

email:  
nFor0 : Str  
a.  
0..  
Revisada e atualizada



# UMA ABORDAGEM PRÁTICA

novatec

# Gilleanes T. A. Guedes

# **UML2**

## **UMA ABORDAGEM PRÁTICA**

### **3<sup>a</sup> EDIÇÃO**

**Gilleane T. A. Guedes**

**Novatec**

Copyright © 2009, 2011, 2018 da Novatec Editora Ltda.

Todos os direitos reservados e protegidos pela Lei 9.610 de 19/02/1998. É proibida a reprodução desta obra, mesmo parcial, por qualquer processo, sem prévia autorização, por escrito, do autor e da Editora.

Editor: Rubens Prates

Revisão de texto: Patrizia Zagni

Editoração eletrônica: Carolina Kuwabata

Capa: Victor Bittow

ISBN: 978-85-7522-644-5

Histórico de edições impressas:

Fevereiro/2018 Terceira edição

Fevereiro/2014 Quarta reimpressão

Julho/2013 Terceira reimpressão

Janeiro/2013 Segunda reimpressão

Janeiro/2012 Primeira reimpressão

Junho/2011 Segunda edição (ISBN: 978-85-7522-281-2)

Maio/2009 Primeira edição (ISBN: 978-85-7522-193-8)

Novatec Editora Ltda.

Rua Luís Antônio dos Santos 110

02460-000 – São Paulo, SP – Brasil

Tel.: +55 11 2959-6529

E-mail: [novatec@novatec.com.br](mailto:novatec@novatec.com.br)

Site: [www.novatec.com.br](http://www.novatec.com.br)

Twitter: [twitter.com/novateceditora](https://twitter.com/novateceditora)

Facebook: [facebook.com/novatec](https://facebook.com/novatec)

LinkedIn: [linkedin.com/in/novatec](https://linkedin.com/in/novatec)

*Dedico este livro aos meus filhos,  
Ulisses e Sophia.*

# **Sumário**

[Sobre o autor](#)

[Prefácio](#)

## [Capítulo 1 ■ Introdução à UML](#)

[1.1 Breve Histórico da UML](#)

[1.2 Por que Modelar Software?](#)

[1.2.1 Modelo de Software – Uma Definição](#)

[1.2.2 Elicitação e Análise de Requisitos](#)

[1.2.3 Prototipação](#)

[1.2.4 Prazos e Custos](#)

[1.2.5 Projeto](#)

[1.2.6 Manutenção](#)

[1.2.7 Documentação Histórica](#)

[1.3 Por que Tantos Diagramas?](#)

[1.4 Rápido Resumo dos Diagramas da UML](#)

[1.4.1 Diagrama de Casos de Uso](#)

[1.4.2 Diagrama de Classes](#)

[1.4.3 Diagrama de Objetos](#)

[1.4.4 Diagrama de Pacotes](#)

[1.4.5 Diagrama de Sequência](#)

[1.4.6 Diagrama de Comunicação](#)

[1.4.7 Diagrama de Máquina de Estados](#)

[1.4.8 Diagrama de Atividade](#)

[1.4.9 Diagrama de Visão Geral de Interação](#)

[1.4.10 Diagrama de Componentes](#)

[1.4.11 Diagrama de Implantação](#)

[1.4.12 Diagrama de Estrutura Composta](#)

[1.4.13 Diagrama de Tempo ou de Temporização](#)

[1.4.14 Diagrama de Perfil](#)

[1.4.15 Síntese Geral dos Diagramas](#)

[1.5 Ferramentas CASE Baseadas na Linguagem UML](#)

## [Capítulo 2 ■ Orientação a Objetos](#)

[2.1 Classificação, Abstração e Instanciação](#)

[2.2 Classes de Objetos](#)  
[2.3 Atributos ou Propriedades](#)  
[2.4 Operações, Métodos ou Comportamentos](#)  
[2.5 Visibilidade](#)  
[2.6 Herança](#)  
    [2.6.1 Herança Múltipla](#)  
[2.7 Polimorfismo](#)

## **Capítulo 3 ■ Diagrama de Casos de Uso**

[3.1 Atores](#)  
[3.2 Como Identificar os Atores?](#)  
[3.3 Casos de Uso](#)  
[3.4 Documentação de Casos de Uso](#)  
[3.5 Como Identificar os Casos de Uso?](#)  
[3.6 Associações](#)  
[3.7 Generalização/Especialização](#)  
[3.8 Inclusão](#)  
[3.9 Extensão](#)  
[3.10 Restrições em Associações de Extensão](#)  
[3.11 Pontos de Extensão](#)  
[3.12 Multiplicidade no Diagrama de Casos de Uso](#)  
[3.13 Estereótipos](#)  
[3.14 Fronteira de Sistema](#)  
[3.15 Exemplo de Diagrama de Casos de Uso – Sistema de Controle Bancário](#)  
[3.16 Documentação do Diagrama de Casos de Uso do Sistema de Controle Bancário](#)  
    [3.16.1 Atores que Interagem com o Sistema](#)  
    [3.16.2 Documentação do Caso de Uso Abrir Conta Especial](#)  
    [3.16.3 Documentação do Caso de Uso Abrir Conta Poupança](#)  
    [3.16.4 Documentação do Caso de Uso Gerenciar Clientes](#)  
    [3.16.5 Documentação do Caso de Uso Realizar Depósito](#)  
    [3.16.6 Documentação do Caso de Uso Emitir Saldo](#)  
    [3.16.7 Documentação do Caso de Uso Emitir Extrato](#)  
    [3.16.8 Documentação do Caso de Uso Realizar Saque](#)  
    [3.16.9 Documentação do Caso de Uso Registrar Movimento](#)  
[3.17 Exemplo de Diagrama de Casos de Uso – Sistema de Telefone Celular](#)  
    [3.17.1 Documentação do Caso de Uso Realizar Ligação](#)  
[3.18 Exemplo de Diagrama de Casos de Uso – Sistema de Biblioteca](#)  
    [3.18.1 Documentação do Caso de Uso Locar Exemplares](#)  
[3.19 Exemplo de Diagrama de Casos de Uso – Sistema de Clínica Veterinária](#)  
    [3.19.1 Documentação do Caso de Uso Atender à Consulta](#)  
[3.20 Exemplo de Diagrama de Casos de Uso – Sistema de Controle de Advocacia](#)  
    [3.20.1 Documentação do Caso de Uso Gerenciar Processos](#)

### 3.21 Exercícios Propostos

- [3.21.1 Sistema de Controle de Cinema](#)
- [3.21.2 Sistema de Controle de Clube Social](#)
- [3.21.3 Sistema de Locação de Veículos](#)
- [3.21.4 Sistema para Controle de Leilão Via Internet](#)
- [3.21.5 Sistema de Controle de Hotelaria](#)
- [3.21.6 Sistema de Controle de Imobiliária](#)

### 3.22 Resolução dos Exercícios

- [3.22.1 Resolução do Exercício Sistema de Controle de Cinema](#)
- [3.22.2 Resolução do Exercício Sistema de Controle de Clube Social](#)
- [3.22.3 Resolução do Exercício Sistema de Locação de Veículos](#)
- [3.22.4 Resolução do Exercício Sistema para Controle de Leilão Via Internet](#)
- [3.22.5 Resolução do Exercício Sistema de Controle de Hotelaria](#)
- [3.22.6 Resolução do Exercício Sistema de Controle de Imobiliária](#)

## **Capítulo 4 ■ Diagrama de Classes**

### 4.1 Atributos e Métodos

- [4.2 Relacionamentos ou Associações](#)
- [4.2.1 Associação Unária ou Reflexiva](#)
- [4.2.2 Associação Binária](#)
- [4.2.3 Associação Ternária ou N-ária](#)
- [4.2.4 Agregação](#)
- [4.2.5 Composição](#)
- [4.2.6 Generalização/Especialização](#)
- [4.2.7 Classe Associativa](#)
- [4.2.8 Associação Qualificada](#)
- [4.2.9 Dependência](#)
- [4.2.10 Realização](#)

### 4.3 Portas

### 4.4 Interfaces

- [4.4.1 Interfaces Fornecidas](#)
- [4.4.2 Interfaces Requeridas](#)

### 4.5 Restrições

- [4.5.1 Restrições em OCL \(Object Constraint Language\)](#)

### 4.6 Estereótipos do Diagrama de Classes

- [4.6.1 Estereótipo <>enumeration>>](#)
- [4.6.2 Estereótipos para Projeto Navegacional](#)
- [4.6.3 Estereótipo <>boundary>>](#)
- [4.6.4 Estereótipo <>control>>](#)
- [4.6.5 Estereótipo <>entity>>](#)

### 4.7 Exemplo de Diagrama de Classes (Modelo Conceitual) – Sistema de Controle Bancário

- [4.8 Como Identificar Classes](#)
- [4.9 Exemplo de Modelo de Domínio](#)
- [4.10 Exemplo de Diagrama de Classes – Sistema de Telefone Celular](#)
- [4.11 Exemplo de Diagrama de Classes – Sistema de Biblioteca](#)
- [4.12 Exemplo de Diagrama de Classes – Sistema de Clínica Veterinária](#)
- [4.13 Exemplo de Diagrama de Classes – Sistema de Controle de Advocacia](#)
- [4.14 Persistência](#)
- [4.15 Mapeamento de Classes em Tabelas](#)
  - [4.15.1 Estereótipo Table](#)
  - [4.15.2 Associações e Chaves Estrangeiras](#)
- [4.16 Padrão Repository](#)
- [4.17 Padrão DAO \(Data Access Object\)](#)
- [4.18 Exercícios Propostos](#)
  - [4.18.1 Sistema de Controle de Cinema](#)
  - [4.18.2 Sistema de Controle de Clube Social](#)
  - [4.18.3 Sistema de Locação de Veículos](#)
  - [4.18.4 Sistema para Controle de Leilão Via Internet](#)
  - [4.18.5 Sistema de Controle de Hotelaria](#)
  - [4.18.6 Sistema de Controle de Imobiliária](#)
- [4.19 Solução dos Exercícios](#)
  - [4.19.1 Sistema de Controle de Cinema](#)
  - [4.19.2 Sistema de Controle de Clube Social](#)
  - [4.19.3 Sistema de Locação de Veículos](#)
  - [4.19.4 Sistema para Controle de Leilão Via Internet](#)
  - [4.19.5 Sistema de Controle de Hotelaria](#)
  - [4.19.6 Sistema de Controle de Imobiliária](#)

## **Capítulo 5 ■ Diagrama de Objetos**

- [5.1 Objeto](#)
- [5.2 Vínculos](#)
- [5.3 Dependência com Estereótipo <<instanceOf>>](#)
- [5.4 Exemplo de Diagrama de Objetos](#)

## **Capítulo 6 ■ Diagrama de Pacotes**

- [6.1 Pacotes](#)
- [6.2 Dependência](#)
- [6.3 Pacotes Contendo Pacotes](#)
- [6.4 Estereótipos Aplicados a Pacotes](#)
- [6.5 Representação de Camadas do Modelo por Meio de Pacotes](#)

## **Capítulo 7 ■ Diagrama de Sequência**

[7.1 Atores](#)

[7.2 Lifelines](#)

[7.3 Mensagens ou Estímulos](#)

[7.3.1 Mensagens entre Atores](#)

[7.3.2 Mensagens entre Lifelines](#)

[7.3.3 Mensagens de Retorno](#)

[7.3.4 Mensagens Construtoras](#)

[7.3.5 Mensagens Destrutoras](#)

[7.3.6 Autochamadas ou Autodelegações](#)

[7.3.7 Mensagens Assíncronas](#)

[7.3.8 Restrição de Duração](#)

[7.3.9 Mensagens Perdidas e Mensagens Encontradas](#)

[7.4 Portas](#)

[7.5 Fragmentos de Interação](#)

[7.6 Usos de Interação \(Ocorrências de Interação antes da UML 2.1.1\)](#)

[7.7 Portões \(Gates\)](#)

[7.8 Fragmentos Combinados e Operadores de Interação](#)

[7.9 Invariante de Estado \(StateInvariant\)](#)

[7.10 Exemplos de Diagramas de Sequência para o Sistema de Controle Bancário](#)

[7.10.1 Processo de Abertura de Conta Comum – Modelo Preliminar](#)

[7.10.2 Processo de Abertura de Conta Comum – Modelo Detalhado](#)

[7.10.3 Processo de Realizar Depósito](#)

[7.10.4 Processo de Emissão de Extrato](#)

[7.11 Padrões Repository e DAO](#)

[7.12 Exemplo de Diagrama de Sequência – Processo de Realizar Ligação para o Sistema de Telefone Celular](#)

[7.13 Exemplo de Diagrama de Sequência – Processo de Locação de Exemplares para o Sistema de Biblioteca](#)

[7.14 Exemplo de Diagrama de Sequência – Processo de Atendimento de Consulta para o Sistema de Clínica Veterinária](#)

[7.15 Exemplo de Diagrama de Sequência – Funcionalidade para Gerenciamento de Processos do Sistema de Controle de Advocacia](#)

[7.16 Exercícios Propostos](#)

[7.16.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos](#)

[7.16.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade](#)

[7.16.3 Sistema de Locação de Veículos – Processo de Locação de Veículo](#)

[7.16.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão](#)

[7.16.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias](#)

[7.16.6 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel](#)

[7.17 Solução dos Exercícios](#)

[7.17.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos](#)

[7.17.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade](#)

[7.17.3 Sistema de Locação de Veículos – Processo de Locação de Veículo](#)

- [7.17.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão](#)
- [7.17.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias](#)
- [7.17.6 Sistema de Controle de Imobiliária](#)

## **Capítulo 8 ■ Diagrama de Comunicação**

- [8.1 Lifelines](#)
- [8.2 Vínculos](#)
- [8.3 Mensagens](#)
- [8.4 Atores](#)
- [8.5 Autochamada](#)
- [8.6 Exemplo de diagrama de comunicação – Processo de Emissão de Saldo](#)
- [8.7 Condições de Guarda e Iterações](#)
- [8.8 Exercícios Propostos](#)
- [8.9 Solução dos Exercícios](#)
  - [8.9.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos](#)
  - [8.9.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade](#)
  - [8.9.3 Sistema de Locação de Veículos – Processo de Locação de Veículo](#)
  - [8.9.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão](#)
  - [8.9.5 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel](#)

## **Capítulo 9 ■ Diagrama de Máquina de Estados**

- [9.1 Estado](#)
  - [9.1.1 Estado Simples](#)
- [9.2 Transições](#)
- [9.3 Estado Inicial](#)
- [9.4 Estado Final](#)
- [9.5 Exemplo de Diagrama de Máquina de Estados – Processo de Emissão de Saldo](#)
- [9.6 Atividades internas](#)
- [9.7 Transições Internas](#)
- [9.8 Autotransições](#)
- [9.9 Pseudoestado de Escolha](#)
- [9.10 Barra de Bifurcação/União](#)
- [9.11 Estados Compostos](#)
- [9.12 Pseudoestado de História](#)
- [9.13 Estados Compostos Ortogonais](#)
- [9.14 Estado de Sincronismo](#)
- [9.15 Estado de Submáquina](#)
- [9.16 Pseudoestado de Junção](#)
- [9.17 Pseudoestado de Ponto de Entrada e Pseudoestado de Ponto de Saída](#)
- [9.18 Pseudoestado de Término](#)
- [9.19 Exemplo de Diagrama de Máquina de Estados – Emitir Extrato](#)
- [9.20 Exemplo de Diagrama de Máquina de Estados – Realizar Depósito](#)

[9.21 Exemplo de Diagrama de Máquina de Estados – Realizar Saque](#)

[9.22 Exemplo de Diagrama de Máquina de Estados – Encerrar Conta](#)

[9.23 Exercícios Propostos](#)

[9.23.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos](#)

[9.23.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade](#)

[9.23.3 Sistema de Locação de Veículos – Processo de Locação de Veículo](#)

[9.23.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão](#)

