texto a fim de representar o nome do nodo. As figuras “*DeviceNodeFigure*” e “*ExecutionEnvironmentFigure*”, que representam os conceitos “*DeviceNode*” e “*ExecutionEnvironment*” respectivamente, foram implementadas se baseando na figura “*NodeFigure*”. Entretanto, elas contém estereótipos e por isso agregam uma figura de texto a mais.

A figura “*ArtifactFigure*”, que representa o conceito “*Artifact*”, foi baseada na “*ComponentFigure*”, uma vez que também necessita de uma pequena imagem no topo da direita da imagem principal, que parece uma folha de papel com a ponta direita dobrada, como se fosse uma “orelha” de caderno. Foi necessário utilizar a figura “*PolyLineFigure*” a fim de desenhar o ícone linha a linha, assim como foi feito na “*ComponentFigure*”. A figura também necessita agregar duas figuras de texto: uma que representa o estereótipo do artefato e outra que representa o nome do artefato.

A fim de representar o conceito de “*Deployment Specification*”, criou-se a figura “*DeploymentSpecificationFigure*”. Esta figura é composta apenas de duas figuras de texto internas: uma figura que representa o estereótipo e outra que representa o nome da especificação de implantação.

Também implementou-se as figuras que representam os dois relacionamentos específicos do diagrama de implantação: “*ManifestationLineConnection*”, que representa o conceito “*Manifestation*”, e “*DeploymentLineConnection*”, que representa o conceito “*Deployment*”. Ambas as classes puderam herdar as propriedades e as operações da classe “*DependencyLineConnection*”, já que visualmente e semanticamente são semelhantes. Foi suficiente sobrescrever o método *drawLine* de forma a inserir ao meio da linha tracejada o estereótipo <<*manifestation*>> ou <<*deploy*>>, que varia de acordo com figura sendo criada.

Por fim, adicionou-se aos nodos a funcionalidade de agregação de componentes, artefatos ou especificações de implantação. O algoritmo utilizado foi similar ao algoritmo de agregação de classes em pacotes.

Exemplo de uso

Pode-se visualizar um exemplo de utilização do diagrama de implantação construído no ambiente SEA na imagem 7.11. Neste exemplo, inspirado em obra de SILVA, o arquivo *GUIconfig.xml* conteria informações referentes à configuração da interface com o usuário do jogo da velha. Com isso, um jogador poderia dispor uma interface com aparência diferentes da do outro jogador.

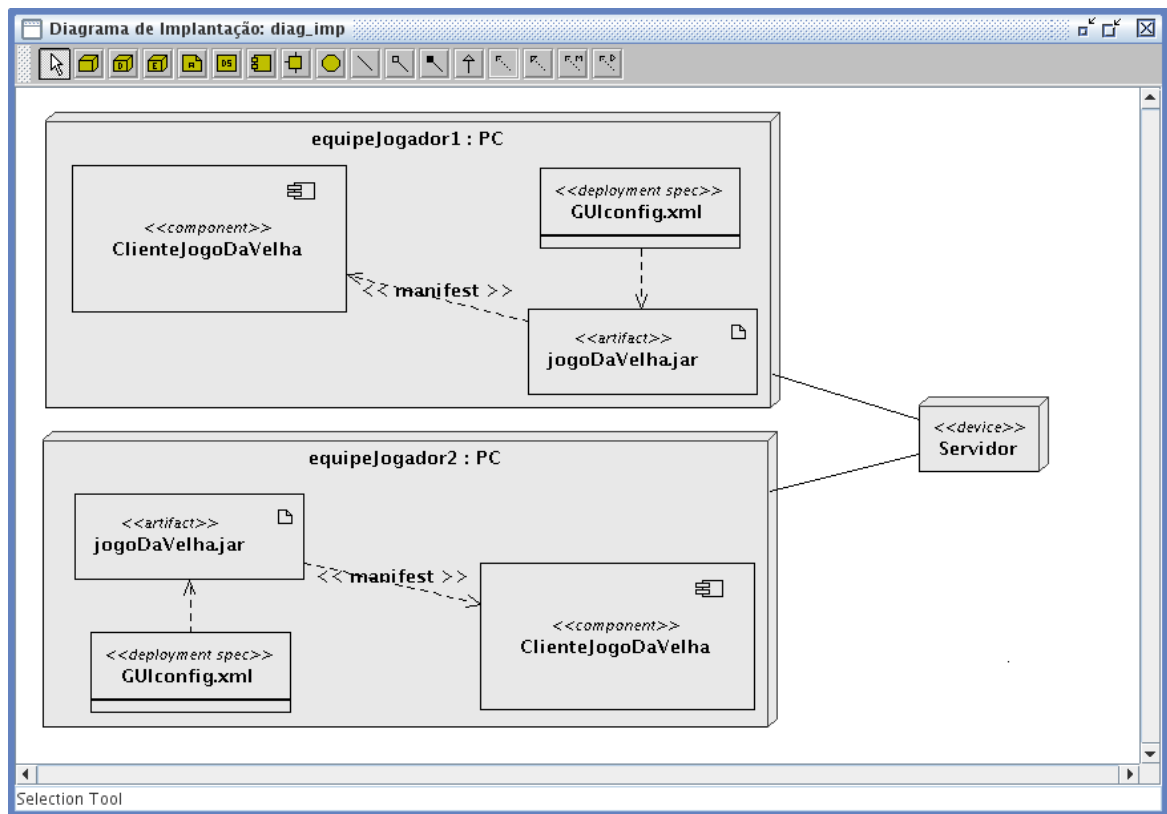


Figura 7.11: Exemplo de diagrama de implantação construído no ambiente SEA.

7.3.5 Diagrama de Comunicação

O diagrama de comunicação era conhecido como diagrama de colaboração até a primeira versão UML, tendo seu nome modificado para diagrama de comunicação a partir da versão 2.0. Esse diagrama está amplamente associado ao diagrama de seqüência, uma vez que se complementam.

As informações mostradas no diagrama de comunicação são, com freqüência, praticamente as mesmas apresentadas no diagrama de seqüência, porém com um enfoque diferente, visto que este diagrama não se preocupa com a temporalidade do processo, concentrando-se em como os

objetos estão vinculados e quais mensagens trocam entre si durante o processo.

As classes concebidas e reusadas para a implementação do modelo de diagrama de comunicação podem ser vistas na figura 7.12.

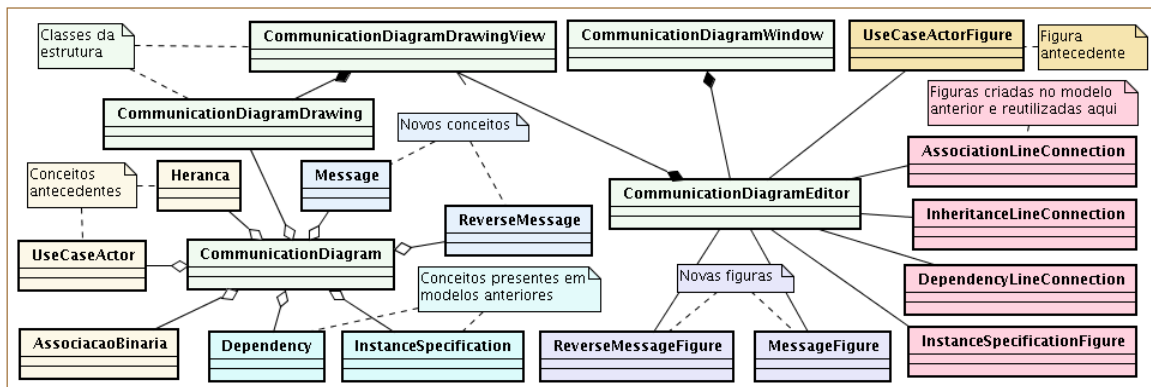


Figura 7.12: Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de comunicação.

Conceitos antecedentes

Foi possível reutilizar alguns conceitos presentes na versão anterior do ambiente SEA e outros conceitos pertencentes a diagramas anteriores. Estes conceitos são apontados a seguir:

- *Association*: Este conceito é representado pela classe “AssociaçãoBinaria”.
- *Generalization*: Este conceito é representado pela classe “Heranca”.
- *Actor*: Um ator representa um conjunto coerente de papéis que os usuários de casos de uso desempenham quando interagem com esses casos de uso. Tipicamente, um ator representa um papel que um ser humano, um dispositivo de hardware ou até outro sistema desempenha com o sistema. Este conceito é representado pela classe “UseCaseActor”.
- *Dependency*: Este conceito é representado pela classe “Dependency”.
- *Instance specification*: Este conceito é representado pela classe “InstanceSpecification”.

Conceitos novos

Apenas dois conceitos foram criados do zero para o diagrama de comunicação. Seguem descrições detalhadas na seqüência:

- *Message*: Uma mensagem é um elemento que representa a interação entre elementos do programa em tempo de execução. É voltada à invocação de um método ou envio de sinal. Este conceito é representado pela classe “*Message*”.
- *Reverse message*: Uma mensagem invertida tem a mesma semântica que uma mensagem típica apenas se apresenta no sentido oposto. Este conceito é representado pela classe “*ReverseMessage*”.

Figuras novas

Muitas figuras puderam ser reusadas de outros diagramas. Assim bastou apenas realizar algumas modificações nas figuras existentes e criar a “*MessageFigure*” e a “*ReverseMessageFigure*”. A figura “*UseCaseActorFigure*”, criada na versão anterior do ambiente SEA, precisou ter sua figura de texto interna centralizada em relação ao corpo do desenho do ator, uma vez que anteriormente estava alinhada à esquerda.

Uma mensagem pode ser criada entre duas instâncias desde que haja uma ligação entre elas. A figura que representa o conceito de mensagem é a “*MessageFigure*”. Ela é composta por uma figura de texto (representada pela classe *TextFigure* do framework JHotDraw) com uma referência para o número da mensagem e sua descrição, que é o nome do método propriamente dito, e uma seta que aponta por padrão para a direita. Uma figura de mensagem invertida se comporta da mesma maneira que uma figura de mensagem típica e apresenta as mesmas características, por isto a “*ReverseMessageFigure*” estende “*MessageFigure*”. A única diferença visual entre elas é que a figura de mensagem invertida tem a seta apontando para o lado oposto ao da figura de mensagem.

Exemplo de uso

O diagrama de comunicação exemplificado na imagem 7.13 procura indicar as mensagens trocadas entre instâncias das classes “Temporizador”, “BombaDeAgua” e “RepositorioDeAgua”. O diagrama mostra a ordem das mensagens usando numeração.

Caso o usuário desejar editar uma mensagem, basta selecionar a figura e clicar com o botão direito. Um *popup* será apresentado ao usuário com as ações possíveis de serem realizadas e uma delas será “Alterar a mensagem”. Basta informar um novo método a ser utilizado ou editar o anterior e confirmar a ação. Também é possível girar a seta de mensagem para torná-la inclinada de acordo com a necessidade, uma vez que nem sempre teremos ligações em linha reta - vide o exemplo.

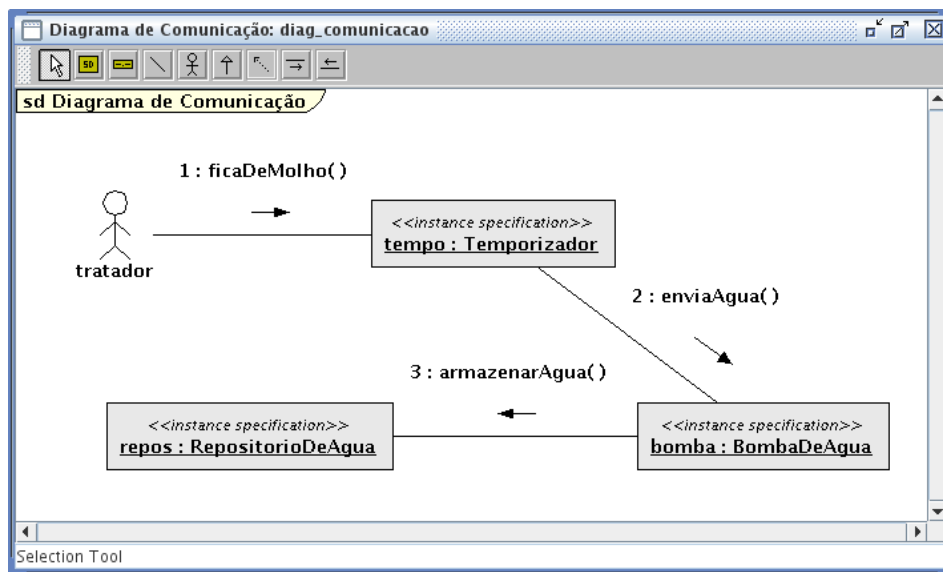


Figura 7.13: Exemplo de diagrama de comunicação construído no ambiente SEA.

7.3.6 Diagrama de Casos de Uso

O diagrama de casos de uso tem o objetivo de auxiliar a comunicação entre os analistas e o cliente. Descreve um cenário que mostra as funcionalidades do sistema do ponto de vista do usuário. O cliente deve ver no diagrama de casos de uso as principais funcionalidades de seu sistema.

O modelo de diagrama de casos de uso já existia na versão do ambiente SEA anterior a este trabalho. Entretanto, não continha alguns conceitos importantes e muitas de suas figuras estavam mal elaboradas. O principal objetivo era tornar o diagrama disponível para uso assim como os outros, com as funcionalidades básicas construídas.

Muitas classes necessárias para a implementação deste diagrama já estavam criadas, então bastou reusá-las e conceber os novos conceitos. Na imagem 7.14, pode-se visualizar os conceitos relacionados a este diagrama e as figuras que os representam.

Conceitos antecedentes

Todos os conceitos listados a seguir já estavam presentes no ambiente SEA antes da construção da extensão para suporte à UML 2. Contudo, vale ressaltar que o conceito de “*Generalization*”, mais comumente chamado de “Herança”, não havia sido incluído ainda no modelo “*UseCaseDiagram*”. Assim, foi suficiente adicionar a herança na lista de conceitos do modelo e inserir um novo ícone na paleta de ferramentas do diagrama, a fim de possibilitar que o usuário

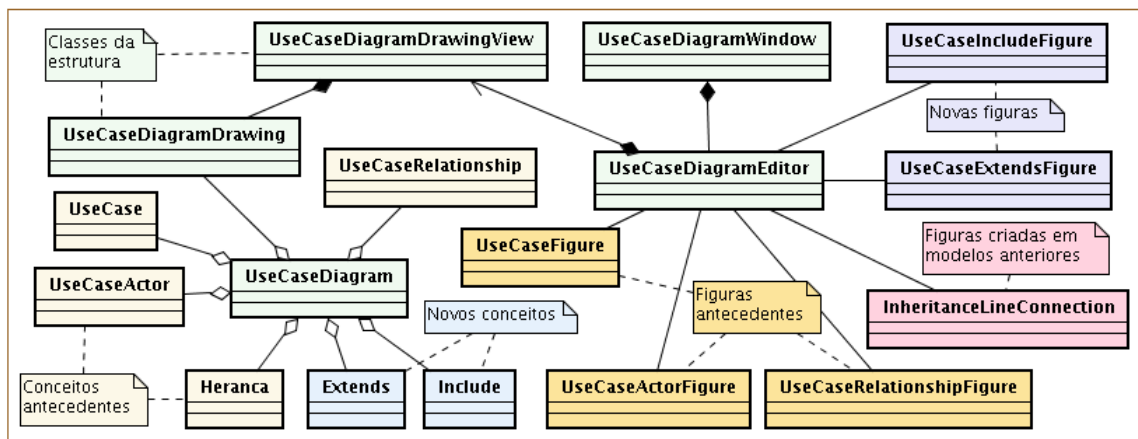


Figura 7.14: Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de casos de uso.

também pudesse servir-se desta característica.

- *Use case*: Um caso de uso é uma funcionalidade atômica de um sistema, subsistema ou classe. Representa apenas a identificação de uma funcionalidade, sem qualquer referência a como ela é executada. Este conceito é representado neste modelo do ambiente SEA pela classe “*UseCase*”.
- *Actor*: Conceito já reusado no diagrama de comunicação, foi concebido inicialmente para o modelo do diagrama de casos de uso. É através do ator que o software recebe interação. Este conceito é representado neste modelo do ambiente SEA pela classe “*UseCaseActor*”.
- *Relationship*: Este conceito representa o relacionamento entre casos de uso e atores, tratando-se de um tipo de associação específica deste modelo. É representado pela classe “*UseCaseRelationship*”.

Conceitos novos

Apenas dois conceitos foram fundados para este modelo neste trabalho. Estes conceitos são apontados detalhadamente na seqüência:

- *Generalization*: A herança já foi tratada em outros diagramas deste trabalho, porém foi inicialmente instituída para o diagrama de classes. Este modelo criou um novo conceito para a generalização, específico para o diagrama de casos de uso, que herda do conceito de herança tratado anteriormente. É representado pela classe “*UseCaseGeneralization*”.

- *Include*: A inclusão estabelece que parte do comportamento inerente a um caso de uso está definida em outro caso de uso, isto é, que um caso de uso contém o comportamento definido em outro caso de uso. Este conceito é representado neste modelo pela classe “*Include*”.
- *Extends*: A extensão estabelece uma relação em que um dos casos de uso tem seu comportamento estendido através do comportamento definido em outro caso de uso. Este conceito é representado neste modelo pela classe “*Extends*”.

Figuras novas

As figuras “*UseCaseIncludeFigure*” e “*UseCaseExtendsFigure*”, que representam graficamente os conceitos “*Include*” e “*Extends*” respectivamente, puderam estender diretamente a figura “*DependencyLineConnection*”, visto que se tratam de dependências específicas do diagrama de casos de uso. São compostas por linhas tracejadas com um determinado estereótipo ao meio: no caso da figura “*UseCaseIncludeFigure*”, o estereótipo é “<< include >>” e há uma seta aberta ao fim da linha; já quando a figura é a “*UseCaseExtendsFigure*”, o estereótipo é “<< extends >>” e há uma seta aberta ao início da linha.

A figura “*UseCaseFigure*”, presente na versão anterior, foi modificada pois estava com uma cor fora do padrão das demais e a elipse que a envolve não trazia espaçamentos entre a borda e o texto interno. Na versão atual, colocou-se as margens e a cor padrão, centralizando também o texto contido no caso de uso.

Assim como o conceito de herança já estava presente no ambiente SEA, sua figura correspondente também havia sido criada, a “*InheritanceLineConnection*”. Foi necessário incluí-la no editor juntamente com as outras figuras deste modelo. Também notou-se que a seta da figura estava ligeiramente arredonda em sua base e isto foi reparado alterando a angulação da seta.

Observa-se que ao criar as novas figuras, procurou-se seguir o padrão de nomenclatura já existente. Diferente dos outros modelos, o diagrama de casos de uso tem todas as suas figuras iniciadas por “*UseCase*”. Assim, as novas figuras foram nomeadas seguindo este padrão.

Exemplo de uso

Pode-se visualizar um exemplo de utilização do diagrama de casos de uso criado no ambiente SEA através da imagem 7.15. O exemplo envolve o uso de atores, casos de uso e seus relacionamentos. O diagrama apresenta possíveis funcionalidades de um sistema de clínica médica, necessitando-se de mais de um ator para descrever o cenário.

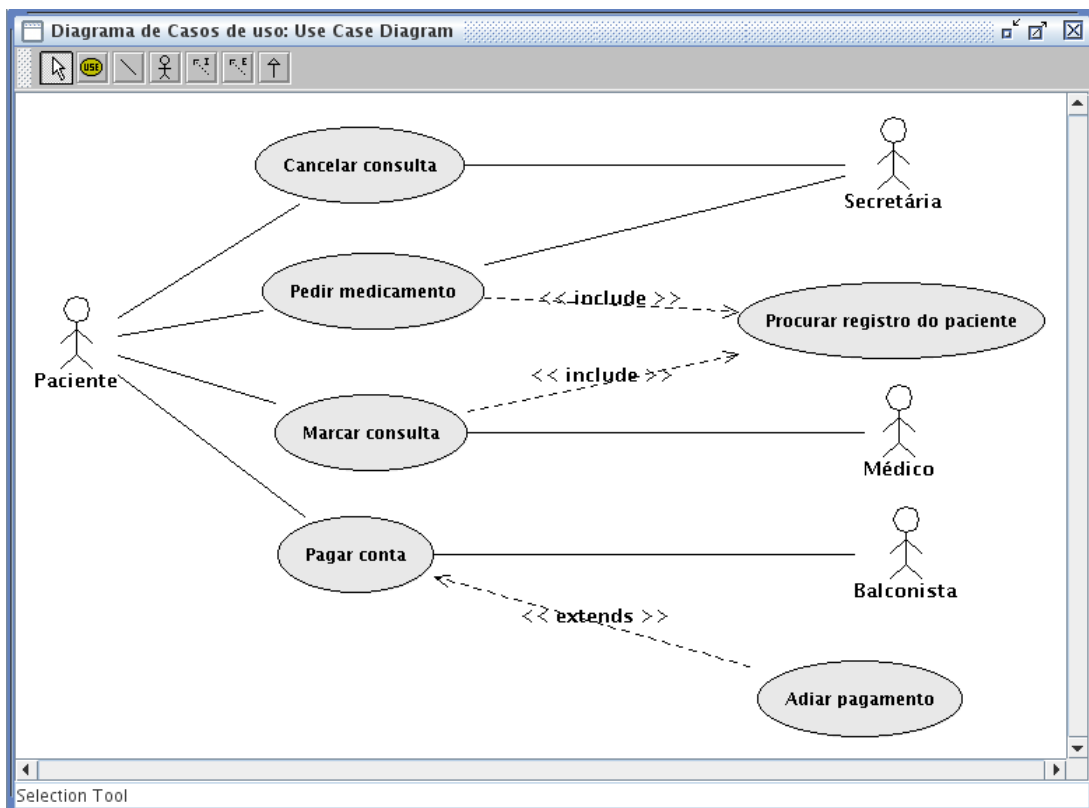


Figura 7.15: Exemplo de diagrama de casos de uso construído no ambiente SEA.

7.3.7 Diagrama de Seqüência

O diagrama de seqüência permite modelar os processos que ocorrem no sistema através da troca de mensagens entre os objetos do sistema. Este diagrama já estava presente na primeira versão de UML e, na segunda versão, foi introduzida a noção de agrupamento de mensagens através do elemento sintático “fragmento combinado”, melhorando a representação de envio de mensagens com o uso de condições e repetições.

O diagrama de seqüência já estava construído antes deste trabalho, entretanto, algumas modificações precisaram ser feitas pois além de ser necessário a inclusão do conceito de “fragmento combinado”, o modelo estava com alguns erros. Na imagem 7.16, pode-se visualizar o conjunto de classes utilizadas para a criação do modelo de diagrama de seqüência no ambiente SEA.

Conceitos antecedentes

Conforme comentado anteriormente, o diagrama de seqüência é um modelo que já pertencia ao ambiente SEA antes da construção deste trabalho. Sendo assim, muitos conceitos já

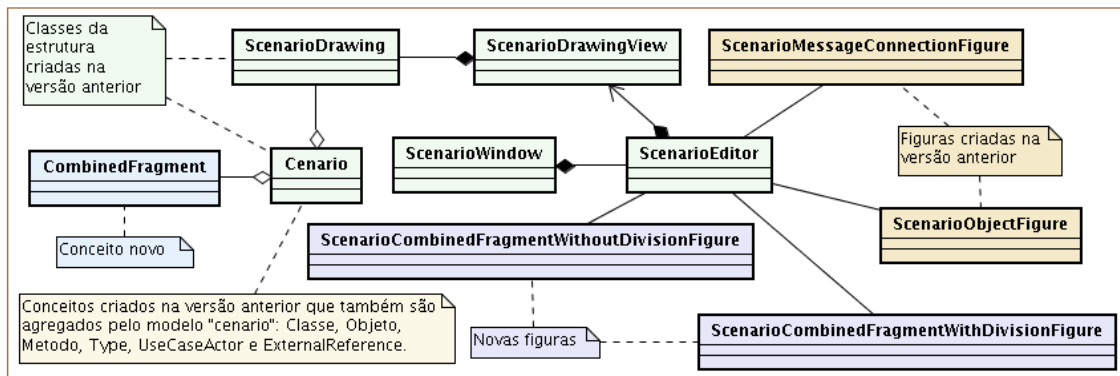


Figura 7.16: Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de seqüência.

estavam presentes. Observe a seguir a descrição de cada um deles. Vale ressaltar que a maioria destes foi desenvolvido e nomeado em português, diferente da forma como se procurou fazer na extensão, que está em inglês.

- *Class*: Este conceito é representado pela classe “Classe”.
- *Object*: Um objeto é uma instância de uma classe. Foi criado mais especificamente para o diagrama de seqüência, onde um objeto, além do nome e da classe, agrega uma linha de vida. É representado neste modelo pela classe “Objeto”.
- *Method*: Um método define uma habilidade de um objeto. O conceito de método também é utilizado pelo diagrama de classes. É representado pela classe “Metodo”.
- *Actor*: Este conceito é representado pela classe “UseCaseActor”.
- *Message*: Uma mensagem é uma chamada a um objeto para invocar um de seus métodos, ativando um comportamento descrito por sua classe. Neste modelo este conceito é representado pela classe “Message”.

Conceitos novos

A principal modificação do diagrama de seqüência na segunda versão de UML foi a inclusão do elemento sintático “fragmento combinado”. O fragmento combinado é uma espécie de invólucro voltado a conter uma fração das mensagens do diagrama em que ele se encontra e a dar uma característica semântica adicional a estas mensagens. Segundo a superestrutura da OMG (OMG, 2007b), a UML 2 apresenta doze tipos de operadores de fragmento combinado. São eles:

- *Opção* [opt]: Aplicado a um fragmento combinado composto por um único conjunto de mensagens. Estabelece uma condição para que essas mensagens sejam efetivamente enviadas.
- *Alternativa* [alt]: Aplicado a um fragmento combinado composto por mais de um conjunto de mensagens. Permite estabelecer escolhas, sendo que cada operando está associado a uma condição.
- *Laço* [loop]: Apresenta um único operando e estabelece que o envio do conjunto de mensagens será repetido certo número de vezes.
- *Quebra* [break]: Apresenta um único operando e estabelece uma quebra do processamento em curso.
- *Região crítica* [critical]: Define uma seqüência de mensagens que, uma vez iniciada, não pode ser interrompida - nem pela ação de outro operador.
- *Paralelo* [par]: Define um conjunto de seqüências de mensagens que podem ocorrer em paralelo.
- *Seqüenciamento fraco* [seq]: Permite inserir não determinismo em um diagrama de seqüência, definindo as frações de interação cuja ordem é indefinida.
- *Seqüenciamento estrito* [strict]: Estabelece uma seqüência de mensagens como válida.
- *Negativo* [neg]: Estabelece uma seqüência de mensagens como inválida.
- *Asserção* [assert]: Estabelece uma seqüência de mensagens como válida, e, adicionalmente, qualquer outra seqüência como inválida.
- *Considerar* [consider]: Declara uma mensagem ou uma coleção de mensagens como válida, ou seja, que podem ocorrer e que são significativas na interação modelada.
- *Ignorar* [ignore]: Declara uma mensagem ou uma coleção de mensagens como inválida, ou seja, que podem ocorrer mas não são significativas na interação modelada.

Na extensão desenvolvida para fornecer suporte à UML 2, criou-se apenas um único conceito para o fragmento combinado, denominado “*CombinedFragment*”. Este conceito apresenta uma característica chamada “*operatorType*”, que pode variar de acordo com a escolha do usuário dentre os doze tipo de operadores.

Figuras novas

As primeiras figuras desenvolvidas para o modelo de diagrama de seqüência foram as representações do conceito de fragmento combinado. Foi necessário a criação de duas, já que a representação de um fragmento combinado divide-se em duas categorias distintas de tipos de operadores: operadores que possuem apenas um operando, isto é, apenas um conjunto de mensagens, sem necessitar de divisão; e operadores que possuem mais de um operando, isto é, mais de um conjunto de mensagens, podendo necessitar de mais de uma divisão. Para cada categoria foi criada uma classe em Java diferente, uma vez que semanticamente e visualmente são distintas. Elas foram denominadas por “*ScenarioCombinedFragmentWithDivisionFigure*” e “*ScenarioCombinedFragmentWithoutDivisionFigure*”. Assim, dependendo do tipo do operador escolhido pelo usuário, tem-se criado um fragmento combinado com ou sem divisão. Além disso, um fragmento combinado também pode trazer operandos com condições. Contudo, esta característica está mais uma vez atrelada à escolha de operador feita pelo usuário. Por exemplo, se o usuário escolher por um operador *opt*, a figura não trará opções para adicionar divisão, somente para editar a condição. Entretanto, se o usuário optar por um operador *seq*, ele poderá adicionar novas divisões mas não poderá incluir condições.

Após a inclusão do conceito e da figura referente ao elemento sintático “fragmento combinado”, percebeu-se que o modelo continha alguns problemas. O algoritmo de ordenação e seqüenciamento de mensagens estava bastante robusto, mas o modelo pecava esteticamente e algumas funcionalidades estavam incompletas. A parte da codificação de inclusão e edição de mensagem não estava concluída, sendo impossível adicionar um método em uma mensagem após esta ser criada. Em seguida à correção deste problema, corrigiu-se o posicionamento da elipse pertencente à mensagem inicial, visto que a elipse se iniciava embaixo da seta de criação de mensagem. O certo seria que o início da mensagem se encaixasse ao núcleo da elipse. Pode-se observar este problema na figura 6.2 do capítulo anterior e verificar a modificação no diagrama de exemplo da próxima seção.

A figura que representa um objeto no diagrama de seqüência, a “*ScenarioObjectFigure*”, também não estava visualmente apropriada, uma vez que estava solta acima da linha de vida do objeto. Faltava um retângulo que a envolvesse até o início da linha de vida. Desta maneira, adicionou-se uma instância de “*BorderDecorator*” na classe *drawing* do modelo, invocando o método *decorate* e passando a figura de texto que contém o nome do objeto como parâmetro. Também coloriu-se o fundo deste objeto com cinza, da mesma forma como as outras figuras implementadas em outros modelos deste trabalho. Além disso, tirou-se a cor preta da linha de vida do objeto e inseriu-se a cor branca.

Vale ressaltar que todos os ícones da barra de ferramentas do diagrama de seqüência foram alterados a fim de torná-los mais intuitivos ao usuário.

Exemplo de uso

Levando em consideração que a implementação de uma ferramenta de criação para cada um dos doze tipos de fragmento combinado iria “poluir” excessivamente a paleta de ferramentas do diagrama de seqüência, optou-se por criar apenas duas ferramentas: uma que serve para criar fragmentos combinados com operadores que possibilitam divisões e outra é usada para criar os tipos de fragmentos combinados que não permitem mais de um operando, isto é, mais de uma divisão. Cada ferramenta cria um dos tipos de figuras, que podem ser com ou sem divisão. O meio utilizado para que o usuário escolha o tipo de operador, após ter escolhido o tipo de fragmento combinado, foi a criação de duas janelas que oferecem a listagem de opções de operadores. Na imagem 7.17 pode-se observar os dois tipos de janelas: a janela da esquerda apresenta ao usuário todos os tipos de operadores sem divisão e a da direita mostra todos os tipos de operadores com divisão.

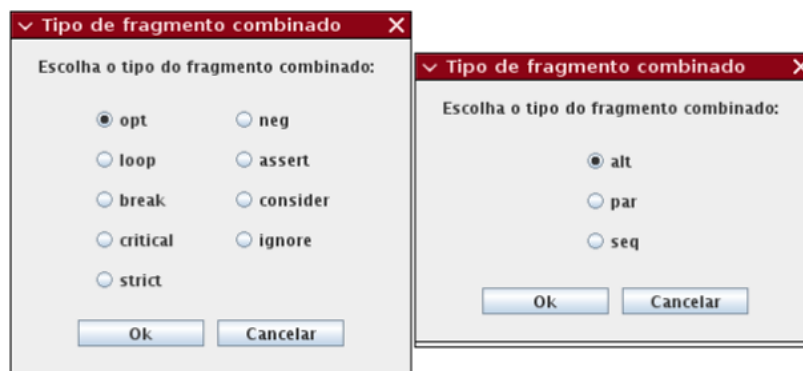


Figura 7.17: Telas para a escolha do tipo do operador do fragmento combinado.

É possível visualizar um exemplo de utilização do diagrama de seqüência criado através do ambiente SEA na imagem 7.18. Utilizou-se o operador *opt* para exemplificar o uso do fragmento combinado.