[9.23.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias](#)

[9.23.6 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel](#)

[9.24 Solução dos Exercícios](#)

[9.24.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos](#)

[9.24.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade](#)

[9.24.3 Sistema de Locação de Veículos – Processo de Locação de Veículo](#)

[9.24.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão](#)

[9.24.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias](#)

[9.24.6 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel](#)

## **Capítulo 10 ■ Diagrama de Atividade**

[10.1 Atividade](#)

[10.2 Nó de Ação](#)

[10.3 Fluxo de Controle](#)

[10.4 Nó Inicial](#)

[10.5 Nó de Final de Atividade](#)

[10.6 Nó de Decisão](#)

[10.7 Exemplo Simples de Diagrama de Atividade](#)

[10.8 Nó de Bifurcação/União](#)

[10.9 Final de Fluxo](#)

[10.10 Fluxo de Objetos](#)

[10.11 Nó de Objeto](#)

[10.12 Alfinetes \(Pins\)](#)

[10.13 Nó de Parâmetro de Atividade](#)

[10.14 Nó de Buffer Central](#)

[10.15 Nó de Repositório de Dados \(Data Store Node\)](#)

[10.16 Exceções](#)

[10.17 Ação de Envio de Sinal \(Ação de Objeto de Envio na versão 2.0\)](#)

[10.18 Ação de Evento de Aceitação](#)

[10.19 Ação de Evento de Tempo de Aceitação](#)

[10.20 Ação de Chamada de Comportamento](#)

[10.21 Ação de Chamada de Operação](#)

[10.22 Partição de Atividade](#)

[10.23 Região de Atividade Interrompível](#)

[10.24 Nó de Atividade Estruturada](#)

- [10.24.1 Nós Condicionais](#)
- [10.24.2 Nós de Laço](#)
- [10.25 Região de Expansão](#)
- [10.26 Conectores](#)
- [10.27 Exemplo de Diagrama de Atividade – Emitir Extrato](#)
- [10.28 Exemplo de Diagrama de Atividade – Realizar Depósito](#)
- [10.29 Exemplo de Diagrama de Atividade – Realizar Saque](#)
- [10.30 Exercícios Propostos](#)
  - [10.30.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos](#)
  - [10.30.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade](#)
  - [10.30.3 Sistema de Locação de Veículos – Processo de Locação de Veículo](#)
  - [10.30.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão](#)
  - [10.30.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias](#)
  - [10.30.6 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel](#)
- [10.31 Solução dos Exercícios](#)
  - [10.31.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos](#)
  - [10.31.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade](#)
  - [10.31.3 Sistema de Locação de Veículos – Processo de Locação de Veículo](#)
  - [10.31.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão](#)
  - [10.31.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias](#)
  - [10.31.6 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel](#)

## **Capítulo 11 ■ Diagrama de Visão Geral de Interação**

- [11.1 Exemplo de Diagrama de Visão Geral de Interação – Processo Geral de Conclusão de Pedido – Sistema de Livraria Digital](#)
- [11.2 Exercícios Propostos](#)
  - [11.2.1 Sistema de Controle de Clube Social – Processo Geral de Associação](#)
  - [11.2.2 Sistema de Controle de Hotel – Processo Geral de Encerramento de Estada](#)
- [11.3 Solução dos Exercícios](#)
  - [11.3.1 Sistema de Controle de Clube Social – Processo Geral de Associação](#)
  - [11.3.2 Sistema de Controle de Hotel – Processo Geral de Encerramento de Estada](#)

## **Capítulo 12 ■ Diagrama de Componentes**

- [12.1 Componente](#)
- [12.2 Interfaces Fornecidas e Requeridas](#)
- [12.3 Classes e Componentes Internos](#)
  - [12.3.1 Portas](#)
- [12.4 Exemplo de Diagrama de Componentes – Sistema de Controle Bancário](#)
- [12.5 Exercícios Propostos](#)
  - [12.5.1 Sistema de Controle de Cinema](#)
  - [12.5.2 Sistema de Controle de Clube Social](#)
  - [12.5.3 Sistema de Locação de Veículos](#)

[12.5.4 Sistema para Controle de Leilão Via Internet](#)

[12.5.5 Sistema de Controle de Hotelaria](#)

[12.5.6 Sistema de Controle de Imobiliária](#)

[12.6 Solução dos Exercícios](#)

[12.6.1 Sistema de Controle de Cinema](#)

[12.6.2 Sistema de Controle de Clube Social](#)

[12.6.3 Sistema de Locação de Veículos](#)

[12.6.4 Sistema para Controle de Leilão Via Internet](#)

[12.6.5 Sistema de Controle de Hotelaria](#)

[12.6.6 Sistema de Controle de Imobiliária](#)

## **Capítulo 13 ■ Diagrama de Implantação**

[13.1 Nós](#)

[13.2 Estereótipos](#)

[13.3 Associações](#)

[13.4 Exemplo de Diagrama de Implantação](#)

[13.5 Artefatos](#)

[13.6 Especificação de Implantação](#)

[13.7 Exemplo de Diagrama de Implantação contendo Artefatos](#)

[13.8 Nós Contendo Pacotes](#)

[13.9 Exercícios Propostos](#)

[13.9.1 Sistema para Controle de Leilão Via Internet](#)

[13.10 Solução dos Exercícios](#)

[13.10.1 Sistema para Controle de Leilão Via Internet](#)

## **Capítulo 14 ■ Diagrama de Estrutura Composta**

[14.1 Colaborações](#)

[14.2 Papéis](#)

[14.3 Ocorrência de Colaboração](#)

[14.4 Portas](#)

[14.5 Propriedades e Partes](#)

## **Capítulo 15 ■ Diagrama de Tempo ou de Temporização**

## **Capítulo 16 ■ Diagrama de Perfil**

[16.1 Conceitos Básicos: Modelos, Metamodelos e Metaclasses](#)

[16.1.1 Metaclasse Classifier](#)

[16.1.2 Metaclasse BehavioredClassifier](#)

[16.1.3 Metaclasse NameSpace](#)

[16.1.4 Metaclasse NamedElement](#)

[16.1.5 Metaclasse DirectedRelationship](#)

[16.1.6 Metaclasse Constraint](#)  
[16.1.7 Metaclasse RedefinableElement](#)  
[16.2 Criação de Perfis](#)  
[16.3 Estereótipos](#)  
[16.4 Extensão](#)

## **Capítulo 17 ■ Estudo de Caso – Sistema de Pizzaria Online – PizzaNet**

[17.1 Descrição do Problema](#)  
[17.2 Solução do Problema](#)  
[17.2.1 Diagramas de Casos de Uso](#)  
[17.2.2 Documentação dos Diagramas de Casos de Uso da PizzaNet](#)  
[17.2.3 Diagrama de Classes – Modelo de Domínio](#)  
[17.2.4 Diagrama de Objetos](#)  
[17.2.5 Diagrama de Pacotes da PizzaNet](#)  
[17.2.6 Diagramas de Sequência da PizzaNet](#)  
[17.2.7 Diagrama de Comunicação Escolher Pizza](#)  
[17.2.8 Diagramas de Máquinas de Estados da PizzaNet](#)  
[17.2.9 Diagramas de Atividade da PizzaNet](#)  
[17.2.10 Diagrama de Visão Geral de Interação – Realizar Pedido](#)  
[17.2.11 Diagrama de Componentes da PizzaNet](#)  
[17.2.12 Diagrama de Implantação da PizzaNet](#)

## **Capítulo 18 ■ A UML 2.5**

[18.1 Áreas Semânticas](#)  
[18.2 Conceitos Básicos: Modelos, Metamodelos e Metaclasses](#)  
[18.3 Estrutura Comum](#)  
[18.3.1 Raiz \(Root\)](#)  
[18.3.1.1 Metaclasse Element](#)  
[18.3.1.2 Metaclasse Comment](#)  
[18.3.1.3 Metaclasse Relationship](#)  
[18.4 Metaclasses Utilizadas para a Modelagem de Classes](#)  
[18.4.1 Metaclasse Classifier](#)  
[18.4.2 Metaclasse StructuredClassifier](#)  
[18.4.3 Metaclasse EncapsulatedClassifier](#)  
[18.4.4 Metaclasse BehavioredClassifier](#)  
[18.4.5 Metaclasse StructuralFeature](#)  
[18.4.6 Metaclasse BehavioralFeature](#)  
[18.4.7 Metaclasse Property](#)  
[18.4.8 Metaclasse Operation](#)  
[18.4.9 Metaclasse Reception](#)  
[18.4.10 Metaclasse Extension](#)

# Sobre o autor

Gilleanes Thorwald Araujo Guedes é doutor em Ciência da Computação pela Universidade Federal do Rio Grande do Sul (UFRGS), mestre em Ciência da Computação pela mesma instituição e bacharel em Informática pela Universidade da Região da Campanha (Urcamp). É professor no curso de Engenharia de Software na Universidade Federal do Pampa (UniPampa) – Campus de Alegrete. Já ministrou diversas palestras e cursos sobre UML em eventos científicos, cursos técnicos e cursos de pós-graduação lato sensu. Pode ser contatado pelo e-mail [gtag@novatec.com.br](mailto:gtag@novatec.com.br).

# Prefácio

A UML (Unified Modeling Language – Linguagem de Modelagem Unificada) é atualmente a linguagem-padrão de modelagem adotada internacionalmente pela indústria de engenharia de software. Em decorrência disso, existe hoje uma grande valorização e procura no mercado por profissionais que dominem essa linguagem.

O objetivo deste livro é ensinar ao leitor como modelar software por meio dos diversos diagramas que compõem a UML. No entanto, é importante destacar que a UML é uma linguagem de modelagem totalmente independente, não estando vinculada a nenhum processo de desenvolvimento específico e menos ainda a qualquer linguagem de programação.

Apesar de a UML oferecer um grande número de diagramas que enfoquem tanto características estruturais quanto comportamentais de um software, o leitor não deve se sentir obrigado a utilizar todos os diagramas propostos na modelagem de seus sistemas, pois cada um deles tem uma função específica e, algumas vezes, alguns deles não são necessários em determinadas situações ou domínios.

Esta obra foi revisada, atualizada e ampliada tomando como base a segunda edição do livro *UML 2 – Uma Abordagem Prática*, lançado em 2011. Contudo, esta nova edição baseia-se nas notações definidas na especificação da UML 2.5, a versão mais atual da linguagem até o momento em que este livro foi escrito. Assim, embora alguns exemplos sejam parecidos com os da edição anterior, estes foram revisados e atualizados sempre que isso se mostrou necessário. Este livro contém uma grande quantidade de novos conteúdos e exemplos não abordados na edição anterior, além da proposição de novos exercícios. O livro está estruturado da seguinte forma:

O capítulo 1 apresenta uma explanação a respeito da necessidade de se modelar software, além de introduzir a UML, destacando em linhas gerais as funções de cada diagrama.

O capítulo 2 enfoca o paradigma de orientação a objetos, uma vez que a UML é uma linguagem baseada nesse paradigma e utilizada principalmente para a modelagem de softwares orientados a objetos.

Os capítulos 3 a 16 abordam, respectivamente, os diagramas de casos de uso, de classes, objetos, pacotes, sequência, comunicação, máquina de estados, atividade, visão geral de interação, componentes, implantação, estrutura composta, tempo e perfis, sendo esse último um diagrama que não existia antes da versão 2.4 da linguagem. Em cada um desses capítulos, procuramos descrever a função de cada diagrama, detalhando seus elementos e apresentando exemplos de como os utilizar e modelar cada diagrama. Ao final da maioria dos capítulos, são sugeridos alguns exercícios para que o leitor possa praticar seu conhecimento. Todos os exercícios encontram-se resolvidos e explicados no final de seus respectivos capítulos.

Ao longo dos capítulos 3 a 14, foi modelado um sistema de controle bancário como ilustração. Embora este seja o sistema base usado para demonstrar a utilização da UML, outros cinco sistemas relativamente simples foram também parcialmente modelados como exemplos durante a explanação da maioria dos capítulos. Além disso, os exercícios propostos referem-se também a seis outros sistemas, parcialmente modelados por meio das soluções dos exercícios.

O capítulo 17 apresenta um estudo de caso referente a um sistema maior e mais complexo, no qual modelamos um sistema para controle de pizzaria online, em que os pedidos dos clientes poderão ser feitos pela internet. Finalmente, no último capítulo, enfocamos a arquitetura da linguagem, discorrendo sobre a estrutura da UML 2.5.

# CAPÍTULO 1

## Introdução à UML

A UML – Unified Modeling Language ou Linguagem de Modelagem Unificada – é uma linguagem visual utilizada para modelar softwares baseados no paradigma de orientação a objetos. É uma linguagem de modelagem de propósito geral que pode ser aplicada a todos os domínios de aplicação. Essa linguagem é atualmente a linguagem-padrão de modelagem adotada internacionalmente pela indústria de engenharia de software.

Deve ficar bem claro, porém, que a UML não é uma linguagem de programação, e sim uma linguagem de modelagem, ou seja, uma notação cujo objetivo é auxiliar os engenheiros de software a definirem as características do sistema, como seus requisitos, seu comportamento, sua estrutura lógica, a dinâmica de seus processos e até mesmo suas necessidades físicas em relação ao equipamento sobre o qual o sistema deverá ser implantado. Tais características podem ser definidas por meio da UML antes de o software começar a ser realmente desenvolvido. Além disso, cumpre destacar que a UML não é um processo de desenvolvimento de software e tampouco está ligada a um de forma exclusiva, sendo totalmente independente, assim, ela pode ser utilizada por qualquer processo de desenvolvimento ou mesmo da forma que o engenheiro de software considerar mais adequada.

### 1.1 Breve Histórico da UML

A UML surgiu da união de três métodos orientados a objeto: o método de Booch, o método OMT (Object Modeling Technique) de Jacobson e o método OOSE (Object-Oriented Software Engineering) de Rumbaugh. Até meados da década de 1990, esses eram os métodos orientados a objeto mais populares entre os profissionais da área de desenvolvimento de software. A união desses métodos contou com o amplo apoio da Rational

Software, que a incentivou e financiou.

O esforço inicial do projeto começou com a união do método de Booch ao OMT de Jacobson, o que resultou no lançamento do Método Unificado no final de 1995. Logo em seguida, Rumbaugh juntou-se a Booch e Jacobson na Rational Software e seu método OOSE começou também a ser incorporado à nova linguagem. Ao longo de seu desenvolvimento, foram também incorporadas à linguagem boas práticas de projeto de linguagens de modelagem, programação orientada a objetos e linguagens de descrição arquitetural. O trabalho de Booch, Jacobson e Rumbaugh, conhecidos popularmente como “Os Três Amigos”, resultou no lançamento, em 1996, da primeira versão da UML propriamente dita. Tão logo a primeira versão foi lançada, muitas empresas atuantes na área de modelagem e desenvolvimento de software passaram a contribuir para o projeto, fornecendo sugestões para melhorar e ampliar a linguagem. Finalmente, a UML foi adotada, em 1997, pela OMG (Object Management Group ou Grupo de Gerenciamento de Objetos), como uma linguagem-padrão de modelagem.

A versão 2.0 da linguagem foi oficialmente lançada em julho de 2005. Atualmente a UML encontra-se na versão 2.5. Essa última versão foi lançada com o objetivo de simplificar a estrutura da linguagem. A documentação oficial da UML pode ser consultada no site da OMG em [www.omg.org](http://www.omg.org) ou mais exatamente em [www.uml.org](http://www.uml.org).

## 1.2 Por que Modelar Software?

Qual a real necessidade de se modelar um software? Muitos “profissionais” podem afirmar que conseguem determinar todas as necessidades de um sistema de informação de cabeça e que sempre trabalharam assim. Qual a real necessidade de se projetar uma casa? Um pedreiro experiente não é capaz de construí-la sem um projeto? Isso pode ser verdade, mas a questão é muito mais ampla, envolvendo fatores complexos, como elicitação e análise de requisitos, projeto, prazos, custos, documentação, manutenção e reusabilidade, entre outros.