7.3.8 Diagrama de Estrutura Composta

O diagrama de estrutura composta foi definido a partir da UML 2 e destina-se à descrição de relacionamentos entre elementos. É utilizado para descrever a colaboração interna de classes, interfaces ou componentes para especificar uma funcionalidade.

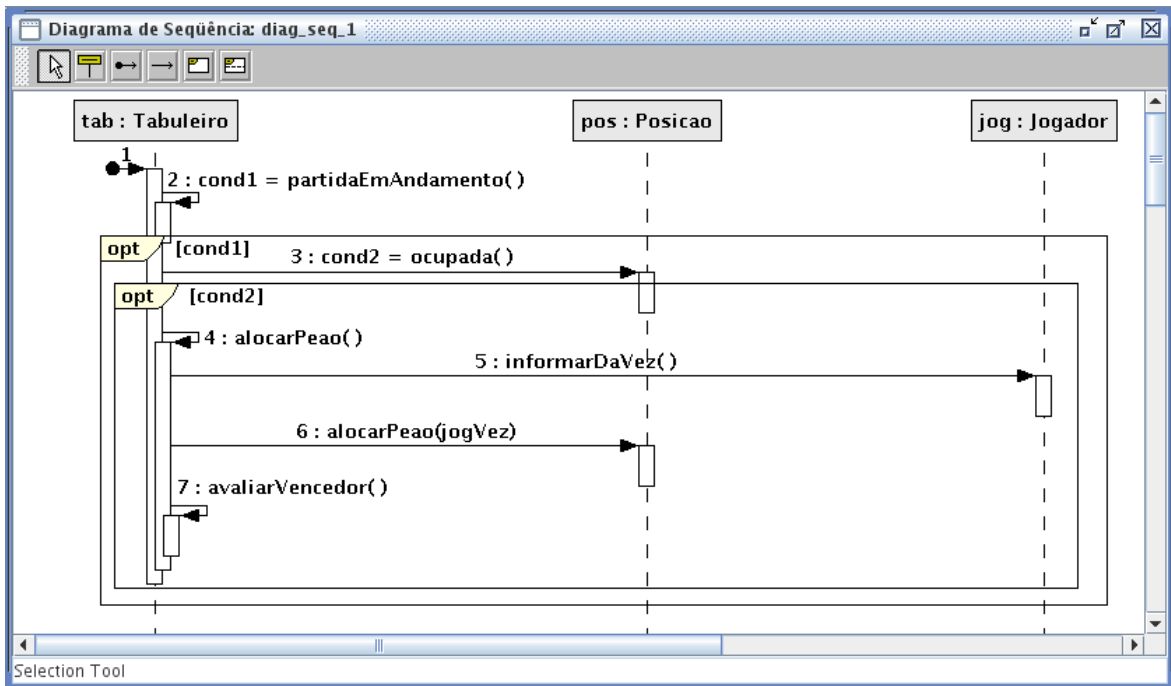


Figura 7.18: Exemplo de diagrama de seqüência construído no ambiente SEA.

Por ser um diagrama novo na UML, teve seu modelo criado do zero no ambiente SEA. Entretanto, alguns conceitos necessários ao diagrama já eram conhecidos. Pode-se observar na imagem 7.19 a estrutura de classes criadas para compor o diagrama de estrutura composta.

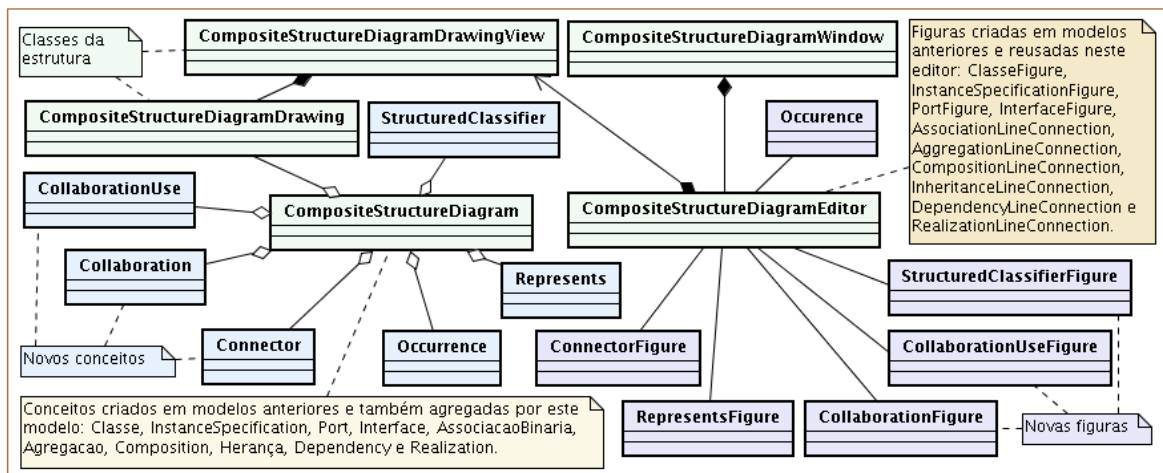


Figura 7.19: Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de estrutura composta.

Conceitos antecedentes

O diagrama de implantação também pôde utilizar alguns conceitos presentes na versão anterior do ambiente SEA e todos os conceitos pertencentes ao diagrama de componentes, conforme comentado anteriormente. Todos os conceitos reusados neste diagrama são apontados a seguir:

- *Association*: Este conceito é representado pela classe “AssociacaoBinaria”.
- *Aggregation*: Este conceito é representado pela classe “Agregacao”.
- *Composition*: Este conceito é representado pela classe “Composition”.
- *Generalization*: Este conceito é representado pela classe “Heranca”.
- *Class*: Este conceito é representado pela classe nomeada “Classe”.
- *Interface*: Este conceito é representado pela classe “Interface”.
- *Port*: Este conceito é representado pela classe “Port”.
- *Dependency*: Este conceito é representado pela classe “Dependency”.
- *Realization*: Este conceito é representado pela classe “Realization”.

Conceitos novos

Apesar de ter usufruído de muitos conceitos já pertencentes à primeira versão de UML, este novo diagrama também trouxe conceitos inéditos. Estes conceitos são apontados e detalhados na seqüência, de acordo com a superestrutura da OMG (OMG, 2007b).

- *Structured classifier*: Conceito que se refere à representação de classes, componentes ou pacotes com a possibilidade de representar sua estrutura interna. Este conceito é representado pela classe “StructuredClassifier”.
- *Collaboration*: É uma estrutura de instâncias interconectadas que desempenham conjuntamente um funcionalidade e, para isso, cada instância assume um papel específico. Este conceito é representado pela classe “Collaboration”.
- *Collaboration Use*: Indica a aplicação do padrão de comportamento representado por uma colaboração a uma situação específica de uma modelagem, em que as classes e as instâncias da especificação tratada assumem os papéis estabelecidos na colaboração. Este conceito é representado pela classe “CollaborationUse”.

- *Connector*: O connector exerce a ligação de portos, interfaces e classificadores estruturados, permitindo interação em tempo de execução. Este conceito é representado pela classe “*Connector*”.
- *Represents*: É um tipo específico de dependência que pode ser usado para ligar uma colaboração a um classificador estruturado, indicando que uma colaboração representa o classificador. Este conceito é representado pela classe “*Represents*”.
- *Occurrence*: É um tipo específico de dependência que pode ser usado para ligar uma colaboração a um classificador estruturado, indicando que uma colaboração é utilizada em um classificador. Este conceito é representado pela classe “*Occurrence*”.

Figuras novas

As figuras que representam os conceitos de colaboração e uso de colaboração tiveram uma complexidade de implementação alta. Tanto a “*CollaborationFigure*” como a “*CollaborationUseFigure*” tem como figura de representação (figura de fundo) uma elipse tracejada, característica que não havia sido necessária em outros modelos. Para isto, criou-se uma nova figura, denominada por “*DashedEllipseFigure*”, que estende a “*EllipseFigure*” e sobrescreve os métodos *drawFrame* e *drawBackground*. Estes métodos utilizam a classe “*BasicStroke*” da biblioteca gráfica *awt* para aplicar o tracejado no momento do desenho da borda da figura. A figura “*CollaborationUseFigure*” também necessitou de uma linha tracejada que separasse o nome do uso de colaboração do resto do conteúdo e, novamente, foi usada a mesma biblioteca gráfica para a codificação. Convém salientar que a figura de colaboração suporta a agregação de conceitos como componentes, classificadores estruturados e interfaces, utilizando-se o mesmo princípio do algoritmo de agregação de classes em pacotes.

A figura “*StructuredClassifierFigure*”, que representa o conceito “*StructuredClassifier*”, foi relativamente simples de ser codificada, uma vez que não necessitou de nenhuma característica particular. Assim como o conceito de componente, um classificador estruturado também agrega portos. Como essa particularidade já havia sido implementada no modelo de diagrama de componentes, bastou reutilizar o algoritmo.

Um conector é um tipo específico de ligação e graficamente ele é bastante semelhante à figura de associação. Sendo assim, criou-se uma figura “*ConnectorFigure*” que herda de “*AssociationLineConnection*”. Entretanto, conectores apresentam descrições particulares que geralmente são nomes de classes e instâncias. Desta forma, precisou-se sobrescrever o método *drawLine* para inserir esta figura de texto ao meio da ligação. Esse texto pode ser editado pelo

usuário através do clique do botão direito do mouse, estando o conector selecionado.

Por fim, duas figuras de dependências específicas foram concebidas: “*RepresentsFigure*” e “*OccurrenceFigure*”. Ambas as figuras puderam herdar de “*DependencyLineConnection*”, já que visualmente são praticamente iguais, exceto que o fato de possuírem estereótipos em sua representação. Assim como na “*ConnectorFigure*”, também se sobrescreveu o método *drawLine* inserindo-se: <<*represents*>>, no caso de uma “*RepresentsFigure*”; ou <<*occurrence*>>, no caso de uma “*OccurrenceFigure*”.

Exemplo de uso

É possível visualizar três exemplos de utilização do diagrama de estrutura composta, dois deles inspirados de exemplos da obra de Ricardo Pereira e Silva (SILVA, 2007), nas imagens 7.20, 7.21 e 7.22. O primeiro exemplo apresenta o uso do padrão de projeto “Decorador” em um sistema de jogo de tabuleiro. Os papéis que rotulam os conectores deixam evidente que elementos estão interligados e de que forma interagem.

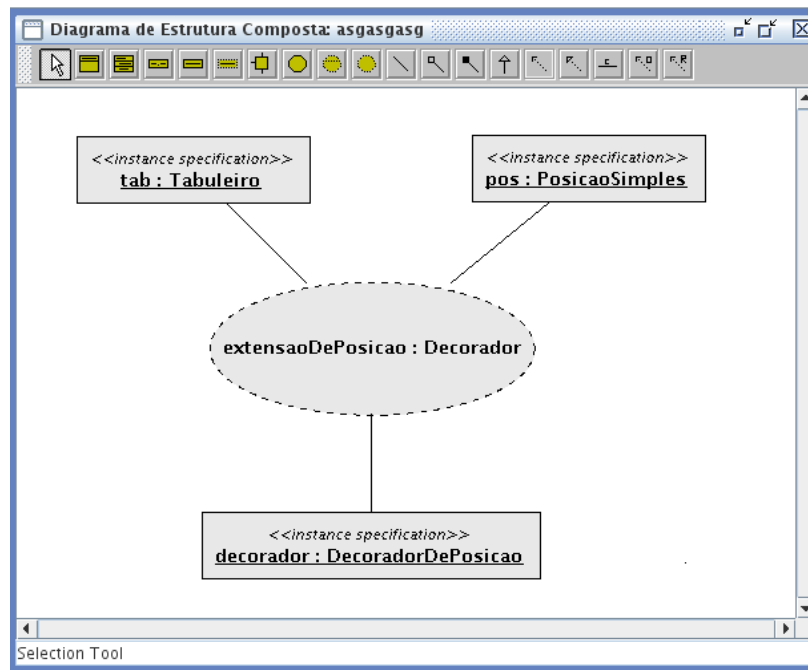


Figura 7.20: Exemplo de diagrama de estrutura composta construído no ambiente SEA utilizando uso de colaboração.

O segundo exemplo define uma colaboração envolvendo instâncias das classes “Decorador-DeSujeito”, “SujeitoConcreto” e “Cliente”. A colaboração apresentada estabelece as conexões previstas entre instâncias e por que portos elas ocorrem. O terceiro exemplo define classificadores estruturados para representar a estrutura interna das classes “Carro”, “Transmissao”,

“Motor” e “SistemaDeTransmissao”.

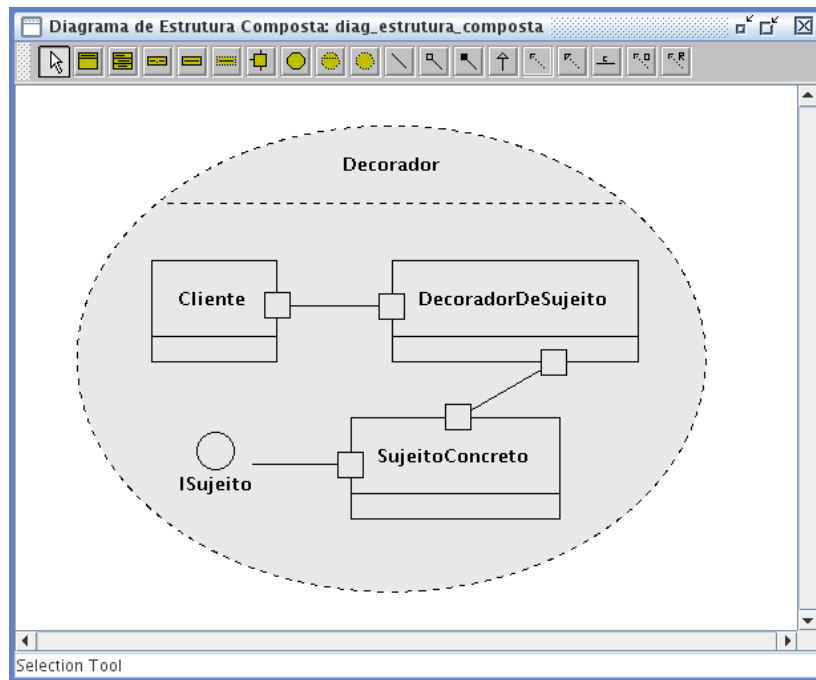


Figura 7.21: Exemplo de diagrama de estrutura composta construído no ambiente SEA utilizando colaboração.

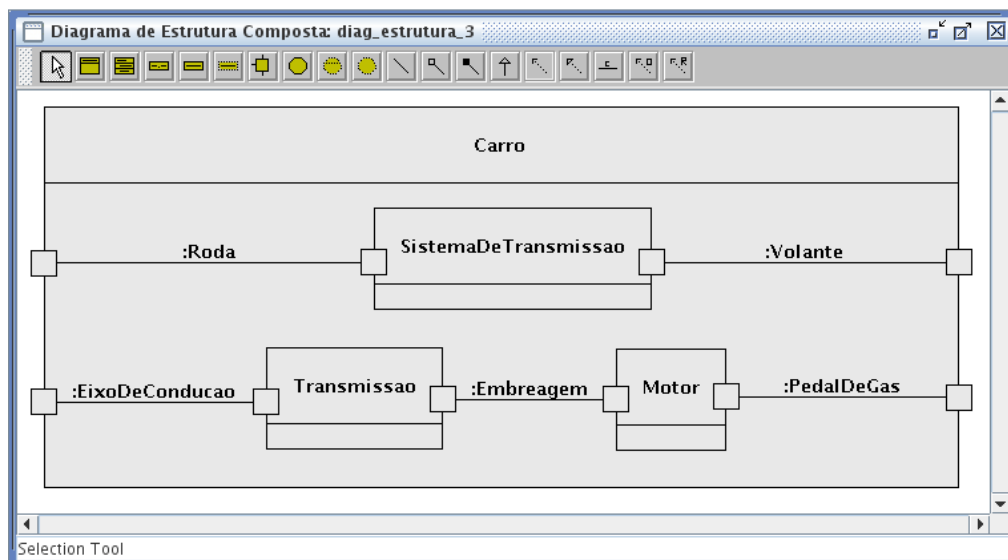


Figura 7.22: Exemplo de diagrama de estrutura composta construído no ambiente SEA utilizando classificador estruturado.

7.3.9 Diagrama de Máquina de Estados

A modelagem dinâmica através de máquina de estados consiste em identificar o conjunto de situações em que o elemento modelado pode estar envolvido ao longo do período tratado pela modelagem e os percursos possíveis através desse conjunto de situações. Além das situações possíveis, também é possível estabelecer as transições possíveis (SILVA, 2007).

Na versão anterior a este trabalho, já haviam algumas classes que começavam a oferecer suporte ao diagrama de *Statechart*, presente na UML 1. Entretanto, o diagrama estava incompleto e inadequado, incluindo apenas o conceito de estado e de transição. Assim, optou-se por refazer este diagrama, não reutilizando as classes anteriores.

Sendo assim, vale salientar que o modelo de diagrama de máquina de estados não reutilizou nenhum conceito criado anteriormente em outros diagramas deste trabalho ou em outras versões do ambiente SEA. Todos os conceitos pertencentes a este modelo são exclusivos deste diagrama e foram criados do zero.

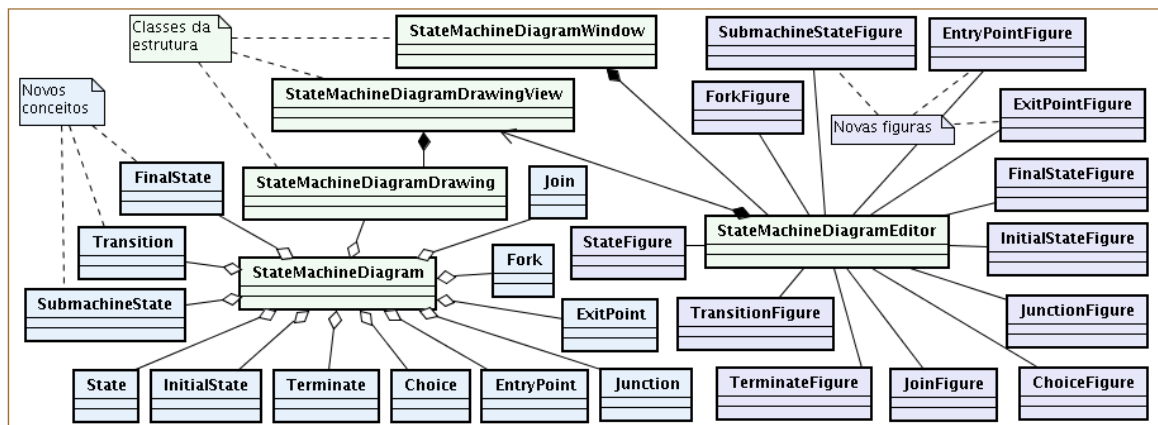


Figura 7.23: Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de máquina de estados.

Conceitos novos

Como descrito anteriormente, todos os conceitos pertencente a este modelo são novos. Cada um teve suas peculiaridades, uma vez que são muito diferentes dos já tratados em diagramas anteriores. A seguir serão listados e detalhados (de acordo com a obra de SILVA) todos os conceitos concebidos neste modelo.

- *State*: Um estado de um sistema é uma situação na qual um sistema se encontra durante

uma operação, durante a qual alguma característica persiste. Este conceito é representado neste modelo pela classe “*State*”.

- *Initial state*: O estado inicial corresponde ao ponto de partida de uma evolução de estados. Não é caracterizado como estado, isto é, não tem valores de atributos que definam uma situação. Este conceito é representado neste modelo pela classe “*InitialState*”.
- *Final state*: O estado final representa uma situação de uma evolução de estados, da qual não é possível sair. Este conceito é representado neste modelo pela classe “*FinalState*”.
- *Entry point*: O conceito de ponto de entrada é um pseudo-estado que o início de uma transição de estado. Geralmente é usado por estados compostos ou submáquinas de estado. Este conceito é representado pela classe “*EntryPoint*”.
- *Exit point*: O conceito de ponto de saída é um pseudo-estado que estabelece o final de uma transição de um estado. Este conceito é representado pela classe “*ExitPoint*”.
- *Transition*: Uma transição é o elemento que produz a passagem de um estado para o outro. Este conceito é representado neste modelo pela classe “*Transition*”.
- *Choice*: O conceito de escolha é representado por um losango com uma transição entrando e duas ou mais saindo. Corresponde a um ponto de decisão na evolução de estados em que o percurso da evolução tomará um caminho dentre os possíveis. Este conceito é representado neste modelo pela classe “*Choice*”.
- *Terminate*: Representado por um “X”, o conceito de término representa a extinção do elemento que é alvo da modelagem de evolução de estados, equivalentemente à destruição de objeto. Este conceito é representado neste modelo pela classe “*Terminate*”.
- *Fork*: O conceito de “fork” estabelece transições concorrentes. Este conceito é representado neste modelo pela classe “*Fork*”.
- *Join*: O conceito de “join” estabelece a sincronização de transições concorrentes. Este conceito é representado neste modelo pela classe “*Join*”.
- *Junction*: O conceito de junção é um pseudo-estado que une vértices a fim de encadear múltiplas transições. A junção é utilizada para construir caminhos de transições compostas entre estados. Este conceito é representado pela classe “*ExitPoint*”.
- *Submachine state*: O conceito de estado de submáquina consiste em uma referência a um diagrama de máquina de estados representada em outro diagrama, na forma de estado. Este conceito é representado neste modelo pela classe “*SubmachineState*”.

Figuras novas

As figuras “*InitialStateFigure*”, “*FinalStateFigure*”, “*EntryPointFigure*”, “*ExitPointFigure*” e “*JunctionFigure*” tiveram a implementação similar, uma vez que todas são elipses e todas têm o mesmo diâmetro fixo: 18 pixels. Entretanto, cada uma delas apresentou algumas características particulares. A figura “*InitialStateFigure*”, que representa conceito de estado inicial, apenas teve o conteúdo da elipse preenchido com a cor preta. Já na figura “*FinalStateFigure*”, que representa o conceito de estado final, necessitou-se utilizar duas elipses: uma preenchida com a cor branca e outra preenchida com a cor preta e com o diâmetro de 12 pixels, localizada no interior da primeira, formando-se apenas uma única figura. A figura “*EntryPointFigure*” foi implementada de maneira bem similar a figura de estado inicial, contudo, teve seu interior preenchido de um tom de cinza e seu contorno em preto. Já a figura “*ExitPointFigure*” teve sua elipse codificada da mesma maneira que a figura de ponto de entrada, no entanto, ainda precisou-se traçar duas linhas internas à elipse de forma a desenhar um “X” ao meio, característica particular da figura que representa o conceito de ponto de saída. A figura “*JunctionFigure*” teve a mesma implementação da figura de estado inicial.

O conceito de escolha no diagrama de máquina de estados foi representado pela figura “*ChoiceFigure*”, que teve uma implementação simples, porém ligeiramente trabalhosa. Esta figura é composta por uma figura de polígono pertencente ao framework JHotDraw denominada “*PolygonFigure*”. Com uma instância de uma figura de polígono, foi possível desenhar um losango adicionando-se pontos através do método *addPoint* a fim de se criar as quatro arestas necessárias à representação do conceito de escolha. Vale ressaltar que optou-se por escolher a figura “*PolygonFigure*” ao invés da figura “*PolyLineFigure*” uma vez que a segunda não permite o fechamento automático de uma figura e nem a inserção de cores no interior de figuras fechadas.

As figuras “*ForkFigure*” e “*JoinFigure*” foram compostas por retângulos com largura e altura fixa, preenchidos com a cor preta. É possível girar a posição destas figuras através do botão direito do mouse.

O conceito de término neste diagrama foi representado pela figura “*TerminateFigure*”. Ela consiste em um retângulo sem preenchimento com duas linhas que se cruzam em seu interior formando um “X”.

Este modelo possui apenas uma figura que herda da especificação “*SpecificationLineFigure*”: a figura “*TransitionFigure*”, que representa o conceito de transição. Como uma transição é composta por uma linha reta com uma seta aberta na extremidade, pôde-se estender direta-

mente da classe “*AssociationLineConnection*” a fim de herdar a linha reta, e desenhou-se a seta através da classe “*ArrowTip*”, adicionando-a como uma decoração através do método *setEndDecoration*.

A figura “*StateFigure*”, que representa o conceito de estado, teve de ser refeita mais de uma vez. Primeiro porque inicialmente não se tinha o conhecimento de que já existia uma classe no framework JHotDraw que fornecia suporte a retângulos com os cantos arredondados e, sem a idéia que esta característica já estava presente, desenhou-se esta figura através de um polígono, gerando muito mais trabalho de implementação. Segundo pois a autora não havia desenhado o estado de acordo com as conformidades da especificação da UML 2.0. Após estes impasses, chegou-se a uma versão final, utilizando-se a figura “*RoundRectangleFigure*” do framework JHotDraw a fim de tornar os cantos arredondados do retângulo automaticamente. Através do método *setArc* foi possível editar a angulação dos arredondamentos. Ao topo do retângulo foi adicionada uma figura de texto para a inserção e edição da descrição do estado. Abaixo desta, inseriu-se uma linha reta que separa a descrição do estado de seus métodos.

A última e mais complexa figura implementada neste modelo foi a “*SubmachineStateFigure*”, que representa o conceito de submáquina de estados. Assim com a “*StateFigure*”, ela também utilizou a figura “*RoundRectangleFigure*” para obter os arredondamentos automaticamente. Não possui linha de separação de conteúdo, mas necessitou do desenho de um pequena figura localizada em seu canto inferior direito, que pode ser observado no exemplo da próxima seção. Para conceber este ícone, utilizou-se a figura “*PolyLineFigure*”, desenhando-o ponto a ponto através do método *addPoint*. Após conferir esta característica, foi necessário oferecer suporte a agregação dos conceitos de ponto de entrada e ponto de saída. Para isto, apenas reusou-se e adaptou-se o algoritmo elaborado no modelo de diagrama de componentes, onde componentes agregam portos.

Exemplo de uso

Pode-se visualizar na imagem 7.24 um exemplo de utilização do diagrama de máquina de estados construído no ambiente SEA. O exemplo não é significativo, mas nele apresenta-se a utilização da maioria dos conceitos concebidos.

7.3.10 Diagrama de Atividades

O diagrama de atividades tem por finalidade descrever o comportamento de um programa, ou seja, descrevê-lo quando em execução. O objetivo desta descrição é o detalhamento em mais

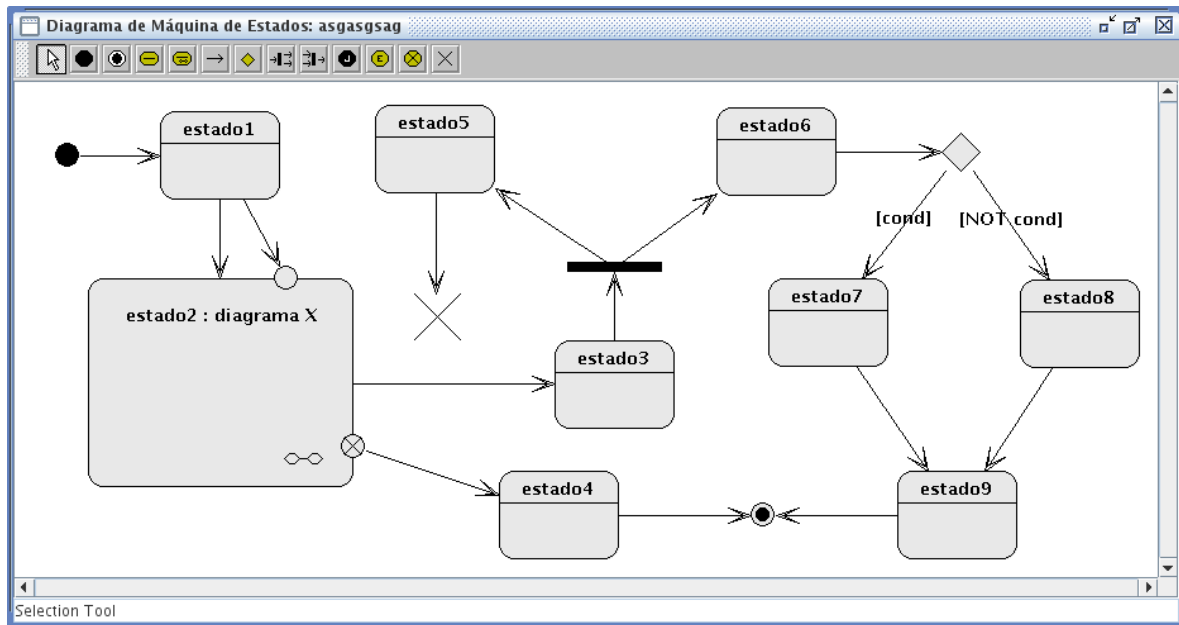


Figura 7.24: Exemplo de diagrama de máquina de estados construído no ambiente SEA.

alto ou em mais baixo nível de abstração daquilo que é feito pelo programa: os procedimentos executados. Isto é modelado em termos de atividades e ações e dos possíveis percursos entre esses elementos (SILVA, 2007).

Na versão anterior a este trabalho, havia um modelo incompleto para o diagrama de atividades presente na especificação orientada a objetos. Entretanto, apenas o conceito de atividade estava criado e as classes da estrutura básica do modelo não estavam concluídas. Desta maneira, optou-se por reescrever um novo modelo que representasse o diagrama de atividades por completo. Ao todo foram desenvolvidos 40 conceitos para este modelo e suas respectivas figuras. Alguns destes conceitos puderam herdar de outros conceitos presentes no diagrama de máquina de estados, contudo todos eles são conceitos novos na especificação UML 2 do ambiente SEA.

Com o intuito de ilustrar a arquitetura de classes criada à construção deste modelo no ambiente SEA, pode-se visualizar o diagrama de classes da figura 7.25. Não foram colocados explicitamente todos os conceitos e figuras criados no diagrama de atividades por estarem em grande número - o que dificultaria o entendimento.

Conceitos novos

Uma das possibilidades de uso do diagrama de atividades é para a modelagem de algoritmo de método de classe. Neste caso, os elementos sintáticos do diagrama são usados para modelar os comandos que compõem código de método (SILVA, 2007). Sabendo que os elementos

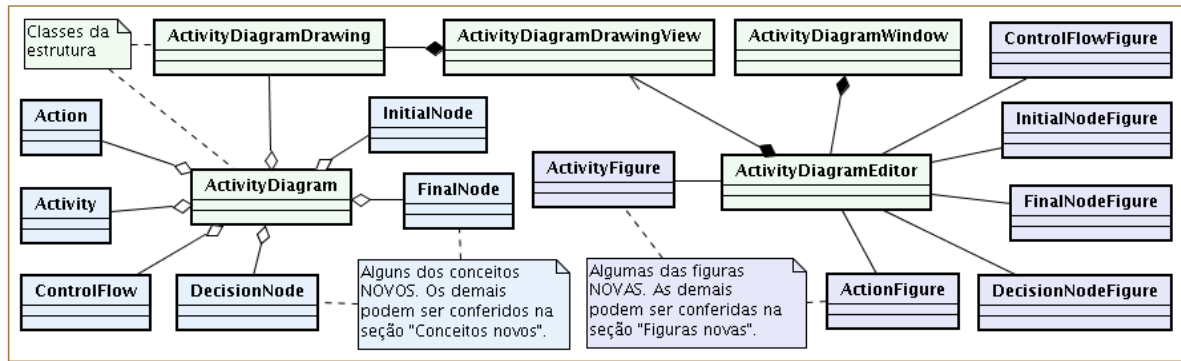


Figura 7.25: Diagrama de classes que expõe a arquitetura criada para a produção do diagrama de atividades.

sintáticos de UML são mapeados em conceitos no ambiente SEA, construiu-se dois grupos de conceitos para o modelo de diagrama de atividades: conceitos genéricos do diagrama de atividades e conceitos específicos para modelagem de algoritmo.

A seguir são listados e detalhados os conceitos genéricos do diagrama de atividades.

- *Action*: O conceito de ação é a unidade de modelagem de comportamento. Consiste em algo que é executado sem interrupção. Neste modelo, é representada pela classe “*Action*”.
- *Activity*: Uma atividade é um elemento não atômico de modelagem de comportamento. Neste modelo, este conceito é representando pela classe “*Activity*”.
- *Initial node*: O nodo inicial estabelece o início da execução de uma atividade. Este conceito é representado neste modelo pela classe “*InitialNode*”.
- *Final node*: O nodo final estabelece o final da execução de uma atividade. É representado pela classe “*FinalNode*”.
- *Send signal*: O conceito de envio de sinal representa uma ação voltada a disparar a execução de um procedimento. É representado pela classe “*SendSignal*”.
- *Accept event action*: O conceito de recebimento de evento é uma ação que modela o recebimento de uma comunicação pelo elemento modelado. É representado pela classe “*AcceptEventAction*”.
- *Accept time event action*: Este conceito é um caso especial de uma ação de recebimento de evento. Consiste em um evento temporizado, que ocorre em intervalos determinados. Neste modelo, é representado pela classe “*AccpetTimeEventAction*”.

- *Object node*: É um conceito do diagrama de atividades que representa uma instância de uma classe. É representado pela classe “*ObjectNode*”.
- *Central buffer node*: O nodo *buffer* central é um nodo objeto que pode ser interligado a outros nodos objeto e que atua como um buffer que intermedia múltiplos fluxos de entrada e de saída. Este conceito é representado neste modelo pela classe “*CentralBufferNode*”.
- *Data store node*: O nodo armazenador de dados é um conceito especializado do *central buffer node* que é voltado ao armazenamento de dados. Modela a noção de persistência dos dados. Este conceito é representado neste modelo pela classe “*DataStoreNode*”.
- *Control flow*: O conceito de fluxo de controle consiste do percurso de uma atividade. Neste modelo, este conceito é representado pela classe “*ControlFlow*”.
- *Object flow*: O fluxo de objeto é um conceito específico do fluxo de controle. Controla o percurso de objetos. É representado pela classe “*ObjectFlow*”.
- *Flow final node*: O nodo final de fluxo é um conceito que estabelece o final de um fluxo de controle em uma atividade. É representado pela classe “*FlowFinalNode*”.
- *Decision node*: O nodo de decisão é um conceito que permite fluxos de controle alternativos, condicionados por expressões booleanas. Neste modelo, este conceito é representado pela classe “*DecisionNode*”.
- *Fork node*: O nodo *fork* estabelece fluxos de controle concorrentes. Este conceito é representado pela classe “*ForkNode*”.
- *Merge node*: O nodo *merge* é um conceito que funde um conjunto de fluxos em um único. É representado pela classe “*MergeNode*”.
- *Join node*: O nodo *join* estabelece a sincronização de fluxos de controle concorrentes. Neste modelo, este conceito é representado pela classe “*JoinNode*”.
- *Exception handler*: O manipulador de exceção é um conceito que especifica uma determinada ação a ser executada no caso de uma exceção durante a execução de um nodo. É representado pela classe “*ExceptionHandler*”.
- *Interruptible activity*: A atividade interrompível é um conceito que representa uma região onde o fluxo normal de execução pode ser interrompido por algum evento que leva o fluxo de controle para fora da região delimitada, sendo que quaisquer ações ou atividades nela contidas e que estejam em execução são encerradas. Neste modelo, este conceito é representado pela classe “*InterruptibleActivity*”.

- *Structured activity node*: O nodo de atividade estruturada é um conceito que agrupa elementos de uma atividade que não fazem parte de outro nodo do mesmo tipo. É representado pela classe “*StructuredActivityNode*”.
- *Loop node*: O nodo laço modela uma estrutura de repetição. É representado pela classe “*LoopNode*”.
- *Sequence node*: O node seqüência é um conceito que estabelece a execução de seus nodos na seqüência em que são modelados. Neste modelo, este conceito é representado pela classe “*SequenceNode*”.
- *Conditional node*: O nodo condicional estabelece a modelagem de escolhas. Este conceito é representado pela classe “*ConditionalNode*”.
- *Vertical swimlane*: Este conceito representa uma partição vertical que envolve elementos do diagrama de atividades. É representado pela classe “*VerticalSwimlane*”.
- *Horizontal swimlane*: Este conceito representa uma partição horizontal que envolve elementos do diagrama de atividades. É representado pela classe “*HorizontalSwimlane*”.
- *Note*: Uma nota é um conceito que apresenta um bloco de texto que descreve um elemento ou conjunto de elementos. Não tem efeitos de execução. Este conceito é representado pela classe “*Note*”.
- *Note link*: É o conceito que liga uma nota ou um comentário a um determinado elemento de interesse. É representado pela classe “*NoteLink*”.

A partir dos conceitos genéricos presentes no diagrama de atividades, o professor Ricardo Pereira e Silva concebeu novos conceitos (SILVA, 2000), específicos para modelagem de algoritmo. A extensão desenvolvida também buscou dar suporte a todos estes conceitos. São eles:

- *Comment*: Assim como o conceito de nota, um comentário não corresponde a um comando executável, apenas volta-se a conter um texto descritivo. A única diferença entre o conceito de nota e o conceito de comentário é que um comentário possui em seu cabeçalho um estereótipo. É representado pela classe “*Comment*”.
- *Assignment*: Este conceito representa um comando voltado à atribuição de variável temporária, atributo, parâmetro, classe ou constante a um atributo da classe que possui o método modelado ou a variável temporária desse método. Neste modelo, é representado pela classe “*Assignment*”.

- *Return*: Este conceito representa um comando para retornar um objeto ou valor. É representado pela classe “*Return*”.
- *Message*: O conceito de mensagem representa um comando de envio de mensagem a objeto, com a opção de atribuição de objeto retornado a atributo ou variável temporária. Neste modelo, este conceito é representado pela classe “*Message*”.
- *Variable*: Este conceito representa um comando de declaração de variáveis temporárias. É representado pela classe “*Variable*”.
- *Task package*: O conceito de pacote de tarefas é preenchido com um texto que, em um processo automático de geração de código, é copiado diretamente para o código-fonte. Neste modelo, este conceito é representado pela classe “*TaskPackage*”.
- *Generic task*: O conceito de tarefa genérica representa um comando que corresponde à definição de uma tarefa em um alto nível de abstração. É representado pela classe “*GenericTask*”.
- *If node*: Conceito que representa o comando *if*, um invólucro de comandos que condiciona a execução destes a uma expressão booleana. Neste modelo, este conceito é representado pela classe “*IfNode*”.
- *If-else node*: Conceito que representa um comando semelhante ao *if*, porém com uma cláusula *else*, um invólucro de comandos no caso da condição booleana resultar false. É representado pela classe “*IfElseNode*”.
- *Switch node*: Conceito que representa um comando que contém um conjunto de invólucros de comandos semelhante ao *if-else*, porém com várias alternativas, em vez de somente duas. É representado pela classe “*SwitchNode*”.
- *While node*: Conceito que representa um comando de repetição que consiste em um invólucro de comandos em que a condição é testada antes de cada execução. É representado pela classe “*WhileNode*”.
- *Do-while node*: Conceito que representa um comando de repetição que consiste em um invólucro de comandos em que a condição é testada após cada execução. É representado pela classe “*DoWhileNode*”.
- *For node*: Conceito que representa um comando de repetição que consiste em um invólucro de comandos em que a quantidade de repetições é definida por uma expressão. Neste modelo, este conceito é representado pela classe “*ForNode*”.

Barra de ferramentas

Em virtude de ter-se duas categorias para os conceitos criados no modelo de diagrama de atividades, optou-se por criar três barras de ferramentas em um mesmo editor: uma barra que apresenta ferramentas típicas do diagrama de atividades, utilizada tanto em modelagem de algoritmo como em outros tipos de modelagem de atividades; uma barra que apresenta ferramentas típicas do diagrama de atividades que não deve ser utilizada em modelagem de algoritmo, somente em outros tipos de modelagem de atividades; e uma barra que apresenta ferramentas específicas de modelagem de algoritmo que não deve ser utilizada em outros tipos de modelagem de atividades. A imagem 7.26 ilustra o resultado obtido na implementação deste conjunto de ferramentas.



Figura 7.26: Barra de ferramentas customizada.

É importante salientar que foi necessário sobrescrever o método *open*, herdado de “*SpecificationEditor*”, para que o editor do diagrama de atividades ganhasse a característica de possuir mais de uma barra de ferramentas, já que por padrão apenas uma barra de ferramentas é criada. Duas novas instâncias de “*JToolBar*” foram inicializadas. A instância que representa a barra de ferramentas específica à modelagem de algoritmos não foi deixada visível por *default*, apenas as instâncias que representam as barras de ferramentas típicas de diagrama de atividades. Assim, para que o usuário possa desfrutar de uma barra de ferramentas escondida, é necessário clicar no menu de opções e escolher a barra desejada para utilização.

Figuras novas

Todas as figuras criadas neste diagrama são novas, entretanto muitas delas puderam estender figuras pertencentes ao diagrama de máquina de estados por serem visualmente idênticas. Este é o caso das figuras: “*InitialNodeFigure*”, que estende a figura “*InitialStateFigure*”; “*FinalNodeFigure*”, que estende a figura “*FinalStateFigure*”; “*FinalFlowNodeFigure*”, que estende

a figura “*ExitPointFigure*”; “*DecisionNodeFigure*” e “*MergeFigure*”, que estendem a figura “*ChoiceFigure*”; “*ForkNodeFigure*”, que estende a figura “*ForkFigure*”; “*JoinNodeFigure*”, que estende “*JoinFigure*”; “*ControlFlowFigure*” e “*ObjectFlowFigure*”, que estendem “*TransitionFigure*”.

A figura “*ActionFigure*”, que representa o conceito de ação, foi implementada com base na figura “*StateFigure*”. Utilizou-se a figura “*RoundRectangleFigure*” do framework JHotDraw para deixar os cantos do retângulo arredondado. Após a inserção desta figura como figura de representação (de fundo), adicionou-se ao meio uma figura de texto que armazena a descrição da ação. A figura “*ActivityFigure*”, que representa o conceito de atividade, teve a codificação muito parecida com a da “*ActionFigure*”, contudo ela necessitou de uma pequena imagem no seu canto inferior direito, muito parecido com um garfo de cabeça para baixo. Para isto, desenhou-se o ícone linha a linha, de maneira que ele ficasse com a posição relativa à posição da figura de texto que armazena a descrição da atividade.

As figuras que representam os conceitos de envio de sinal e recebimento de evento tiveram a implementação custosa e complexa, uma vez que a figura de fundo utilizada, a “*PolygonFigure*”, comportava-se de maneira inesperada ao inserir texto em seu interior. A implementação passou por inúmeras tentativas e pesquisas de formas de utilização antes de se chegar a uma versão final. Depois de descoberto qual era a melhor maneira de utilizar a figura que representa um polígono, conseguiu-se fazer três tipos de polígono para as respectivas figuras: “*SendSignalFigure*”, “*AcceptEventActionFigure*” e “*AcceptTimeEventActionFigure*”. Observe na imagem 7.27 o resultado obtido.



Figura 7.27: Algumas das figuras que utilizaram a “*PolygonFigure*”.

Três figuras representam os três conceitos relativos a objetos pertencentes ao diagrama de atividades: “*ObjectNodeFigure*”, “*CentralBufferNodeFigure*”, “*DataStoreNodeFigure*”. Estas figuras foram relativamente simples de serem codificadas se comparadas a todas as figuras já desenhadas. São compostas por retângulos que contém o nome do objeto em seu interior. A figura “*CentralBufferNodeFigure*” tem a particularidade do estereótipo <<central buffer>> no topo, assim como a “*DataStoreNodeFigure*” apresenta o estereótipo <<data store>>.

Pode-se dizer que as figuras onde se encontrou mais dificuldade de implementação no diagrama de atividades foram todas as referentes aos nodos de repetição e condição. Todas

fugiram muito dos padrões até então desenvolvidos e como não haviam tutorias para desenhar figuras complexas, bastante tempo foi consumido. Estas figuras, “*InterruptibleActivityFigure*”, “*StructuredActivityFigure*”, “*SequenceFigure*”, “*ConditionalNodeFigure*” e “*LoopNodeFigure*”, basicamente consistem em retângulos arredondados com as bordas tracejadas. A figura “*InterruptibleActivityFigure*” não possui conteúdo inicialmente, mas todas as outras tiveram o estereótipo `<<structured>>` adicionado no topo através de uma figura de texto, já que todas são noções concretas de atividades estruturadas. Vale lembrar também que as figuras “*ConditionalNodeFigure*” e “*LoopNodeFigure*” são filhas de “*SequenceFigure*” e possuem representações internas desta figura. A figura “*LoopNodeFigure*” contém três instâncias de “*SequenceFigure*” que compõem as três seções necessárias a um laço: *setup*, *test* e *body*. Já a figura “*ConditionalNodeFigure*” contém pelo menos duas instâncias de “*SequenceFigure*” que compõem as duas seções necessárias ao nodo condicional: *test* e *body*. É importante observar que o framework JHotDraw não possuía suporte à *layout* horizontal, apenas vertical. Portanto, não era possível implementar a figura “*ConditionalNodeFigure*”, já que quando uma figura é adicionada em uma figura do tipo “*GraphicalCompositeFigure*”, ela por padrão é adicionada embaixo da última figura adicionada, como uma pilha, e a figura que representa o nodo condicional precisava adicionar figuras lado a lado. De maneira a atender às necessidades da extensão, foi necessário criar um novo tipo de *layout* para este caso específico de composição de figuras. A imagem 7.28 exemplifica o resultado obtido na criação da figura “*ConditionalNodeFigure*”.

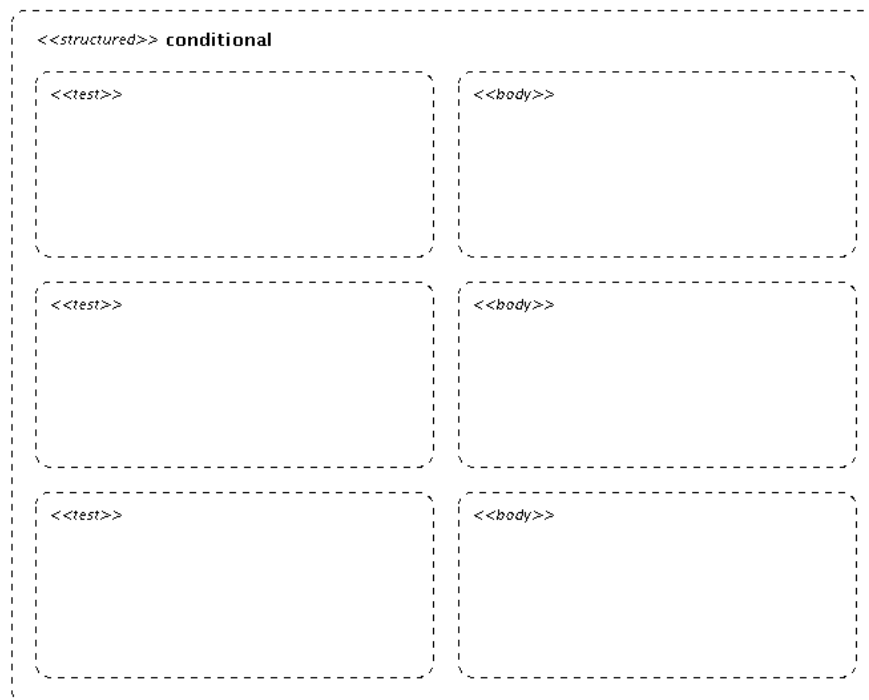


Figura 7.28: A figura “*ConditionalNodeFigure*”.

Especializações das figuras de nodos condicionais e de repetições também foram criados a fim de ajudar a modelagem de algoritmos. Visualmente, as principais diferenças que apresentam são os estereótipos específicos. A figura “*WhileNodeFigure*”, que representa o conceito “*WhileNode*”, apresenta o estereótipo <<while>>. Já a figura “*DoWhileNodeFigure*”, que representa o conceito “*DoWhileNode*”, apresenta o estereótipo <<do-while>>. A figura “*ForNodeFigure*”, que representa o conceito “*ForNode*”, apresenta o estereótipo <<for>>. Já a figura “*IfNodeFigure*”, que representa o conceito “*IfNode*”, além de apresentar o estereótipo <<if>>, possui nodos de fusão e decisão auxiliares, que ficam externos à figura. A figura “*IfElseNodeFigure*”, que representa o conceito “*IfElseNode*”, apresenta os estereótipos <<if>> e <<else>>, cada um em uma “*SequenceFigure*” particular. Também possui nodos de fusão e decisão auxiliares. A imagem 7.29 apresenta o resultado final obtido na implementação destas figuras, adicionando-se ações e fluxos de controle para a utilização.

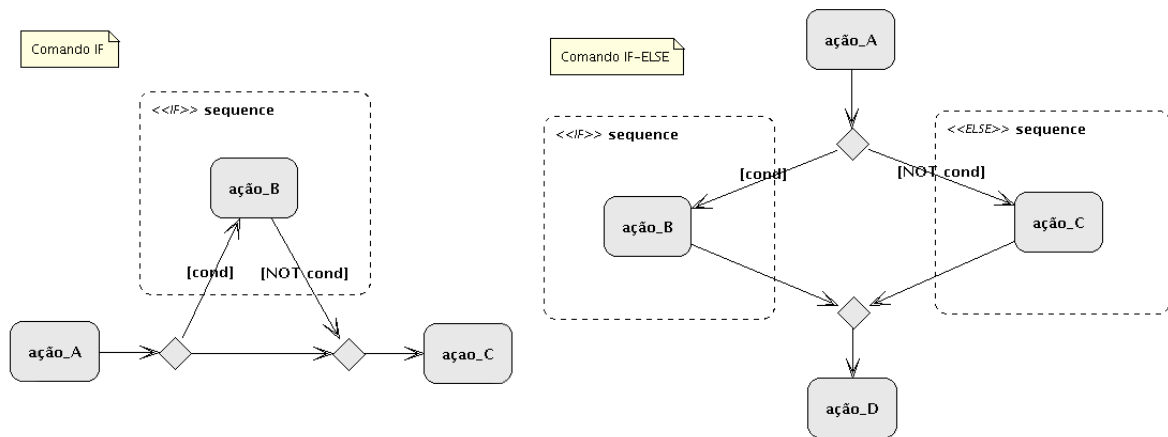


Figura 7.29: As figuras “*IfNodeFigure*” e “*IfElseNodeFigure*”.

Foram implementadas figuras de comentários e notas ao diagrama. São representadas pelas classes “*NoteFigure*” e “*CommentFigure*”. Ambas são figuras compostas por um retângulo com uma ponta dobrada no lado direito (como se fosse uma “orelha” de caderno), desenhada ponto a ponto através da figura “*PolyLineFigure*”. Diferente da “*NoteFigure*”, a “*CommentFigure*” apresenta no topo o estereótipo <<comment>>. Também foi criado a figura “*NoteLinkFigure*” que é uma linha tracejada que representa o relacionamento entre conceitos e comentários e notas.

As figuras “*AssignmentFigure*”, “*ReturnFigure*” e “*MessageFigure*”, que representam os conceitos “*Assignment*”, “*Return*” e “*Message*” respectivamente, são especializações da figura “*ActionFigure*”, contudo apresentam particularidades semânticas. A figura “*AssignmentFigure*” possui o estereótipo <<assignment>>, a “*ReturnFigure*” contém o estereótipo <<return>> e, por fim, a “*MessageFigure*” possui o estereótipo <<message>>.

O conceito de pacotes de tarefas foi representado pela figura “*TaskPackageFigure*” e o conceito de tarefa genérica pela figura “*GenericTaskFigure*”. Estas figuras também são especializações da “*ActionFigure*” possuindo a mais os estereótipos <<*task package*>> e <<*generic task*>>.

Exemplo de uso

Os diagramas presentes nas imagens 7.30 e 7.31, sendo o primeiro inspirado em um exemplo na obra de Ricardo Pereira e Silva (SILVA, 2007), exemplificam formas de aplicação do modelo de diagrama de atividades no ambiente SEA.

O diagrama da figura 7.30 apresenta a modelagem do procedimento de *proceder lance* em um jogo de tabuleiro comum, separando as responsabilidades entre instâncias. Através do uso das partições, é possível explicitar o que cada instância faz.

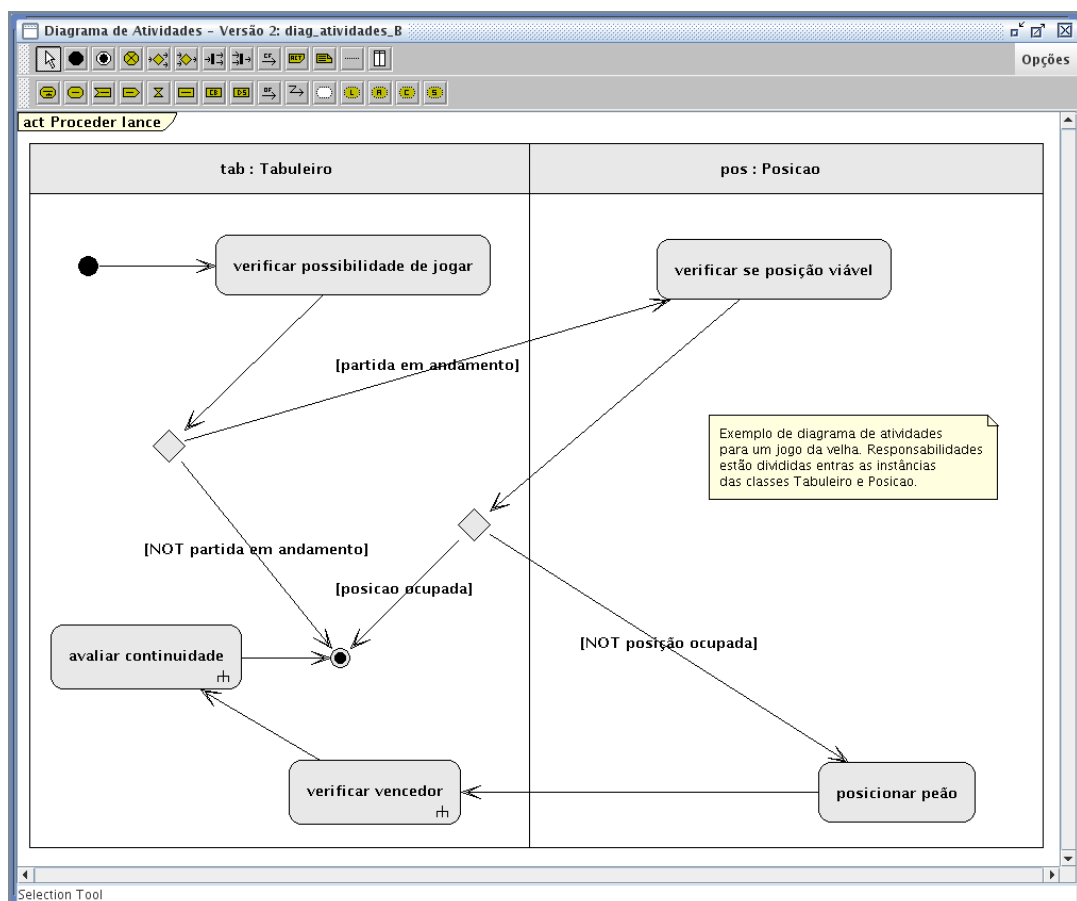


Figura 7.30: Exemplo de diagrama de atividades construído no ambiente SEA utilizando partições.

A figura 7.31 apresenta um diagrama de atividades que modela um algoritmo de método.

O método tratado no exemplo é *imprimeStatusAluno* de uma classe de um sistema para cálculo de médias.

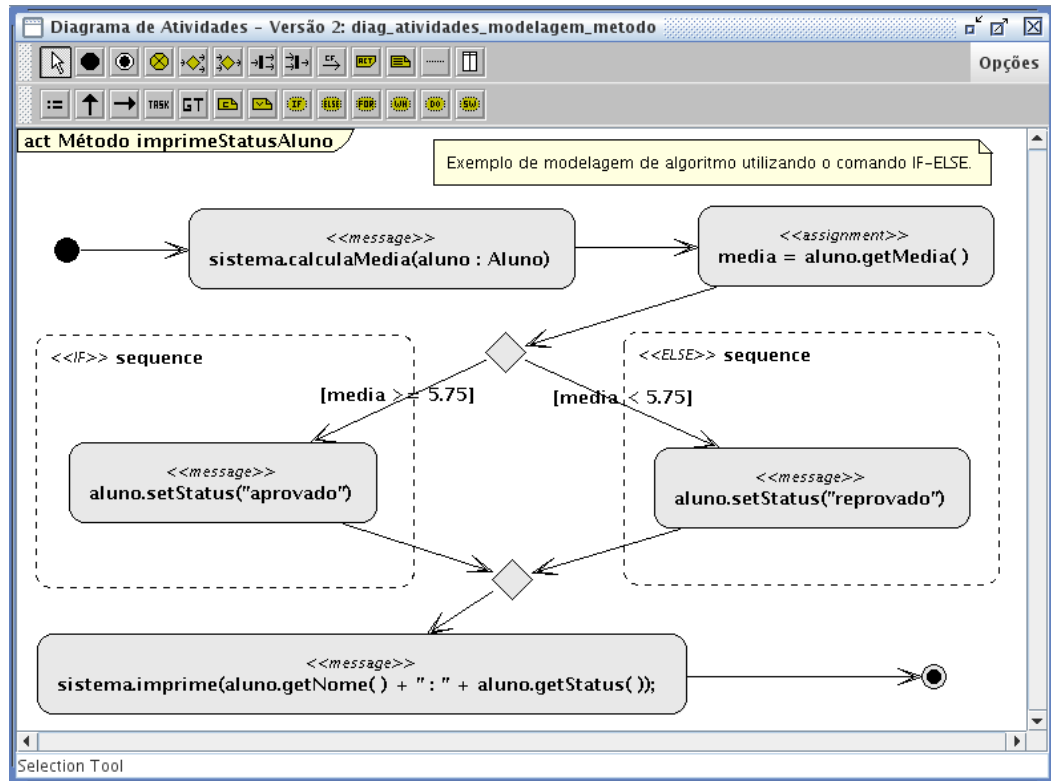


Figura 7.31: Exemplo de diagrama de atividades construído no ambiente SEA utilizando ferramentas específicas para modelagem de algoritmo.

7.4 Suporte à Internacionalização e Localização

A internacionalização e localização são processos de desenvolvimento e adaptação de um produto, em geral softwares de computadores, para uma língua e cultura de um país. A internacionalização de um produto não fabrica o produto novamente, somente adapta a língua e a cultura local as mensagens do sistema. Isto é importante pois permite que o desenvolvedor de software respeite as particularidades de cada língua e cultura de cada país (THEPPITAK, 2005).

A palavra “internacionalização” também é escrita acronimamente como i18n, vindo da palavra inglesa *internationalization* - retira-se a primeira e a última letra e o número de letras que contém a palavra. Da mesma forma, a palavra “localização” (em inglês *localization*) também é escrita acronimamente como l10n.

A diferença entre internacionalização e localização é somente fundamental. A internacionalização é uma adaptação de um produto para melhoramento e a localização é uma adição de

características específicas de uma região. Os dois são evidentemente complementares.

Alguns dos elementos específicos da localização são: tradução lingüística, suporte a várias línguas, símbolos, métodos de ordenação de listas, valores culturais e contexto social. No desenvolvimento de software, depois de internacionalizado um produto, o termo “localização” se refere ao processo necessário para o produto internacionalizado esteja também pronto para mercados específicos (THEPPITAK, 2005).

Por esta razão pode-se falar que um produto internacionalizado satisfaz a comunidade internacional, mas não para um mercado específico. A preparação para um mercado específico é chamado de localização.

A internacionalização e localização trazem inúmeros benefícios como a redução significativamente a quantidade de treinamento necessária para os usuários finais para o uso de um sistema de computador, facilitando a introdução da informática em pequenas e médias empresas, permitindo que empregados trabalhem inteiramente na sua língua nativa e facilitando o desenvolvimento dos sistemas e para controlar bases de dados de nomes e de dados locais da língua, facilitando a descentralização dos dados em níveis provinciais e de distrito.

Buscou-se dar suporte à internacionalização no ambiente SEA à medida que ía-se construindo a extensão para UML 2. É importante lembrar que o projeto inicial do ambiente não apresentou este requisito. Sendo assim, no que se refere às classes implementadas anteriormente a este trabalho, forneceu-se a internacionalização apenas às principais, em virtude da falta de tempo. Já no que se refere à extensão implementada, todas as classes possuem suporte à internacionalização, uma vez que foi projetada desde o início para fornecer esta característica.

Vale ressaltar que a internacionalização no ambiente SEA se ateu à língua, tendo como objetivo a codificação dos textos apresentados ao usuário em diferentes sistemas de escrita. Outros pontos importantes da internacionalização como formatos de data, tempo e moeda não foram aplicados pois não são empregados no ambiente SEA.

7.4.1 Classes utilitárias

A fim de prover suporte à internacionalização no ambiente SEA foi necessário a criação e implementação de algumas classes utilitárias. A principal delas foi a classe “*II8NProperties*”, que apresenta atributos e métodos públicos e estáticos para possibilitar a utilização em todo o sistema sem a necessidade de instanciação. As funcionalidades oferecidas por esta classe são:

- *Inicializar a localidade*: Funcionalidade fornecida através do método *initializeLocale*. É neste ponto onde uma instância da classe “*Locale*” é criada. Um objeto “*Locale*” em Java representa um região geográfica, política ou cultural específica. Para criar um objeto “*Locale*” de uma determinada localização, é necessário passar como parâmetro na execução do ambiente SEA a língua e o país desejado para a utilização no sistema. Por exemplo, se o usuário desejar que a língua utilizada no ambiente SEA seja inglês e o país Estados Unidos, basta executar o programa com parâmetros da seguinte forma:

```
$>java Splash en US
```

Segundo a ²API da ³*Sun Microsystems*, estes valores, mapeados no sistema como “*String*”, são padronizados conforme normas da ⁴ISO. O parâmetro que se refere à língua responde a norma *ISO-639* e o parâmetro que se refere ao país responde a norma *ISO-3166*. Caso o usuário não passar parâmetros na execução do sistema, o sistema terá a língua *default* como português e o país *default* como Brasil.

- *Buscar a tradução*: Funcionalidade fornecida através do método *getString*. Este método recebe como parâmetro uma palavra-chave em língua inglesa e retorna o texto referente traduzido para a língua setada pelo usuário na inicialização do ambiente SEA. Essa tradução é feita através de buscas de *strings* em arquivos de tradução. Para isto, é necessário a criação de um objeto de uma classe utilitária chamada *ResourceBundle*. É com uma instância desta classe que o método *getString* conseguirá varrer todas as *strings* de todos os arquivos de propriedades existentes em um determinado caminho passado por parâmetro na sua criação. Um exemplo de utilização desta funcionalidade, estando a língua setada para o português:

```
String text = I18NProperties.getString(‘‘change.state’’);
```

A saída da string será: “Alterar a descrição do estado”.

7.4.2 Arquivos de tradução

Estando as classes utilitárias criadas, era necessário criar os arquivos de tradução. Na linguagem de programação Java, estes tipos de arquivos são comumente mapeados como arquivos de propriedades, com a extensão *.properties*. Estes arquivos de propriedades são formados por

²É a sigla para *Application Programming Interface* ou, em português, “Interface de Programação de Aplicativos”. É um conjunto de rotinas e padrões estabelecidos por um software para utilização de suas funcionalidades por programas aplicativos.

³Empresa responsável pela linguagem Java.

⁴É a sigla para *International Organization for Standardization* ou, em português, “Organização Internacional para Padronização”. É uma entidade que aglomera os grêmios de normalização de 158 países.

vários pares de *strings* chave e valor, separados por “=” . Em geral, a chave é escrita em inglês e representa um *string* oficial para um determinado texto. É através dessa chave que um texto traduzido será localizado. O valor pode ser escrito em qualquer língua, contanto que esteja conizente com o nome do arquivo. Tanto a chave como o seu respectivo valor são tratados como *strings* no sistema.