Existe uma diferença gritante entre construir uma pequena casa e construir um prédio de vários andares. Obviamente, para se construir um edifício, é necessário, em primeiro lugar, desenvolver um projeto muito

bem elaborado, cujos cálculos têm de estar corretos e precisos. Além disso, é preciso fornecer uma estimativa de custos, determinar em quanto tempo a construção estará concluída, avaliar a quantidade de profissionais necessária à execução da obra, especificar a quantidade de material a ser adquirida para a construção, escolher o local onde o prédio será erguido etc. Grandes projetos não podem ser modelados de cabeça, nem mesmo a maioria dos pequenos projetos pode sê-lo, exceto, talvez, aqueles extremamente simples.

Na realidade, por mais simples que seja, todo sistema deve ser modelado antes de se iniciar sua implementação, entre outras coisas, porque os sistemas de informação frequentemente costumam ter tendência a “crescer”, isto é, aumentar em tamanho, complexidade e abrangência. Alguns profissionais costumam afirmar que sistemas de informação são “vivos” porque nunca estão completamente finalizados. Na verdade, o termo correto seria “dinâmicos”, pois normalmente os sistemas de informação estão em constante mudança. Tais mudanças são devidas a diversos fatores, como:

- Os clientes desejam constantemente modificações ou melhorias no sistema.
- O mercado está sempre mudando, o que força a adoção de novas estratégias pelas empresas e, consequentemente, de seus sistemas.
- O governo frequentemente promulga novas leis e cria novos impostos e alíquotas ou, ainda, modifica as leis, os impostos e as alíquotas já existentes, o que acarreta a manutenção no software.

Assim, um sistema de informação precisa ter uma documentação detalhada, precisa e atualizada para ser mantido com facilidade, rapidez e correção, sem produzir novos erros ao corrigir os antigos. Modelar um sistema é uma forma bastante eficiente de documentá-lo, mas a modelagem não serve apenas para isso: a documentação é apenas uma das vantagens fornecidas pela modelagem. Existem muitas outras que serão discutidas nas próximas seções.

### **1.2.1 Modelo de Software – Uma Definição**

A modelagem de um software implica criar modelos de software, mas o

que é realmente um modelo de software? Um modelo de software captura uma visão de um sistema físico, é uma abstração do sistema com um certo propósito, como descrever aspectos estruturais ou comportamentais do software. Esse propósito determina o que deve ser incluído no modelo e o que é considerado irrelevante. Assim, um modelo descreve completamente aqueles aspectos do sistema físico relevantes ao propósito do modelo, no nível apropriado de detalhe.

Dessa forma, um modelo de casos de uso fornecerá uma visão dos requisitos necessários ao sistema, identificando as funcionalidades do software e os atores que poderão utilizá-las, não se preocupando em detalhar nada além disso. Já um modelo conceitual identificará as classes relacionadas ao domínio do problema, sem detalhar seus métodos, enquanto um modelo de domínio ampliará o modelo conceitual, incluindo informações relativas à solução do problema, como os métodos necessários a essa solução.

### **1.2.2 Elicitação e Análise de Requisitos**

Uma das primeiras fases de um processo de desenvolvimento de software consiste na Elicitação ou Levantamento de Requisitos. As outras etapas, sugeridas por muitos autores, são Análise de Requisitos, Projeto, que se constitui na principal fase da modelagem, Codificação, Testes e Implantação. Dependendo do método/processo adotado, essas etapas ganham, por vezes, nomenclaturas diferentes, podendo algumas delas ser condensadas em uma etapa única, ou uma etapa pode ser dividida em duas ou mais etapas.

Se tomarmos como exemplo o Processo Unificado (Unified Process), um método de desenvolvimento de software, veremos que este se divide em quatro fases: Concepção, em que é feita a elicitação de requisitos inicial e determina-se a viabilidade de desenvolver o software; Elaboração, em que são feitos a análise dos requisitos e o projeto do software; Construção, em que o software é implementado e testado; Transição, em que o software será implantado. As fases de Elaboração e Construção ocorrem, sempre que possível, em ciclos iterativos, dessa forma, sempre que um ciclo é completado pela fase de Construção, volta-se à fase de Elaboração para tratar do ciclo seguinte, até todo o software ser finalizado.

As etapas de elicição e análise de requisitos trabalham com o domínio do problema e tentam determinar “o que” o software deve fazer e se é realmente possível desenvolver o software solicitado. Na etapa de elicição de requisitos, o engenheiro de software busca compreender as necessidades do usuário e o que ele deseja que o sistema a ser desenvolvido realize. Isso é feito sobretudo por meio de entrevistas, nas quais o engenheiro tenta compreender como funciona atualmente o processo a ser informatizado e quais serviços o cliente precisa que o software forneça.

Devem ser realizadas tantas entrevistas quantas forem necessárias para que as necessidades do usuário sejam bem compreendidas. Durante as entrevistas, o engenheiro deve auxiliar o cliente a definir quais informações deverão ser fornecidas, quais deverão ser produzidas, como deverão ser combinadas e/ou transformadas para auxiliar no processo de tomada de decisão, quais as regras de negócio referentes a cada processo, quais os níveis de desempenho, confiabilidade e proteção exigidos do software, entre outros requisitos.

Um dos principais problemas enfrentados na fase de elicição de requisitos é a comunicação, que se constitui em um dos maiores desafios da engenharia de software, caracterizando-se pela dificuldade em conseguir compreender um conjunto de conceitos vagos, nebulosos e difusos que representam as necessidades e os desejos dos clientes e transformá-los em conceitos concretos e inteligíveis.

Além disso, há um grave problema adicional causado pela maneira como os conceitos são interpretados, já que um mesmo termo pode ter múltiplos significados, dependendo da área em que é utilizado. O significado das palavras e termos pode ter significados e interpretações muito diferentes, dependendo do domínio em que são aplicados, da região onde são utilizados e até mesmo da faixa etária em que são empregados. Dessa forma, podem causar interpretações errôneas de conceitos e, por conseguinte, grandes prejuízos por não atender às reais necessidades dos clientes. Para evitar isto, é importante elaborar um glossário de termos nessa fase, em que os conceitos elicitados tenham uma interpretação única e clara, de forma a evitar ambiguidades no momento de compreender o que os clientes desejam.

A fase de elicição de requisitos deve identificar dois tipos de requisitos:

os funcionais e os não funcionais. Os requisitos funcionais correspondem ao que o cliente quer que o sistema realize, ou seja, as funcionalidades do software. Já os requisitos não funcionais correspondem a restrições, condições, consistências e validações que devem ser levadas a efeito sobre os requisitos funcionais. Por exemplo, em um sistema bancário, deve ser oferecida a opção de abrir novas contas-correntes, o que é um requisito funcional. Já determinar que somente pessoas maiores de idade possam abrir contas-correntes é um requisito não funcional.

Alguns requisitos não funcionais identificam regras de negócio, ou seja, políticas, normas e condições estabelecidas pela empresa que devem ser seguidas na execução de uma funcionalidade. Por exemplo, estabelecer que depois de abrir uma conta é necessário depositar um valor mínimo inicial é uma regra de negócio adotada por um determinado banco e que não necessariamente é seguida por outras instituições bancárias. Outro exemplo de regra de negócio seria determinar que, em um sistema de biblioteca, só se poderia realizar um novo empréstimo para um sócio se o seu limite máximo de exemplares para locação ainda não tivesse sido atingido, caso contrário ele deveria devolver algum exemplar antes de realizar um novo empréstimo.

Existem ainda diversos tipos de requisitos não funcionais, como de usabilidade, desempenho, confiabilidade, segurança ou interface, que se referem ao sistema como um todo e não estão atrelados a um requisito funcional específico.

Logo após a elicitação de requisitos, passa-se à fase em que as necessidades apresentadas pelo cliente são analisadas. Essa etapa é conhecida como análise de requisitos. Aqui, o engenheiro examina os requisitos enunciados pelos usuários-chave ou stakeholders (profissionais com experiência que têm conhecimento profundo sobre o funcionamento de determinados processos na empresa) verificando se estes foram especificados corretamente e se foram realmente bem compreendidos. A partir da etapa de análise de requisitos são determinadas as reais necessidades do sistema de informação.

A grande questão é: como saber se as necessidades dos clientes e usuários-chave foram realmente bem compreendidas? Um dos objetivos da análise de requisitos consiste em determinar se os requisitos dos usuários

foram entendidos de maneira correta, verificando se alguma questão deixou de ser abordada, determinando se algum requisito foi especificado incorretamente ou se algum conceito precisa ser mais bem explicado, ou seja, se a declaração dos requisitos está clara, consistente, completa, sem conflitos e ambiguidades.

Durante a análise de requisitos, uma linguagem de modelagem auxilia a levantar questões que não foram concebidas durante as entrevistas iniciais. Tais questões devem ser sanadas o quanto antes, para que o projeto do software não tenha que sofrer modificações quando seu desenvolvimento já estiver em andamento, o que pode causar significativos atrasos no desenvolvimento do software, sendo por vezes necessário reiniciar o projeto do começo.

Além daquele concernente à comunicação, outro grande problema encontrado durante as entrevistas consiste no fato de que, na maioria das vezes, os usuários não têm realmente certeza do que querem e não conseguem enxergar as reais potencialidades de um sistema de informação. Em geral, os engenheiros de software precisam sugerir inúmeras características e funções do sistema que o cliente não sabia como formular ou sequer havia imaginado. Na realidade, na maior parte das vezes, esses profissionais precisam reestruturar o modo como as informações são geridas e utilizadas pela empresa e apresentar maneiras de combiná-las e apresentá-las de maneira que possam ser mais bem aproveitadas pelos usuários, auxiliando-os na tomada de decisões.

Em muitos casos é realmente isso o que os clientes esperam dos engenheiros de software, porém, em outros, os engenheiros encontram fortes resistências a qualquer mudança na forma como a empresa manipula suas informações, tornando-se necessário um significativo esforço para provar ao cliente que as modificações sugeridas permitirão um melhor desempenho do software, além de ser útil para a própria empresa, obviamente, uma vez que permitirão combinar e comparar as informações de tal forma que estas possam auxiliar os gerentes na tomada de decisões empresariais tornando a empresa mais competitiva. Na realidade, nesse último caso é fundamental trabalhar bastante o aspecto social da implantação de um sistema informatizado na empresa, pois muitas vezes a resistência não é tanto da gerência, mas dos usuários finais, que serão

obrigados a mudar a forma como estavam acostumados a trabalhar e aprender a utilizar uma nova tecnologia. Deve-se, assim, procurar destacar como o novo sistema melhorará e facilitará o trabalho desses usuários.

### **1.2.3 Prototipação**

A prototipação é uma técnica bastante popular e de fácil aplicação que permite validar se os requisitos do software foram compreendidos corretamente e se a proposta representada pelo protótipo satisfará realmente as necessidades do cliente. Essa técnica consiste em desenvolver rapidamente um “rascunho” do que seria o sistema de informação quando estivesse finalizado. Um protótipo normalmente apresenta pouco mais do que a interface do software a ser desenvolvido, ilustrando como as informações seriam inseridas e recuperadas no sistema, apresentando alguns exemplos com dados fictícios de quais seriam os resultados apresentados pelo software, principalmente em forma de relatórios. A utilização de um protótipo pode evitar que, após meses ou até anos de desenvolvimento, descubra-se, ao implantar o sistema, que o software não atende completamente às necessidades do cliente em razão, sobretudo, de falhas de comunicação durante as entrevistas iniciais.

Hoje, é possível desenvolver protótipos com extrema rapidez e facilidade, por meio da utilização de ferramentas conhecidas como RAD (Rapid Application Development ou Desenvolvimento Rápido de Aplicações), que são encontradas na maioria dos ambientes de desenvolvimento das linguagens de programação atuais, como NetBeans, Delphi, Visual Basic ou C++ Builder, entre outras. Essas ferramentas disponibilizam ambientes de desenvolvimento que permitem a construção de interfaces de forma muito rápida, além de permitirem também modificar tais interfaces de maneira igualmente veloz, na maioria das vezes sem alterar qualquer código porventura já escrito.

As ferramentas RAD permitem criar formulários e inserir componentes neles, de forma muito simples, rápida e fácil, bastando ao desenvolvedor selecionar o componente (botões, caixas de texto, labels, combos etc.) em uma barra de ferramentas e clicar com o mouse sobre o formulário. Além disso, tais ferramentas permitem ao usuário mudar a posição dos componentes depois de terem sido colocados no formulário simplesmente

selecionando o componente com o mouse e o arrastando para a posição desejada.

Esse tipo de ferramenta é extremamente útil no desenvolvimento de protótipos pela facilidade de produzir e modificar as interfaces. Assim, depois de determinar quais as modificações necessárias ao sistema de informação após o protótipo ter sido apresentado aos usuários, pode-se modificar a interface do protótipo de acordo com as novas especificações e reapresentá-lo ao cliente de forma muito rápida.

Seguindo esse raciocínio, a etapa de análise de requisitos deve, obrigatoriamente, produzir um protótipo para demonstrar como se apresentará e comportará o sistema em essência, bem como quais informações deverão ser inseridas no sistema e que tipo de informações deverá ser fornecido pelo software. Um protótipo é de vital importância durante as primeiras fases de engenharia de um sistema de informação. Por meio da ilustração que um protótipo pode apresentar, a maioria das dúvidas e erros de especificação pode ser sanada pelo fato de um protótipo demonstrar visualmente um exemplo de como funcionará o sistema depois de concluído, como será sua interface, de que maneira os usuários interagirão com ele, que tipos de relatórios serão fornecidos etc., facilitando a compreensão do cliente.

Apesar das grandes vantagens advindas do uso da técnica de prototipação, é necessária, ainda, uma ressalva: um protótipo pode induzir o cliente a acreditar que o software encontra-se em um estágio bastante avançado de desenvolvimento. Com frequência, o cliente não comprehende o conceito de um protótipo. Para ele, o esboço apresentado já é o próprio sistema praticamente acabado. Por isso, muitas vezes não entende nem aceita prazos longos, os quais considera absurdos, já que o sistema foi-lhe apresentado já funcionando, necessitando de poucos ajustes. Por isso, é preciso deixar bem claro ao usuário que o software que lhe está sendo apresentado é apenas uma “maquete” do que será o sistema de informação quando estiver finalizado e que seu desenvolvimento ainda não foi realmente iniciado.

#### **1.2.4 Prazos e Custos**

Em seguida, vem a espinhosa e desagradável, porém fundamental, questão

dos prazos e custos. Como determinar o prazo real de entrega de um software? Como determinar a real complexidade de desenvolvimento? Quantos profissionais deverão trabalhar no projeto? Qual será o custo total para concluir o projeto? Qual deverá ser o valor estipulado para produzir o sistema? Geralmente, após as primeiras entrevistas, os clientes estão bastante interessados em saber quanto lhes custará o software e em quanto tempo o terão implantado e funcionando em sua empresa.

A estimativa de tempo é realmente um tópico extremamente complexo da engenharia de software. Na realidade, por mais bem modelado que um sistema tenha sido, ainda, assim, fica difícil determinar com exatidão os prazos finais de entrega do software. Uma boa modelagem auxilia a estimar a complexidade de desenvolvimento de um sistema, o que, por sua vez, ajuda a determinar o prazo final em que o software será entregue. No entanto, é preciso ter diversos sistemas de informação com níveis de dificuldade e características semelhantes aos do software que será construído, previamente desenvolvidos e bem documentados, para determinar com mais exatidão a estimativa de prazos.

Para auxiliar a estimativa de prazos e custos de um software, a documentação da empresa desenvolvedora deverá ter registros das datas de início e término de cada projeto já concluído, além do custo real de desenvolvimento que tais projetos acarretaram, o número de profissionais envolvidos em cada um deles, bem como as funções por eles desempenhadas e os salários recebidos, devendo envolver também custos com manutenção.