No ambiente SEA, foram criados dois arquivos de tradução (imagem 7.32): um arquivo para português (“pt”) do Brasil (“BR”), o *I18NProperties_pt_BR.properties*; e um arquivo para inglês (“en”) dos Estados Unidos (“US”), o *I18NProperties_en_US.properties*. Todas os textos utilizados nesta extensão do ambiente SEA estão internacionalizados nestas duas línguas. Se outro desenvolvedor do ambiente SEA desejar dar suporte a uma nova língua, basta criar um novo arquivo de tradução com a nomenclatura padrão:

I18NProperties_<língua>_<país>.properties

<pre> 1# ----- 2# Properties - en/US 3# ----- 4 5sea=SEA - Reusable Software Artifacts Environment 6 7uml2.specification=UML 2 Specification 8uml2.specification.for.tree=UML 2 Specification 9 10class=CLASS 11new.class=New class 12 13object=OBJECT 14new.object=New object 15 16object.concept.name=Object 17object.concept.name.for.tree=Object: 18 19object.model.name=Object Diagram 20object.model.name.for.tree=Object Diagram: 21 22object.not.created=Object couldn't be created. 23class.not.created=Class couldn't be created. 24component.not.created=Component couldn't be created. 25port.not.created=Port couldn't be created. 26interface.not.created=Interface couldn't be created. 27node.not.created=Node couldn't be created. 28label.not.created=Label couldn't be created. 29 30actor=ACTOR 31new.actor=New actor 32 33label=LABEL 34new.label=New label 35 36message=MESSAGE 37new.message=New message </pre>	<pre> 1# ----- 2# Properties - pt/BR 3# ----- 4 5sea=SEA - Ambiente de Artefatos de Software Reusaveis 6 7uml2.specification=Especificação UML 2 8uml2.specification.for.tree=Especificação UML 2 9 10class=CLASS 11new.class=Nova classe 12 13object=OBJECT 14new.object=Novo objeto 15 16object.concept.name=Objeto 17object.concept.name.for.tree=Objeto: 18 19object.model.name=Diagrama de Objetos 20object.model.name.for.tree=Diagrama de Objetos: 21 22object.not.created=Objeto não pôde ser criado. 23class.not.created=Classe não pôde ser criada. 24component.not.created=Componente não pôde ser criado. 25port.not.created=Porta não pôde ser criada. 26interface.not.created=Interface não pôde ser criada. 27node.not.created=Nodo não pôde ser criado. 28label.not.created=Rotulo não pôde ser criado. 29 30actor=ACTOR 31new.actor=Novo ator 32 33label=LABEL 34new.label=Novo rotulo 35 36message=MESSAGE 37new.message=Nova mensagem </pre>
--	---

Figura 7.32: Arquivos de tradução criados no ambiente SEA.

Por exemplo, se a língua desejada para a internacionalização for o alemão (“de”) da Dinamarca (“DK”), o arquivo a ser criado deve ter o seguinte nome: *I18NProperties_de_DK.properties*. Após isto, é necessário sobrescrever todas as chaves já existentes dando-as valores traduzidos

na língua escolhida. A figura 7.32 ilustra o início dos dois arquivos de tradução criados para o ambiente SEA no decorrer deste trabalho.

7.5 Modificações extras

À medida que a extensão foi construída, alguns inconvenientes com relação a interface do ambiente SEA íam surgindo. Apesar de não ser o foco do trabalho, algumas modificações foram necessárias para a melhoria da produtividade do desenvolvimento do trabalho. Dentre as principais alterações na interface pode-se destacar:

- *Tela principal maximizada*: Antes do presente trabalho, o ambiente SEA era inicializado por *default* em resolução 800x600 e a tela não ficava centralizada. Sendo assim, usuários de resoluções maiores sempre necessitavam clicar no botão de maximizar para conseguir trabalhar no ambiente. Isto se tornou demasiado custoso no momento em que se testava o desenvolvimento da extensão, uma vez que a todo momento era necessário maximizar. Sendo assim, optou-se por deixar a tela por padrão maximizada e na resolução de tela do sistema operacional do usuário. Isto é, se o usuário tem a resolução de tela 1024x768, ele verá o ambiente SEA aberto nessa resolução ao iniciar.
- *Editores maximizados*: Os editores de cada diagrama não vinham maximizados por padrão. Eles abriam em resolução 300x300. Esta característica se tornou muito inconveniente, pois dificilmente se cria um diagrama somente neste espaço, então toda vez que o usuário necessitasse criar um diagrama, precisaria clicar no botão de maximizar. Desta maneira, optou-se por deixar todos os editores maximizados por *default*.
- *Diálogo de escolha de especificação melhorado*: Quando um usuário clica para criar uma nova especificação, uma diálogo é aberto apresentando-se todos os tipos de especificações existentes. Entretanto, este diálogo não era centralizado em relação a tela e a listagem de especificações era trazida em um campo de seleção com uma altura extremamente pequena, dificultando a seleção. Além disso, o botão “Cancelar” estava com o texto interno cortado ao meio. Sendo assim, centralizou-se a tela de criação de especificação, aumentou-se o diálogo de listagem de especificações e aumentou-se a largura do botão de cancelamento.

8 *Conclusão*

Conhecendo as necessidades encontradas no mercado e em disciplinas acadêmicas de análise de projeto e engenharia de software, o trabalho de conclusão de curso realizado propôs uma nova versão em Java do ambiente SEA, mais robusta e consistente que a versão original. Para isso, foram inseridas novas técnicas e elementos sintáticos da modelagem UML 2. A nova versão possibilitou uma validação mais precisa e confiável do framework OCEAN, uma vez que está mais estável e abrangente.

Como o trabalho consistiu em representar uma linguagem de notação gráfica, diversas imagens estiveram presentes para fins ilustrativos. Desta forma, é importante destacar que a maioria dos diagramas apresentados foram projetados pela autora utilizando o ambiente SEA. Os diagramas de visão geral de integração e de temporização, presentes no capítulo 3, foram modelados utilizando as ferramentas JUDE e Visual Paradigm, respectivamente, em virtude do ambiente SEA ainda não ter suporte a estes diagramas. Além destes, os diagramas de classes inseridos no capítulo 7, que mostram a arquitetura de classes criadas para a construção de cada modelo, também não foram modelados no ambiente SEA e sim pela ferramenta JUDE, uma vez que necessitaram de elementos sintáticos com cores, característica ainda não apresentada no ambiente SEA. Nenhum diagrama exemplificado foi extraído de outros trabalhos, todos foram desenhados pela autora em ferramentas de modelagem. A maioria destes diagramas também foram idealizados pela autora. Alguns deles foram inspirados em exemplos da obra de Ricardo Pereira e Silva (SILVA, 2007), mas apresentam citações. Os novos ícones de conceitos de UML inseridos no ambiente SEA foram desenhados com o auxílio da ferramenta GIMP e Microsoft Paint. As demais imagens e ícones presentes foram desenhadas na ferramenta Inkscape e muitas destas imagens também foram criadas e produzidas pela autora.

A ferramenta Eclipse Europa, ambiente de desenvolvimento integrado *open source*, foi utilizada para a codificação em linguagem Java da extensão do ambiente SEA.

Este trabalho seguiu os padrões da ABNT (Associação Brasileira de Normas Técnicas) e foi desenvolvido em $\text{L}^{\text{A}}\text{T}_{\text{E}}\text{X}$, uma linguagem de marcação de documento.

8.1 Comparativo entre ferramentas CASE e o ambiente SEA

No capítulo 3, pôde-se acompanhar um comparativo entre algumas ferramentas CASE, julgadas como as mais populares. No quadro 8.1, tem-se o mesmo comparativo com a inserção do ambiente SEA como ferramenta de comparação.

Tabela 8.1: Quadro comparativo entre as ferramentas, incluindo o Ambiente SEA.

		FERRAMENTAS CASE							
CARACTERÍSTICAS		Argo	Umbrello	NetBeans	JUDE	Rational	EA	VP	SEA
1	Plataforma Windows	✓	✗	✓	✓	✓	✓	✓	✓
2	Plataforma Linux	✓	✓	✓	✓	✓	✗	✓	✓
3	Dicionário unificado de dados	✗	✗	✗	✗	✓	✓	✓	✗
4	Compartilhamento do repositório	✗	✗	✓	✗	✓	✓	✓	✗
5	Permissões e grupos de usuários	✗	✗	✗	✗	✓	✓	✓	✗
6	Prototipação de telas	✗	✗	✗	✗	✗	✓	✓	✗
7	Processos de negócio	✗	✗	✗	✗	✓	✓	✓	✗
8	Geração de código	✓	✓	✓	✗	✓	✓	✓	✓
9	Engenharia reversa	✓	✗	✓	✗	✓	✓	✓	✓
10	Proprietária	✗	✗	✗	✗	✓	✓	✓	✗
11	Livre	✓	✓	✓	✓	✗	✗	✗	✗
12	Gratuita	✓	✓	✓	✓	✗	✗	✗	✓
13	UML 1	✓	✓	✓	✓	✓	✓	✓	✓
14	UML 2	✗	✗	✗	✗	✓	✓	✓	✓
15	Armazenamento XMI	✓	✓	✓	✗	✓	✓	✓	✗
16	Controle de Versões	✗	✗	✓	✗	✓	✓	✓	✗
17	Relatórios	✗	✗	✓	✗	✓	✓	✓	✗

LEGENDA:

✓ Presença da característica

✗ Ausência da característica

Observando-se o quadro comparativo, é possível concluir que o ambiente SEA ainda não é uma ferramenta com tantos recursos como outras ferramentas pagas, uma vez que ainda é um protótipo sob desenvolvimento. Além disso, muitas das funcionalidades que provêm um diferencial ao ambiente SEA ainda estão em versão em linguagem Smalltalk, necessitando de uma futura conversão para a linguagem Java. Por outro lado, atenta-se que o ambiente SEA é uma entre poucas ferramentas com suporte à UML 2, foco do presente trabalho. Contudo, na versão atual em Java, ele ainda não se sobressai em relação às demais ferramentas.

8.2 Resultados Obtidos

Tratando os resultados obtidos de maneira quantitativa, foi possível efetuar um levantamento das realizações adquiridas no desenvolvimento deste trabalho:

- 230 classes foram criadas e codificadas em linguagem de programação Java;
- 30 classes que já pertenciam ao ambiente SEA necessitaram de modificações;
- 280 ícones foram desenhados para as barras de ferramentas pertencentes aos modelos;
- 1 nova especificação orientada a objetos foi concebida;
- 8 modelos foram criados;
- 2 modelos foram corrigidos e melhorados;
- 85 conceitos foram criados;
- 5 conceitos foram aperfeiçoados;
- 2 arquivos de tradução foram desenvolvidos.

Em suma, todos estes itens compõem a principal realização deste trabalho: o desenvolvimento de uma extensão do ambiente SEA que fornece suporte à UML 2. Na versão atual, o ambiente possibilita a criação de onze tipos de diagramas. A fim de alcançar este objetivo, foi necessário:

- Conhecer e entender o funcionamento o framework OCEAN e o ambiente SEA de maneira aprofundada;
- Criar uma nova especificação orientada a objetos no ambiente SEA adequadamente;
- Conhecer os tipos de figuras gráficas que o framework JHotDraw, embutido no OCEAN, fornece suporte;
- Estudar a segunda versão da linguagem UML e os seus principais elementos sintáticos, fazendo um levantamento dos diagramas possíveis de serem implementados em tempo;
- Criar os modelos que representam os diagramas de UML levantados, juntamente com seus editores gráficos;
- Criar um conceito para cada elemento sintático que optou-se por implementar;

- Criar uma figura gráfica para cada conceito criado;
- Editar modelos, conceitos e figuras anteriores a este trabalho que não estavam apropriados;
- Criar e incluir novos ícones às barras de ferramentas presentes nos editores gráficos dos modelos.

8.3 Avaliação

O desenvolvimento de uma extensão para uma aplicativo não é uma tarefa trivial quando não se tem o conhecimento prévio das melhores formas de uso dos frameworks deste aplicativo. Assim, a implementação da extensão do ambiente SEA foi bastante penosa inicialmente, uma vez que não se sabia muito como usufruir do framework OCEAN. Foi necessário a leitura de muitos trabalhos relacionados e a participação de um treinamento de uso do framework, fornecido por Ademir Coelho (COELHO, 2007). Todavia, mesmo obtendo-se esta preparação, ainda encontraram-se diversas dificuldades, mas à medida que mais recursos eram implementados à extensão, mais domínio sobre o framework era adquirido.

O grande desafio deste trabalho era que este necessitava de várias características ainda não presentes na versão Java do ambiente SEA. Isto é, não existiam formas de implementação parecidas com as formas de implementação requeridas em determinadas funcionalidades, como a agregação de conceitos, por exemplo. Desta maneira, foi preciso realizar muitas pesquisas e tentativas de codificação, principalmente quando os desenvolvedores pioneiros do ambiente e do framework não conseguiam ajudar a resolver alguns problemas levantados. Este aspecto, atrelado à falta de experiência da autora e ao grande escopo deste trabalho, gerou a necessidade de um grande consumo de tempo para o desenvolvimento da extensão.

Uma das maiores limitações ao se desenvolver este trabalho foi a falta de documentação do framework JHotDraw, uma vez que a parte mais custosa foi a implementação das figuras que representavam os elementos sintáticos. Dos poucos tutoriais encontrados na internet sobre formas de utilização do JHotDraw, nenhum trouxe explicações palpáveis, apenas explicações simplórias. Era necessário que existissem manuais que exemplificassem maneiras de se criarem figuras complexas, aspecto que não costuma estar presente em comentários de código-fonte.

Sabendo que a parte gráfica deste trabalho foi que mais consumiu tempo por apresentar grande complexidade, alguns tratamentos de consistência tiveram que ser deixados de lado. Muitas restrições também não foram implementadas, mas anotações foram deixadas para que

se possa dar continuidade em novos trabalhos.

8.4 Trabalhos Futuros

Lembrando que são ao todo treze diagramas pertencentes à UML 2, a versão Java do ambiente SEA passou a dar suporte a onze diagramas com a implementação deste trabalho. Dos diagramas que já existiam antes da versão atual do ambiente, escrita em Java, o único que não foi criado ou modificado significativamente neste trabalho foi o diagrama de classes, por já apresentar as funcionalidades básicas. Assim, dois diagramas inéditos da UML 2 não foram cobertos por falta de tempo hábil: o diagrama de temporização e o diagrama de visão geral de interação. Ambos ficam sujeitos a serem implementados em trabalhos futuros.

Conforme comentado anteriormente, alguns tratamentos semânticos não puderam ser explorados ao máximo neste trabalho. É necessário que se faça um levantamento completo de todas as restrições necessárias aos diagramas e todas as verificações de consistência. Estes aspectos também podem ser inseridos nos próximos trabalhos relacionados ao ambiente SEA.

Convém ressaltar que seria proveitoso que se realizasse um *refactoring* no framework OCEAN e em algumas classes pertencentes à estrutura básica do ambiente SEA. Conforme apontado em capítulos anteriores, existe uma falta de padronização e documentação muito grande em classes antecedentes a este trabalho. Além disso, vários recursos da nova versão da linguagem Java não estão sendo desfrutados, causando uma queda de desempenho no aplicativo, uma vez que acumularam-se muitos *warnings* devido ao uso de recursos arcaicos. Outro aspecto a se considerar é a mesclagem de formas de escrita (ora em inglês, ora em português) que dificulta bastante a legibilidade do sistema como um todo, tornando-se um impasse para que novos desenvolvedores possam surgir e novas extensões possam existir.

Uma outra possibilidade que deve ser estudada para implementação em trabalhos futuros é a atualização do JHotDraw, que atualmente segue no framework OCEAN na versão 5.3. Infelizmente, o JHotDraw tornou-se incompatível com as versões anteriores na versão 7.0 (a mais atual em maio de 2008), e uma migração para esta versão no OCEAN causaria sérios danos a figuras existentes e em muitas desenvolvidas neste trabalho, mas em contrapartida traria inúmeras novas funcionalidades e características. Acredita-se que seria o caso de se realizar apenas uma atualização e não uma migração completa.

8.5 Considerações Finais

A principal motivação de implementar esta extensão ao ambiente SEA é a realização de uma melhor validação do framework OCEAN e a busca por tornar o ambiente SEA cada vez mais adequado à utilização em disciplinas acadêmicas. O trabalho realizado dispõe de um protótipo em Java da nova versão do ambiente SEA, mais abrangente que a versão original, com suporte às técnicas de UML 2.

Considera-se que este trabalho seja de grande importância para outros trabalhos que venham a realizar manutenções e melhorias no ambiente SEA, uma vez que muitas características novas foram implementadas e estas poderão servir de exemplo a futuras extensões e correções. Além disso, contribuiu-se à difusão de abordagens voltadas ao reuso, que geram aumento de produtividade e qualidade no desenvolvimento de software.

APÊNDICE A – Cookbook para criação de documentos gráficos

Em seu trabalho de conclusão de curso, João de Amorim (AMORIM, 2006) desenvolveu um *cookbook* para criação de editores gráficos sob o framework OCEAN, descrevendo os passos a serem realizados para a obtenção de um novo editor. O objetivo da concepção deste *cookbook* era que todo novo desenvolvimento de diagrama pudesse seguir os passos do *cookbook* a fim de que a criação se tornasse mais rápida e que cobrisse todas as características básicas necessárias para o seu correto funcionamento.

Este apêndice apresenta uma versão mais atualizada do *cookbook* de João de Amorim trazendo novos comentários sobre os passos que puderam ser vivenciados na criação dos documentos gráficos deste trabalho de conclusão de curso.

Assim, a seguir serão transcritos os passos necessários identificados para que seja suportado um novo diagrama.

A.1 Passo 1: Criação da especificação de projeto

A fim de criar uma nova especificação de projeto no ambiente SEA, deve-se criar uma nova classe que herda da classe *Specification*. Os métodos que devem ser sobrescritos são listados a seguir.

1. **Método *getId()***: Deve retornar o identificador da nova especificação.
2. **Método *getIdType()***: Deve retornar o tipo do identificador da nova especificação.
3. **Método *getNameForTree()***: Deve retornar um nome para a especificação que pertencerá à árvore de especificações presente na interface.

4. **Método *modeloLista()***: Deve retornar uma lista dos modelos que estarão presentes no diagrama - a forma de criação deste modelos poderá ser vista no próximo passo. Um exemplo da sobrescrição deste método, utilizado na criação da especificação UML 2 deste trabalho, seria:

Listagem A.1: Exemplo de implementação do método **modeloLista()**.

```

1  /*
2   * (non-Javadoc)
3   *
4   * @see ocean.framework.Specification#modeloLista()
5   */
6  @SuppressWarnings( { "deprecation", "unchecked" })
7  public OceanVector modeloLista()
8  {
9      OceanVector aList = super.modeloLista();
10     aList.add(ObjectDiagram.class);
11     aList.add(PackageDiagram.class);
12     aList.add(ComponentDiagram.class);
13     aList.add(DeploymentDiagram.class);
14     aList.add(CommunicationDiagram.class);
15     aList.add(CompositeStructureDiagram.class);
16     aList.add(StateMachineDiagram.class);
17     aList.add(ActivityDiagram.class);
18     return aList;
19 }

```

Caso não haja necessidade de se criar uma nova especificação, é possível modificar a classe *ocean.documents.oo.specifications.SEASpecification* e adicionar os modelos necessários, criados no próximo passo do *cookbook*, no método *modeloLista()*.

A.2 Passo 2: Criação do modelo

Tratando-se de UML, um modelo pode ser definido como uma representação conceitual de um diagrama. Em termos de código, um modelo é uma classe em Java que herda de *ocean.framework.ConceptualModel*. Os métodos que devem ser sobrescritos em um modelo são apontados abaixo.

1. **Método *initialize()***: Deve inicializar a lista *elementKeeperList* com os conceitos associados ao modelo em questão. A forma de criação destes conceitos será vista no próximo

passo. Um exemplo de sobrescrição deste método, utilizado neste trabalho, pode ser observado a seguir.

Listagem A.2: Exemplo de implementação do método **initialize()**.

```

1  /*
2   * (non-Javadoc)
3   *
4   * @see ocean.framework.ConceptualModel#initialize()
5   */
6  @SuppressWarnings( { "deprecation", "unchecked" })
7  public void initialize ()
8  {
9      elementKeeperList = new OceanVector ();
10     elementKeeperList.add( SpecificationElement.construtor (
11         InstanceSpecification.class ));
12     elementKeeperList.add( SpecificationElement.construtor (
13         AssociacaoBinaria.class ));
14     elementKeeperList.add( SpecificationElement.construtor (Heranca.
15         class ));
16     elementKeeperList.add( SpecificationElement.construtor (Agregacao.
17         class ));
18     elementKeeperList.add( SpecificationElement.construtor (Composition.
19         class ));
20     elementKeeperList.add( SpecificationElement.construtor (Dependency.
21         class ));
22     elementKeeperList.add( SpecificationElement.construtor (Realization.
23         class ));
24 }

```

2. **Método *conceitoLista()***: Deve retornar uma lista com os conceitos que compõem o modelo em questão. Um exemplo de sobrescrição deste método, utilizado neste trabalho, pode ser observado a seguir.

Listagem A.3: Exemplo de implementação do método **conceitoLista()**.

```

1  /*
2   * (non-Javadoc)
3   *
4   * @see ocean.framework.ConceptualModel#conceitoLista()
5   */
6  @SuppressWarnings( { "deprecation", "unchecked" })
7  public OceanVector<Class> conceitoLista ()

```



```

8  {
9      OceanVector<Class> aList = new OceanVector<Class>();
10     aList.add(InstanceSpecification.class);
11     aList.add(AssociacaoBinaria.class);
12     aList.add(Agregacao.class);
13     aList.add(Composition.class);
14     aList.add(Heranca.class);
15     aList.add(Dependency.class);
16     aList.add(Realization.class);
17     return aList;
18 }

```

3. **Método *modelName()***: Deve retornar o nome do modelo.
4. **Método *getNameForTree()***: Deve retornar um nome para o modelo o qual pertencerá à árvore de especificações presente na interface.
5. **Método *createEmptyDrawing()***: Retorna um novo objeto da classe que representa a *Drawing* específica do modelo em questão - a forma de criação de uma classe *Drawing* poderá ser vista no passo 4.1. Um exemplo da sobrescrição deste método, utilizado neste trabalho, seria:

Listagem A.4: Exemplo de implementação do método **createEmptyDrawing()**.

```

1  /*
2   * (non-Javadoc)
3   *
4   * @see ocean.framework.ConceptualModel#createEmptyDrawing()
5   */
6  @Override
7  public SpecificationDrawing createEmptyDrawing()
8  {
9      return new ObjectDiagramDrawing();
10 }

```

A.3 Passo 3: Criação dos conceitos

Tratando-se de UML, um conceito pode ser definido como uma representação conceitual de um elemento sintático presente em um diagrama de UML. Em termos de código, um conceito é uma classe que herda de *ocean.framework.Concept*. As classes dos conceitos devem definir seus

respectivos métodos construtores, chamando o construtor da superclasse *Concept*. Os métodos que devem ser sobrescritos em um conceito são apresentados a seguir.

1. **Método *conceptName()***: Deve retornar o nome do conceito.
2. **Método *getNameForTree()***: Deve retornar um nome para o conceito o qual pertencerá à árvore de especificações presente na interface.
3. **Método *mustHaveName()***: Deve retornar `false`, caso não se queira que o conceito tenha um nome. Em caso contrário, quando é requerido que o conceito tenha nome, não é necessário sobrescrever este método, pois ele retorna `true` por *default*.
4. **Método *mustHavePoint()***: Para definir o armazenamento de especificações previamente criadas, ao se salvar um modelo criado em um editor de diagrama, são salvos os pontos das figuras que representam cada conceito, para que posteriormente, em uma restauração do ambiente, seja possível redesenhar as figuras corretamente. No caso de conceitos que são representados por figuras do tipo linha, não é necessário salvar o ponto onde tal figura se encontra, no momento do salvamento. Por *default*, os conceitos não necessitam ter pontos. Assim, este método deve retornar `true` quando deseja-se definir que o ponto da figura que representa um conceito deve ser salvo no armazenamento de especificações. Caso contrário, não é necessário sobrescrever este método.

Quando um conceito representa uma ligação entre dois outros conceitos, como no caso de um fluxo de controle em um diagrama de atividades, devem ser definidos construtores que recebam como parâmetros os conceitos ligados pelo conceito em questão. Observe a seguir um exemplo de sobrecarga de construtores feita na implementação do diagrama de atividades do presente trabalho. O construtor sobrecarregado foi criado a fim de prover o fluxo de controle entre o nodo inicial e uma atividade.

Listagem A.5: Sobrecarga de construtores em conceitos.

```

1  /**
2   * Official constructor.
3   */
4  public ControlFlow ()
5  {
6     super ();
7     name(I18NProperties.getString("control.flow.concept.name"));
8  }
9
```

```

10  /**
11   * Overloaded constructor.
12   *
13   * @param initialNode
14   * @param activity
15   */
16  public ControlFlow(InitialNode initialNode , Activity activity)
17  {
18      this ();
19      this . setInitialTerminal1 ( initialNode );
20      this . setActivityTerminal2 ( activity );
21  }

```

A.4 Passo 4: Criação das classes gráficas

Após ter-se criado as classes conceituais do novo diagrama, é preciso conceber as classes gráficas, que darão o suporte visual aos modelos e aos conceitos. São ao todo quatro classes gráficas necessárias ao diagrama. Nas próximas subseções, serão apresentados os passos para a concepção destas classes.

A.4.1 Passo 4.1: Criação da classe *Drawing*

As classes *Drawing* auxiliam na criação das figuras, recebendo conceitos associados a uma figura específica. São muito úteis para a criação de diagramas a partir de um modelo existente (com conceitos previamente criados). Assim, para compor o diagrama, deve-se criar uma classe *Drawing* que estende da classe *ocean.jhotdraw.SpecificationDrawing*. Os métodos que devem ser sobrescritos nesta classe são:

1. **Método *createDesiredFigureForConcept(Concept aComponent)***: Deve definir a criação das figuras dos conceitos associados ao diagrama, exceto para os conceitos cuja figura de representação é uma linha. Segue um exemplo de sobrescrição deste método no trecho de código abaixo.

Listagem A.6: Exemplo de implementação do método ***createDesiredFigureForConcept()***

```

1  /*
2   * (non-Javadoc)
3   *

```

```

4  * @see ocean.jhotdraw.SpecificationDrawing#
      createDesiredFigureForConcept(ocean.framework.Concept)
5  */
6  @Override
7  public SpecificationCompositeFigure createDesiredFigureForConcept(
      Concept component)
8  {
9      SpecificationCompositeFigure figure = null;
10     if (component instanceof documents.concepts.package_diagram.
      Package)
11     {
12         figure = new PackageFigure();
13     }
14
15     return figure;
16 }

```

2. **Método *createDesiredLineForConcept(Concept aComponent)***: Deve definir a criação apenas das figuras dos conceitos associados ao diagrama, cuja figura de representação é uma linha. Observe um exemplo de sobrescrição deste método no trecho de código que segue:

Listagem A.7: Exemplo de implementação do método **createDesiredLineForConcept()**

```

1  /*
2   * (non-Javadoc)
3   *
4   * @see ocean.jhotdraw.SpecificationDrawing#
      createDesiredLineForConcept(ocean.framework.Concept)
5   */
6  public SpecificationLineFigure createDesiredLineForConcept(Concept
      aComponent)
7  {
8      SpecificationLineFigure aLine = null;
9
10     String concNameHeranca = Concept.conceptName(Heranca.class);
11     String concNameAssociacao = Concept.conceptName(AssociacaoBinaria.
      class);
12
13     if (aComponent.conceptName().equals(concNameAssociacao))
14     {
15         aLine = new MObjRelationshipLineFigure();
16     }

```

```

17     else if ( aComponent . conceptName () . equals ( concNameHeranca ))
18     {
19         aLine = new MObjInheritanceLineFigure () ;
20     }
21
22     return aLine ;
23 }

```

3. **Método *startFigureFor(Concept aComponent)***: Deve retornar a figura do conceito de partida do conceito representado no diagrama, que liga dois outros conceitos. Observe um exemplo de sobrescrição deste método no trecho de código a seguir.

Listagem A.8: Exemplo de implementação do método **startFigureFor()**

```

1  /*
2  * (non-Javadoc)
3  *
4  * @see ocean.jhotdraw.SpecificationDrawing#startFigureFor(ocean.
5  *     framework . Concept )
6  */
7  public SpecificationCompositeFigure startFigureFor ( Concept aComponent )
8  {
9      Concept auxComp = null ;
10     String concNameHeranca = Concept . conceptName ( Heranca . class ) ;
11     String concNameAssociacao = Concept . conceptName ( AssociacaoBinaria .
12         class ) ;
13
14     if ( aComponent . conceptName () . equals ( concNameAssociacao ) )
15     {
16         auxComp = ( ( AssociacaoBinaria ) aComponent ) . classeTerminal1 () ;
17     }
18     else if ( aComponent . conceptName () . equals ( concNameHeranca ) )
19     {
20         auxComp = ( ( Heranca ) aComponent ) . subclasse () ;
21     }
22
23     return ( SpecificationCompositeFigure ) getFigureOfConcept ( auxComp ) ;
24 }

```

4. **Método *stopFigureFor(Concept aComponent)***: Deve retornar a figura do conceito de chegada do conceito representado no diagrama, que liga dois outros conceitos. Veja um exemplo de sobrescrição deste método no trecho de código a seguir.

Listagem A.9: Exemplo de implementação do método **stopFigureFor()**

```

1  /*
2   * (non-Javadoc) mponent.conceptName().equals(concNameAssociacao))
3   *
4   * @see ocean.jhotdraw.SpecificationDrawing#stopFigureFor(ocean.
5   *   framework.Concept)
6   */
7  public SpecificationCompositeFigure stopFigureFor(Concept aComponent)
8  {
9      Concept auxComp = null;
10     String concNameHeranca = Concept.conceptName(Heranca.class);
11     String concNameAssociacao = Concept.conceptName(AssociacaoBinaria.
12         class);
13
14     if (aComponent.conceptName().equals(concNameAssociacao))
15     {
16         auxComp = ((AssociacaoBinaria) aComponent).classeTerminal2();
17     }
18     else if (aComponent.conceptName().equals(concNameHeranca))
19     {
20         auxComp = ((Heranca) aComponent).superclasse();
21     }
22     return (SpecificationCompositeFigure) getFigureOfConcept(auxComp);
23 }

```

A.4.2 Passo 4.2: Criação da classe DrawingView

Também é necessário criar uma classe que estende de *ocean.jhotdraw.SpecificationDrawingView*. Apenas os construtores da superclasse devem ser sobrescritos, com parâmetros específicos para o diagrama em questão. Observe abaixo um exemplo de implementação utilizado na construção do diagrama de atividades deste trabalho:

Listagem A.10: Exemplo de implementação dos construtores da DrawingView.

```

1  /**
2   * Constructor.
3   *
4   * @param editor
5   */
6  public ActivityDiagramDrawingView(ActivityDiagramEditor editor)

```

```

7 {
8     super( editor );
9 }
10
11 /**
12  * Constructor .
13  *
14  * @param editor
15  * @param width
16  * @param height
17  */
18 public ActivityDiagramDrawingView( ActivityDiagramEditor editor , int width ,
19     int height )
20 {
21     super( editor , width , height );
22 }

```

Na superclasse *SpecificationDrawingView* é definido o método *updateConcepts()*, para que quando fosse necessário (após a remoção de uma figura em um editor, por exemplo) os conceitos fossem atualizados. Cada *DrawingView* específica para um novo diagrama, pode sobrescrever este método para realizar as ações necessárias para atualizar os conceitos daquele diagrama. A sobrescrição deste método é opcional, e depende da necessidade do diagrama que está sendo criado. Segue um trecho de código que exemplifica a sobrescrição deste método.

Listagem A.11: Exemplo de sobrescrição do método **updateConcepts()**.

```

1  /*
2  * (non-Javadoc)
3  *
4  * @see ocean.jhotdraw.SpecificationDrawingView#updateConcepts()
5  */
6  public void updateConcepts ()
7  {
8      (( ScenarioDrawing ) this . drawing () ) . makeOrderDictionary () ;
9      (( ScenarioDrawing ) this . drawing () ) . setMessageOrder ( 0 ) ;
10     (( ScenarioDrawing ) this . drawing () ) . updateScenarioDrawing ( true ) ;
11 }

```

A.4.3 Passo 4.3: Criação da classe Window

Outra classe que deve ser criada para compor a estrutura gráfica de um diagrama é a *Window*. Esta classe deve estender de *ocean.jhotdraw.EditorAuxWindow* e deve implementar a interface *ocean.smalltalk.IComunicacao*. Os métodos que devem ser sobrescritos nesta classe são apontados e detalhados abaixo.