Na verdade, uma empresa de desenvolvimento de software que nunca desenvolveu um sistema de informação antes e, portanto, não tem documentação histórica de projetos anteriores, terá dificuldades em apresentar uma estimativa correta de prazos e custos, principalmente porque a equipe de desenvolvimento não saberá com certeza quanto tempo levará desenvolvendo o sistema nem, já que o tempo de desenvolvimento influencia diretamente o custo do sistema, logicamente, estimar corretamente o valor a ser cobrado pelo software.

Contudo, mesmo com o auxílio dessa documentação, ainda assim é muito difícil estipular uma data exata. O máximo que se pode conseguir é apresentar uma data aproximada, com base na experiência documentada

de desenvolvimento de outros softwares. Isso se deve ao fato de que cada software apresenta um desafio novo, trata-se de um problema para o qual é necessário criar uma solução (representada pelo software) e, por isso, é muito difícil determinar a complexidade real do novo sistema a ser desenvolvido. Assim, é recomendável acrescentar alguns meses à data de entrega, o que servirá como margem de segurança para possíveis erros de estimativa.

Uma maneira de determinar a complexidade de um sistema e, por conseguinte, estimar seu prazo e custo de desenvolvimento é o método de cálculo que utiliza pontos de casos de uso. Esse método calcula o peso dos atores que interagem com o sistema e o peso dos casos de uso fornecidos pelo software, considerando o tipo de atores envolvidos e o número de transações ou entidades envolvidas em cada caso de uso. Por meio desse cálculo, é possível fazer uma estimativa mais próxima da realidade.

É importante estimar o mais corretamente possível os prazos e custos de desenvolvimento, pois se a estimativa de prazo estiver errada, cada dia a mais de desenvolvimento do projeto acarretará prejuízos para a empresa que desenvolve o sistema, decorrentes, por exemplo, de pagamentos de salários aos profissionais envolvidos no projeto que não haviam sido previstos e desgaste dos equipamentos utilizados. Isso sem levar em conta prejuízos mais difíceis de contabilizar, como manter inúmeros profissionais ocupados em projetos que já deveriam estar concluídos, que deixam de trabalhar em novos projetos, além da insatisfação dos clientes por não receberem o produto no prazo estimado e a propaganda negativa daí decorrente.

### **1.2.5 Projeto**

Enquanto a fase de análise trabalha com o domínio do problema, a fase de projeto trabalha com o domínio da solução, procurando estabelecer “como” o sistema fará o que foi determinado na fase de análise, ou seja, qual será a solução para o problema identificado. Na etapa de projeto é realizada a maior parte da modelagem do software a ser desenvolvido, ou seja, é produzida a arquitetura do sistema.

A etapa de projeto toma a modelagem iniciada na fase de análise e a enriquece com profundos acréscimos, melhorias e detalhamentos.

Enquanto na análise foram identificadas as funcionalidades necessárias ao software e suas restrições, na fase de projeto será estabelecido como essas funcionalidades deverão realizar o que foi solicitado.

A fase de projeto leva em consideração os recursos tecnológicos existentes para que o problema apresentado pelo cliente possa ser solucionado. É nesse momento que será selecionada a linguagem de programação a ser utilizada, o sistema gerenciador de banco de dados a ser empregado, como será a interface final do sistema e até mesmo como o software será distribuído fisicamente na empresa, especificando o hardware necessário para a sua implantação e funcionamento correto.

## 1.2.6 Manutenção

Possivelmente, a questão mais importante dentre todas as outras já enunciadas é a manutenção. Alguns autores afirmam que muitas vezes a manutenção de um software pode representar de 40% a 60% do custo total do projeto. Alguém poderá, então, dizer que a modelagem é necessária para diminuir os custos com a manutenção – se a modelagem estiver correta, o sistema não apresentará erros e, então, não precisará sofrer manutenções.

Embora um dos objetivos de modelar um software seja realmente diminuir a necessidade de mantê-lo, a modelagem não serve apenas para isso. Na verdade, na maioria dos casos, a manutenção de um software é inevitável, pois, como foi dito, as necessidades de uma empresa são dinâmicas e mudam constantemente, o que faz surgir novas necessidades que não existiam na época em que o software foi projetado, isso sem mencionar as frequentes mudanças em leis, alíquotas, impostos, taxas ou formato de notas fiscais, por exemplo. Levando tudo em consideração, é bastante provável que um sistema de informação, por mais bem modelado que esteja, precise sofrer manutenções.

Nesse caso, a modelagem não serve apenas para diminuir a necessidade de futuras manutenções, mas também para facilitar a compreensão do sistema por quem tiver que o manter, já que, em geral, a manutenção de um sistema é considerada uma tarefa ingrata pelos profissionais de desenvolvimento, por normalmente exigir que estes despendam grandes esforços para compreender códigos escritos por outros cujos estilos de

desenvolvimento são diferentes e que, em geral, não se encontram mais na empresa.

Esse tipo de código é conhecido como “software legado” ou “código alienígena”. O termo refere-se a códigos que não seguem as regras atuais de desenvolvimento da empresa, não foram modelados nem documentados e, por conseguinte, têm pouca ou nenhuma documentação. Além disso, nenhum dos profissionais da equipe atual trabalhou no projeto inicial e, para piorar, o código sofreu manutenções anteriores por outros profissionais que também não se encontram mais na empresa e cada um deles tinha um estilo de desenvolvimento diferente, ou seja, como se diz no meio de desenvolvimento, o código encontra-se “remendado”.

Assim, uma modelagem correta, aliada a uma documentação completa e atualizada de um sistema de informação, torna mais rápido o processo de manutenção e impede que erros sejam cometidos, já que é muito comum que, depois de manter uma rotina, método ou função de um software, outras rotinas ou funções do sistema que antes funcionavam perfeitamente passem a apresentar erros ou simplesmente deixem de funcionar. Tais erros são conhecidos como “efeitos colaterais” da manutenção.

Além disso, qualquer manutenção a ser realizada em um software deve ser também modelada e documentada, para não desatualizar a documentação do sistema e prejudicar futuras manutenções, já que muitas vezes uma documentação desatualizada pode ser mais prejudicial à manutenção do sistema do que nenhuma documentação.

### **1.2.7 Documentação Histórica**

Finalmente, existe a questão da perspectiva histórica, ou seja, novamente a já tão falada documentação de software. Neste caso, referimo-nos à documentação histórica dos projetos anteriores já concluídos pela empresa. É por meio dessa documentação histórica que a empresa pode responder a perguntas como:

- A empresa está evoluindo?
- O processo de desenvolvimento tornou-se mais rápido?
- As metodologias hoje adotadas são superiores às práticas aplicadas anteriormente?

- As ferramentas CASE atualmente utilizadas apresentam melhores resultados do que as anteriores?
- As estimativas de prazos e custos estão mais próximas da realidade?
- A qualidade do software produzido está melhorando?

Uma empresa ou setor de desenvolvimento de software necessita de um registro detalhado de cada um de seus sistemas de informação antes desenvolvidos para poder determinar, entre outros, fatores como:

- Qual é a média de custo e de tempo gasto na análise de um software?
- Qual é a média de custo e de tempo gasto no projeto?
- Qual é a média de custo e de tempo para codificar e testar um sistema?
- Quantos profissionais são necessários envolver, em média, em cada fase de desenvolvimento do software?
- A média de manutenções que um sistema sofre dentro de um determinado período de tempo.

Essas informações são computadas nos orçamentos de desenvolvimento de novos softwares e são de grande auxílio no momento de determinar prazos e custos mais próximos da realidade.

Além disso, a documentação pode ser muito útil em outra área: a Reusabilidade. Um dos grandes desejos e muitas vezes necessidade dos clientes é que o software esteja concluído o mais rápido possível. Uma das formas de agilizar o processo de desenvolvimento é a reutilização de rotinas, funções e algoritmos previamente desenvolvidos em outros sistemas. Nesse caso, a documentação correta do sistema pode auxiliar a sanar questões como:

- Onde as rotinas se encontram?
- Para que foram utilizadas?
- Em que projetos estão documentadas?
- São adequadas ao software atualmente em desenvolvimento?
- Qual é o nível necessário de adaptação dessas rotinas para serem utilizadas na construção do sistema atual?

## 1.3 Por que Tantos Diagramas?

Por que a UML é composta de tantos diagramas? O objetivo disso é fornecer múltiplas visões do sistema a ser modelado, analisando-o e modelando-o sob diversos aspectos, procurando-se, assim, atingir a completude da modelagem, permitindo que cada diagrama complemente os outros.

Cada diagrama da UML analisa o sistema, ou parte dele, sob uma determinada óptica. É como se o sistema fosse modelado em camadas. Alguns diagramas enfocam o sistema de forma mais geral, apresentando uma visão externa do sistema, como é o objetivo do Diagrama de Casos de Uso, enquanto outros oferecem uma visão de uma camada mais profunda do software, apresentando um enfoque mais técnico ou, ainda, visualizando apenas uma característica específica do sistema ou um determinado processo. A utilização de diversos diagramas permite que falhas sejam descobertas, diminuindo a possibilidade da ocorrência de erros futuros.

Tomando novamente o exemplo da construção de um edifício, percebemos que ao se projetar uma construção, esta não tem apenas uma planta, mas diversas, enfocando o projeto de construção do prédio sob diferentes formas, algumas referentes ao layout dos andares, outras apresentando a planta hidráulica e outras, ainda, abordando a planta elétrica, por exemplo. Isso torna o projeto do edifício completo, abrangendo todas as características da construção. Da mesma maneira, os diversos diagramas fornecidos pela UML permitem analisar o sistema em diferentes níveis, podendo enfocar a organização estrutural do sistema, o comportamento de um processo específico, a definição de um determinado algoritmo ou até mesmo as necessidades físicas para implantar o software.

## 1.4 Rápido Resumo dos Diagramas da UML

A seguir, descreveremos rapidamente cada um dos diagramas oferecidos pela UML, destacando suas principais características.

### 1.4.1 Diagrama de Casos de Uso

O diagrama de casos de uso tem por objetivo apresentar uma visão externa geral das funcionalidades que o sistema deverá oferecer aos usuários, sem se preocupar muito com a questão de como tais funcionalidades serão

implementadas. Costuma ser utilizado principalmente nas fases de elicitação e análise de requisitos do sistema, embora venha a ser consultado durante todo o processo de modelagem e possa servir de base para diversos outros diagramas. Procura apresentar uma linguagem simples e de fácil compreensão para que os usuários possam ter uma ideia geral de como o sistema vai se comportar. Procura identificar os atores (usuários, outros sistemas ou até mesmo algum hardware especial) que utilizarão, de alguma forma, o software, bem como os serviços, ou seja, as funcionalidades que o sistema disponibilizará a esses atores, conhecidas nesse diagrama como casos de uso. Veja na figura 1.1 um exemplo desse diagrama.

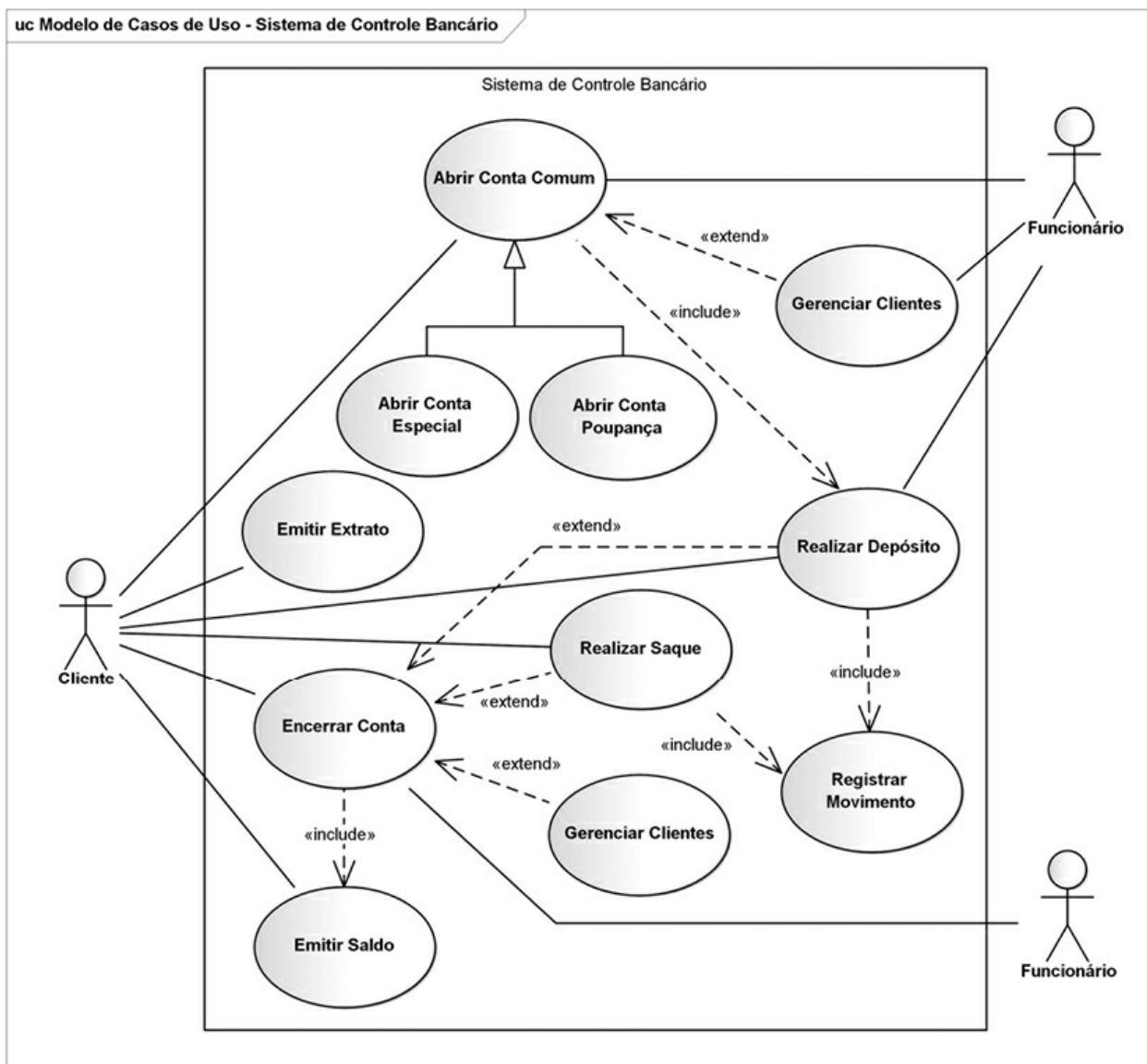


Figura 1.1 – Exemplo de Diagrama de Casos de Uso.

## 1.4.2 Diagrama de Classes

O diagrama de classes é um dos mais importantes e utilizados da UML. Seu principal enfoque está em permitir a visualização das classes que compõem o sistema com seus respectivos atributos e métodos, bem como em demonstrar como as classes do diagrama se relacionam, complementam e transmitem informações entre si. Esse diagrama apresenta uma visão estática de como as classes estão organizadas, preocupando-se em como definir a estrutura lógica delas. O diagrama de classes serve ainda como apoio para a construção da maioria dos outros diagramas da linguagem UML. A figura 1.2 apresenta um exemplo desse diagrama.