1. **Método *getSelected()***: Deve retornar o elemento selecionado na *DrawingView* do diagrama, que é exibida nesta *Window*. Um exemplo de sobrescrição deste método seria:

Listagem A.12: Exemplo de implementação do método **getSelected()**.

```

1  /*
2   * (non-Javadoc)
3   *
4   * @see ocean.jhotdraw.EditorAuxWindow#getSelected()
5   */
6  public Serializable getSelected()
7  {
8      return getDrawingView().getSelected();
9  }

```

2. **Método *show()***: Deve definir a janela como visível. Um exemplo de sobrescrição deste método seria:

Listagem A.13: Exemplo de implementação do método **show()**.

```

1  /*
2   * (non-Javadoc)
3   * @see ocean.jhotdraw.EditorAuxWindow#show()
4   */
5  public void show()
6  {
7      this.setVisible(true);
8  }

```

3. **Método *setDocument(Object obj)***: Deve definir qual o documento que será exibido naquela *Window*. Um exemplo de sobrescrição deste método seria:

Listagem A.14: Exemplo de implementação do método **setDocument()**.

```

1  /*

```



```

2  * (non-Javadoc)
3  *
4  * @see ocean.jhotdraw.EditorAuxWindow#setDocument(java.lang.Object)
5  */
6  public void setDocument(Object obj)
7  {
8      this.handledElement((ConceptualModel) obj);
9      ((PackageDiagramEditor) getEditor()).setDocument(obj);
10     this.getEditor().view().setDrawing(((ConceptualModel) obj).drawing
11         ());
12     this.setDrawing((SpecificationDrawing) getEditor().view().drawing
13         ());
14 }

```

4. **Método *getDocument()***: Deve retornar o elemento tratado através do método *handledElement*, da classe *AuxWindowAppModel*. Um exemplo de sobrescrição deste método seria:

Listagem A.15: Exemplo de implementação do método **getDocument()**.

```

1  /**
2  * (non-Javadoc)
3  *
4  * @see ocean.jhotdraw.EditorAuxWindow#getDocument()
5  */
6  public Object getDocument()
7  {
8      return handledElement();
9  }

```

O construtor da classe *Window* sendo criada deve inicializar o editor e a classe *DrawingView* associados ao modelo em questão. Observe a seguir um exemplo de criação deste construtor, utilizado no diagrama de pacotes implementado no presente trabalho.

Listagem A.16: Exemplo de implementação do construtor de uma classe **Window**.

```

1  /**
2  * Official constructor.
3  */
4  public PackageDiagramWindow()
5  {
6      super(new PackageDiagramEditor());

```

```

7     setEditor(new PackageDiagramaEditor());
8     setDrawingView((SpecificationDrawingView) getEditor().view());
9 }

```

A.4.4 Passo 4.4: Criação da classe Editor

É necessário conceber um editor a fim de apresentar ao usuário as ferramentas disponíveis pelo modelo em questão. Assim, deve-se criar uma classe que estende de *ocean.jhotdraw.SpecificationEditor*. Devem ser sobrescritos os dois construtores da superclasse, mas os parâmetros devem ser específicos do modelo. No corpo de cada método construtor, deve ser chamado o construtor correspondente da superclasse. Um exemplo de implementação pode ser conferido abaixo:

Listagem A.17: Exemplo de implementação de construtores de um editor.

```

1 /**
2  * Official constructor.
3  */
4 public PackageDiagramEditor()
5 {
6     super();
7 }
8
9 /**
10 * Overloaded constructor.
11 *
12 * @param view
13 */
14 public PackageDiagramEditor(PackageDiagramDrawingView view)
15 {
16     super(view);
17 }

```

Os métodos que devem ser sobrescritos em um editor são apontados e detalhados na sequência.

1. **Método *createDrawingViewObject()***: Deve retornar uma nova *DrawingView* específica do modelo em questão. Este método serve para criar a instância de *DrawingView* usada no editor. Um exemplo de sobrescrição deste método seria:

Listagem A.18: Exemplo de implementação do método **createDrawingViewObject()**.

```

1  /*
2   * (non-Javadoc)
3   *
4   * @see ocean.jhotdraw.SpecificationEditor#createDrawingViewObject()
5   */
6  @Override
7  protected SpecificationDrawingView createDrawingViewObject()
8  {
9      return new PackageDiagramDrawingView(this);
10 }

```

2. **Método *createSpecificationDrawing()***: Deve retornar uma nova *Drawing* específica do modelo em questão. Este método serve para criar a instância de *Drawing* usada no editor. Um exemplo de sobrescrição deste método seria:

Listagem A.19: Exemplo de implementação do método **createSpecificationDrawin()**.

```

1  /*
2   * (non-Javadoc)
3   *
4   * @see ocean.jhotdraw.SpecificationEditor#createSpecificationDrawing
5     (*)
6   */
7  @Override
8  protected SpecificationDrawing createSpecificationDrawing()
9  {
10     return new PackageDiagramDrawing();

```

3. **Método *createTools()***: Deve acrescentar as ferramentas específicas do diagrama. Cada figura do modelo em questão deve ter uma ferramenta de criação associada e definida no editor. Na criação de cada ferramenta é definida a qual figura ela está associada. Dentro do corpo da classe é criada a nova ferramenta. É através dos métodos que tratam eventos do mouse que as classes que implementam a interface *Tool* permitem que sejam definidas quais ações devem ser executadas quando determinado evento do mouse ocorrer. No caso do método *mouseUp*, deve-se primeiramente requisitar ao framework OCEAN que crie (ou modifique) o conceito associado a determinada figura que se deseja criar (ou modificar). Se o OCEAN, por algum motivo, não conseguir executar a operação, então a figura não deve ser criada, pois em OCEAN, uma figura só existe se associada a um conceito. Caso seja possível a criação da figura, define-se o conceito desta figura criada

como sendo o conceito que compõe o modelo. A figura criada é adicionada na lista de observadores do conceito que ela representa. A figura então pode ser desenhada no editor. Um exemplo de sobrescrição do método *createTools()*, utilizado na implementação do diagrama de pacotes deste trabalho, seria:

Listagem A.20: Exemplo de implementação do método **createTools()**.

```

1  /*
2   * (non-Javadoc) ackage
3   *
4   * @see ocean.jhotdraw.SpecificationEditor#createTools(javax.swing.
      JToolBar)
5   */
6  @Override
7  protected void createTools(JToolBar palette)
8  {
9      super.createTools(palette);
10
11     // PACKAGE
12     Tool tool = new CreationTool(this, new PackageFigure())
13     {
14
15         /*
16          * (non-Javadoc)
17          *
18          * @see CH.ifa.draw.standard.CreationTool#mouseUp(java.awt.
              event.MouseEvent, int, int)
19          */
20         public void mouseUp(MouseEvent e, int x, int y)
21         {
22             Concept conc = getModel().createComponent(
23                 documents.concepts.package_diagram.Package.class);
24
25             if (conc == null)
26             {
27                 I18NPrinter.printMessage("package.not.created");
28             }
29             else
30             {
31                 PackageFigure packageFig = ((PackageFigure)
32                     getCreatedFigure());
33                 packageFig.concept(conc);
34                 conc.getObservable().addObserver(packageFig);

```

```

34         conc.redraw();
35     }
36     super.mouseUp(e, x, y);
37 }
38
39 };
40
41 ToolButton tB = createToolButton(FG.PACKAGE.DIAGRAM.PATH
42     + I18NProperties.getString("package"), I18NProperties.
43     getString("new.package"),
44     tool);
45 tB.setEnabled(true);
46 palette.add(tB);
47 }

```

A.5 Passo 5: Criação das figuras

Através de figuras do framework JHotDraw, é possível representar visualmente os conceitos relacionados a um modelo. Uma figura é uma classe em Java que, no caso de ser uma composição de diversas figuras, estende a classe *SpecificationCompositeFigure*, e que, no caso de representar um conceito de relação (como associação binária), deve estender a classe *SpecificationLineFigure*.

Podem ser definidos diferentes métodos construtores para as figuras: sem parâmetros, chamando apenas o construtor da superclasse; recebendo um conceito como parâmetro, e chamando um construtor da superclasse que recebe como parâmetro um conceito; entre outros, de acordo com a necessidade. Observe o exemplo a seguir, utilizado no diagrama de componentes implementado no presente trabalho, que cria dois tipos de construtores.

Listagem A.21: Exemplo de sobrecarga de construtores em uma figura.

```

1  /**
2   * Default constructor.
3   */
4  public PortFigure()
5  {
6      super(new RectangleFigure());
7  }
8
9  /**

```

```

10 * Overloaded constructor.
11 *
12 * @param port
13 */
14 public PortFigure (Port port)
15 {
16     super (new RectangleFigure () , port);
17 }

```

Os métodos que devem ser sobrescritos em uma figura que representa um conceito pertencente a um modelo são detalhados a seguir.

1. **Método *initialize()***: Deve inicializar as características iniciais da figura. Após estas definições, deve-se fazer uma chamada ao método *initialize()* da superclasse. Um exemplo de sobrescrição deste método, utilizado neste trabalho, pode ser observado a seguir.

Listagem A.22: Exemplo de implementação do método ***initialize()***.

```

1 /*
2 * (non-Javadoc)
3 *
4 * @see ocean.jhotdraw.SpecificationCompositeFigure#initialize ()
5 */
6 public void initialize ()
7 {
8     removeAll ();
9
10    setAttribute (Figure .POPUP_MENU, createPopupMenu ());
11
12    super . initialize ();
13 }

```

2. **Método *relatedConceptClass()***: Deve retornar a classe do conceito associado a figura sendo criada. Um exemplo de sobrescrição deste método pode ser conferido abaixo.

Listagem A.23: Exemplo de implementação do método ***relatedConceptClass()***.

```

1 /*
2 * (non-Javadoc)
3 *
4 * @see ocean.jhotdraw.SpecificationCompositeFigure#
   relatedConceptClass ()

```

```

5  */
6  @Override
7  public Class< ? extends Activity> relatedConceptClass ()
8  {
9      return Activity.class ;
10 }

```

3. **Método *handles()***: Deve retornar um vector com os *handles* específicos da figura. Um exemplo de uso é apontado no trecho de código que segue.

Listagem A.24: Exemplo de implementação do método **handles()**.

```

1  /*
2   * (non-Javadoc)
3   *
4   * @see ocean.jhotdraw.SpecificationCompositeFigure#handles ()
5   */
6  @SuppressWarnings ("unchecked")
7  public Vector handles ()
8  {
9      Vector handles = new Vector ();
10     BoxHandleKit.addHandles (this , handles );
11     return handles ;
12 }

```

4. **Método *redraw()***: Deve definir todos os valores dos elementos que compõem a figura principal e desenhar todas as figuras internas à figura principal. Um exemplo de implementação deste método, utilizado para criar a figura que representa o conceito de porto no diagrama de componentes, pode ser visualizado no trecho de código que segue.

Listagem A.25: Exemplo de implementação do método **redraw()**.

```

1  /*
2   * (non-Javadoc)
3   *
4   * @see ocean.jhotdraw.SpecificationCompositeFigure#redraw ()
5   */
6  public void redraw ()
7  {
8      removeAll ();
9
10     GraphicalCompositeFigure gFigure = new GraphicalCompositeFigure ();
11     gFigure.getLayouter ().setInsets (new Insets (10, 5, 10, 5));

```

```

12     gFigure.setAttribute("FillColor", new Color(0xE8E8E8));
13     gFigure.setAttribute("FrameColor", new Color(0x000000));
14     add(gFigure);
15
16     update();
17 }

```

As figuras que representam ligações (apenas linhas) podem ter decorações associadas a elas, caso se queira colocar setas no início da linha, por exemplo. Os métodos para se acrescentar decorações no início ou no término de uma figura de linha são o *setStartDecoration(LineDecoration l)* e o *setEndDecoration(LineDecoration l)*. Um exemplo de uso desde métodos é no construtor da figura que representa o conceito de realização. Observe o trecho de código a seguir.

Listagem A.26: Exemplo de uso de decorações em linhas.

```

1  /**
2   * Default constructor.
3   */
4  public RealizationLineConnection()
5  {
6     super();
7
8     setStartDecoration(null);
9     ArrowTip arrow = new ArrowTip(0.35, 20.0, 20.0);
10    arrow.setBorderColor(java.awt.Color.black);
11    arrow.setFillColor(Color.white);
12    setEndDecoration(arrow);
13    setEndDecoration(arrow);
14 }

```

A.6 Passo 6: Edição da classe ReferenceManager

Com o objetivo de acrescentar a nova janela criada ao dicionário de janelas de conceitos, é necessário associar a classe do modelo com a classe da janela (uma instância de *Window*) pertencente a este modelo. Isto é feito no método *initialize()* da classe *ocean.accessories.ReferenceManager*. Basta adicionar no hash *concModelWindowDict* a classe do modelo como chave e a classe da janela como valor. Observe o exemplo a seguir:

Listagem A.27: Edição da classe ReferenceManager.


```
1  /*
2   * Initializes the diagrams.
3   */
4  public void initialize ()
5  {
6      // ...
7      concModelWindowDict.at_put (ObjectDiagram.class , ObjectDiagramWindow.class);
8      // ..
9  }
```

ANEXO A – Artigo: A História de UML e seus diagramas

Abstract. *This paper describes the UML history since the decade of 1990 so far today. It presents the organization of the thirteen UML diagrams, classifying them in structural and behavioral diagrams. The four documents belonging to the specification are cited and explained as well. Finally, each UML 2 diagram is described in detail.*

Resumo. *Este artigo descreve a história de UML desde a década de 1990 até o momento atual. Apresenta-se a organização dos treze diagramas de UML, classificando-os em diagramas estruturais e comportamentais. Os quatro documentos pertencentes à especificação também são mencionados e explicados. Por fim, cada diagrama de UML 2 é descrito em detalhes.*

A.1 Introdução

A modelagem de software é a atividade de construir modelos que expliquem as características ou o comportamento de um software ou de um sistema de software. Na construção do software os modelos podem ser usados na identificação das características e funcionalidades que o software deverá prover (análise de requisitos), e no planejamento de sua construção. Frequentemente a modelagem de software usa algum tipo de notação gráfica e são apoiados pelo uso de ferramentas.

A modelagem de software normalmente implica a construção de modelos gráficos que simbolizam os artefatos dos componentes de software utilizados e os seus interrelacionamentos. Uma forma comum de modelagem de programas orientados a objeto é através da linguagem unificada UML.

A UML (Unified Modeling Language) é uma linguagem para especificação, documentação, visualização e desenvolvimento de sistemas orientados a objetos. Sintetiza os principais métodos existentes, sendo considerada uma das linguagens mais expressivas para modelagem de sistemas orientados a objetos. Por meio de seus diagramas é possível representar sistemas de

softwares sob diversas perspectivas de visualização. Facilita a comunicação de todas as pessoas envolvidas no processo de desenvolvimento de um sistema - gerentes, coordenadores, analistas, desenvolvedores - por apresentar um vocabulário de fácil entendimento.

A.2 História

No início da utilização do paradigma de orientação a objetos, diversos métodos foram apresentados para a comunidade. Chegaram a mais de cinquenta entre os anos de 1989 a 1994, porém a maioria deles cometeu o erro de tentar estender os métodos estruturados da época. Com isso, os maiores prejudicados foram os usuários que não conseguiam encontrar uma maneira satisfatória de modelar seus sistemas. Foi a partir da década de 90 que começaram a surgir teorias que procuravam trabalhar de forma mais ativa com o paradigma da orientação a objetos. Diversos autores renomados contribuíram com publicações de seus respectivos métodos.

Por volta de 1993 existiam três métodos que mais cresciam no mercado, eram eles: *Booch'93* de *Grady Booch*, *OMT-2* de *James Rumbaugh* e *OOSE* de *Ivar Jacobson*. Cada um deles possuía pontos fortes em algum aspecto. O *OOSE* possuía foco em casos de uso (*use cases*), *OMT-2* se destacava na fase de análise de sistemas de informação e *Booch'93* era mais forte na fase de projeto. O sucesso desses métodos foi, principalmente, devido ao fato de não terem tentado estender os métodos já existentes. Seus métodos já convergiam de maneira independente, então seria mais produtivo continuar de forma conjunta (SAMPAIO, 2007).

Em outubro de 1994, começaram os esforços para unificação dos métodos. Já em outubro de 1995, *Booch* e *Rumbaugh* lançaram um rascunho do “Método Unificado” unindo o *Booch'93* e o *OMT-2*. Após isso, *Jacobson* se juntou a equipe do projeto e o “Método Unificado” passou a incorporar o *OOSE*. Em junho de 1996, *Booch*, *Rumbaugh* e *Jacobson* lançaram a primeira versão de uma nova linguagem de notação diagramática batizada de UML, que reuniu os com os três métodos (FOWLER, 2003). Posteriormente, foram lançadas novas versões.

A *OMG (Object Management Group)* lançou uma *RFP (Request for Proposals)* para que outras empresas pudessem contribuir com a evolução de UML, chegando à versão 1.1. Após alcançar esta versão, a *OMG* passou a adotá-la como padrão e a se responsabilizar (através da *RTF – Revision Task Force*) pelas revisões. Atualmente, estas revisões são, de certa forma, controladas a não provocar uma grande mudança no escopo original. Ao observar as diferenças ocorridas com as versões atuais, é possível notar que de uma versão para a outra não há grande impacto. Esta é uma característica que facilita a disseminação mundial de UML.

A.3 Estrutura da especificação

A especificação de UML é composta por quatro documentos: infra-estrutura de UML (OMG, 2007a), superestrutura de UML (OMG, 2007b), *Object Constraint Language* (OCL) (OMG, 2006a) e intercâmbio de diagramas (OMG, 2006b).

- *Infra-estrutura de UML*: O conjunto de diagramas de UML é constituído por uma linguagem definida a partir de outra linguagem que define os elementos construtivos fundamentais. Esta linguagem que suporta a definição dos diagramas é apresentada no documento “Infra-estrutura de UML”.
- *Superestrutura de UML*: Documento que complementa o documento de infra-estrutura e que define os elementos da linguagem no nível do usuário.
- *Linguagem para Restrições de Objetos (OCL)*: Documento que apresenta a linguagem usada para descrever expressões em modelos UML, com pré-condições, pós-condições e invariantes.
- *Intercâmbio de diagramas de UML*: Apresenta uma extensão do meta-modelo voltado a informações gráficas. A extensão permite a geração de uma descrição no estilo XMI (*XML Metadata Interchange*) orientada a aspectos gráficos que, em conjunto com o XMI original, permite produzir representações portáteis de especificações UML.

A.4 Organização dos Diagramas de UML

A linguagem UML 2 é composta por treze diagramas, classificados em diagramas estruturais e diagramas de comportamento. A figura A.1 apresenta a estrutura das categorias utilizando a notação de diagramas de classes (OMG, 2007a).

Os diagramas estruturais, ilustrados na imagem A.2 conforme a especificação da OMG (OMG, 2007a), tratam o aspecto estrutural tanto do ponto de vista do sistema quanto das classes. Existem para visualizar, especificar, construir e documentar os aspectos estáticos de um sistema, ou seja, a representação de seu esqueleto e estruturas “relativamente estáveis”. Os aspectos estáticos de um sistema de software abrangem a existência e a colocação de itens como classes, interfaces, colaborações e componentes.

Os diagramas de comportamento, ilustrados na imagem A.3 conforme a especificação da OMG (OMG, 2007a), são voltados a descrever o sistema computacional modelado quando em

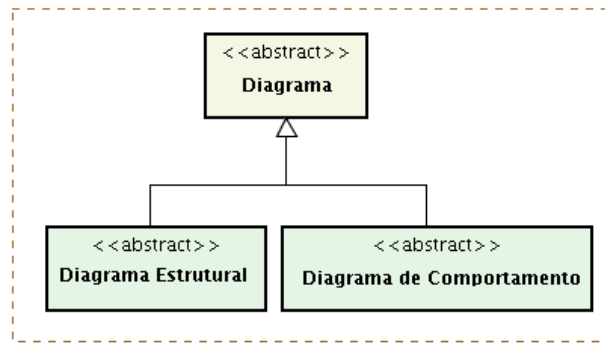


Figura A.1: Organização geral dos diagramas de UML 2.

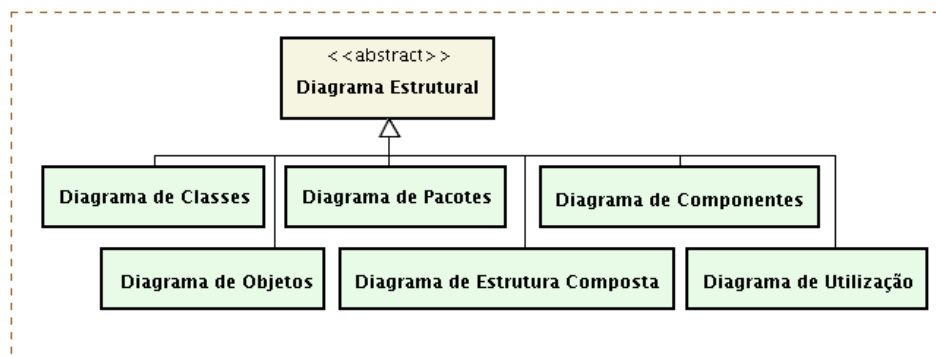


Figura A.2: Diagramas estruturais.

execução, isto é, como a modelagem dinâmica do sistema. São usados para visualizar, especificar, construir e documentar os aspectos dinâmicos de um sistema que é a representação das partes que “sofrem alterações”, como por exemplo, o fluxo de mensagens ao longo do tempo e a movimentação física de componentes em uma rede.

A.5 Diagramas de UML

Um diagrama é uma representação gráfica de um conjunto de elementos (classes, interfaces, colaborações, componentes, nós, etc) utilizados para visualizar o sistema sob diferentes perspectivas. A UML define um número de diagramas que permite dirigir o foco para aspectos diferentes do sistema de maneira independente. Se bem empregados, os diagramas facilitam a compreensão do sistema que está sendo desenvolvido.

Nas próximas seções, serão apresentados os treze diagramas que compõem a linguagem UML 2. Cada diagrama trará um exemplo de utilização, com objetivos meramente ilustrativos.

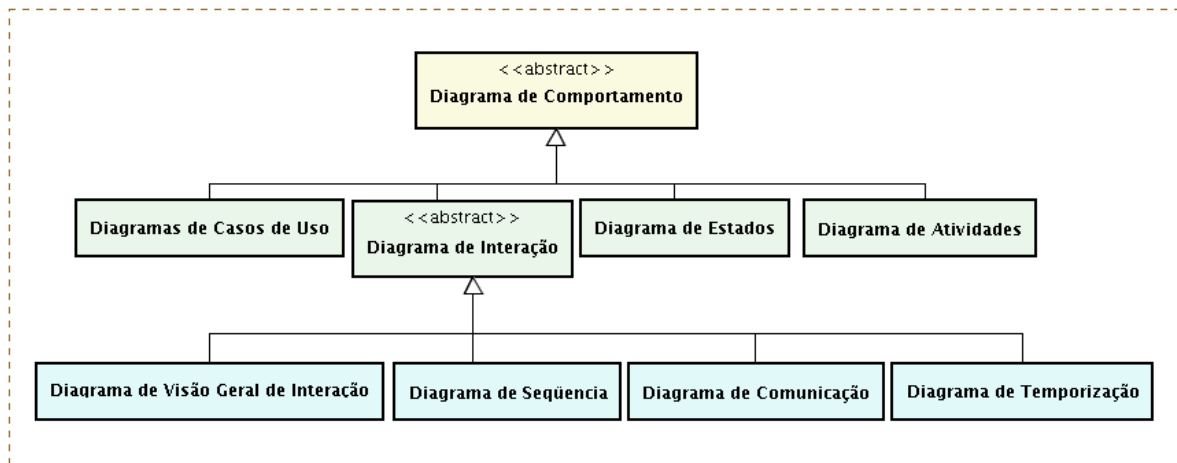


Figura A.3: Diagramas de comportamento.

A.5.1 Diagrama de Casos de Uso

O diagrama de casos de uso especifica um conjunto de funcionalidades, através do elemento sintático “casos de uso”, e os elementos externos que interagem com o sistema, através do elemento sintático “ator” (SILVA, 2007).

Além de casos de uso e atores, este diagrama contém relacionamentos de dependência (inclusão e extensão), generalização e associação, sendo basicamente usados para fazer a modelagem de visão estática do caso de uso do sistema. Essa visão proporciona suporte principalmente para o comportamento de um sistema, ou seja, os serviços externamente visíveis que o sistema fornece no contexto de seu ambiente. Neste caso, os diagramas de caso de uso são usados para fazer a modelagem do contexto de um sistema e fazer a modelagem dos requisitos de um sistema.

A.5.2 Diagrama de Classes

Um diagrama de classes é um modelo fundamental de uma especificação orientada a objetos. Produz a descrição mais próxima da estrutura do código de um programa, ou seja, mostra o conjunto de classes com seus atributos e métodos e os relacionamentos entre classes. Classes e relacionamentos constituem os elementos sintáticos básicos do diagrama de classes (SILVA, 2007).

O elemento sintático “classe” é representado por um retângulo dividido em três partes. A primeira divisão é utilizada para o nome da classe; na segunda divisão, coloca-se as informações de atributos; e a última divisão é utilizada para identificar os métodos.

A.5.3 Diagrama de Objetos

O diagrama de objetos consiste em uma variação do diagrama de classes em que, em vez de classes, são representadas instâncias e ligações entre instâncias. A finalidade é descrever um conjunto de objetos e seus relacionamentos em um ponto no tempo.

As instâncias e as ligações entre estas instâncias são usadas para fazer a modelagem da visão de projeto estática de um sistema a partir da perspectiva de instâncias reais ou prototípicas.

A.5.4 Diagrama de Pacotes

O pacote é um elemento sintático voltado a conter elementos sintáticos de uma especificação orientada a objetos. Esse elemento foi definido na primeira versão de UML para ser usado nos diagramas então existentes, como diagrama de classes, por exemplo. Na segunda versão da linguagem, foi introduzido um novo diagrama, o diagrama de pacotes, voltado a conter exclusivamente pacotes e relacionamentos entre pacotes (SILVA, 2007). Sua finalidade é tratar a modelagem estrutural do sistema dividindo o modelo em divisões lógicas e descrevendo as interações entre eles em alto nível.

A.5.5 Diagrama de Estrutura Composta

O diagrama de estrutura composta fornece meios de definir a estrutura de um elemento e de focalizá-la no detalhe, na construção e em relacionamentos internos. É um dos novos diagramas propostos na segunda versão de UML, voltado a detalhar elementos de modelagem estrutural, como classes, pacotes e componentes, descrevendo sua estrutura interna.

O diagrama de estrutura composta introduz a noção de “porto”, um ponto de conexão do elemento modelado, a quem podem ser associadas interfaces. Também utiliza a noção de “colaboração”, que consiste em um conjunto de elementos interligados através de seus portos para a execução de uma funcionalidade específica – recurso útil para a modelagem de padrões de projeto (SILVA, 2007).

A.5.6 Diagrama de Componentes

O diagrama de componentes é um dos dois diagramas de UML voltados a modelar software baseado em componentes. Tem por finalidade indicar os componentes do software e seus relacionamentos. Este diagrama mostra os artefatos de que os componentes são feitos, como arquivos

de código fonte, bibliotecas de programação ou tabelas de bancos de dados. As interfaces é que possibilitam as associações entre os componentes.

A.5.7 Diagrama de Utilização ou Implantação

O diagrama de utilização, também denominado diagrama de implantação, consiste na organização do conjunto de elementos de um sistema para a sua execução. O principal elemento deste diagrama é o nodo, que representa um recurso computacional. Podem ser representados em um diagrama tanto os nodos como instâncias de nodos.

A.5.8 Diagrama de Seqüência

O diagrama de seqüência mostra a troca de mensagens entre diversos objetos, em uma situação específica e delimitada no tempo. Coloca ênfase especial na ordem e nos momentos nos quais mensagens para os objetos são enviadas.

Em diagramas de seqüência, objetos são representados através de linhas verticais tracejadas (denominadas como linha de existência), com o nome do objeto no topo. O eixo do tempo é também vertical, aumentando para baixo, de modo que as mensagens são enviadas de um objeto para outro na forma de setas com a operação e os nomes dos parâmetros.

A.5.9 Diagrama de Comunicação

Os elementos de um sistema trabalham em conjunto para cumprir os objetivos do sistema e uma linguagem de modelagem precisa poder representar esta característica. O diagrama de comunicação é voltado a descrever objetos interagindo e seus principais elementos sintáticos são “objeto” e “mensagem”. Corresponde a um formato alternativo para descrever interação entre objetos. Ao contrário do diagrama de seqüência, o tempo não é modelado explicitamente, uma vez que a ordem das mensagens é definida através de enumeração.

Vale ressaltar que tanto o diagrama de comunicação como o diagrama de seqüência são diagramas de interação.

A.5.10 Diagrama de Máquina de Estados

O diagrama de máquina de estados tem como elementos principais o estado, que modela uma situação em que o elemento modelado pode estar ao longo de sua existência, e a transição,

que leva o elemento modelado de um estado para o outro.

O diagrama de máquina de estados vê os objetos como máquinas de estados ou autômatos finitos que poderão estar em um estado pertencente a uma lista de estados finita e que poderão mudar o seu estado através de um estímulo pertencente a um conjunto finito de estímulos.

A.5.11 Diagrama de Atividades

O diagrama de atividades representa a execução das ações e as transições que são acionadas pela conclusão de outras ações ou atividades.

Uma atividade pode ser descrita como um conjunto de ações e um conjunto de atividades. A diferença básica entre os dois conceitos que descrevem comportamento é que a ação é atômica, não admitindo particionamento, o que não se aplica a atividade, que pode ser detalhada em atividades e ações (SILVA, 2007).

A.5.12 Diagrama de Visão Geral de Interação

O diagrama de visão geral de interação é uma variação do diagrama de atividades, proposto na segunda versão de UML.

Os elementos sintáticos deste diagrama são os mesmos do diagrama de atividades, exceto os nodos que são substituídos por interações. As interações que fazem parte do diagrama de visão geral de interação podem ser referências a diagramas de interação existentes na especificação tratada.

A.5.13 Diagrama de Temporização

O diagrama de temporização consiste na modelagem de restrições temporais do sistema. É um diagrama introduzido na segunda versão de UML, classificado como diagrama de interação. Este diagrama modela interação e evolução de estados.

A.6 Conclusão

Embora a UML defina uma linguagem precisa, ela não é uma barreira para futuros aperfeiçoamentos nos conceitos de modelagem. O desenvolvimento da UML foi baseado em técnicas antigas e marcantes da orientação a objetos, mas muitas outras influenciarão a linguagem em suas

próximas versões. Muitas técnicas avançadas de modelagem podem ser definidas usando UML como base, podendo ser estendida sem se fazer necessário redefinir a sua estrutura interna.

A UML está sendo a base para muitas ferramentas de desenvolvimento, incluindo modelagem visual, simulações e ambientes de desenvolvimento. Em breve, ferramentas de integração e padrões de implementação baseados em UML estarão disponíveis para qualquer um.

A UML integrou muitas idéias adversas, e esta integração acelera o uso do desenvolvimento de softwares orientados a objetos.

ANEXO B – Código-fonte: Extensão do Ambiente SEA

No desenvolvimento deste trabalho de conclusão de curso, a autora concebeu cerca de 230 classes em linguagem Java - sem contar as classes que foram modificadas e as classes de bibliotecas e frameworks que foram utilizadas. Sendo assim, a inserção de todos estes arquivos na presente monografia tornaria o documento extremamente extenso e, por este motivo, optou-se por anexar apenas as classes principais da implementação deste trabalho e as classes referentes à criação de apenas um dos diagramas codificados: o diagrama de componentes.

As listagens que seguem apresentam o código-fonte dos arquivos julgados mais relevantes no contexto deste trabalho.

Listagem B.1: Classe **UML2SEASpecification.java**

```

1 package documents . specifications ;
2
3 import ocean . documents . oo . specifications . SEASpecification ;
4 import ocean . smalltalk . OceanVector ;
5 import documents . models . * ;
6 import documents . util . I18NProperties ;
7
8 public class UML2SEASpecification extends SEASpecification
9 {
10
11     /**
12      * The serial version UID.
13      */
14     private static final long serialVersionUID = 3621842892413105154L;
15
16     /**
17      * Constructor.
18      */
19     public UML2SEASpecification ()

```

```

20     {
21         super ();
22     }
23
24     /**
25      * @param documentCode The document code.
26      */
27     public UML2SEASpecification(String documentCode)
28     {
29         super(documentCode);
30     }
31
32     /*
33      * (non-Javadoc)
34      *
35      * @see ocean.framework.Specification#modeloLista()
36      */
37     @SuppressWarnings( { "deprecation", "unchecked" })
38     public OceanVector modeloLista()
39     {
40         OceanVector aList = super.modeloLista();
41         aList.add(ObjectDiagram.class);
42         aList.add(PackageDiagram.class);
43         aList.add(ComponentDiagram.class);
44         aList.add(DeploymentDiagram.class);
45         aList.add(CommunicationDiagram.class);
46         aList.add(CompositeStructureDiagram.class);
47         aList.add(StateMachineDiagram.class);
48         aList.add(ActivityDiagram.class);
49         return aList;
50     }
51
52     /**
53      * Gets the id type.
54      */
55     public static String getIdType()
56     {
57         return "uml2.specification";
58     }
59
60     /**
61      * Gets the ID.
62      */

```

```

63     public static String getId()
64     {
65         return I18NProperties.getString("uml2.specification");
66     }
67
68     /*
69     * (non-Javadoc)
70     *
71     * @see ocean.framework.OceanDocument#getNameForTree()
72     */
73     public String getNameForTree()
74     {
75         return I18NProperties.getString("uml2.specification.for.tree");
76     }
77
78 }

```

Listagem B.2: Classe **ActivityDiagram.java**

```

1  package documents.models;
2
3  import ocean.framework.ConceptualModel;
4  import ocean.framework.SpecificationElement;
5  import ocean.jhotdraw.SpecificationDrawing;
6  import ocean.smalltalk.OceanVector;
7  import documents.concepts.activity_diagram.*;
8  import documents.concepts.communication_diagram.Label;
9  import documents.graphical.activity_diagram.ActivityDiagramDrawing;
10 import documents.util.I18NProperties;
11
12 public class ActivityDiagram extends ConceptualModel
13 {
14
15     /**
16     * The serial version UID.
17     */
18     private static final long serialVersionUID = -73604485105160495L;
19
20     /*
21     * (non-Javadoc)
22     *
23     * @see ocean.framework.ConceptualModel#initialize()
24     */

```

```

25     @SuppressWarnings( { "deprecation", "unchecked" })
26     public void initialize ()
27     {
28         elementKeeperList = new OceanVector();
29         elementKeeperList.add( SpecificationElement.construtor(
30             AcceptEventAction.class ));
31         elementKeeperList.add( SpecificationElement.construtor(
32             AcceptTimeEventAction.class ));
33         elementKeeperList.add( SpecificationElement.construtor( Action.class )
34             );
35         elementKeeperList.add( SpecificationElement.construtor( Activity.
36             class ));
37         elementKeeperList.add( SpecificationElement.construtor(
38             CentralBufferNode.class ));
39         elementKeeperList.add( SpecificationElement.construtor(
40             ConditionalNode.class ));
41         elementKeeperList.add( SpecificationElement.construtor( ControlFlow.
42             class ));
43         elementKeeperList.add( SpecificationElement.construtor( DataStoreNode
44             .class ));
45         elementKeeperList.add( SpecificationElement.construtor( DecisionNode.
46             class ));
47         elementKeeperList.add( SpecificationElement.construtor(
48             ExceptionHandler.class ));
49         elementKeeperList.add( SpecificationElement.construtor( FinalNode.
50             class ));
51         elementKeeperList.add( SpecificationElement.construtor( FlowFinalNode
52             .class ));
53         elementKeeperList.add( SpecificationElement.construtor( ForkNode.
54             class ));
55         elementKeeperList.add( SpecificationElement.construtor(
56             HorizontalSwinlane.class ));
57         elementKeeperList.add( SpecificationElement.construtor( InitialNode.
58             class ));
59         elementKeeperList.add( SpecificationElement.construtor(
60             InterruptibleActivity.class ));
61         elementKeeperList.add( SpecificationElement.construtor( JoinNode.
62             class ));
63         elementKeeperList.add( SpecificationElement.construtor( LoopNode.
64             class ));
65         elementKeeperList.add( SpecificationElement.construtor( MergeNode.
66             class ));
67         elementKeeperList.add( SpecificationElement.construtor( ObjectFlow.

```

```
        class));
49 elementKeeperList.add( SpecificationElement.construtor( ObjectNode .
        class));
50 elementKeeperList.add( SpecificationElement.construtor( SendSignal .
        class));
51 elementKeeperList.add( SpecificationElement.construtor( SequenceNode .
        class));
52 elementKeeperList.add( SpecificationElement.construtor(
        StructuredActivityNode . class));
53 elementKeeperList.add( SpecificationElement.construtor(
        VerticalSwinlane . class));
54 elementKeeperList.add( SpecificationElement.construtor( Assignment .
        class));
55 elementKeeperList.add( SpecificationElement.construtor( Return . class)
        );
56 elementKeeperList.add( SpecificationElement.construtor( Message . class
        ));
57 elementKeeperList.add( SpecificationElement.construtor( Comment . class
        ));
58 elementKeeperList.add( SpecificationElement.construtor( NoteLink .
        class));
59 elementKeeperList.add( SpecificationElement.construtor( Variable .
        class));
60 elementKeeperList.add( SpecificationElement.construtor( TaskPackage .
        class));
61 elementKeeperList.add( SpecificationElement.construtor( GenericTask .
        class));
62 elementKeeperList.add( SpecificationElement.construtor( Label . class)
        ;
63 elementKeeperList.add( SpecificationElement.construtor( IfNode . class)
        );
64 elementKeeperList.add( SpecificationElement.construtor( IfElseNode .
        class));
65 elementKeeperList.add( SpecificationElement.construtor( ForNode . class
        ));
66 elementKeeperList.add( SpecificationElement.construtor( WhileNode .
        class));
67 elementKeeperList.add( SpecificationElement.construtor( DoWhileNode .
        class));
68 elementKeeperList.add( SpecificationElement.construtor( SwitchNode .
        class));
69 elementKeeperList.add( SpecificationElement.construtor( Note . class));
70 }
```

```

71
72  /*
73   * (non-Javadoc)
74   *
75   * @see ocean.framework.ConceptualModel#createEmptyDrawing()
76   */
77  @Override
78  public SpecificationDrawing createEmptyDrawing()
79  {
80      return new ActivityDiagramDrawing();
81  }
82
83  /*
84   * (non-Javadoc)
85   *
86   * @see ocean.framework.ConceptualModel#conceitoLista()
87   */
88  @SuppressWarnings( { "deprecation", "unchecked" })
89  public OceanVector<Class> conceitoLista()
90  {
91      OceanVector<Class> aList = new OceanVector<Class>();
92      aList.add(AcceptEventAction.class);
93      aList.add(AcceptTimeEventAction.class);
94      aList.add(Action.class);
95      aList.add(Activity.class);
96      aList.add(CentralBufferNode.class);
97      aList.add(ConditionalNode.class);
98      aList.add(ControlFlow.class);
99      aList.add(DataStoreNode.class);
100     aList.add(DecisionNode.class);
101     aList.add(ExceptionHandler.class);
102     aList.add(FinalNode.class);
103     aList.add(FlowFinalNode.class);
104     aList.add(ForkNode.class);
105     aList.add(HorizontalSwimlane.class);
106     aList.add(InitialNode.class);
107     aList.add(InterruptibleActivity.class);
108     aList.add(JoinNode.class);
109     aList.add(LoopNode.class);
110     aList.add(MergeNode.class);
111     aList.add(ObjectFlow.class);
112     aList.add(ObjectNode.class);
113     aList.add(SendSignal.class);

```



```

114     aList.add(SequenceNode.class);
115     aList.add(StructuredActivityNode.class);
116     aList.add(VerticalSwimlane.class);
117     aList.add(Assignment.class);
118     aList.add(Return.class);
119     aList.add(Message.class);
120     aList.add(Comment.class);
121     aList.add(NoteLink.class);
122     aList.add(Variable.class);
123     aList.add(TaskPackage.class);
124     aList.add(GenericTask.class);
125     aList.add(Label.class);
126     aList.add(IfNode.class);
127     aList.add(IfElseNode.class);
128     aList.add(ForNode.class);
129     aList.add(WhileNode.class);
130     aList.add(DoWhileNode.class);
131     aList.add(SwitchNode.class);
132     aList.add(Note.class);
133
134     return aList;
135 }
136
137 /*
138  * (non-Javadoc) Realization
139  *
140  * @see ocean.framework.ConceptualModel#modelName()
141  */
142 public String modelName()
143 {
144     return I18NProperties.getString("activity.diagram.name");
145 }
146
147 /*
148  * (non-Javadoc)
149  *
150  * @see ocean.framework.OceanDocument#getNameForTree()
151  */
152 public String getNameForTree()
153 {
154     return I18NProperties.getString("activity.diagram.name.for.tree") +
155         name;
156 }

```

156
157 }

Listagem B.3: Classe **CommunicationDiagram.java**

```

1  package documents . models ;
2
3  import ocean . documents . oo . concepts . AssociacaoBinaria ;
4  import ocean . documents . oo . concepts . Heranca ;
5  import ocean . documents . oo . concepts . UseCaseActor ;
6  import ocean . framework . ConceptualModel ;
7  import ocean . framework . SpecificationElement ;
8  import ocean . jhotdraw . SpecificationDrawing ;
9  import ocean . smalltalk . OceanVector ;
10 import documents . concepts . communication_diagram . Label ;
11 import documents . concepts . communication_diagram . Message ;
12 import documents . concepts . communication_diagram . ReverseMessage ;
13 import documents . concepts . object_diagram . Dependency ;
14 import documents . concepts . object_diagram . InstanceSpecification ;
15 import documents . graphical . communication_diagram .
    CommunicationDiagramDrawing ;
16 import documents . util . I18NProperties ;
17
18 public class CommunicationDiagram extends ConceptualModel
19 {
20
21     /**
22      * The serial version UID.
23      */
24     private static final long serialVersionUID = -73604485105160495L;
25
26     /*
27      * (non-Javadoc)
28      *
29      * @see ocean.framework.ConceptualModel#initialize()
30      */
31     @SuppressWarnings( { "deprecation", "unchecked" })
32     public void initialize ()
33     {
34         elementKeeperList = new OceanVector ();
35         elementKeeperList . add ( SpecificationElement . construtor ( Label . class ) )
36         ;
37         elementKeeperList . add ( SpecificationElement . construtor (

```

```

        InstanceSpecification.class));
37     elementKeeperList.add(SpecificationElement.construtor(
        AssociacaoBinaria.class));
38     elementKeeperList.add(SpecificationElement.construtor(Message.class
        ));
39     elementKeeperList.add(SpecificationElement.construtor(
        ReverseMessage.class));
40     elementKeeperList.add(SpecificationElement.construtor(UseCaseActor.
        class));
41     elementKeeperList.add(SpecificationElement.construtor(Heranca.class
        ));
42     elementKeeperList.add(SpecificationElement.construtor(Dependency.
        class));
43 }
44
45 /*
46  * (non-Javadoc)
47  *
48  * @see ocean.framework.ConceptualModel#createEmptyDrawing()
49  */
50 @Override
51 public SpecificationDrawing createEmptyDrawing()
52 {
53     return new CommunicationDiagramDrawing();
54 }
55
56 /*
57  * (non-Javadoc)
58  *
59  * @see ocean.framework.ConceptualModel#conceitoLista()
60  */
61 @SuppressWarnings( { "deprecation", "unchecked" })
62 public OceanVector<Class> conceitoLista()
63 {
64     OceanVector<Class> aList = new OceanVector<Class>();
65     aList.add(Label.class);
66     aList.add(InstanceSpecification.class);
67     aList.add(AssociacaoBinaria.class);
68     aList.add(Message.class);
69     aList.add(ReverseMessage.class);
70     aList.add(UseCaseActor.class);
71     aList.add(Heranca.class);
72     aList.add(Dependency.class);

```

```

73     return aList;
74 }
75
76 /*
77  * (non-Javadoc) Realization
78  *
79  * @see ocean.framework.ConceptualModel#modelName()
80  */
81 public String modelName()
82 {
83     return I18NProperties.getString("communication.diagram.name");
84 }
85
86 /*
87  * (non-Javadoc)
88  *
89  * @see ocean.framework.OceanDocument#getNameForTree()
90  */
91 public String getNameForTree()
92 {
93     return I18NProperties.getString("communication.diagram.name.for.
94         tree") + name;
95 }
96 }

```

Listagem B.4: Classe **ComponentDiagram.java**

```

1 package documents.models;
2
3 import ocean.documents.oo.concepts.*;
4 import ocean.framework.ConceptualModel;
5 import ocean.framework.SpecificationElement;
6 import ocean.jhotdraw.SpecificationDrawing;
7 import ocean.smalltalk.OceanVector;
8 import documents.concepts.component_diagram.Component;
9 import documents.concepts.component_diagram.Interface;
10 import documents.concepts.component_diagram.Port;
11 import documents.concepts.object_diagram.Dependency;
12 import documents.concepts.object_diagram.Realization;
13 import documents.graphical.component_diagram.ComponentDiagramDrawing;
14 import documents.util.I18NProperties;
15

```

```

16 public class ComponentDiagram extends ConceptualModel
17 {
18
19     /**
20      * The serial version UID.
21      */
22     private static final long serialVersionUID = 6117649257803752678L;
23
24     /* (non-Javadoc)
25      * @see ocean.framework.ConceptualModel#initialize ()
26      */
27     @SuppressWarnings( { "deprecation", "unchecked" })
28     public void initialize ()
29     {
30         elementKeeperList = new OceanVector ();
31         elementKeeperList.add( SpecificationElement.construtor( Port.class ));
32         elementKeeperList.add( SpecificationElement.construtor( Interface.class ));
33         elementKeeperList.add( SpecificationElement.construtor( Component.class ));
34         elementKeeperList.add( SpecificationElement.construtor( Dependency.class ));
35         elementKeeperList.add( SpecificationElement.construtor( Realization.class ));
36         elementKeeperList.add( SpecificationElement.construtor( Heranca.class
37             ));
38         elementKeeperList.add( SpecificationElement.construtor(
39             AssociacaoBinaria.class ));
40         elementKeeperList.add( SpecificationElement.construtor( Agregacao.class ));
41         elementKeeperList.add( SpecificationElement.construtor( Composition.class ));
42     }
43
44     /*
45      * (non-Javadoc)
46      *
47      * @see ocean.framework.ConceptualModel#createEmptyDrawing ()
48      */
49     @Override
50     public SpecificationDrawing createEmptyDrawing ()
51     {
52         return new ComponentDiagramDrawing ();
53     }
54 }

```

```

51     }
52
53     /*
54      * (non-Javadoc)
55      *
56      * @see ocean.framework.ConceptualModel#conceitoLista()
57      */
58     @SuppressWarnings( { "deprecation", "unchecked" })
59     public OceanVector<Class> conceitoLista()
60     {
61         OceanVector<Class> aList = new OceanVector<Class>();
62         aList.add(Port.class);
63         aList.add(Interface.class);
64         aList.add(Component.class);
65         aList.add(Dependency.class);
66         aList.add(Realization.class);
67         aList.add(Heranca.class);
68         aList.add(AssociacaoBinaria.class);
69         aList.add(Agregacao.class);
70         aList.add(Composition.class);
71         return aList;
72     }
73
74     /*
75      * (non-Javadoc) Realization
76      *
77      * @see ocean.framework.ConceptualModel#modelName()
78      */
79     public String modelName()
80     {
81         return I18NProperties.getString("component.diagram.name");
82     }
83
84     /*
85      * (non-Javadoc)
86      *
87      * @see ocean.framework.OceanDocument#getNameForTree()
88      */
89     public String getNameForTree()
90     {
91         return I18NProperties.getString("component.diagram.name.for.tree")
92             + name;
93     }

```

93
94 }

Listagem B.5: Classe **CompositeStructuredDiagram.java**

```

1  package documents . models ;
2
3  import ocean . documents . oo . concepts . * ;
4  import ocean . framework . ConceptualModel ;
5  import ocean . framework . SpecificationElement ;
6  import ocean . jhotdraw . SpecificationDrawing ;
7  import ocean . smalltalk . OceanVector ;
8  import documents . concepts . component_diagram . Interface ;
9  import documents . concepts . component_diagram . Port ;
10 import documents . concepts . composite_structure_diagram . * ;
11 import documents . concepts . object_diagram . Dependency ;
12 import documents . concepts . object_diagram . InstanceSpecification ;
13 import documents . concepts . object_diagram . Realization ;
14 import documents . graphical . composite_structure_diagram .
    CompositeStructureDiagramDrawing ;
15 import documents . util . I18NProperties ;
16
17 public class CompositeStructureDiagram extends ConceptualModel
18 {
19
20     /**
21      * The serial version UID.
22      */
23     private static final long serialVersionUID = -73604485105160495L;
24
25     /*
26      * (non-Javadoc)
27      *
28      * @see ocean . framework . ConceptualModel # initialize ()
29      */
30     @SuppressWarnings( { "deprecation", "unchecked" })
31     public void initialize ()
32     {
33         elementKeeperList = new OceanVector ();
34         elementKeeperList . add ( SpecificationElement . construtor ( Classe . class )
35                                 );
35         elementKeeperList . add ( SpecificationElement . construtor (
36                                 InstanceSpecification . class ) );

```

```

36     elementKeeperList.add( SpecificationElement.construtor( Part.class ));
37     elementKeeperList.add( SpecificationElement.construtor( Property .
        class ));
38     elementKeeperList.add( SpecificationElement.construtor( Port.class ));
39     elementKeeperList.add( SpecificationElement.construtor( Interface .
        class ));
40     elementKeeperList.add( SpecificationElement.construtor(
        AssociacaoBinaria.class ));
41     elementKeeperList.add( SpecificationElement.construtor( Agregacao .
        class ));
42     elementKeeperList.add( SpecificationElement.construtor( Composition .
        class ));
43     elementKeeperList.add( SpecificationElement.construtor( Heranca.class
        ));
44     elementKeeperList.add( SpecificationElement.construtor( Dependency .
        class ));
45     elementKeeperList.add( SpecificationElement.construtor( Realization .
        class ));
46     elementKeeperList.add( SpecificationElement.construtor( Connector .
        class ));
47     elementKeeperList.add( SpecificationElement.construtor( Collaboration
        .class ));
48     elementKeeperList.add( SpecificationElement.construtor(
        CollaborationUse.class ));
49     elementKeeperList.add( SpecificationElement.construtor( Represents .
        class ));
50     elementKeeperList.add( SpecificationElement.construtor( Occurrence .
        class ));
51 }
52
53 /*
54  * (non-Javadoc)
55  *
56  * @see ocean.framework.ConceptualModel#createEmptyDrawing()
57  */
58 @Override
59 public SpecificationDrawing createEmptyDrawing ()
60 {
61     return new CompositeStructureDiagramDrawing ();
62 }
63
64 /*
65  * (non-Javadoc)

```



```

66      *
67      * @see ocean.framework.ConceptualModel#conceitoLista()
68      */
69      @SuppressWarnings( { "deprecation", "unchecked" })
70      public OceanVector<Class> conceitoLista()
71      {
72          OceanVector<Class> aList = new OceanVector<Class>();
73          aList.add(InstanceSpecification.class);
74          aList.add(Classe.class);
75          aList.add(Part.class);
76          aList.add(Property.class);
77          aList.add(Port.class);
78          aList.add(Interface.class);
79          aList.add(AssociacaoBinaria.class);
80          aList.add(Agregacao.class);
81          aList.add(Composition.class);
82          aList.add(Heranca.class);
83          aList.add(Dependency.class);
84          aList.add(Realization.class);
85          aList.add(Connector.class);
86          aList.add(Collaboration.class);
87          aList.add(CollaborationUse.class);
88          aList.add(Represents.class);
89          aList.add(Occurrence.class);
90          return aList;
91      }
92
93      /*
94      * (non-Javadoc) Realization
95      *
96      * @see ocean.framework.ConceptualModel#modelName()
97      */
98      public String modelName()
99      {
100         return I18NProperties.getString("composite.diagram.name");
101     }
102
103     /*
104     * (non-Javadoc)
105     *
106     * @see ocean.framework.OceanDocument#getNameForTree()
107     */
108     public String getNameForTree()

```

```

109     {
110         return I18NProperties.getString("composite.diagram.name.for.tree")
            + name;
111     }
112
113 }

```

Listagem B.6: Classe **DeploymentDiagram.java**

```

1  package documents.models;
2
3  import ocean.documents.oo.concepts.*;
4  import ocean.framework.ConceptualModel;
5  import ocean.framework.SpecificationElement;
6  import ocean.jhotdraw.SpecificationDrawing;
7  import ocean.smalltalk.OceanVector;
8  import documents.concepts.component_diagram.Component;
9  import documents.concepts.component_diagram.Interface;
10 import documents.concepts.component_diagram.Port;
11 import documents.concepts.deployment_diagram.*;
12 import documents.concepts.object_diagram.Dependency;
13 import documents.concepts.object_diagram.Realization;
14 import documents.graphical.deployment_diagram.DeploymentDiagramDrawing;
15 import documents.util.I18NProperties;
16
17 public class DeploymentDiagram extends ConceptualModel
18 {
19
20     /**
21      * The serial version UID.
22      */
23     private static final long serialVersionUID = -2759341821171253471L;
24
25     /**
26      * (non-Javadoc)
27      *
28      * @see ocean.framework.ConceptualModel#initialize()
29      */
30     @SuppressWarnings( { "deprecation", "unchecked" })
31     public void initialize()
32     {
33         elementKeeperList = new OceanVector();
34         elementKeeperList.add(SpecificationElement.construtor( Artifact .

```

```

    class));
35 elementKeeperList.add( SpecificationElement . constructor ( Deployment .
    class));
36 elementKeeperList.add( SpecificationElement . constructor (
    DeploymentSpecification . class));
37 elementKeeperList.add( SpecificationElement . constructor ( DeviceNode .
    class));
38 elementKeeperList.add( SpecificationElement . constructor ( Node . class));
39 elementKeeperList.add( SpecificationElement . constructor ( Manifestation
    . class));
40 elementKeeperList.add( SpecificationElement . constructor (
    ExecutionEnvironment . class));
41 elementKeeperList.add( SpecificationElement . constructor ( Port . class));
42 elementKeeperList.add( SpecificationElement . constructor ( Interface .
    class));
43 elementKeeperList.add( SpecificationElement . constructor ( Component .
    class));
44 elementKeeperList.add( SpecificationElement . constructor ( Dependency .
    class));
45 elementKeeperList.add( SpecificationElement . constructor ( Realization .
    class));
46 elementKeeperList.add( SpecificationElement . constructor ( Heranca . class
    ));
47 elementKeeperList.add( SpecificationElement . constructor (
    AssociacaoBinaria . class));
48 elementKeeperList.add( SpecificationElement . constructor ( Agregacao .
    class));
49 elementKeeperList.add( SpecificationElement . constructor ( Composition .
    class));
50 }
51
52 /*
53  * (non-Javadoc)
54  *
55  * @see ocean.framework.ConceptualModel#createEmptyDrawing()
56  */
57 @Override
58 public SpecificationDrawing createEmptyDrawing ()
59 {
60     return new DeploymentDiagramDrawing ();
61 }
62
63 /*

```

```

64     * (non-Javadoc)
65     *
66     * @see ocean.framework.ConceptualModel#conceitoLista()
67     */
68     @SuppressWarnings( { "deprecation", "unchecked" })
69     public OceanVector<Class> conceitoLista()
70     {
71         OceanVector<Class> aList = new OceanVector<Class>();
72         aList.add(Artifact.class);
73         aList.add(Deployment.class);
74         aList.add(DeploymentSpecification.class);
75         aList.add(DeviceNode.class);
76         aList.add(Node.class);
77         aList.add(Manifestation.class);
78         aList.add(ExecutionEnvironment.class);
79         aList.add(Port.class);
80         aList.add(Interface.class);
81         aList.add(Component.class);
82         aList.add(Dependency.class);
83         aList.add(Realization.class);
84         aList.add(Heranca.class);
85         aList.add(AssociacaoBinaria.class);
86         aList.add(Agregacao.class);
87         aList.add(Composition.class);
88         return aList;
89     }
90
91     /*
92     * (non-Javadoc) Realization
93     *
94     * @see ocean.framework.ConceptualModel#modelName()
95     */
96     public String modelName()
97     {
98         return I18NProperties.getString("deployment.diagram.name");
99     }
100
101     /*
102     * (non-Javadoc)
103     *
104     * @see ocean.framework.OceanDocument#getNameForTree()
105     */
106     public String getNameForTree()

```

```

107     {
108         return I18NProperties.getString("deployment.diagram.name.for.tree")
            + name;
109     }
110
111 }

```

Listagem B.7: Classe **ObjectDiagram.java**

```

1  package documents.models;
2
3  import ocean.documents.oo.concepts.*;
4  import ocean.framework.ConceptualModel;
5  import ocean.framework.SpecificationElement;
6  import ocean.jhotdraw.SpecificationDrawing;
7  import ocean.smalltalk.OceanVector;
8  import documents.concepts.object_diagram.Dependency;
9  import documents.concepts.object_diagram.Realization;
10 import documents.graphical.object_diagram.ObjectDiagramDrawing;
11 import documents.util.I18NProperties;
12
13 public class ObjectDiagram extends ConceptualModel
14 {
15
16     /**
17      * The serial version UID.
18      */
19     private static final long serialVersionUID = 5511381289512273235L;
20
21     /*
22      * (non-Javadoc)
23      *
24      * @see ocean.framework.ConceptualModel#initialize()
25      */
26     @SuppressWarnings( { "deprecation", "unchecked" })
27     public void initialize()
28     {
29         elementKeeperList = new OceanVector();
30         elementKeeperList.add(SpecificationElement
31             .construtor(documents.concepts.object_diagram.
32                 InstanceSpecification.class));
33         elementKeeperList.add(SpecificationElement.construtor(
34             AssociacaoBinaria.class));

```

```

33     elementKeeperList.add( SpecificationElement.construtor( Heranca.class
34         ));
35     elementKeeperList.add( SpecificationElement.construtor( Agregacao.
36         class));
37     elementKeeperList.add( SpecificationElement.construtor( Composition.
38         class));
39     elementKeeperList.add( SpecificationElement.construtor( Dependency.
40         class));
41     elementKeeperList.add( SpecificationElement.construtor( Realization.
42         class));
43 }
44
45 /*
46  * (non-Javadoc)
47  *
48  * @see ocean.framework.ConceptualModel#createEmptyDrawing()
49  */
50 @Override
51 public SpecificationDrawing createEmptyDrawing()
52 {
53     return new ObjectDiagramDrawing();
54 }
55
56 /*
57  * (non-Javadoc)
58  *
59  * @see ocean.framework.ConceptualModel#conceitoLista()
60  */
61 @SuppressWarnings( { "deprecation", "unchecked" })
62 public OceanVector<Class> conceitoLista()
63 {
64     OceanVector<Class> aList = new OceanVector<Class>();
65     aList.add( documents.concepts.object_diagram.InstanceSpecification.
66         class);
67     aList.add( AssociacaoBinaria.class);
68     aList.add( Agregacao.class);
69     aList.add( Composition.class);
70     aList.add( Heranca.class);
71     aList.add( Dependency.class);
72     aList.add( Realization.class);
73     return aList;
74 }

```

```

70     /*
71     * (non-Javadoc)
72     *
73     * @see ocean.framework.ConceptualModel#modelName()
74     */
75     public String modelName()
76     {
77         return I18NProperties.getString("object.model.name");
78     }
79
80     /*
81     * (non-Javadoc)
82     *
83     * @see ocean.framework.OceanDocument#getNameForTree()
84     */
85     public String getNameForTree()
86     {
87         return I18NProperties.getString("object.model.name.for.tree") +
88             name;
89     }
90 }

```

Listagem B.8: Classe **PackageDiagram.java**

```

1  package documents.models;
2
3  import ocean.documents.oo.concepts.AssociacaoBinaria;
4  import ocean.documents.oo.concepts.Classe;
5  import ocean.documents.oo.concepts.Heranca;
6  import ocean.framework.ConceptualModel;
7  import ocean.framework.SpecificationElement;
8  import ocean.jhotdraw.SpecificationDrawing;
9  import ocean.smalltalk.OceanVector;
10 import documents.concepts.object_diagram.Dependency;
11 import documents.concepts.object_diagram.Realization;
12 import documents.concepts.package_diagram.*;
13 import documents.concepts.package_diagram.Package;
14 import documents.graphical.package_diagram.PackageDiagramDrawing;
15 import documents.util.I18NProperties;
16
17 public class PackageDiagram extends ConceptualModel
18 {

```

```

19
20  /**
21   * The serial version UID.
22   */
23  private static final long serialVersionUID = 1728189157639079058L;
24
25  /**
26   * (non-Javadoc)
27   *
28   * @see ocean.framework.ConceptualModel#initialize()
29   */
30  @SuppressWarnings( { "deprecation", "unchecked" })
31  public void initialize ()
32  {
33      elementKeeperList = new OceanVector ();
34      elementKeeperList.add( SpecificationElement.construtor(
35          AssociacaoBinaria.class ));
36      elementKeeperList.add( SpecificationElement.construtor( Heranca.class
37          ));
38      elementKeeperList.add( SpecificationElement.construtor( Dependency.class
39          ));
40      elementKeeperList.add( SpecificationElement.construtor( Realization.class
41          ));
42      elementKeeperList.add( SpecificationElement.construtor( Access.class
43          ));
44      elementKeeperList.add( SpecificationElement.construtor( Import.class
45          ));
46      elementKeeperList.add( SpecificationElement.construtor( Merge.class )
47          );
48      elementKeeperList.add( SpecificationElement.construtor( Package.class
49          ));
50      elementKeeperList.add( SpecificationElement.construtor( Classe.class
51          ));
52  }
53
54  /**
55   * (non-Javadoc)
56   *
57   * @see ocean.framework.ConceptualModel#createEmptyDrawing()
58   */
59  @Override
60  public SpecificationDrawing createEmptyDrawing ()
61  {

```



```

53     return new PackageDiagramDrawing ();
54 }
55
56 /*
57  * (non-Javadoc)
58  *
59  * @see ocean.framework.ConceptualModel#conceitoLista ()
60  */
61 @SuppressWarnings( { "deprecation", "unchecked" })
62 public OceanVector<Class> conceitoLista ()
63 {
64     OceanVector<Class> aList = new OceanVector<Class>();
65     aList.add(AssociacaoBinaria.class);
66     aList.add(Heranca.class);
67     aList.add(Dependency.class);
68     aList.add(Realization.class);
69     aList.add(Access.class);
70     aList.add(Import.class);
71     aList.add(Merge.class);
72     aList.add(Package.class);
73     // aList.add(Subsystem.class);
74     aList.add(Classe.class);
75     return aList;
76 }
77
78 /*
79  * (non-Javadoc) Realization
80  *
81  * @see ocean.framework.ConceptualModel#modelName ()
82  */
83 public String modelName ()
84 {
85     return I18NProperties.getString("package.diagram.name");
86 }
87
88 /*
89  * (non-Javadoc)
90  *
91  * @see ocean.framework.OceanDocument#getNameForTree ()
92  */
93 public String getNameForTree ()
94 {
95     return I18NProperties.getString("package.diagram.name.for.tree") +

```

```

        name ;
96     }
97
98 }

```

Listagem B.9: Classe `StateMachineDiagram.java`

```

1  package documents . models ;
2
3  import ocean . framework . ConceptualModel ;
4  import ocean . framework . SpecificationElement ;
5  import ocean . jhotdraw . SpecificationDrawing ;
6  import ocean . smalltalk . OceanVector ;
7  import documents . concepts . communication_diagram . Label ;
8  import documents . concepts . state_machine_diagram . * ;
9  import documents . graphical . state_machine_diagram . StateMachineDiagramDrawing
   ;
10 import documents . util . I18NProperties ;
11
12 public class StateMachineDiagram extends ConceptualModel
13 {
14
15     /**
16      * The serial version UID.
17      */
18     private static final long serialVersionUID = 2232046358676034753L ;
19
20     /*
21      * (non-Javadoc)
22      *
23      * @see ocean . framework . ConceptualModel # initialize ()
24      */
25     @SuppressWarnings ( { "deprecation" , "unchecked" })
26     public void initialize ()
27     {
28         elementKeeperList = new OceanVector () ;
29         elementKeeperList . add ( SpecificationElement . construtor ( Choice . class )
   );
30         elementKeeperList . add ( SpecificationElement . construtor ( EntryPoint .
   class ) ) ;
31         elementKeeperList . add ( SpecificationElement . construtor ( ExitPoint .
   class ) ) ;
32         elementKeeperList . add ( SpecificationElement . construtor ( FinalState .