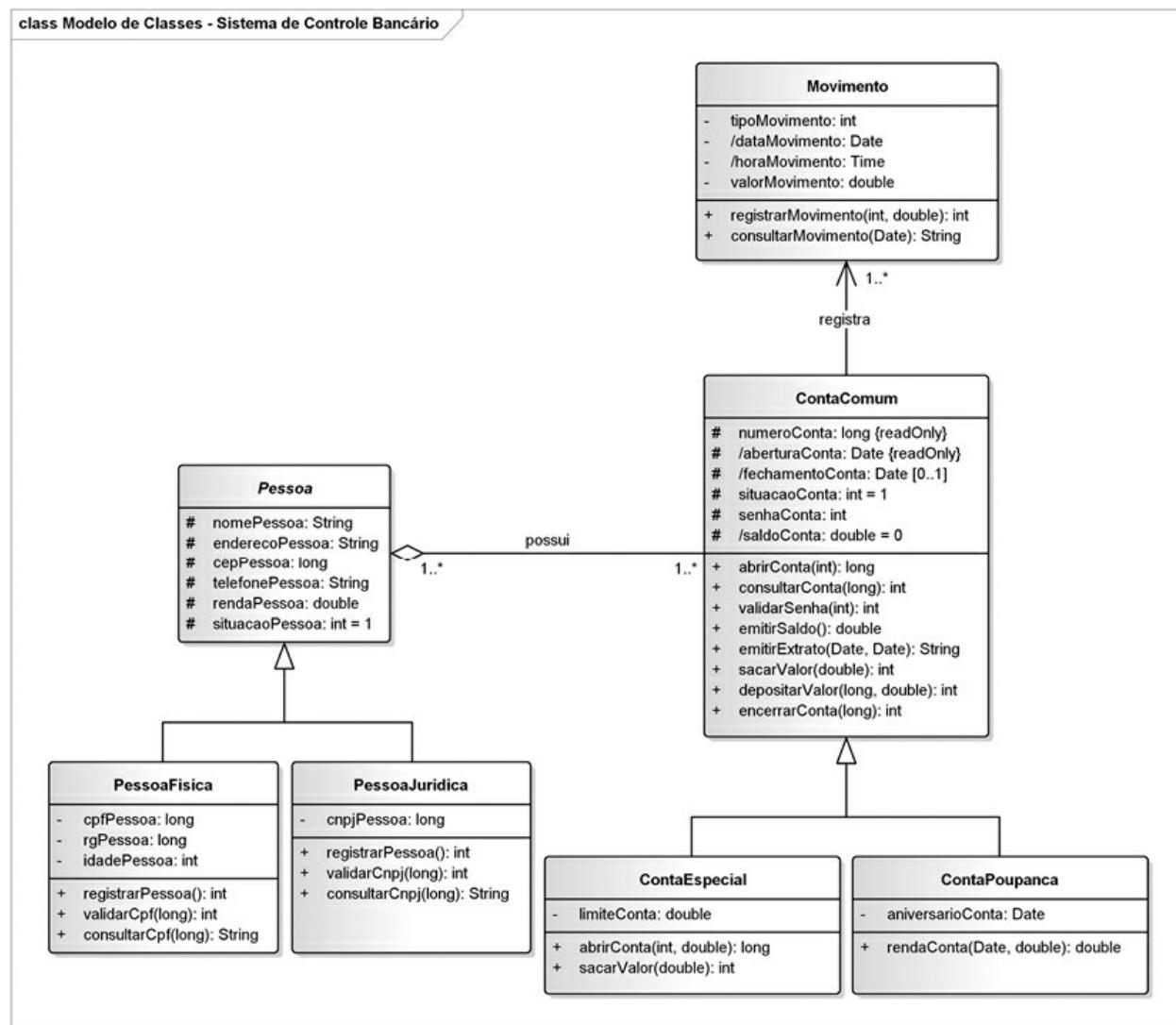


Figura 1.2 – Exemplo de Diagrama de Classes.

### 1.4.3 Diagrama de Objetos

O diagrama de objetos está amplamente associado ao diagrama de classes. Na verdade, o diagrama de objetos é praticamente um complemento do diagrama de classes e bastante dependente deste. O diagrama fornece uma visão dos valores armazenados pelos objetos de um diagrama de classes em um determinado momento da execução de um processo do software. A figura 1.3 apresenta um exemplo de diagrama de objetos.

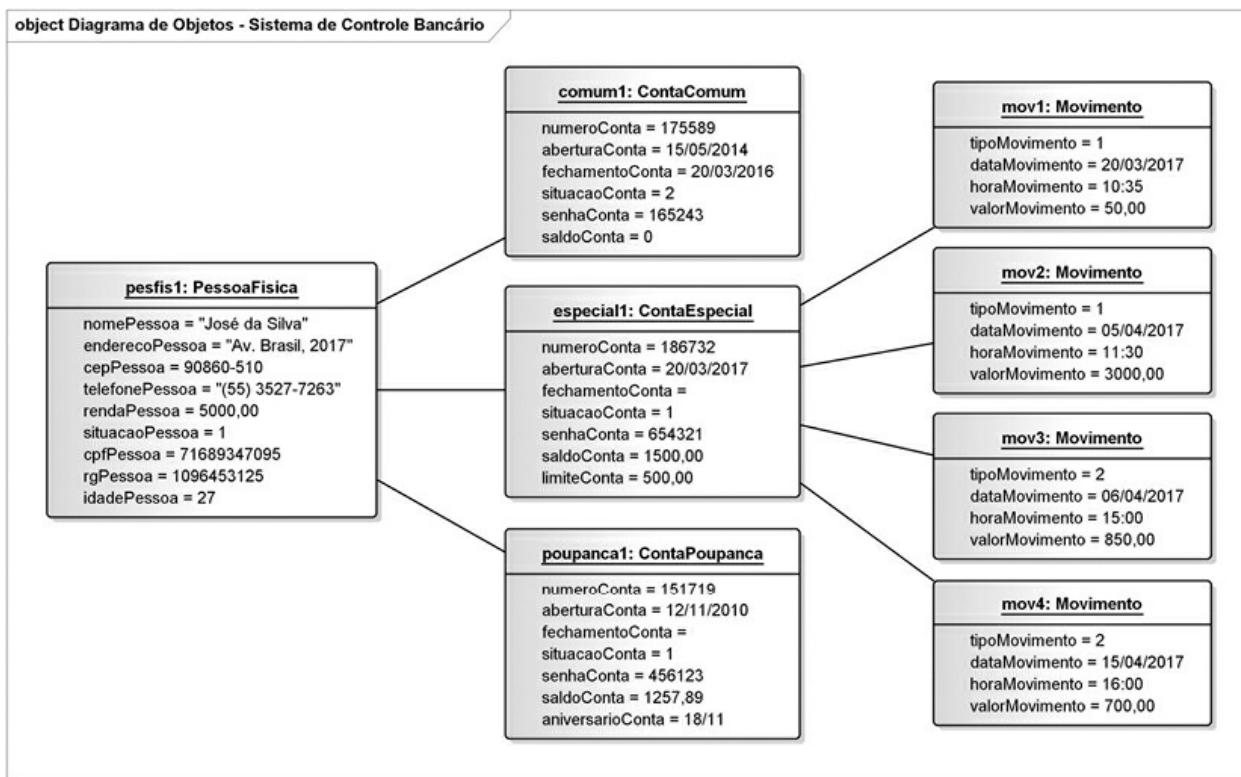
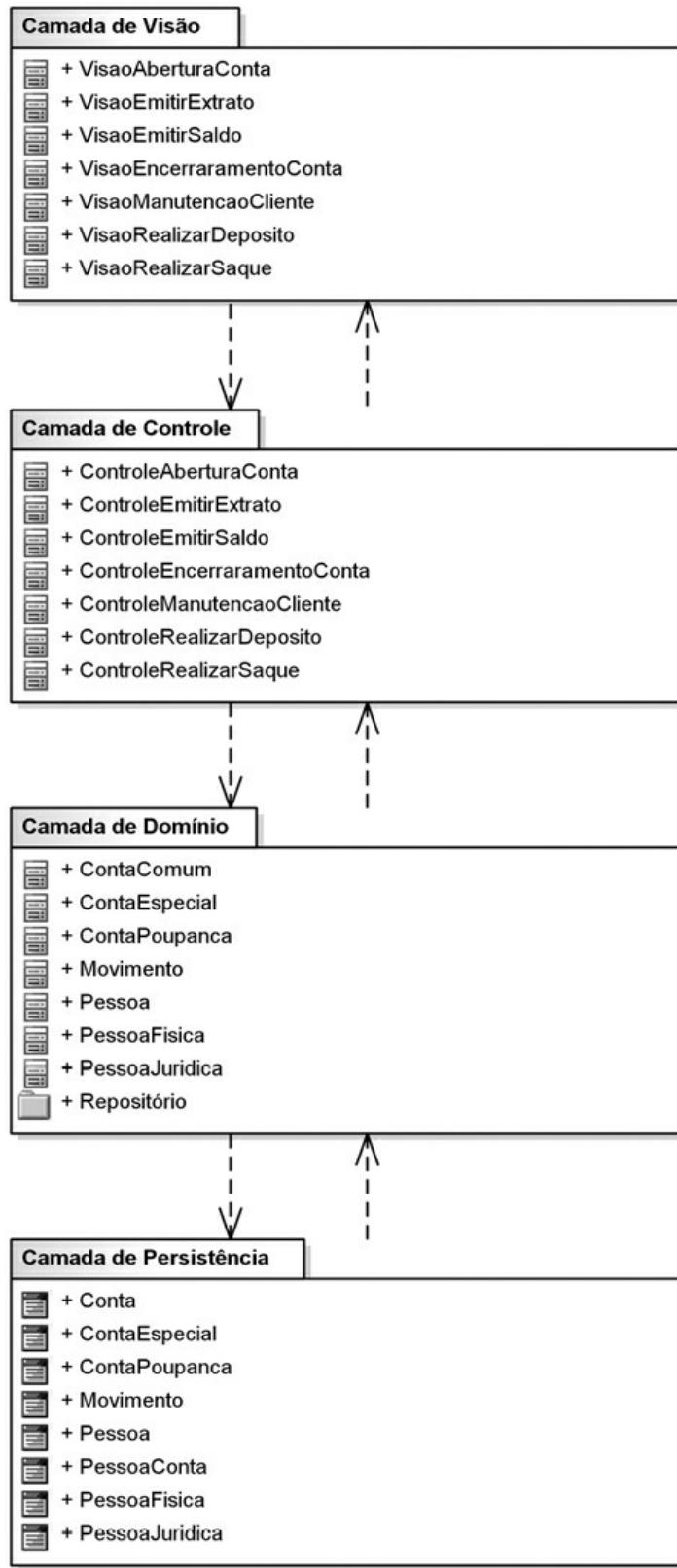


Figura 1.3 – Exemplo de Diagrama de Objetos.

### 1.4.4 Diagrama de Pacotes

O diagrama de pacotes é um diagrama estrutural que tem por objetivo representar como os elementos do modelo estão divididos logicamente. Tais elementos podem ser, por exemplo, subsistemas ou componentes englobados por um sistema ou as camadas que o compõem, entre outras possibilidades. Essas divisões lógicas são denominadas Pacotes. Esse diagrama pode ser utilizado de maneira independente ou associado a outros diagramas. A figura 1.4 apresenta um exemplo desse diagrama.

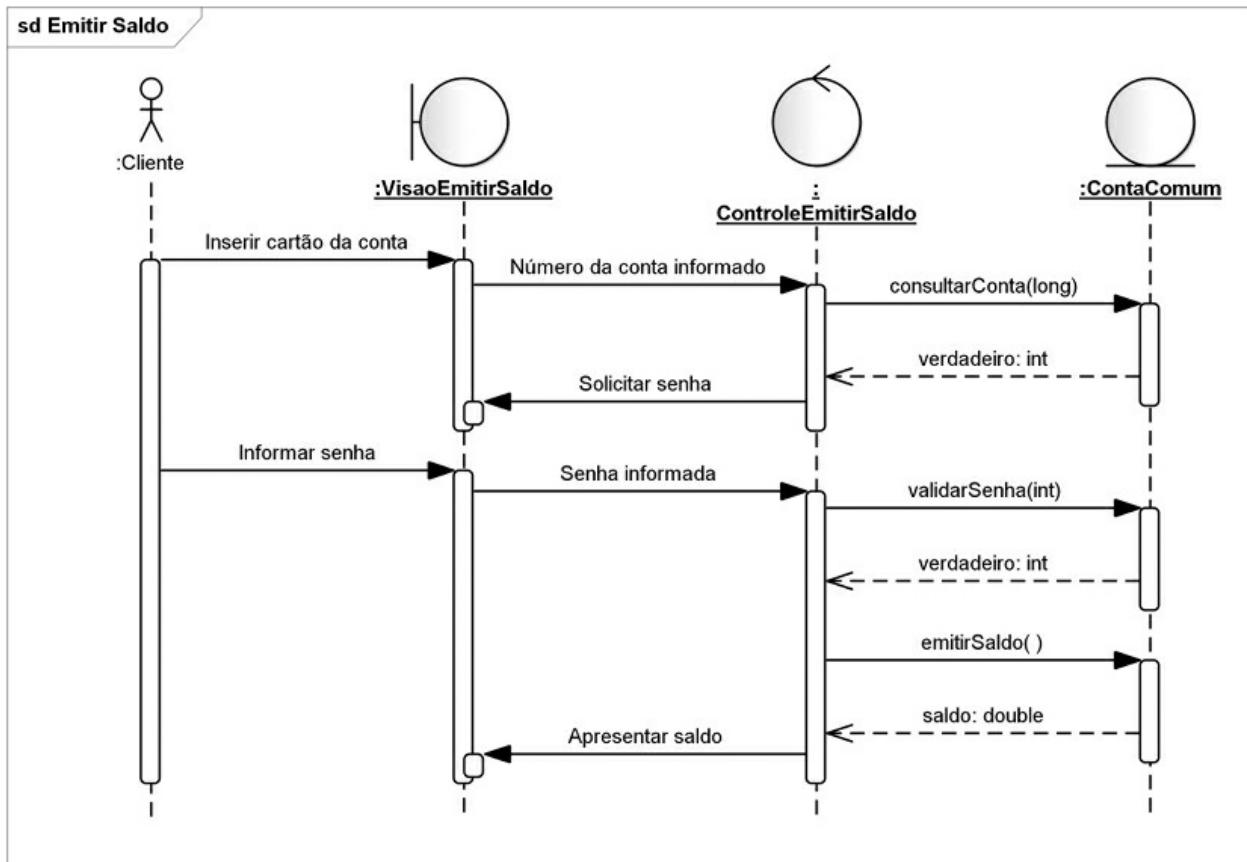
**pkg Camadas do Projeto - Sistema de Controle Bancário**



*Figura 1.4 – Exemplo de Diagrama de Pacotes.*

### 1.4.5 Diagrama de Sequência

O diagrama de sequência é um diagrama comportamental que se preocupa com a ordem temporal em que as mensagens são trocadas entre os objetos envolvidos em um determinado processo. Em geral, baseia-se em um caso de uso definido pelo diagrama de mesmo nome e apoia-se no diagrama de classes para determinar os objetos das classes envolvidas em um processo. Um diagrama de sequência costuma identificar o evento gerador do processo modelado, bem como o ator responsável por esse evento, e determina como o processo deve se desenrolar e ser concluído por meio da chamada de métodos disparados por mensagens enviadas entre os objetos. A figura 1.5 apresenta um exemplo desse diagrama.



*Figura 1.5 – Exemplo de Diagrama de Sequência.*

### 1.4.6 Diagrama de Comunicação

O diagrama de comunicação era conhecido como de colaboração até a versão 1.5 da UML, tendo seu nome modificado para diagrama de comunicação a partir da versão 2.0. Está amplamente associado ao diagrama de sequência: na verdade, um complementa o outro. As informações mostradas no diagrama de comunicação com frequência são praticamente as mesmas apresentadas no de sequência, porém com um enfoque distinto, visto que esse diagrama não se preocupa com a temporalidade do processo, concentrando-se em como os elementos do diagrama estão vinculados e quais mensagens trocam entre si durante o processo. A figura 1.6 apresenta um exemplo de diagrama de comunicação.

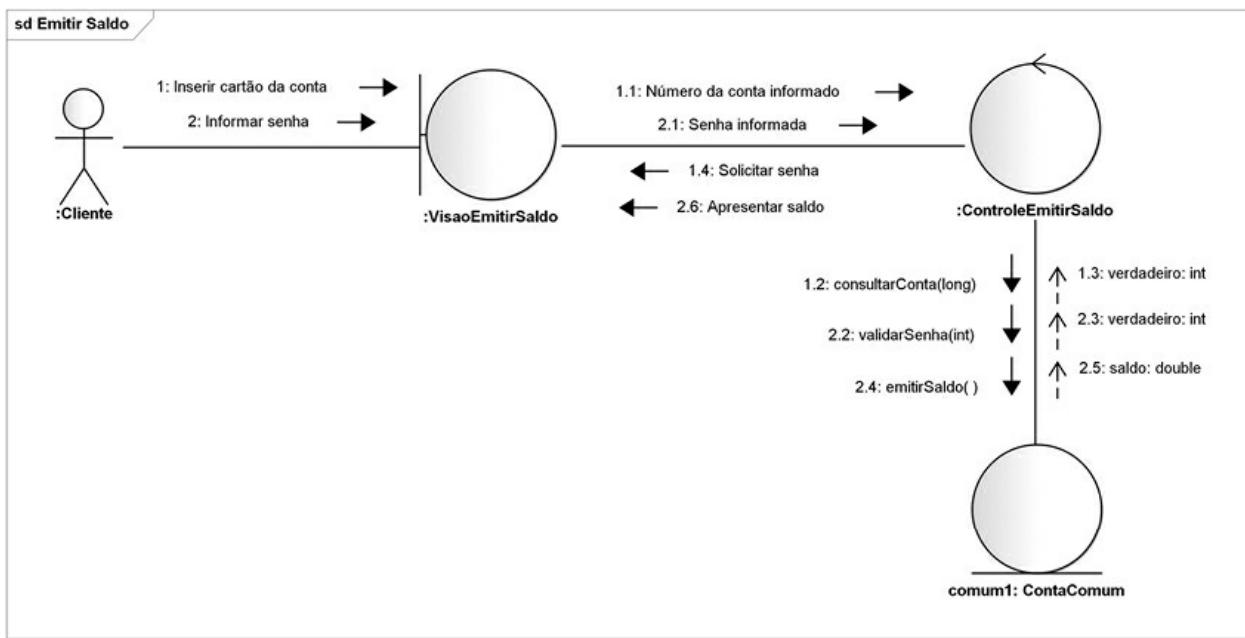
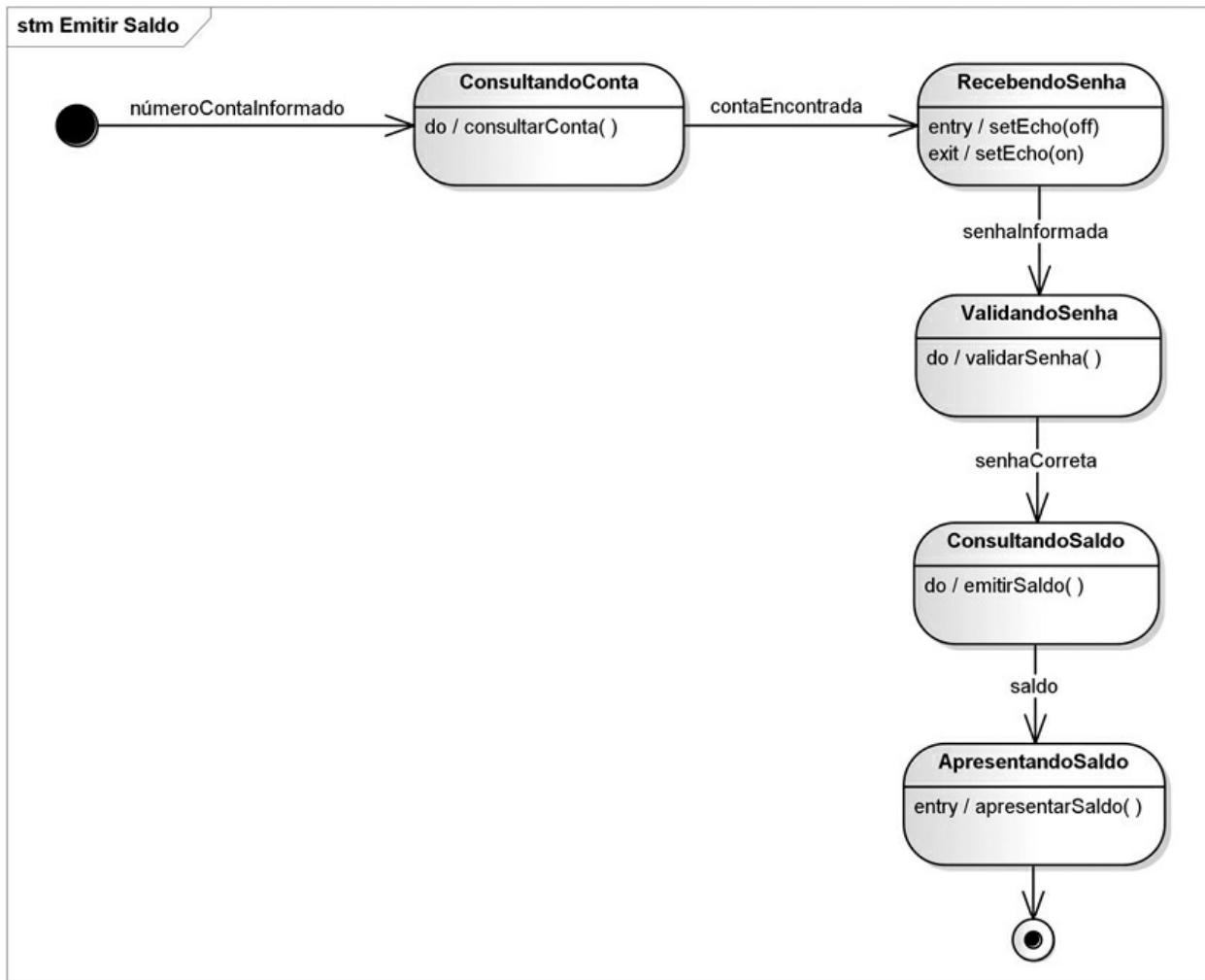


Figura 1.6 – Exemplo de Diagrama de Comunicação.

#### 1.4.7 Diagrama de Máquina de Estados

O diagrama de máquina de estados demonstra o comportamento de um elemento por meio de um conjunto finito de transições de estado. Esse diagrama pode ser utilizado para expressar o comportamento de uma parte do sistema, quando é chamado de máquina de estado comportamental, ou o protocolo de uso de parte de um sistema, quando identifica uma máquina de estados de protocolo. Uma máquina de estados comportamental pode ser usada para especificar o comportamento de vários elementos do modelo. O elemento modelado muitas vezes é uma

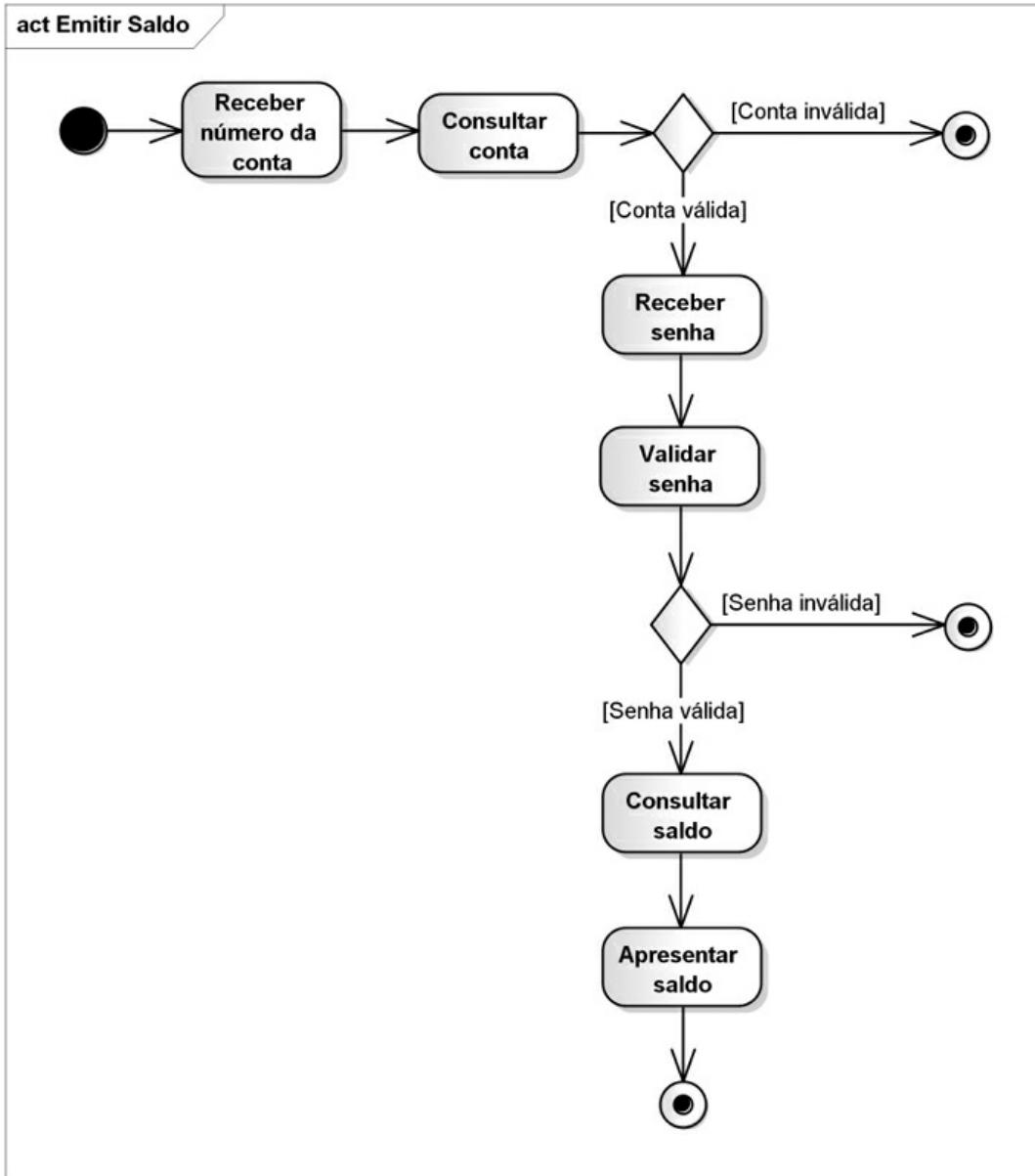
instância de uma classe. No entanto, pode-se usar esse diagrama para modelar o comportamento de um caso de uso, por exemplo. A figura 1.7 apresenta um exemplo de diagrama de máquina de estados.



*Figura 1.7 – Exemplo de Diagrama de Máquina de Estados.*

### 1.4.8 Diagrama de Atividade

O diagrama de atividade preocupa-se em descrever os passos a serem percorridos para a conclusão de uma atividade específica, podendo esta ser representada por um método com certo grau de complexidade, um algoritmo, ou mesmo um processo completo. O diagrama de atividade concentra-se na representação do fluxo de controle e de objetos de uma atividade. A figura 1.8 apresenta um exemplo desse diagrama.



*Figura 1.8 – Exemplo de Diagrama de Atividade.*

#### 1.4.9 Diagrama de Visão Geral de Interação

O diagrama de visão geral de interação é uma variação do diagrama de atividade que fornece uma visão geral dentro de um sistema ou processo de negócio, podendo englobar vários subprocessos. Esse diagrama passou a existir apenas a partir da UML 2. A figura 1.9 apresenta um exemplo do diagrama em questão.

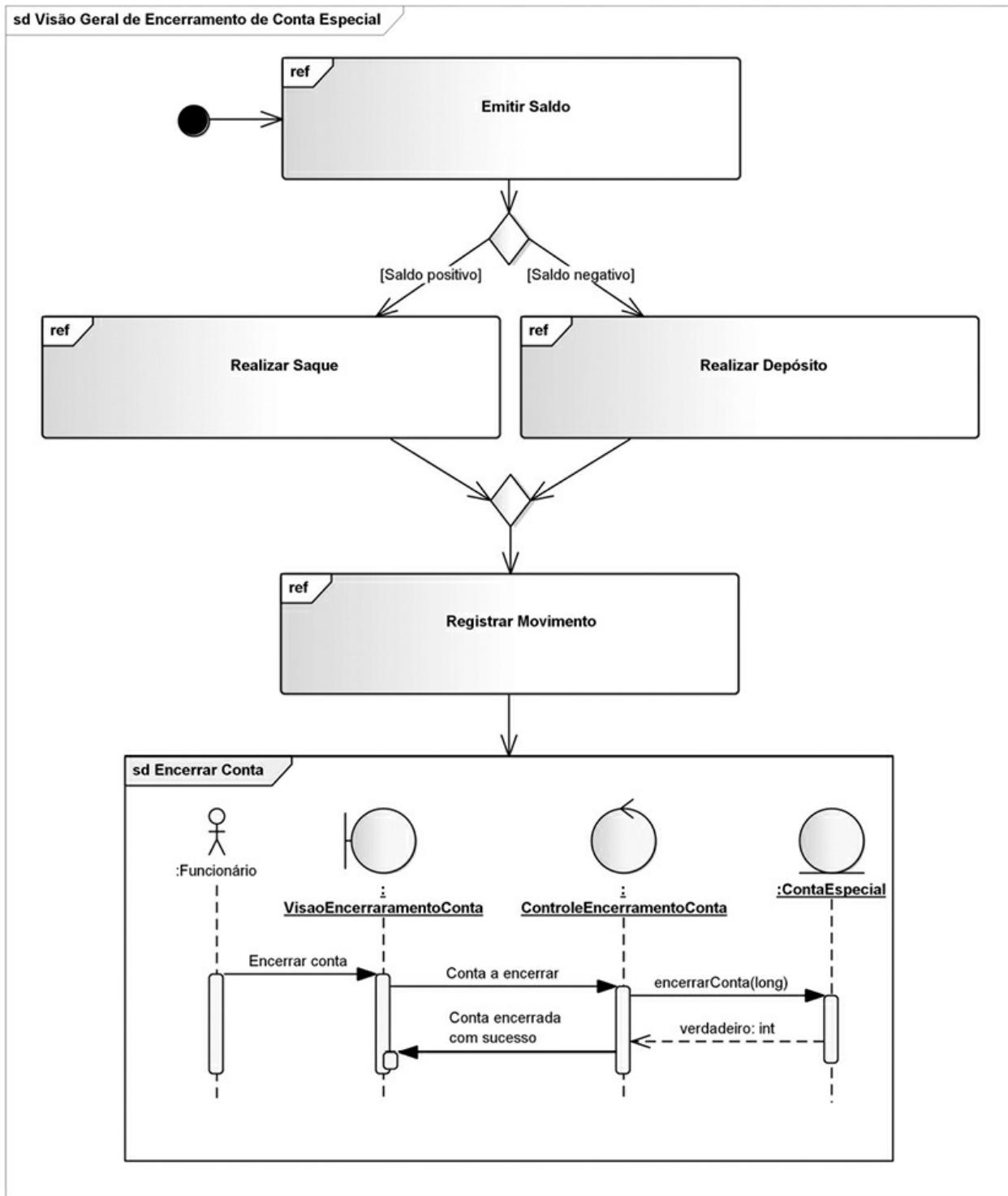


Figura 1.9 – Exemplo de Diagrama de Visão Geral de Interação.

#### 1.4.10 Diagrama de Componentes

O diagrama de componentes, como seu próprio nome indica, identifica os componentes que fazem parte de um sistema, um subsistema ou mesmo os

componentes ou classes internas de um componente individual. Um componente pode representar tanto um componente lógico (um componente de negócio ou de processo) quanto um componente físico, como arquivos contendo código-fonte, arquivos de ajuda (help), bibliotecas, arquivos executáveis etc. A figura 1.10 apresenta um exemplo desse diagrama.