```

```

        class));
33     elementKeeperList.add( SpecificationElement.construtor( Fork.class));
34     elementKeeperList.add( SpecificationElement.construtor( InitialState.class));
        class));
35     elementKeeperList.add( SpecificationElement.construtor( Join.class));
36     elementKeeperList.add( SpecificationElement.construtor( Junction.class));
        class));
37     elementKeeperList.add( SpecificationElement.construtor( State.class))
        ;
38     elementKeeperList.add( SpecificationElement.construtor(
        SubmachineState.class));
39     elementKeeperList.add( SpecificationElement.construtor( Terminate.class));
        class));
40     elementKeeperList.add( SpecificationElement.construtor( Transition.class));
        class));
41 }
42
43 /*
44  * (non-Javadoc)
45  *
46  * @see ocean.framework.ConceptualModel#createEmptyDrawing()
47  */
48 @Override
49 public SpecificationDrawing createEmptyDrawing()
50 {
51     return new StateMachineDiagramDrawing();
52 }
53
54 /*
55  * (non-Javadoc)
56  *
57  * @see ocean.framework.ConceptualModel#conceitoLista()
58  */
59 @SuppressWarnings( { "deprecation", "unchecked" })
60 public OceanVector<Class> conceitoLista()
61 {
62     OceanVector<Class> aList = new OceanVector<Class>();
63     aList.add( Choice.class);
64     aList.add( EntryPoint.class);
65     aList.add( ExitPoint.class);
66     aList.add( FinalState.class);
67     aList.add( Fork.class);
68     aList.add( InitialState.class);

```

```

69     aList.add(Join.class);
70     aList.add(Junction.class);
71     aList.add(State.class);
72     aList.add(SubmachineState.class);
73     aList.add(Terminate.class);
74     aList.add(Transition.class);
75     return aList;
76 }
77
78 /*
79  * (non-Javadoc) Realization
80  *
81  * @see ocean.framework.ConceptualModel#modelName()
82  */
83 public String modelName()
84 {
85     return I18NProperties.getString("state.machine.diagram.name");
86 }
87
88 /*
89  * (non-Javadoc)
90  *
91  * @see ocean.framework.OceanDocument#getNameForTree()
92  */
93 public String getNameForTree()
94 {
95     return I18NProperties.getString("state.machine.diagram.name.for.
96         tree") + name;
97 }
98 }

```

Listagem B.10: Classe **Component.java**

```

1 package documents.concepts.component_diagram;
2
3 import ocean.framework.Concept;
4 import ocean.smalltalk.OceanVector;
5 import documents.util.I18NProperties;
6
7 public class Component extends Concept
8 {
9

```

```

10  /**
11   * The serial version UID.
12   */
13  private static final long serialVersionUID = -3127608563350981688L;
14
15  /**
16   * The name of component.
17   */
18  private String componentName = null;
19
20  /**
21   * The type of component.
22   */
23  private String componentType = null;
24
25  /**
26   * Official constructor.
27   */
28  public Component()
29  {
30      super ();
31      this.componentName = new String("componente");
32      this.componentType = new String("<<component>>");
33      name("componente");
34  }
35
36  /*
37   * (non-Javadoc)
38   *
39   * @see ocean.framework.Concept#conceptName()
40   */
41  public String conceptName()
42  {
43      return I18NProperties.getString("component.concept.name");
44  }
45
46  /*
47   * (non-Javadoc)
48   *
49   * @see ocean.framework.OceanDocument#getNameForTree()
50   */
51  public String getNameForTree()
52  {

```

```

53         return I18NProperties.getString("component.concept.name.for.tree")
           + name();
54     }
55
56     /*
57     * (non-Javadoc)
58     *
59     * @see ocean.framework.Concept#mustHaveName()
60     */
61     public boolean mustHaveName()
62     {
63         return false;
64     }
65
66     /**
67     * @return the componentName
68     */
69     public String getComponentName()
70     {
71         return componentName;
72     }
73
74     /**
75     * @param componentName the componentName to set
76     */
77     public void setComponentName(String componentName)
78     {
79         this.componentName = componentName;
80     }
81
82     /**
83     * @return the componentType
84     */
85     public String getComponentType()
86     {
87         return componentType;
88     }
89
90     /**
91     * @param componentType the componentType to set
92     */
93     public void setComponentType(String componentType)
94     {

```

```

95     this.componentType = componentType;
96 }
97
98 /*
99  * FIXME (non-Javadoc)
100  *
101  * @see ocean.framework.Concept#duplicityIn(ocean.smalltalk.OceanVector
102     )
103  */
104 @SuppressWarnings( { "deprecation", "unchecked" })
105 public Concept duplicityIn(OceanVector aConceptList)
106 {
107     return super.duplicityIn(aConceptList);
108 }
109 }

```

Listagem B.11: Classe **Port.java**

```

1 package documents.concepts.component_diagram;
2
3 import documents.util.I18NProperties;
4 import ocean.framework.Concept;
5
6 public class Port extends Concept
7 {
8
9     /**
10     * The serial version UID.
11     */
12     private static final long serialVersionUID = -6662722305450121658L;
13
14     /**
15     * Official constructor.
16     */
17     public Port()
18     {
19         super();
20         name("port");
21     }
22
23     /*
24     * (non-Javadoc)

```

```

25     *
26     * @see ocean.framework.Concept#conceptName()
27     */
28     public String conceptName()
29     {
30         return I18NProperties.getString("port.concept.name");
31     }
32
33     /*
34     * (non-Javadoc)
35     *
36     * @see ocean.framework.OceanDocument#getNameForTree()
37     */
38     public String getNameForTree()
39     {
40         return I18NProperties.getString("port.concept.name.for.tree") +
41             name();
42     }
43     /*
44     * (non-Javadoc)
45     *
46     * @see ocean.framework.Concept#mustHaveName()
47     */
48     public boolean mustHaveName()
49     {
50         return false;
51     }
52
53 }

```

Listagem B.12: Classe **Interface.java**

```

1 package documents.concepts.component_diagram;
2
3 import documents.util.I18NProperties;
4 import ocean.framework.Concept;
5
6 public class Interface extends Concept
7 {
8
9     /**
10     * The serial version UID.

```



```

11     */
12     private static final long serialVersionUID = 5550319748632533836L;
13
14     /**
15      * The name of the interface.
16      */
17     private String interfaceName = null;
18
19     /**
20      * Official constructor.
21      */
22     public Interface ()
23     {
24         super ();
25         this.interfaceName = new String ("interface");
26         name ("interface");
27     }
28
29     /*
30      * (non-Javadoc)
31      *
32      * @see ocean.framework.Concept#conceptName ()
33      */
34     public String conceptName ()
35     {
36         return I18NProperties.getString ("interface.concept.name");
37     }
38
39     /*
40      * (non-Javadoc)
41      *
42      * @see ocean.framework.OceanDocument#getNameForTree ()
43      */
44     public String getNameForTree ()
45     {
46         return I18NProperties.getString ("interface.concept.name.for.tree")
47             + name ();
48     }
49
50     /*
51      * (non-Javadoc)
52      *
53      * @see ocean.framework.Concept#mustHaveName ()

```

```

53     */
54     public boolean mustHaveName ()
55     {
56         return false ;
57     }
58
59     /**
60     * @return the interfaceName
61     */
62     public String getInterfaceName ()
63     {
64         return interfaceName ;
65     }
66
67     /**
68     * @param interfaceName the interfaceName to set
69     */
70     public void setInterfaceName (String interfaceName)
71     {
72         this.interfaceName = interfaceName ;
73     }
74
75 }

```

Listagem B.13: Classe **ComponentDiagramaDrawing.java**

```

1  package documents . graphical . component_diagram ;
2
3  import ocean . documents . oo . concepts . * ;
4  import ocean . framework . Concept ;
5  import ocean . jhotdraw . * ;
6  import documents . concepts . component_diagram . Component ;
7  import documents . concepts . component_diagram . Interface ;
8  import documents . concepts . component_diagram . Port ;
9  import documents . concepts . object_diagram . Dependency ;
10 import documents . concepts . object_diagram . Realization ;
11 import documents . graphical . object_diagram . * ;
12 import documents . graphical . object_diagram . AssociationLineConnection ;
13
14 public class ComponentDiagramDrawing extends SpecificationDrawing
15 {
16
17     /**

```

```

18     * The serial version UID.
19     */
20     private static final long serialVersionUID = 8993678782536928458L;
21
22     /**
23     * Official constructor.
24     */
25     public ComponentDiagramDrawing ()
26     {
27         super ();
28     }
29
30     /*
31     * (non-Javadoc)
32     *
33     * @see ocean.jhotdraw.SpecificationDrawing#
34         createDesiredFigureForConcept (ocean.framework.Concept)
35     */
36     @Override
37     public SpecificationCompositeFigure createDesiredFigureForConcept (
38         Concept aConcept)
39     {
40         SpecificationCompositeFigure figure = null;
41         if (aConcept instanceof Component)
42         {
43             figure = new ComponentFigure ();
44         }
45         else if (aConcept instanceof Interface)
46         {
47             figure = new InterfaceFigure ();
48         }
49         else if (aConcept instanceof Port)
50         {
51             figure = new PortFigure ();
52         }
53         return figure;
54     }
55
56     /*
57     * (non-Javadoc)
58     *
59     * @see ocean.jhotdraw.SpecificationDrawing#startFigureFor (ocean.
60         framework.Concept)

```

```

58     */
59     public SpecificationCompositeFigure startFigureFor(Concept aConcept)
60     {
61         Concept auxComp = null;
62
63         String concNameDependency = Concept.conceptName(Dependency.class);
64         String concNameAgregacao = Concept.conceptName(Agregacao.class);
65         String concNameAssociacao = Concept.conceptName(AssociacaoBinaria.
66             class);
67         String concNameRealization = Concept.conceptName(Realization.class)
68             ;
69         String concNameInheritance = Concept.conceptName(Heranca.class);
70         String concNameComposition = Concept.conceptName(Composition.class)
71             ;
72
73         if (aConcept.conceptName().equals(concNameDependency))
74         {
75             auxComp = ((Dependency) aConcept).dependentComponent();
76         }
77         else if (aConcept.conceptName().equals(concNameAgregacao))
78         {
79             auxComp = ((Agregacao) aConcept).agreggadoComponent();
80         }
81         else if (aConcept.conceptName().equals(concNameAssociacao))
82         {
83             auxComp = ((AssociacaoBinaria) aConcept).componentTerminal1();
84         }
85         else if (aConcept.conceptName().equals(concNameRealization))
86         {
87             auxComp = ((Realization) aConcept).realizedComponent();
88         }
89         else if (aConcept.conceptName().equals(concNameInheritance))
90         {
91             auxComp = ((Heranca) aConcept).subComponent();
92         }
93         else if (aConcept.conceptName().equals(concNameComposition))
94         {
95             auxComp = ((Composition) aConcept).agreggadoComponent();
96         }
97         return (SpecificationCompositeFigure) getFigureOfConcept(auxComp);
98     }
99     /*

```

```

98     * (non-Javadoc) mponent.conceptName().equals(concNameAssociacao))
99     *
100    * @see ocean.jhotdraw.SpecificationDrawing#stopFigureFor(ocean.
101      framework.Concept)
102    */
103    public SpecificationCompositeFigure stopFigureFor(Concept aConcept)
104    {
105        Concept auxComp = null;
106
107        String concNameDependency = Concept.conceptName(Dependency.class);
108        String concNameAgregacao = Concept.conceptName(Agregacao.class);
109        String concNameAssociacao = Concept.conceptName(AssociacaoBinaria.
110            class);
111        String concNameRealization = Concept.conceptName(Realization.class)
112            ;
113        String concNameInheritance = Concept.conceptName(Heranca.class);
114        String concNameComposition = Concept.conceptName(Composition.class)
115            ;
116
117        if (aConcept.conceptName().equals(concNameDependency))
118        {
119            auxComp = ((Dependency) aConcept).dependencyComponent();
120        }
121        else if (aConcept.conceptName().equals(concNameAgregacao))
122        {
123            auxComp = ((Agregacao) aConcept).agreggaterComponent();
124        }
125        else if (aConcept.conceptName().equals(concNameAssociacao))
126        {
127            auxComp = ((AssociacaoBinaria) aConcept).componentTerminal2();
128        }
129        else if (aConcept.conceptName().equals(concNameRealization))
130        {
131            auxComp = ((Realization) aConcept).realizaterComponent();
132        }
133        else if (aConcept.conceptName().equals(concNameInheritance))
134        {
135            auxComp = ((Heranca) aConcept).superComponent();
136        }
137        else if (aConcept.conceptName().equals(concNameComposition))
138        {
139            auxComp = ((Composition) aConcept).agreggaterComponent();
140        }

```

```

137     return ( SpecificationCompositeFigure ) getFigureOfConcept (auxComp);
138 }
139
140 /*
141  * (non-Javadoc)
142  *
143  * @see ocean.jhotdraw.SpecificationDrawing#createDesiredLineForConcept
144  *      (ocean.framework.Concept)
145  */
146 public SpecificationLineFigure createDesiredLineForConcept (Concept
147     aConcept)
148 {
149     SpecificationLineFigure aLine = null;
150
151     String concNameDependency = Concept.conceptName (Dependency.class);
152     String concNameHeranca = Concept.conceptName (Heranca.class);
153     String concNameAgregacao = Concept.conceptName (Agregacao.class);
154     String concNameAssociacao = Concept.conceptName (AssociacaoBinaria.
155         class);
156     String concNameRealization = Concept.conceptName (Realization.class)
157         ;
158     String concNameInheritance = Concept.conceptName (Heranca.class);
159     String concNameComposition = Concept.conceptName (Composition.class)
160         ;
161
162     if ( aConcept.conceptName () . equals ( concNameDependency ) )
163     {
164         aLine = new DependencyLineConnection ();
165     }
166     else if ( aConcept.conceptName () . equals ( concNameAgregacao ) )
167     {
168         aLine = new AggregationLineConnection ();
169     }
170     else if ( aConcept.conceptName () . equals ( concNameAssociacao ) )
171     {
172         aLine = new AssociationLineConnection ();
173     }
174     else if ( aConcept.conceptName () . equals ( concNameHeranca ) )
175     {

```

```

175         aLine = new RealizationLineConnection ();
176     }
177     else if ( aConcept.conceptName().equals(concNameInheritance))
178     {
179         aLine = new InheritanceLineConnection ();
180     }
181     else if ( aConcept.conceptName().equals(concNameComposition))
182     {
183         aLine = new CompositionLineConnection ();
184     }
185
186     return aLine;
187 }
188
189 }

```

Listagem B.14: Classe **ComponentDiagramDrawingView.java**

```

1 package documents.graphical.component_diagram;
2
3 import ocean.jhotdraw.SpecificationDrawingView;
4 import ocean.jhotdraw.SpecificationEditor;
5
6 public class ComponentDiagramDrawingView extends SpecificationDrawingView
7 {
8
9     /**
10      * The serial version UID.
11      */
12     private static final long serialVersionUID = 1959402307798478308L;
13
14     /**
15      * @param editor
16      */
17     public ComponentDiagramDrawingView(SpecificationEditor editor)
18     {
19         super(editor);
20     }
21
22     /**
23      * Constructor.
24      *
25      * @param editor

```

```

26     * @param width
27     * @param height
28     */
29     public ComponentDiagramDrawingView( SpecificationEditor editor , int
        width , int height )
30     {
31         super( editor , width , height );
32     }
33
34 }

```

Listagem B.15: Classe **ComponentDiagramEditor.java**

```

1  package documents . graphical . component _ diagram ;
2
3  import java . awt . event . MouseEvent ;
4
5  import javax . swing . JToolBar ;
6
7  import ocean . framework . Concept ;
8  import ocean . jhotdraw . * ;
9  import CH . ifa . draw . framework . Tool ;
10 import CH . ifa . draw . standard . ConnectionTool ;
11 import CH . ifa . draw . standard . CreationTool ;
12 import CH . ifa . draw . standard . ToolButton ;
13 import documents . concepts . component _ diagram . Component ;
14 import documents . concepts . component _ diagram . Interface ;
15 import documents . concepts . component _ diagram . Port ;
16 import documents . graphical . object _ diagram . * ;
17 import documents . graphical . object _ diagram . AssociationLineConnection ;
18 import documents . util . I18NPrinter ;
19 import documents . util . I18NProperties ;
20
21 public class ComponentDiagramEditor extends SpecificationEditor
22 {
23
24     /**
25     * The serial version UID .
26     */
27     private static final long serialVersionUID = -5413087507406938906L ;
28
29     /**
30     * Diagram images path .

```



```

31     */
32     protected final static String FG_OBJECT_DIAGRAM_PATH = new String (
33         "/ocean/documents/oo/graphical/objectDiagram/");
34
35     protected final static String FG_CLASS_DIAGRAM_PATH = new String (
36         "/ocean/documents/oo/graphical/classDiagram/");
37
38     protected final static String FG_COMPONENT_DIAGRAM_PATH = new String (
39         "/ocean/documents/oo/graphical/componentDiagram/");
40
41     /**
42      * Overloaded constructor.
43      *
44      * @param view
45      */
46     public ComponentDiagramEditor(ComponentDiagramDrawingView view)
47     {
48         super(view);
49     }
50
51     /**
52      * Official constructor.
53      */
54     public ComponentDiagramEditor()
55     {
56         super();
57     }
58
59     /*
60      * (non-Javadoc)
61      *
62      * @see ocean.jhotdraw.SpecificationEditor#createDrawingViewObject()
63      */
64     @Override
65     protected SpecificationDrawingView createDrawingViewObject()
66     {
67         return new ComponentDiagramDrawingView(this);
68     }
69
70     /*
71      * (non-Javadoc)
72      *
73      * @see ocean.jhotdraw.SpecificationEditor#createSpecificationDrawing()

```

```

74     */
75     @Override
76     protected SpecificationDrawing createSpecificationDrawing ()
77     {
78         return new ComponentDiagramDrawing ();
79     }
80
81     /*
82     * (non-Javadoc) ackage
83     *
84     * @see ocean.jhotdraw.SpecificationEditor#createTools(javax.swing.
85         JToolBar)
86     */
87     @Override
88     protected void createTools(JToolBar palette)
89     {
90
91         // COMPONENT
92         Tool tool = new CreationTool(this, new ComponentFigure())
93         {
94
95             /*
96             * (non-Javadoc)
97             *
98             * @see CH.ifa.draw.standard.CreationTool#mouseUp(java.awt.
99                 event.MouseEvent, int, int)
100            */
101            public void mouseUp(MouseEvent e, int x, int y)
102            {
103                Concept conc = getModel().createComponent(Component.class);
104
105                if (conc == null)
106                {
107                    I18NPrinter.printMessage("component.not.created");
108                }
109                else
110                {
111                    ComponentFigure componentFig = (ComponentFigure)
112                        getCreatedFigure();
113                    componentFig.concept(conc);
114                    conc.getObservable().addObserver(componentFig);
115                    conc.redraw();

```

```

114         }
115         super.mouseUp(e, x, y);
116     }
117
118 };
119 ToolButton tB = createToolButton(FG_COMPONENT_DIAGRAM_PATH
120     + I18NProperties.getString("component"), I18NProperties.
121     getString("new.component"),
122     tool);
123 tB.setEnabled(true);
124 palette.add(tB);
125
126 // PORT
127 tool = new CreationTool(this, new PortFigure())
128 {
129     /*
130     * (non-Javadoc)
131     *
132     * @see CH.ifa.draw.standard.CreationTool#mouseUp(java.awt.
133     * event.MouseEvent, int, int)
134     */
135     public void mouseUp(MouseEvent e, int x, int y)
136     {
137         Concept conc = getModel().createComponent(Port.class);
138
139         if (conc == null)
140         {
141             I18NPrinter.printMessage("port.not.created");
142         }
143         else
144         {
145             PortFigure portFig = (PortFigure) getCreatedFigure();
146             portFig.concept(conc);
147             portFig.setDrawing((ComponentDiagramDrawing) getDrawing
148                 ());
149             conc.getObservable().addObserver(portFig);
150             conc.redraw();
151         }
152         super.mouseUp(e, x, y);
153     }
154 };

```

```

154     tB = createToolButton (FG_COMPONENT_DIAGRAM_PATH + I18NProperties .
        getString ("port"),
155         I18NProperties . getString ("new.port"), tool);
156     tB.setEnabled (true);
157     palette.add (tB);
158
159     // INTERFACE
160     tool = new CreationTool (this, new InterfaceFigure ())
161     {
162
163         /*
164         * (non-Javadoc)
165         *
166         * @see CH.ifa.draw.standard.CreationTool#mouseUp(java.awt.
            event.MouseEvent, int, int)
167         */
168         public void mouseUp (MouseEvent e, int x, int y)
169         {
170             Concept conc = getModel ().createComponent (Interface.class);
171
172             if (conc == null)
173             {
174                 I18NPrinter .printMessage ("interface.not.created");
175             }
176             else
177             {
178                 ((InterfaceFigure) getCreatedFigure ().concept (conc);
179                 conc.getObservable ().addObserver ((InterfaceFigure)
                    getCreatedFigure ());
180                 conc.redraw ();
181             }
182             super.mouseUp (e, x, y);
183         }
184
185     };
186     tB = createToolButton (FG_COMPONENT_DIAGRAM_PATH + I18NProperties .
        getString ("interface"),
187         I18NProperties . getString ("new.interface"), tool);
188     tB.setEnabled (true);
189     palette.add (tB);
190
191     // ASSOCIATION
192     tool = new ConnectionTool (this, new AssociationLineConnection ())

```

```

193     {
194
195         /*
196         * (non-Javadoc)
197         *
198         * @see CH.ifa.draw.standard.ConnectionTool#mouseUp(java.awt.
199         *     event.MouseEvent, int, int)
200         */
201     public void mouseUp(MouseEvent e, int x, int y)
202     {
203         AssociationLineConnection line = (AssociationLineConnection
204             ) getConnection();
205         super.mouseUp(e, x, y);
206
207         if (stopCreateConcept(line))
208         {
209             return;
210         }
211
212         Concept start = ((SpecificationCompositeFigure) line.
213             startFigure()).concept();
214         Concept end = ((SpecificationCompositeFigure) line.
215             endFigure()).concept();
216
217         Concept conc = getModel().createComponent_with_and(line.
218             relatedConceptClass(),
219             start, end);
220
221         if (conc == null)
222         {
223             getDrawing().remove(line);
224         }
225
226         line.concept(conc);
227     }
228 };
229 tB = createToolButton(FG_CLASS_DIAGRAM_PATH + I18NProperties.
230     getString("association"),
231     I18NProperties.getString("new.association"), tool);
232 tB.setEnabled(true);
233 palette.add(tB);

```

```

230 // AGREGATION
231 tool = new ConnectionTool(this, new AggregationLineConnection())
232 {
233
234     /*
235     * (non-Javadoc)
236     *
237     * @see CH.ifa.draw.standard.ConnectionTool#mouseUp(java.awt.
238     *     event.MouseEvent, int, int)
239     */
240     public void mouseUp(MouseEvent e, int x, int y)
241     {
242         AggregationLineConnection line = (AggregationLineConnection
243             ) getConnection();
244         super.mouseUp(e, x, y);
245
246         if (stopCreateConcept(line))
247         {
248             return;
249         }
250
251         Concept start = ((SpecificationCompositeFigure) line.
252             startFigure()).concept();
253         Concept end = ((SpecificationCompositeFigure) line.
254             endFigure()).concept();
255
256         Concept conc = getModel().createComponent_with_and(line.
257             relatedConceptClass(),
258             start, end);
259
260         if (conc == null)
261         {
262             getDrawing().remove(line);
263         }
264
265         line.concept(conc);
266     }
267 };
268 tB = createToolButton(FG_OBJECT_DIAGRAM_PATH + I18NProperties.
269     getString("aggregation"),
270     I18NProperties.getString("new.aggregation"), tool);
271 tB.setEnabled(true);

```

```

267     palette.add(tB);
268
269     // COMPOSITION
270     tool = new ConnectionTool(this, new CompositionLineConnection())
271     {
272
273         /*
274          * (non-Javadoc)
275          *
276          * @see CH.ifa.draw.standard.ConnectionTool#mouseUp(java.awt.
277           event.MouseEvent, int, int)
278          */
279         public void mouseUp(MouseEvent e, int x, int y)
280         {
281             CompositionLineConnection line = (CompositionLineConnection
282                 ) getConnection();
283             super.mouseUp(e, x, y);
284
285             if (stopCreateConcept(line))
286             {
287                 return;
288             }
289
290             Concept start = ((SpecificationCompositeFigure) line.
291                 startFigure()).concept();
292             Concept end = ((SpecificationCompositeFigure) line.
293                 endFigure()).concept();
294
295             Concept conc = getModel().createComponent_with_and(line.
296                 relatedConceptClass(),
297                 start, end);
298
299             if (conc == null)
300             {
301                 getDrawing().remove(line);
302             }
303
304             line.concept(conc);
305         }
306     };
307     tB = createToolButton(FG_OBJECT_DIAGRAM_PATH + I18NProperties.
308         getString("composition"),

```

```

304         I18NProperties.getString("new.composition"), tool);
305     tB.setEnabled(true);
306     palette.add(tB);
307
308     // INHERITANCE
309     tool = new ConnectionTool(this, new InheritanceLineConnection())
310     {
311
312         /*
313          * (non-Javadoc)
314          *
315          * @see CH.ifa.draw.standard.ConnectionTool#mouseUp(java.awt.
316           event.MouseEvent, int, int)
317          */
318         public void mouseUp(MouseEvent e, int x, int y)
319         {
320             InheritanceLineConnection line = (InheritanceLineConnection
321             ) getConnection();
322             super.mouseUp(e, x, y);
323
324             if (stopCreateConcept(line))
325             {
326                 return;
327             }
328
329             Concept start = ((SpecificationCompositeFigure) line.
330             startFigure()).concept();
331             Concept end = ((SpecificationCompositeFigure) line.
332             endFigure()).concept();
333
334             Concept conc = getModel().createComponent_with_and(line.
335             relatedConceptClass(),
336             start, end);
337
338             if (conc == null)
339             {
340                 getDrawing().remove(line);
341             }
342
343             line.concept(conc);
344         }
345     };

```



```

342     tB = createToolButton(FG_CLASS_DIAGRAM_PATH + I18NProperties .
           getString("inheritance"),
343         I18NProperties . getString("new.inheritance"), tool);
344     tB.setEnabled(true);
345     palette.add(tB);
346
347     // DEPENDENCY
348     tool = new ConnectionTool(this, new DependencyLineConnection())
349     {
350
351         /*
352          * (non-Javadoc)
353          *
354          * @see CH.ifa.draw.standard.ConnectionTool#mouseUp(java.awt.
           event.MouseEvent, int, int)
355          */
356         public void mouseUp(MouseEvent e, int x, int y)
357         {
358             DependencyLineConnection line = (DependencyLineConnection)
           getConnection();
359             super.mouseUp(e, x, y);
360
361             if (stopCreateConcept(line))
362             {
363                 return;
364             }
365
366             Concept start = ((SpecificationCompositeFigure) line .
           startFigure()).concept();
367             Concept end = ((SpecificationCompositeFigure) line .
           endFigure()).concept();
368
369             Concept conc = getModel().createComponent_with_and(line .
           relatedConceptClass(),
370                 start, end);
371
372             if (conc == null)
373             {
374                 getDrawing().remove(line);
375             }
376
377             line.concept(conc);
378

```

```

379     }
380 };
381 tB = createToolButton(FG_CLASS_DIAGRAM_PATH + I18NProperties .
    getString("dependency"),
382     I18NProperties . getString("new.dependency"), tool);
383 tB.setEnabled(true);
384 palette.add(tB);
385
386 // REALIZATION
387 tool = new ConnectionTool(this, new RealizationLineConnection())
388 {
389
390     /*
391     * (non-Javadoc)
392     *
393     * @see CH.ifa.draw.standard.ConnectionTool#mouseUp(java.awt.
    event.MouseEvent, int, int)
394     */
395     public void mouseUp(MouseEvent e, int x, int y)
396     {
397         RealizationLineConnection line = (RealizationLineConnection
    ) getConnection();
398         super.mouseUp(e, x, y);
399
400         if (stopCreateConcept(line))
401         {
402             return;
403         }
404
405         Concept start = ((SpecificationCompositeFigure) line .
    startFigure()).concept();
406         Concept end = ((SpecificationCompositeFigure) line .
    endFigure()).concept();
407
408         Concept conc = getModel().createComponent_with_and(line .
    relatedConceptClass(),
409             start, end);
410
411         if (conc == null)
412         {
413             getDrawing().remove(line);
414         }
415

```

```

416         line . concept ( conc );
417
418     }
419 };
420 tB = createToolButton ( FG.OBJECT_DIAGRAM_PATH + I18NProperties .
         getString ( "realization" ),
421     I18NProperties . getString ( "new.realization" ), tool );
422 tB . setEnabled ( true );
423 palette . add ( tB );
424 }
425
426 }

```

Listagem B.16: Classe **ComponentDiagramWindow.java**

```

1  package documents . graphical . component_diagram ;
2
3  import java . io . Serializable ;
4
5  import ocean . framework . ConceptualModel ;
6  import ocean . jhotdraw . * ;
7  import ocean . smalltalk . IComunicacao ;
8
9  public class ComponentDiagramWindow extends EditorAuxWindow implements
        IComunicacao
10 {
11
12     /**
13      * The serial version UID.
14      */
15     private static final long serialVersionUID = 1316124310994994030L ;
16
17     /**
18      * Official constructor.
19      */
20     public ComponentDiagramWindow ()
21     {
22         super ( new ComponentDiagramEditor () );
23         setDrawingView ( ( SpecificationDrawingView ) getEditor () . view () );
24     }
25
26     /**
27      * Official constructor.

```

```

28     *
29     * @param editor The specification editor.
30     */
31     public ComponentDiagramWindow( SpecificationEditor editor )
32     {
33         super( editor );
34     }
35
36     /*
37     * (non-Javadoc)
38     *
39     * @see ocean.jhotdraw.EditorAuxWindow#getSelected()
40     */
41     public Serializable getSelected()
42     {
43         return getDrawingView().getSelected();
44     }
45
46     /*
47     * (non-Javadoc)
48     *
49     * @see ocean.jhotdraw.EditorAuxWindow#setDocument(java.lang.Object)
50     */
51     public void setDocument( Object obj )
52     {
53         this.handledElement( ( ConceptualModel ) obj );
54         (( ComponentDiagramEditor ) getEditor()).setDocument( obj );
55         this.getEditor().view().setDrawing( (( ConceptualModel ) obj).drawing() );
56         this.setDrawing( ( SpecificationDrawing ) getEditor().view().drawing() );
57     }
58
59     /*
60     * (non-Javadoc)
61     *
62     * @see ocean.jhotdraw.EditorAuxWindow#getDocument()
63     */
64     public Object getDocument()
65     {
66         return handledElement();
67     }
68

```

```

69     /* (non-Javadoc)
70     * @see ocean.jhotdraw.EditorAuxWindow#show()
71     */
72     public void show()
73     {
74         this.setVisible(true);
75     }
76
77 }

```

Listagem B.17: Classe **ComponentFigure.java**

```

1  package documents.graphical.component_diagram;
2
3  import java.awt.*;
4  import java.awt.event.ActionEvent;
5  import java.util.LinkedList;
6  import java.util.List;
7  import java.util.Vector;
8
9  import javax.swing.AbstractAction;
10 import javax.swing.JPopupMenu;
11
12 import ocean.accessories.SingleConnector;
13 import ocean.jhotdraw.SpecificationCompositeFigure;
14 import ocean.smalltalk.Constantes;
15 import ocean.smalltalk.OceanVector;
16 import CH.ifa.draw.contrib.GraphicalCompositeFigure;
17 import CH.ifa.draw.figures.PolyLineFigure;
18 import CH.ifa.draw.figures.RectangleFigure;
19 import CH.ifa.draw.framework.Figure;
20 import CH.ifa.draw.framework.FigureEnumeration;
21 import CH.ifa.draw.standard.NullHandle;
22 import CH.ifa.draw.standard.RelativeLocator;
23 import documents.concepts.component_diagram.Component;
24 import documents.concepts.deployment_diagram.Node;
25 import documents.graphical.deployment_diagram.DeploymentDiagramDrawing;
26 import documents.graphical.deployment_diagram.ExtendedTextFigure;
27 import documents.graphical.deployment_diagram.NodeFigure;
28 import documents.util.I18NProperties;
29
30 public class ComponentFigure extends SpecificationCompositeFigure
31 {

```

```

32
33  /**
34   * The serial version UID.
35   */
36  private static final long serialVersionUID = -3360539347429614894L;
37
38  /**
39   * A list of port figures.
40   */
41  private List<PortFigure> portFigureList = new LinkedList<PortFigure>();
42
43  /**
44   * A node figure.
45   */
46  private NodeFigure nodeFig;
47
48  /**
49   * The drawing.
50   */
51  private DeploymentDiagramDrawing drawing;
52
53  /**
54   * Default constructor.
55   */
56  public ComponentFigure ()
57  {
58      super(new RectangleFigure ());
59  }
60
61  /**
62   * Overloaded constructor.
63   *
64   * @param object
65   */
66  public ComponentFigure(Component component)
67  {
68      super (component);
69  }
70
71  /*
72   * (non-Javadoc)
73   *
74   * @see ocean.jhotdraw.SpecificationCompositeFigure#relatedConceptClass

```

```

    ()
75     */
76     @Override
77     public Class<Component> relatedConceptClass ()
78     {
79         return Component.class ;
80     }
81
82     /**
83     * @return
84     */
85     @SuppressWarnings( { "deprecation", "unchecked" })
86     public static OceanVector memberRelatedConceptClass ()
87     {
88         OceanVector auxList = new OceanVector ();
89         auxList.add(Component.class);
90         return auxList;
91     }
92
93     /*
94     * (non-Javadoc)
95     *
96     * @see ocean.jhotdraw.SpecificationCompositeFigure#initialize ()
97     */
98     public void initialize ()
99     {
100         removeAll ();
101
102         setAttribute (Figure.POPUP_MENU, createPopupMenu ());
103
104         super.initialize ();
105     }
106
107     /*
108     * (non-Javadoc)
109     *
110     * @see CH.ifa.draw.standard.AbstractFigure#moveBy (int , int )
111     */
112     public void moveBy (int dx, int dy)
113     {
114
115         // It doesn't belong to a node.
116         if ((nodeFig == null) && (drawing != null))

```

```

117     {
118         int xComp = this.displayBox().x;
119         int yComp = this.displayBox().y;
120         int wComp = this.displayBox().width;
121         int hComp = this.displayBox().height;
122
123         FigureEnumeration enumeration = drawing.figures();
124         while (enumeration.hasMoreElements())
125         {
126             Figure nextFigure = enumeration.nextFigure();
127             if (nextFigure instanceof NodeFigure)
128             {
129                 Rectangle displayBox = nextFigure.displayBox();
130                 int xNode = displayBox.x;
131                 int yNode = displayBox.y;
132                 int wNode = displayBox.width;
133                 int hNode = displayBox.height;
134
135                 // Checks the restrictions.
136                 if (((xComp > xNode) && (xComp < (xNode + wNode - wComp)
137                    && (yComp < (yNode + hNode - hComp)) && (yComp
138                    > yNode))
139                    || ((xComp < (xNode + wNode - wComp)) && (xComp
140                    > xNode) /* RIGHT */
141                    && (yComp > yNode) && (yComp < (yNode +
142                    hNode - hComp)))
143                    || ((yComp > yNode) && (yComp < (yNode + hNode
144                    - hComp)) /* TOP */
145                    && (xComp > xNode) && (xComp < (xNode +
146                    wNode - wComp)))
147                    || ((xComp > xNode) && (xComp < (xNode + wNode
148                    - wComp)) /* LEFT */
149                    && (yComp > yNode) && (yComp < (yNode +
150                    hNode - hComp))))

```



```

151     }
152 }
153 // It belongs to a package.
154 else if ((nodeFig != null) && (drawing != null))
155 {
156
157     int x = nodeFig.getPresentationFigure().displayBox().x;
158     int y = nodeFig.getPresentationFigure().displayBox().y;
159     int width = nodeFig.getPresentationFigure().displayBox().width;
160     int height = nodeFig.getPresentationFigure().displayBox().
161         height;
162
163     final int w = (width == 0) ? 0 : width;
164     final int h = (height == 0) ? 0 : height;
165
166     int oldX = this.displayBox().x;
167     int oldY = this.displayBox().y;
168     int classWidth = this.displayBox().width;
169     int classHeight = this.displayBox().height;
170
171     int leftBoundary = x + 10;
172     int rightBoundary = x + w - 10 - classWidth;
173     int topBoundary = y + 5;
174     int bottomBoundary = y + h - 10 - classHeight;
175
176     int newX = dx;
177     int newY = dy;
178
179     // Add restriction inside of the package.
180     if (((oldX + dx) < leftBoundary) || ((oldX + dx) >
181         rightBoundary)
182         || ((oldY + dy) < topBoundary) || ((oldY + dy) >
183         bottomBoundary))
184     {
185         // Don't move!!!
186         return;
187     }
188
189     super.moveBy(newX, newY);
190 }

```

```

191         super.moveBy(dx, dy);
192     }
193
194     for (PortFigure portFig : portFigureList)
195     {
196         portFig.moveBy(dx, dy);
197     }
198 }
199
200 /*
201  * (non-Javadoc)
202  *
203  * @see ocean.jhotdraw.SpecificationCompositeFigure#redraw()
204  */
205 @SuppressWarnings("deprecation")
206 public void redraw()
207 {
208
209     removeAll();
210
211     Component component = (Component) this.concept();
212     if (component.getComponentName() == null || "".equals(component.
213         getComponentName()))
214     {
215         component.setComponentName("componente");
216     }
217
218     if (component.getComponentType() == null || "".equals(component.
219         getComponentType()))
220     {
221         component.setComponentType("<<component>>");
222     }
223
224     // NAME
225     Font fontName = new Font("Helvetica", Font.BOLD, 14);
226     FontMetrics metrics1 = Toolkit.getDefaultToolkit().getFontMetrics(
227         fontName);
228     int widthFig1 = metrics1.stringWidth(component.getComponentName());
229
230     // TYPE
231     Font fontType = new Font("Helvetica", Font.ITALIC, 11);
232     FontMetrics metrics2 = Toolkit.getDefaultToolkit().getFontMetrics(
233         fontType);

```

```

230     int widthFig2 = metrics2.stringWidth(component.getComponentType());
231
232     ExtendedTextFigure figure;
233     ExtendedTextFigure figure2;
234     int w = 0;
235
236     if (widthFig1 >= widthFig2)
237     {
238         w = widthFig1;
239         // NAME
240         figure = new ExtendedTextFigure();
241         // TYPE
242         figure2 = new ExtendedTextFigure((widthFig1 - widthFig2) / 2);
243     }
244     else
245     {
246         w = widthFig2;
247         // NAME
248         figure = new ExtendedTextFigure((widthFig2 - widthFig1) / 2);
249         // TYPE
250         figure2 = new ExtendedTextFigure();
251     }
252
253     // NAME
254     figure.setFont(fontName);
255     figure.setText(component.getComponentName());
256     // TYPE
257     figure2.setFont(fontType);
258     figure2.setText(component.getComponentType());
259
260     GraphicalCompositeFigure gFigure = new GraphicalCompositeFigure();
261     gFigure.getLayouter().setInsets(new Insets(40, 30, 20, 2));
262     gFigure.setAttribute("FillColor", new Color(0xE8E8E8));
263     gFigure.add(figure2);
264     gFigure.add(figure);
265
266     int x = this.getPresentationFigure().displayBox().x;
267     int y = this.getPresentationFigure().displayBox().y;
268
269     PolyLineFigure poly = new PolyLineFigure();
270
271     poly.addPoint(x + w + 19, y + 15);
272     poly.addPoint(x + w + 29, y + 15);

```

```
273     poly.addPoint(x + w + 29, y + 18);
274     poly.addPoint(x + w + 19, y + 18);
275     poly.addPoint(x + w + 19, y + 15);
276
277     poly.setAttribute("FillColor", new Color(0xE8E8E8));
278     poly.setAttribute("FrameColor", Color.BLACK);
279
280     gFigure.add(poly);
281
282     poly = new PolyLineFigure();
283
284     poly.addPoint(x + w + 19, y + 20);
285     poly.addPoint(x + w + 29, y + 20);
286     poly.addPoint(x + w + 29, y + 23);
287     poly.addPoint(x + w + 19, y + 23);
288     poly.addPoint(x + w + 19, y + 20);
289
290     poly.setAttribute("FillColor", new Color(0xE8E8E8));
291     poly.setAttribute("FrameColor", Color.BLACK);
292
293     gFigure.add(poly);
294
295     poly = new PolyLineFigure();
296
297     poly.addPoint(x + w + 24, y + 18);
298     poly.addPoint(x + w + 24, y + 20);
299
300     poly.setAttribute("FillColor", new Color(0xE8E8E8));
301     poly.setAttribute("FrameColor", Color.BLACK);
302
303     gFigure.add(poly);
304
305     poly = new PolyLineFigure();
306
307     poly.addPoint(x + w + 24, y + 15);
308     poly.addPoint(x + w + 24, y + 13);
309     poly.addPoint(x + w + 39, y + 13);
310     poly.addPoint(x + w + 39, y + 25);
311     poly.addPoint(x + w + 24, y + 25);
312     poly.addPoint(x + w + 24, y + 23);
313
314     poly.setAttribute("FillColor", new Color(0xE8E8E8));
315     poly.setAttribute("FrameColor", Color.BLACK);
```

```

316
317     gFigure.add(poly);
318     add(gFigure);
319
320     update();
321 }
322
323 /*
324  * (non-Javadoc)
325  *
326  * @see ocean.jhotdraw.SpecificationCompositeFigure#handles()
327  */
328 @SuppressWarnings("unchecked")
329 public Vector handles()
330 {
331     Vector handles = new Vector();
332     handles.addElement(new NullHandle(this, RelativeLocator.northWest()));
333     handles.addElement(new NullHandle(this, RelativeLocator.northEast()));
334     handles.addElement(new NullHandle(this, RelativeLocator.southWest()));
335     handles.addElement(new NullHandle(this, RelativeLocator.southEast()));
336     return handles;
337 }
338
339 /**
340  * Creates a popup menu to edit the class or object name.
341  */
342 protected JPopupMenu createPopupMenu()
343 {
344     JPopupMenu popupMenu = new JPopupMenu();
345
346     popupMenu.add(new AbstractAction(I18NProperties.getString("change.name.component"))
347     {
348
349         /**
350          * The serial version UID.
351          */
352         private static final long serialVersionUID = 676546L;
353

```

```

354     /*
355     * (non-Javadoc)
356     *
357     * @see java.awt.event.ActionListener#actionPerformed(java.awt.
      event.ActionEvent)
358     */
359     public void actionPerformed(ActionEvent event)
360     {
361         if (SingleConnector.managerView().openDialogInterface(
      Constantes.NAME_SPEC,
362         ((Component) concept()).getComponentName().trim()))
363         {
364             ((Component) concept()).setComponentName((
      SingleConnector.manager()
365             .getCreationName()));
366             SingleConnector.manager().updateRedrawCode();
367             concept().redraw();
368         }
369     }
370 });
371
372 popupMenu.add(new AbstractAction(I18NProperties.getString("change.
      type.component"))
373 {
374
375     /**
376     * The serial version UID.
377     */
378     private static final long serialVersionUID = 676546L;
379
380     /*
381     * (non-Javadoc)
382     *
383     * @see java.awt.event.ActionListener#actionPerformed(java.awt.
      event.ActionEvent)
384     */
385     public void actionPerformed(ActionEvent event)
386     {
387         String type = ((Component) concept()).getComponentType().
      trim();
388         if (type.startsWith("<<") && type.endsWith(">>"))
389         {
390             type = type.substring(2, type.length() - 2);

```

```

391         }
392
393         if (SingleConnector.managerView().openDialogInterface(
394             Constantes.NAME_SPEC, type))
395         {
396             String newType = (SingleConnector.manager()).
397                 getCreationName();
398             if ((newType == null) || ("").equals(newType) || (" ").
399                 equals(newType))
400             {
401                 newType = new String("componente");
402             }
403             ((Component) concept()).setComponentType("<<" + newType
404                 + ">>");
405             SingleConnector.manager().updateRedrawCode();
406             concept().redraw();
407         }
408     }
409 });
410
411     popupMenu.setLightWeightPopupEnabled(true);
412     return popupMenu;
413 }
414
415 /**
416  * @return the portFigureList
417  */
418 public List<PortFigure> getPortFigureList()
419 {
420     return portFigureList;
421 }
422
423 /**
424  * @param portFigureList the portFigureList to set
425  */
426 public void setPortFigureList(List<PortFigure> portFigureList)
427 {
428     this.portFigureList = portFigureList;
429 }
430
431 /**
432  * @return the drawing
433  */

```

```

430     public DeploymentDiagramDrawing getDrawing ()
431     {
432         return drawing;
433     }
434
435     /**
436      * @param drawing the drawing to set
437      */
438     public void setDrawing(DeploymentDiagramDrawing drawing)
439     {
440         this.drawing = drawing;
441     }
442 }

```

Listagem B.18: Classe **PortFigure.java**

```

1  package documents.graphical.component_diagram;
2
3  import java.awt.Color;
4  import java.awt.Insets;
5  import java.awt.Rectangle;
6
7  import ocean.jhotdraw.SpecificationCompositeFigure;
8  import ocean.jhotdraw.SpecificationDrawing;
9  import ocean.smalltalk.OceanVector;
10 import CH.ifa.draw.contrib.GraphicalCompositeFigure;
11 import CH.ifa.draw.figures.RectangleFigure;
12 import CH.ifa.draw.framework.Figure;
13 import CH.ifa.draw.framework.FigureEnumeration;
14 import documents.concepts.component_diagram.Port;
15 import documents.graphical.composite_structure_diagram.
    CompositeStructureDiagramDrawing;
16 import documents.graphical.composite_structure_diagram.
    StructuredClassifierFigure;
17
18 public class PortFigure extends SpecificationCompositeFigure
19 {
20
21     /**
22      * The serial version UID.
23      */
24     private static final long serialVersionUID = 2731329396930134885L;
25

```



```

26     /**
27      * An instance of classifier figure.
28      */
29     private StructuredClassifierFigure classifier;
30
31     /**
32      * An instance of component figure.
33      */
34     private ComponentFigure component;
35
36     /**
37      * The drawing.
38      */
39     private SpecificationDrawing drawing;
40
41     /**
42      * Default constructor.
43      */
44     public PortFigure ()
45     {
46         super(new RectangleFigure ());
47     }
48
49     /**
50      * Overloaded constructor.
51      *
52      * @param port
53      */
54     public PortFigure(Port port)
55     {
56         super(new RectangleFigure (), port);
57     }
58
59     /**
60      * (non-Javadoc)
61      *
62      * @see ocean.jhotdraw.SpecificationCompositeFigure#relatedConceptClass
63      * ()
64      */
65     @Override
66     public Class<Port> relatedConceptClass ()
67     {
68         return Port.class;
69     }

```

```

68     }
69
70     /**
71      * @return
72      */
73     @SuppressWarnings( { "deprecation", "unchecked" })
74     public static OceanVector memberRelatedConceptClass ()
75     {
76         OceanVector auxList = new OceanVector ();
77         auxList.add(Port.class);
78         return auxList;
79     }
80
81     /*
82      * (non-Javadoc)
83      *
84      * @see ocean.jhotdraw.SpecificationCompositeFigure#initialize ()
85      */
86     public void initialize ()
87     {
88         removeAll ();
89
90         super.initialize ();
91     }
92
93     /*
94      * (non-Javadoc)
95      *
96      * @see ocean.jhotdraw.SpecificationCompositeFigure#redraw ()
97      */
98     public void redraw ()
99     {
100
101         removeAll ();
102
103         GraphicalCompositeFigure gFigure = new GraphicalCompositeFigure ();
104         gFigure.getLayouter().setInsets(new Insets(10, 5, 10, 5));
105         gFigure.setAttribute("FillColor", new Color(0xE8E8E8));
106         gFigure.setAttribute("FrameColor", new Color(0x000000));
107         add(gFigure);
108
109         update ();
110     }

```

```

111
112  /*
113   * (non-Javadoc)
114   *
115   * @see CH.ifa.draw.standard.AbstractFigure#moveBy(int, int)
116   */
117  @Override
118  public void moveBy(int dx, int dy)
119  {
120
121      if (drawing instanceof CompositeStructureDiagramDrawing)
122      {
123
124          // It doesn't belong to a package.
125          if ((classifier == null) && (drawing != null))
126          {
127              int xPort = this.displayBox().x;
128              int yPort = this.displayBox().y;
129              int wPort = this.displayBox().width;
130              int hPort = this.displayBox().height;
131
132              FigureEnumeration enumeration = drawing.figures();
133              while (enumeration.hasMoreElements())
134              {
135                  Figure nextFigure = enumeration.nextFigure();
136                  if (nextFigure instanceof StructuredClassifierFigure)
137                  {
138                      Rectangle displayBox = nextFigure.displayBox();
139                      int xClass = displayBox.x;
140                      int yClass = displayBox.y;
141                      int wClass = displayBox.width;
142                      int hClass = displayBox.height;
143
144                      // Checks the restrictions.
145                      if ((xPort > (xClass - (wPort / 2)))
146                          && (xPort < (xClass + wClass - (wPort / 2))
147                              )
148                          && (yPort > (yClass - (hPort / 2)))
149                          && (yPort < (yClass + hClass - (hPort / 2))
150                              ))
151                      {
152                          classifier = (StructuredClassifierFigure)
153                              nextFigure;

```

```

151         classifier.getPortFigureList().add(this);
152         return;
153     }
154 }
155 }
156 }
157 // It belongs to a package.
158 else if ((classifier != null) && (drawing != null))
159 {
160
161     // CLASSIFIER
162     int classX = classifier.getPresentationFigure().displayBox
163         ().x;
164     int classY = classifier.getPresentationFigure().displayBox
165         ().y;
166     int classW = classifier.getPresentationFigure().displayBox
167         ().width;
168     int classH = classifier.getPresentationFigure().displayBox
169         ().height;
170
171     // PORT
172     int portX = this.displayBox().x;
173     int portY = this.displayBox().y;
174     int portW = this.displayBox().width;
175     int portH = this.displayBox().height;
176
177     int leftBoundary = classX - (portW / 2);
178     int rightBoundary = classX + classW - (portW / 2);
179     int topBoundary = classY - (portH / 2);
180     int bottomBoundary = classY + classH - (portH / 2);
181
182     int newX = dx;
183     int newY = dy;
184
185     // Add restriction inside of the package.
186     if (((portX + dx) < leftBoundary) || ((portX + dx) >
187         rightBoundary)
188         || ((portY + dy) < topBoundary) || ((portY + dy) >
189         bottomBoundary))
190     {
191         return;
192     }
193 }

```

```

188     else if (!( (((portX + dx) > leftBoundary) && ((portX + dx)
        < rightBoundary) && (((portY + dy) == topBoundary) || ((
        portY + dy) == bottomBoundary))) || (((portY + dy) >
        topBoundary)
189         && ((portY + dy) < bottomBoundary) && (((portX + dx
            ) == leftBoundary) || ((portX + dx) ==
            rightBoundary))))))
190     {
191         return ;
192     }
193
194     super .moveBy(newX, newY);
195
196 }
197
198 }
199 else if (drawing instanceof ComponentDiagramDrawing)
200 {
201     // It doesn't belong to a package.
202     if ((component == null) && (drawing != null))
203     {
204         int xPort = this .displayBox().x;
205         int yPort = this .displayBox().y;
206         int wPort = this .displayBox().width;
207         int hPort = this .displayBox().height;
208
209         FigureEnumeration enumeration = drawing.figures();
210         while (enumeration.hasMoreElements())
211         {
212             Figure nextFigure = enumeration.nextFigure();
213             if (nextFigure instanceof ComponentFigure)
214             {
215                 Rectangle displayBox = nextFigure.displayBox();
216                 int xClass = displayBox.x;
217                 int yClass = displayBox.y;
218                 int wClass = displayBox.width;
219                 int hClass = displayBox.height;
220
221                 // Checks the restrictions.
222                 if ((xPort > (xClass - (wPort / 2)))
223                     && (xPort < (xClass + wClass - (wPort / 2))
224                         )
                )
                && (yPort > (yClass - (hPort / 2)))

```

```

225         && (yPort < (yClass + hClass - (hPort / 2))
226             ))
227         {
228             component = (ComponentFigure) nextFigure;
229             component.getPortFigureList().add(this);
230             return;
231         }
232     }
233 }
234 // It belongs to a package.
235 else if ((component != null) && (drawing != null))
236 {
237
238     // CLASSIFIER
239     int classX = component.getPresentationFigure().displayBox()
240         .x;
241     int classY = component.getPresentationFigure().displayBox()
242         .y;
243     int classW = component.getPresentationFigure().displayBox()
244         .width;
245     int classH = component.getPresentationFigure().displayBox()
246         .height;
247
248     // PORT
249     int portX = this.displayBox().x;
250     int portY = this.displayBox().y;
251     int portW = this.displayBox().width;
252     int portH = this.displayBox().height;
253
254     int leftBoundary = classX - (portW / 2);
255     int rightBoundary = classX + classW - (portW / 2);
256     int topBoundary = classY - (portH / 2);
257     int bottomBoundary = classY + classH - (portH / 2);
258
259     int newX = dx;
260     int newY = dy;
261
262     // Add restriction inside of the package.
263     if (((portX + dx) < leftBoundary) || ((portX + dx) >
264         rightBoundary)
265         || ((portY + dy) < topBoundary) || ((portY + dy) >
266         bottomBoundary))

```

```

261         {
262             return ;
263         }
264
265         else if (!( (((portX + dx) > leftBoundary) && ((portX + dx)
                < rightBoundary) && (((portY + dy) == topBoundary) || ((
                portY + dy) == bottomBoundary))) || (((portY + dy) >
                topBoundary)
266                 && ((portY + dy) < bottomBoundary) && (((portX + dx
                ) == leftBoundary) || ((portX + dx) ==
                rightBoundary))))))
267         {
268             return ;
269         }
270
271         super.moveBy(newX, newY);
272
273     }
274 }
275
276 if ((classifier == null) && (component == null))
277 {
278     super.moveBy(dx, dy);
279 }
280
281 }
282
283 /**
284  * @return the drawing
285  */
286 public SpecificationDrawing getDrawing()
287 {
288     return drawing;
289 }
290
291 /**
292  * @param drawing the drawing to set
293  */
294 public void setDrawing(SpecificationDrawing drawing)
295 {
296     this.drawing = drawing;
297 }
298

```

299 }

Listagem B.19: Classe **InterfaceFigure.java**

```

1  package documents . graphical . component_diagram ;
2
3  import java . awt . * ;
4  import java . awt . event . ActionEvent ;
5
6  import javax . swing . AbstractAction ;
7  import javax . swing . JPopupMenu ;
8
9  import ocean . accessories . SingleConnector ;
10 import ocean . jhotdraw . SpecificationCompositeFigure ;
11 import ocean . smalltalk . Constantes ;
12 import ocean . smalltalk . OceanVector ;
13 import CH . ifa . draw . contrib . GraphicalCompositeFigure ;
14 import CH . ifa . draw . figures . EllipseFigure ;
15 import CH . ifa . draw . figures . RectangleFigure ;
16 import CH . ifa . draw . figures . TextFigure ;
17 import CH . ifa . draw . framework . Figure ;
18 import documents . concepts . component_diagram . Interface ;
19 import documents . util . I18NProperties ;
20
21 public class InterfaceFigure extends SpecificationCompositeFigure
22 {
23
24     /**
25      * The serial version UID.
26      */
27     private static final long serialVersionUID = -8939607608294996977L;
28
29     /**
30      * A empty fill rectangle figure.
31      *
32      * @author thania
33      * @since Apr 7, 2008
34      */
35     class EmptyFillRectangle extends RectangleFigure
36     {
37
38         /**
39          * The generated serial version UID.

```



```

40     */
41     private static final long serialVersionUID = 7460260942123340735L;
42
43     /*
44     * (non-Javadoc)
45     *
46     * @see CH.ifa.draw.figures.RectangleFigure#drawFrame(java.awt.
47     *     Graphics)
48     */
49     public void drawFrame(Graphics g)
50     {
51         // EMPTY.
52     }
53
54     /*
55     * (non-Javadoc)
56     *
57     * @see CH.ifa.draw.figures.RectangleFigure#drawBackground(java.awt
58     *     .Graphics)
59     */
60     public void drawBackground(Graphics g)
61     {
62         // EMPTY.
63     }
64
65     /**
66     * Default constructor.
67     */
68     public InterfaceFigure()
69     {
70         super();
71         this.setPresentationFigure(new EmptyFillRectangle());
72     }
73
74     /**
75     * Overloaded constructor.
76     *
77     * @param object
78     */
79     public InterfaceFigure(Interface interfaceConcept)
80     {
81         super(interfaceConcept);

```

```

81     }
82
83     /*
84      * (non-Javadoc)
85      *
86      * @see ocean.jhotdraw.SpecificationCompositeFigure#relatedConceptClass
87      *      ()
88      */
89     @Override
90     public Class<Interface> relatedConceptClass ()
91     {
92         return Interface.class;
93     }
94
95     /**
96      * @return
97      */
98     @SuppressWarnings( { "deprecation", "unchecked" })
99     public static OceanVector memberRelatedConceptClass ()
100    {
101        OceanVector auxList = new OceanVector();
102        auxList.add(Interface.class);
103        return auxList;
104    }
105
106    /*
107     * (non-Javadoc)
108     *
109     * @see ocean.jhotdraw.SpecificationCompositeFigure#initialize ()
110     */
111    public void initialize ()
112    {
113        removeAll();
114
115        setAttribute (Figure.POPUP_MENU, createPopupMenu());
116
117        super.initialize();
118    }
119
120    /*
121     * (non-Javadoc)
122     *
123     * @see ocean.jhotdraw.SpecificationCompositeFigure#redraw ()

```

```

123     */
124     public void redraw ()
125     {
126
127         removeAll ();
128
129         Interface inter = (Interface) this.concept ();
130         if (inter.getInterfaceName () == null || ("").equals (inter .
131             getInterfaceName ()))
132         {
133             inter.setInterfaceName ("interface");
134         }
135         TextFigure figure = new TextFigure ();
136         figure.setFont (new Font ("Helvetica", Font.BOLD, 14));
137         figure.setText (inter.getInterfaceName ());
138
139         final int w = figure.textDisplayBox ().width;
140
141         EllipseFigure ef = new EllipseFigure ()
142         {
143             /**
144              * The serial version UID.
145              */
146             private static final long serialVersionUID =
147                 -5990021166034781909L;
148
149             /*
150              * (non-Javadoc)
151              * @see CH.ifa.draw.figures.EllipseFigure#drawBackground (java .
152                  awt.Graphics)
153              */
154             public void drawBackground (Graphics g)
155             {
156                 Rectangle r = displayBox ();
157                 g.fillOval (r.x + w / 2 - 14, r.y, 28, 28);
158             }
159
160             /*
161              * (non-Javadoc)
162              * @see CH.ifa.draw.figures.EllipseFigure#drawFrame (java.awt.

```

```

        Graphics)
163     */
164     public void drawFrame(Graphics g)
165     {
166         Rectangle r = displayBox();
167         g.drawOval(r.x + w / 2 - 14, r.y, 28, 28);
168     }
169 };
170 ef.setAttribute("FillColor", new Color(0xE8E8E8));
171 ef.setAttribute("FrameColor", new Color(0x000000));
172
173 GraphicalCompositeFigure gf = new GraphicalCompositeFigure(new
        EmptyFillRectangle());
174 gf.getLayouter().setInsets(new Insets(10, 0, 10, 10));
175 gf.setAttribute("FillColor", Color.WHITE);
176 gf.setAttribute("FrameColor", new Color(0xFFFFFFFF));
177 gf.add(ef);
178 add(gf);
179
180 add(figure);
181
182 update();
183 }
184
185 /**
186  * Creates a popup menu to edit the class or object name.
187  */
188 protected JPopupMenu createPopupMenu()
189 {
190     JPopupMenu popupMenu = new JPopupMenu();
191
192     popupMenu.add(new AbstractAction(I18NProperties.getString("change.
        name.interface"))
193     {
194
195         private static final long serialVersionUID = 676546L;
196
197         /*
198          * (non-Javadoc)
199          *
200          * @see java.awt.event.ActionListener#actionPerformed(java.awt.
        event.ActionEvent)
201          */

```

```
202     public void actionPerformed(ActionEvent event)
203     {
204         if (SingleConnector.managerView().openDialogInterface(
205             Constantes.NAME_SPEC,
206             ((Interface) concept()).getInterfaceName()))
207         {
208             ((Interface) concept()).setInterfaceName((
209                 SingleConnector.manager())
210                 .getCreationName());
211             SingleConnector.manager().updateRedrawCode();
212             concept().redraw();
213         }
214     }
215     });
216     popupMenu.setLightWeightPopupEnabled(true);
217     return popupMenu;
218 }
219 }
```

Referências Bibliográficas

- AMORIM, J. de. *Integração dos frameworks JHotDraw e OCEAN para a produção de objetos visuais a partir do framework OCEAN*. 2006.
- AYOAMA, M. *New age of software development: How component-based engineering changes the way of software development?* <http://www.sei.cmu.edu/pacc/icse98/papers/p14.pdf>: [s.n.], 2002.
- BONFIM, V. D. *Tratamento de Documentos Textuais Estruturados no Ambiente SEA*. Dissertação de Mestrado, Universidade Federal de Santa Catarina: [s.n.], 2004.
- BOSCH, J. et al. *Building Application Framework : Object Oriented Foundations of Framework Design*. [S.l.: s.n.], 1999.
- BRANT, J. *HotDraw Home Page*. <http://st-www.cs.uiuc.edu/users/brant/HotDraw/>: [s.n.], 1999.
- COELHO, A. *Reengenharia do Framework OCEAN*. 2007.
- CROSS, E. J. C. J. H. *Reverse engineering and design recovery: A taxonomy*. Los Alamitos, CA, USA: [s.n.], 1990. 13-17 p.
- CUNHA, R. S. da. *Suporte à Análise de Compatibilidade Comportamental e Estrutural entre Componentes no Ambiente SEA*. Dissertação de Mestrado, Universidade Federal de Santa Catarina: [s.n.], 2005.
- FILHO, A. M. da S. Sobre a importância do reuso. *Revista Espaço Acadêmico*, v. 1, n. 92, 2007.
- FOWLER, M. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. [S.l.: s.n.], 2003.
- JOHNSON, R. E. Components, frameworks e patterns. *Proceedings of the 1997 symposium on Software reusability*, ACM Press, p. 10–17, 1997.
- JOHNSON, R. E.; FOOTE, B. Designing reusable classes. *Journal of Object-Oriented Programming*, v. 1, n. 2, p. 22–35, 1988.
- KAISER, W. *JHotDraw as Open-Source Project*. <http://www.jhotdraw.org/>: [s.n.], 2007.
- MACHADO, T. A. A. S. da R. *Reengenharia da Interface do Ambiente SEA*. 2007.
- MEYER, B. *Object-oriented software construction*. [S.l.]: Prentice Hall, 1988.
- OMG. *OCL 2.0 Specification*. 2006.

OMG. *Unified Modeling Language: Diagram Interchange*. 2006.

OMG. *Unified Modeling Language: Infrastructure*. 2007.

OMG. *Unified Modeling Language: Superstructure*. 2007.

PEREIRA, O. A. F. *Suporte à Engenharia Reversa para o Ambiente SEA*. 2004.

ROCHA, A. R. C.; MALDONADO, J. C.; WEBER, K. C. *Qualidade de Software*. São Paulo: Pretince Hall, 2001.

SAMPAIO, M. C. *História de UML*.

http://www.dsc.ufcg.edu.br/~sampaio/cursos/2007.1/Graduacao/SI-II/Uml/historia_uml/historia_uml.htm: [s.n.], 2007.

SARTORI, G. M. S. *Suporte à Geração Semi-Automatizada de Adaptação para Componentes no Ambiente SEA*. Dissertação de Mestrado, Universidade Federal de Santa Catarina: [s.n.], 2005.

SCHEIDT, N. *Introdução de Suporte Gerencial Baseado em Workflow e CMM a um ambiente de desenvolvimento de software*. Dissertação de Mestrado, Universidade Federal de Santa Catarina: [s.n.], 2003.

SILVA, R. P. e. *Suporte ao desenvolvimento e uso de frameworks e componentes*. Tese de Doutorado, Universidade Federal do Rio Grande do Sul: [s.n.], 2000.

SILVA, R. P. e. *UML 2 em Modelagem Orientada a Objetos*. Florianópolis: Visual Books, 2007.

SZYPERSKI, C. Summary of the second internacional workshop on component-oriented programming. 1997.

THEPPITAK, S. *Free e Open Source Software: Localization*. [S.l.: s.n.], 2005.

WEBER, K. C.; ROCHA, A. R. C. *Qualidade e Produtividade em Software*. São Paulo: Makron Book, 1999.

WIRFS-BROCK R.; JOHNSON, R. E. *Surveying current research in object-oriented design*. New York: [s.n.], 1990.