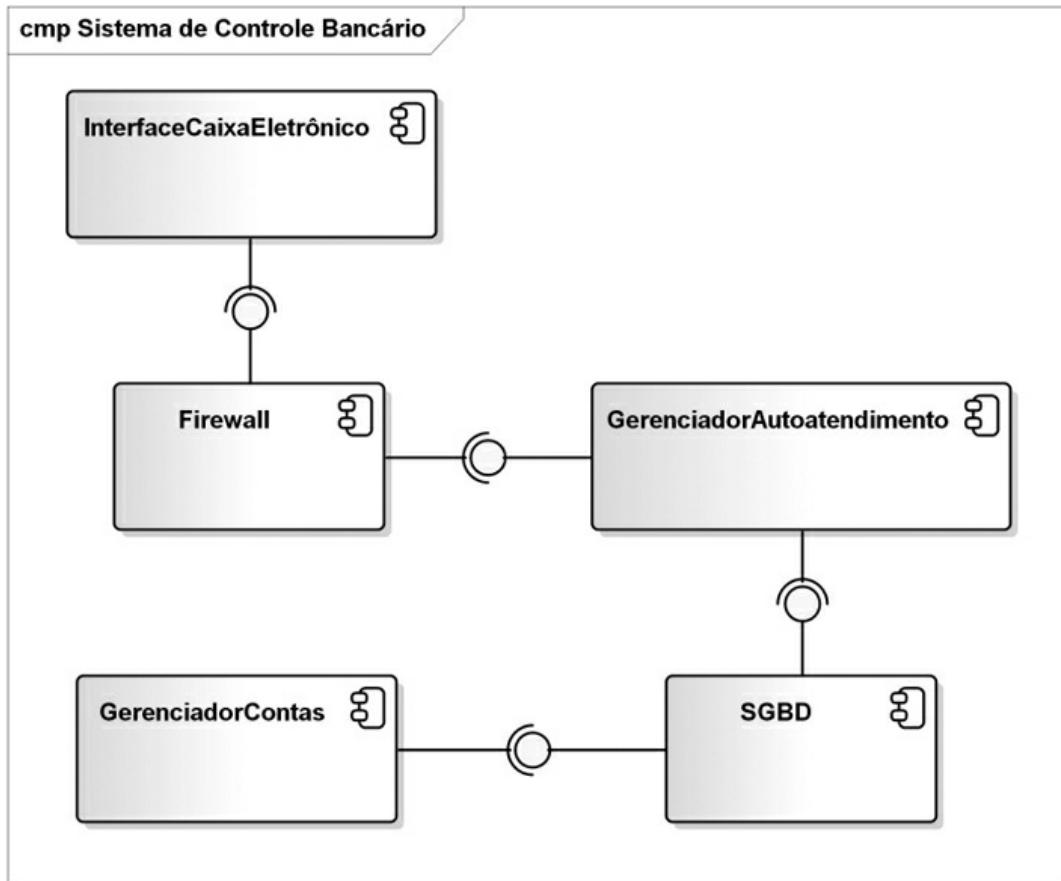


Figura 1.10 – Exemplo de Diagrama de Componentes.

#### 1.4.11 Diagrama de Implantação

O diagrama de implantação determina as necessidades de hardware do sistema, como servidores, estações, topologias e protocolos de comunicação, ou seja, todo o aparato físico sobre o qual o sistema deverá ser executado. Esse diagrama permite demonstrar também como se dará a distribuição dos módulos do sistema, em situações em que estes forem executados em mais de um servidor. A figura 1.11 apresenta um exemplo desse diagrama.

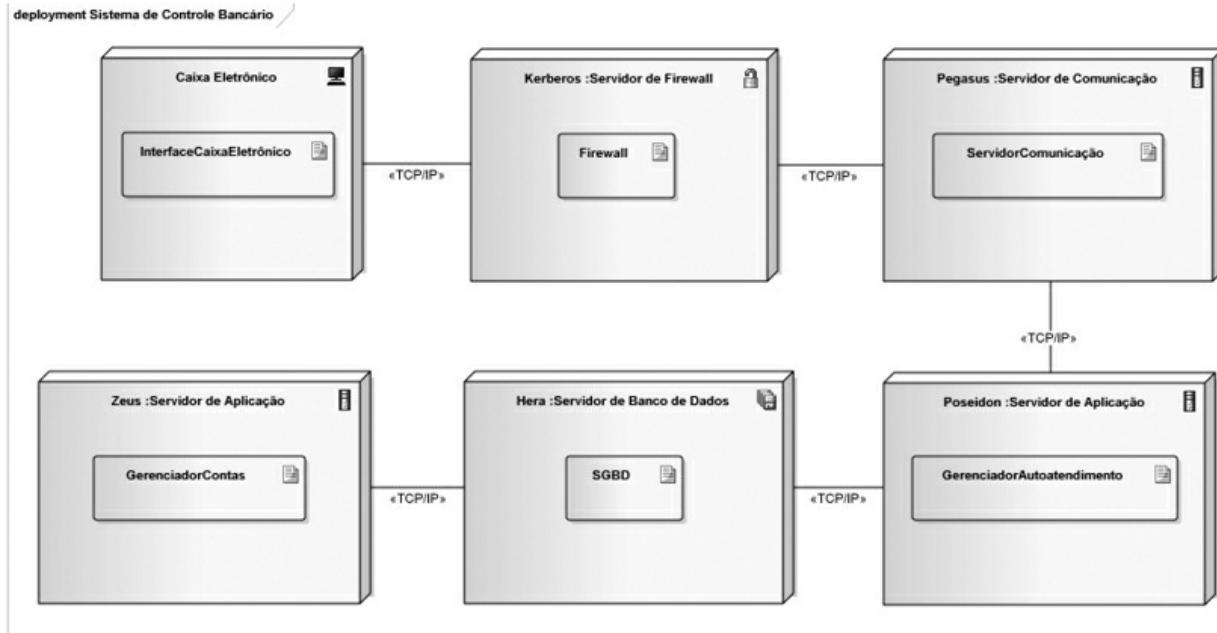
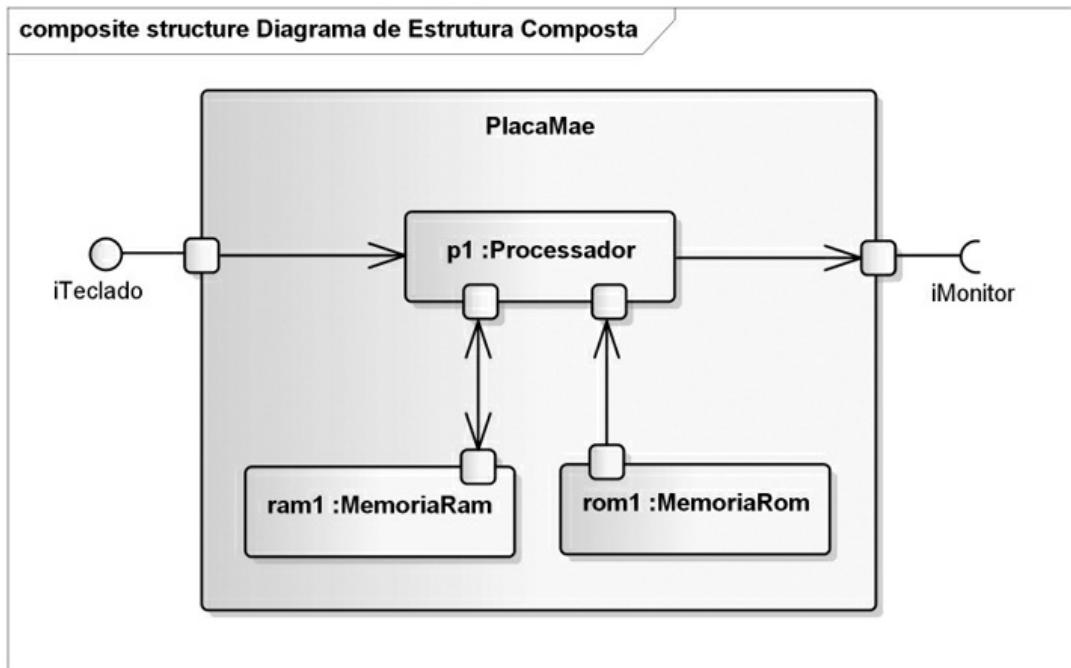


Figura 1.11 – Exemplo de Diagrama de Implantação.

#### 1.4.12 Diagrama de Estrutura Composta

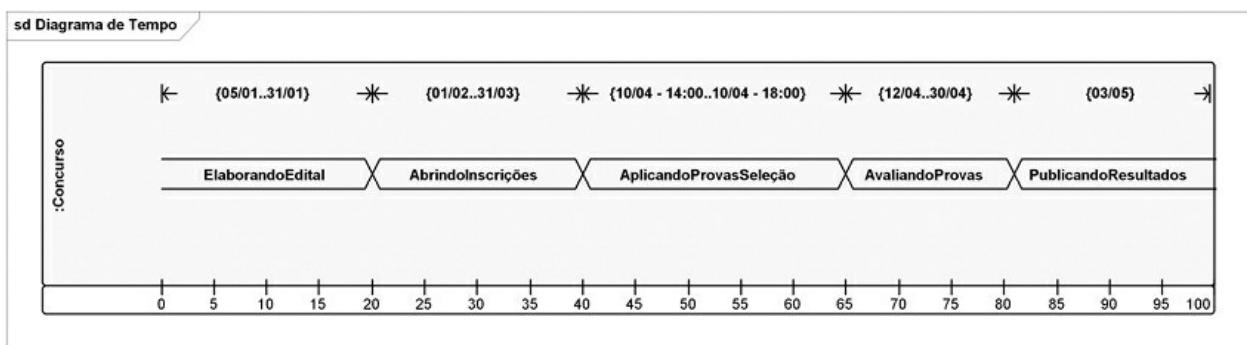
O diagrama de estrutura composta descreve a estrutura interna de um classificador, como uma classe ou componente, detalhando as partes internas que o compõem, como estas se comunicam e colaboram entre si. Também é utilizado para descrever uma colaboração em que um conjunto de instâncias coopera entre si para realizar uma tarefa. A figura 1.12 mostra um exemplo de diagrama de estrutura composta.



*Figura 1.12 – Exemplo de Diagrama de Estrutura Composta.*

### 1.4.13 Diagrama de Tempo ou de Temporização

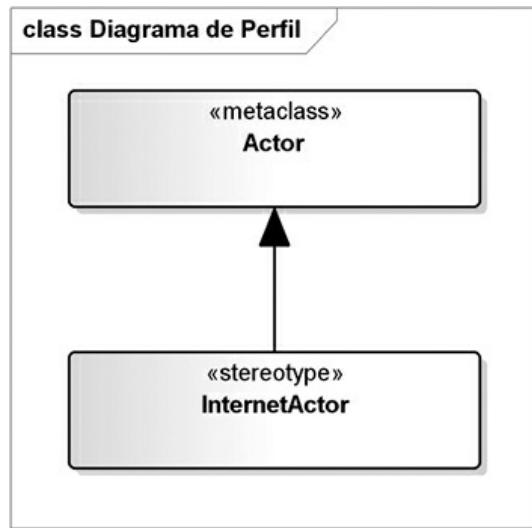
O diagrama de tempo descreve a mudança no estado ou condição de uma instância de uma classe ou o papel que ela assume em um período específico de tempo. É tipicamente utilizado para demonstrar a mudança no estado de um objeto em um tempo exato, em resposta a eventos externos. Pode ser utilizado, por exemplo, na modelagem de sistemas de tempo real; sistemas que utilizem recursos de multimídia/hipermídia, em que o tempo em que um objeto executa algo é muitas vezes importante; ou, ainda, para modelar processos de rede em que o sincronismo entre os eventos é essencial em algumas situações. A figura 1.13 apresenta um exemplo desse diagrama.



*Figura 1.13 – Exemplo de Diagrama de Tempo.*

#### **1.4.14 Diagrama de Perfil**

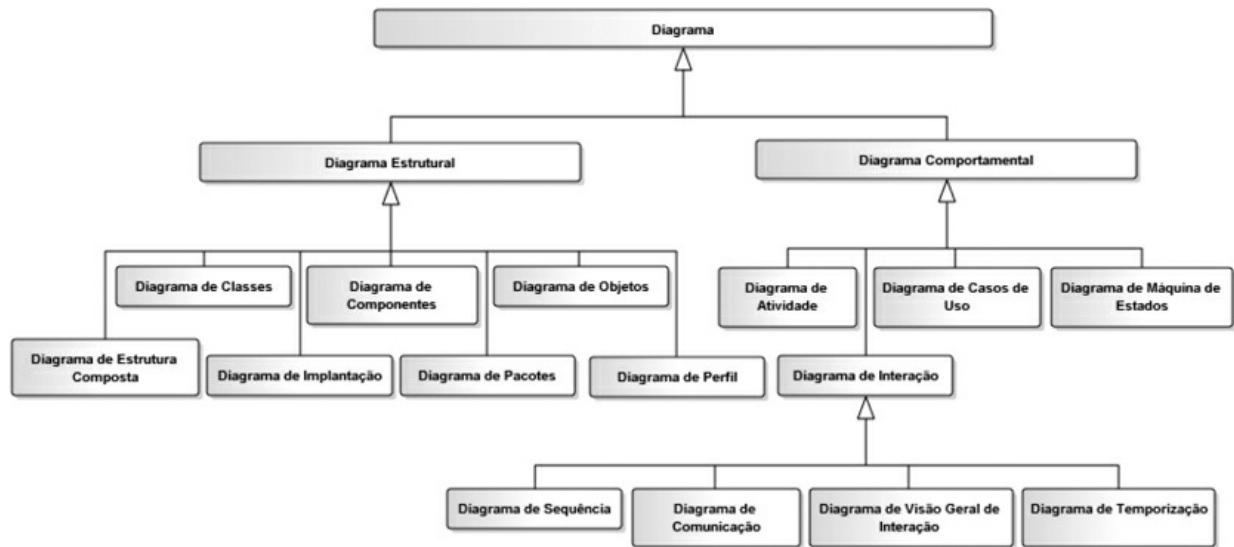
O diagrama de perfil é um tanto mais abstrato que os descritos anteriormente. Esse diagrama permite adaptar a UML a uma plataforma ou domínio ao qual a linguagem UML não foi projetada originalmente e, portanto, não possui recursos para modelar as características particulares da plataforma, tecnologia ou domínio em questão. Sendo assim, por meio da criação de perfis, é possível estender a linguagem, criando-se novas metaclasses e estereótipos que permitam a modelagem desses novos domínios. A figura 1.14 apresenta um exemplo bastante simples desse diagrama.



*Figura 1.14 – Exemplo de Diagrama de Perfil.*

#### **1.4.15 Síntese Geral dos Diagramas**

Os diagramas da UML dividem-se em diagramas estruturais e comportamentais, contendo os últimos ainda uma subdivisão representada pelos diagramas de interação, conforme pode ser verificado na figura 1.15.



*Figura 1.15 – Diagramas da UML.*

Como podemos observar, os diagramas estruturais abrangem os diagramas de classes, de estrutura composta, de objetos, de componentes, de implantação, de pacotes e de perfil, enquanto os comportamentais englobam os de casos de uso, atividade, máquina de estados, sequência, comunicação, visão geral de interação e temporização; os últimos quatro correspondem aos diagramas da subdivisão de diagramas de interação.

## 1.5 Ferramentas CASE Baseadas na Linguagem UML

Ferramentas CASE (Computer-Aided Software Engineering ou Engenharia de Software Auxiliada por Computador) são softwares que, de alguma maneira, colaboram para a execução de uma ou mais atividades realizadas durante o processo de engenharia de software. A maioria das ferramentas CASE atuais (se não todas) suporta a UML, sendo essa, em geral, uma de suas principais características. Entre as diversas ferramentas existentes hoje no mercado, podemos citar:

- **Enterprise Architect** – É uma ferramenta muito fácil de utilizar. Embora não seja livre nem ofereça uma edição para a comunidade, é uma das ferramentas que mais oferecem recursos compatíveis com a UML em sua última versão. Apesar de não dispor de uma edição para a comunidade, a Sparx Systems, a empresa que produz a Enterprise Architect, disponibiliza uma versão trial, que pode ser utilizada por cerca de 60 dias, no site [www.sparxsystems.com.au](http://www.sparxsystems.com.au). Praticamente todos os

exemplos apresentados neste livro foram produzidos por meio dessa ferramenta.

- **Visual Paradigm for UML ou VP-UML** – A ferramenta pode ser encontrada no site [www.visual-paradigm.com](http://www.visual-paradigm.com) e oferece uma edição para a comunidade, ou seja, uma versão da ferramenta que pode ser baixada gratuitamente de sua página. Logicamente, a edição para a comunidade não suporta todos os serviços e opções disponíveis nas suas versões standard ou professional. No entanto, para quem deseja praticar a UML, a edição para a comunidade é uma boa alternativa, apresentando um ambiente amigável e de fácil compreensão.
- **Poseidon for UML** – Esta ferramenta também tem uma edição para a comunidade, apresentando bem menos restrições do que a edição para a comunidade da Visual-Paradigm, mas o ambiente da Poseidon é sensivelmente inferior ao da VP-UML, além de apresentar desempenho um pouco inferior. Uma cópia da Poseidon for UML pode ser adquirida no site [www.gentleware.com](http://www.gentleware.com).
- **Astah** – Esta ferramenta era chamada anteriormente de Jude. É bastante popular e utilizada. Da mesma forma que as ferramentas anteriores, existe uma edição para a comunidade desta ferramenta, além disso sua versão profissional é gratuita para estudantes, bastando que o estudante preencha uma requisição. Também é fornecida uma versão trial da ferramenta profissional, válida por 50 dias. É razoavelmente fácil de usar e suporta muitos dos recursos atuais da UML. Esta ferramenta pode ser adquirida no site [astah.net](http://astah.net).
- **ArgoUML** – Trata-se de uma ferramenta um tanto limitada, e sua usabilidade não é das mais amigáveis e intuitivas. Porém, apresenta uma característica bastante interessante e atrativa: é totalmente livre. O projeto ArgoUML constitui-se em um projeto acadêmico, no qual os códigos-fonte dessa ferramenta podem ser baixados e utilizados até mesmo para o desenvolvimento de ferramentas comerciais, como foi o caso da Poseidon for UML. Os usuários dessa ferramenta podem perceber muitas semelhanças entre ambas, mas a Poseidon tem uma interface muito melhor e é, em geral, superior à ArgoUML. O projeto de código aberto ArgoUML exige apenas que quaisquer empresas que utilizarem seus códigos-fonte como base para uma nova ferramenta

disponibilizem uma edição para a comunidade gratuitamente. Uma cópia da ArgoUML pode ser encontrada no **site** [www.argouml.tigris.org](http://www.argouml.tigris.org).

## CAPÍTULO 2

# Orientação a Objetos

A UML está totalmente inserida no paradigma de orientação a objetos. Por esse motivo, para compreendê-la corretamente, precisamos, antes, compreender os conceitos da orientação a objetos.

## 2.1 Classificação, Abstração e Instanciação

No início da infância, o ser humano aprende e pensa de maneira bastante semelhante à filosofia da orientação a objetos, representando seu conhecimento por meio de abstrações e classificações (na verdade, continuamos fazendo isso mesmo quando adultos, mas desenvolvemos outras técnicas que também utilizamos em paralelo). As crianças aprendem conceitos simples, como pessoa, carro e casa, por exemplo, e, ao fazerem isso, definem classes, ou seja, grupos de objetos, sendo cada um deles um exemplo de um determinado grupo, tendo as mesmas características e comportamentos de qualquer objeto do grupo em questão. A partir desse momento, qualquer coisa que tiver cabeça, tronco e membros torna-se uma pessoa, qualquer construção na qual as pessoas possam entrar passa a ser uma casa e qualquer peça de metal com quatro rodas que se locomova de um lugar para outro transportando pessoas recebe a denominação de carro.

Na verdade, esse processo por si só envolve um grande esforço de abstração, em razão, por exemplo, de os carros apresentarem diferentes formatos, cores e estilos. Uma criança deve se sentir um pouco confusa no começo ao descobrir que o objeto amarelo e o objeto vermelho têm, ambos, a mesma classificação: carro. A partir desse momento, precisa abstrair o conceito de carro para chegar à conclusão de que “carro” é um termo geral que se refere a muitos objetos. No entanto, cada um dos objetos-carro tem características semelhantes entre si, por exemplo, todos têm quatro rodas e, no mínimo, duas portas, além de luzes de farol e freio,

bem como vidros frontais e laterais. Além disso, os objetos-carro podem realizar determinadas tarefas, sendo a principal delas transportar pessoas de um lugar para outro.

No momento em que a criança comprehende esse conceito, percebe que “carro” é a denominação de um grupo, ou seja, ela abstraiu uma classe: a classe carro. Assim, sempre que perceber a presença de um objeto com as características já determinadas, concluirá que aquele objeto é um exemplo do grupo carro, ou seja, uma instância da classe carro.

Assim, uma das formas de o ser humano aprender e representar conhecimento é, ao menos no início, orientada a objetos, ou seja, aprendemos por meio de classificações. Aprendemos a classificar praticamente tudo, criando grupos de objetos com características iguais, sendo cada um deles equivalente a uma classe. Sempre que precisamos compreender um conceito novo, criamos uma nova classe para esse conceito, muitas vezes derivando-a de classes mais simples (ou seja, conceitos mais simples), e determinamos que todo objeto com as características dessa classe é um exemplo, uma instância dela. Assim, instanciação constitui-se simplesmente em criar um exemplo de um tipo, um grupo, uma classe.

Quando instanciamos um objeto de uma classe, estamos criando um novo item do conjunto representado por essa classe, com as mesmas características e comportamentos de todos os outros objetos já instanciados. Além disso, depois de abstrair um conceito inicial, costumamos criar subdivisões dentro das classes, isto é, grupos dentro de grupos, o que já caracteriza um novo nível de abstração. Podemos criar subgrupos dentro da classe **Carro**, classificando-os por marca ou modelo, por exemplo, ou dentro da classe **Pessoa**, classificando os novos grupos por profissão, grau de parentesco ou status social.

No entanto, deve-se ter em mente que, apesar de terem os mesmos atributos, os objetos de uma classe não são exatamente iguais, pois cada objeto armazena valores diferentes em seus atributos. Por exemplo, todos os objetos da classe **Carro** possuem o atributo **placa**, mas cada um dos objetos possui um valor diferente para sua placa específica. Da mesma forma, todos os objetos da classe **Pessoa** podem possuir o atributo **nome** ou o atributo **CPF**, mas os valores armazenados nesses atributos variam de

pessoa para pessoa.

## 2.2 Classes de Objetos

Conforme foi dito, uma classe representa uma categoria e os objetos são os membros ou exemplos dessa categoria.

Na UML, uma classe é representada por um retângulo que pode ter até três divisões. A primeira divisão armazena o nome pelo qual a classe é identificada (e essa é a única divisão obrigatória), a segunda enuncia os possíveis atributos pertencentes à classe e a terceira lista as possíveis operações (métodos) que a classe contém. Em geral, uma classe tem atributos e métodos, mas é possível encontrar classes que contenham apenas uma dessas características ou mesmo nenhuma delas, como no caso de classes abstratas. A figura 2.1 apresenta um exemplo de classe.

Nesse caso, identificamos uma classe chamada **Pessoa**. Observe que essa classe tem apenas uma divisão que contém seu nome, pois não é obrigatório representar uma classe totalmente expandida, embora seja mais comum encontrar exemplos assim. Além disso, a classe pode simplesmente não conter atributos nem métodos quando se tratar de classes abstratas. As demais divisões de uma classe serão explicadas nas seções a seguir.



Figura 2.1 – Exemplo de uma classe.

## 2.3 Atributos ou Propriedades

Classes costumam definir atributos, também conhecidos como propriedades. Os atributos representam as características de uma classe, ou seja, as peculiaridades que costumam variar de um objeto para outro, como o nome, o CPF ou a idade em um objeto da classe **Pessoa** ou a placa ou a cor em um objeto da classe **Carro**, e que permitem diferenciar um objeto de outro da mesma classe em razão de tais variações.

Os atributos são apresentados na segunda divisão da classe e contêm, normalmente, duas informações: o nome que identifica o atributo e o tipo de dado que o atributo armazena, como `integer`, `float` ou `String`. Essa última informação não é obrigatória, mas muitas vezes é útil e recomendável.

Na realidade, não é exatamente a classe que contém os atributos, mas, sim, as instâncias, os objetos dessa classe. Não é realmente possível trabalhar com uma classe apenas com suas instâncias. Por exemplo, a classe `Pessoa` não existe realmente: é uma abstração, uma forma de classificar e identificar um grupo de objetos semelhantes. Nunca poderemos trabalhar com a classe `Pessoa` propriamente dita, apenas com suas instâncias, como João, Pedro ou Paulo, que são nomes que identificam três objetos da classe `Pessoa`. Da mesma forma, ninguém pode morar na planta de uma casa. A planta exprime o conceito da classe e, com base nela, constroem-se quantas casas forem necessárias e nessas casas reais é que se pode morar, ou seja, a casa construída com base na planta é um objeto, enquanto a planta da casa é uma classe.

Assim, os objetos têm os atributos relativos à classe à qual pertencem. Tais atributos são as características do objeto, como placa, cor e número de portas, no caso de uma instância da classe carro, ou CPF, nome e idade em uma instância da classe `Pessoa`. Todas as instâncias de uma mesma classe têm exatamente os mesmos atributos, no entanto estes podem assumir valores diversos. Assim, o atributo nome do objeto `pessoa1` pode assumir o valor “João”, enquanto no objeto `pessoa2` o valor do atributo `nome` pode ser “Paulo”. A figura 2.2 demonstra um exemplo de classe com atributos.

O exemplo apresentado na figura 2.2 toma a mesma classe ilustrada na figura 2.1, mas, dessa vez, com duas divisões. A segunda divisão armazena os atributos da classe `Pessoa`, que, nesse caso, são cpf, nome e idade. No exemplo, não foram definidos os tipos dos atributos por ainda não serem necessários.

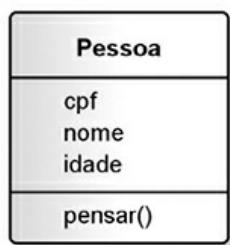


*Figura 2.2 – Exemplo de classe com atributos.*

## 2.4 Operações, Métodos ou Comportamentos

Classes costumam ter métodos, também conhecidos como operações ou comportamentos (a UML usa o termo operação). Um método representa uma atividade que um objeto de uma classe pode executar. Um método pode receber ou não parâmetros (valores utilizados durante a execução do método) e, em geral, tende a retornar valores. Esses valores podem representar o resultado da operação executada ou simplesmente indicar se o processo foi concluído com sucesso ou não.

Assim, um método representa um conjunto de instruções executadas quando o método é chamado. Por exemplo, um objeto da classe **Pessoa** pode executar a atividade de pensar. Grande parte da codificação propriamente dita dos sistemas de informação orientados a objeto está contida nos métodos definidos em suas classes. Os métodos são armazenados na terceira divisão de uma classe. A figura 2.3 apresenta um exemplo de uma classe com métodos.



*Figura 2.3 – Exemplo de classe com métodos.*

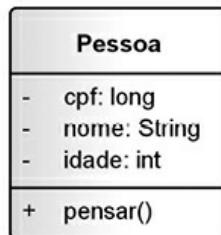
Novamente, tomamos a mesma classe **Pessoa**, ilustrada na figura 2.1, e acrescentamos uma terceira divisão para armazenar o método **pensar()**.

## 2.5 Visibilidade

A visibilidade é utilizada para indicar o nível de acessibilidade de um determinado atributo ou método, sendo representada à esquerda destes. Existem basicamente quatro modos de visibilidade: público, protegido, privado e pacote.

- A visibilidade privada é representada por um símbolo de menos (-) e significa que somente os objetos da classe detentora do atributo ou método poderão enxergá-lo.
- A visibilidade protegida é representada pelo símbolo de sustenido (#) e determina que, além dos objetos da classe detentora do atributo ou método, também os objetos de suas subclasses poderão ter acesso a este.
- A visibilidade pública é representada por um símbolo de mais (+) e determina que o atributo ou método pode ser utilizado por qualquer objeto.
- A visibilidade pacote é representada por um símbolo de til (~) e determina que o atributo ou método é visível por qualquer objeto dentro do pacote. Somente elementos que fazem parte de um pacote podem ter essa visibilidade. Nenhum elemento fora do pacote poderá ter acesso a um atributo ou método com essa visibilidade.

A figura 2.4 apresenta um exemplo de classe com atributos e métodos com sua visibilidade representada à esquerda de seus nomes.



*Figura 2.4 – Exemplo de Visibilidade.*

Como se pode verificar nessa figura, utilizamos novamente o mesmo exemplo de classe anterior, acrescentando visibilidade a seus atributos e métodos. No caso, é possível percebermos que os atributos “cpf”, “nome” e “idade” têm visibilidade privada, pois apresentam o símbolo de menos (-) à esquerda de sua descrição e, portanto, somente as instâncias da classe **Pessoa** propriamente ditas podem enxergar esses atributos. Já o método

`pensar()` tem visibilidade pública, como indica o símbolo de mais (+) acrescentado à sua descrição, o que permite que objetos de outras classes enxerguem o método.

É importante destacar que normalmente os atributos costumam ser privados ou protegidos, enquanto os métodos costumam ser públicos. A declaração de atributos como privados, ou eventualmente protegidos, é altamente recomendada, pois isto garante o encapsulamento dos atributos. Um atributo privado, além de só ser visível por objetos de sua classe, só poderá ser acessado por meio de métodos. Assim, objetos de outras classes não terão conhecimento sobre quais atributos estão contidos na classe em questão e nem poderão acessá-los. Eles saberão da existência dos métodos da classe (quando forem públicos), mas não como o método manipula seus atributos.

Nesse exemplo, pode-se observar que optamos por incluir também os tipos dos atributos. Assim, o atributo `cpf` recebeu o tipo `long`, enquanto `nome` e `idade` receberam os atributos `String` e `int`, respectivamente.

## 2.6 Herança

Como o polimorfismo, que será visto a seguir, a herança é uma das características mais poderosas e importantes da orientação a objetos. Isso se deve ao fato de permitir o reaproveitamento de atributos e métodos, otimizando o tempo de desenvolvimento, além de permitir a diminuição de linhas de código, bem como facilitar futuras manutenções.

A herança na orientação a objetos trabalha com os conceitos de superclasses e subclasses. Uma superclasse, também chamada de classe-mãe, contém classes derivadas dela, chamadas subclasses, também conhecidas como classes-filha. As subclasses, ao serem derivadas de uma superclasse, herdam suas características, ou seja, seus atributos e métodos.

A vantagem do uso da herança é óbvia: ao declararmos uma classe com atributos e métodos específicos e, depois disso, derivarmos uma subclass da classe já criada, não precisamos redeclarar os atributos e métodos previamente definidos: a subclass herda-os automaticamente, permitindo reutilização do código já pronto. Assim, só precisamos nos preocupar em declarar os atributos ou métodos exclusivos da subclass, o que torna muito mais ágil o processo de desenvolvimento, além de facilitar

igualmente futuras manutenções, sendo necessário apenas alterar o método da superclasse para que todas as subclasses estejam também atualizadas imediatamente.

Além disso, a herança permite trabalhar com especializações. Podemos criar classes gerais, com características compartilhadas por muitas classes, mas que tenham pequenas diferenças entre si. Assim, criamos uma classe geral com as características comuns a todas as classes e diversas subclasses a partir dela, detalhando apenas os atributos ou métodos exclusivos destas. Uma subclass pode se tornar uma superclasse a qualquer momento, bastando para tanto que se derive uma subclass dela. A figura 2.5 apresenta um exemplo de herança.

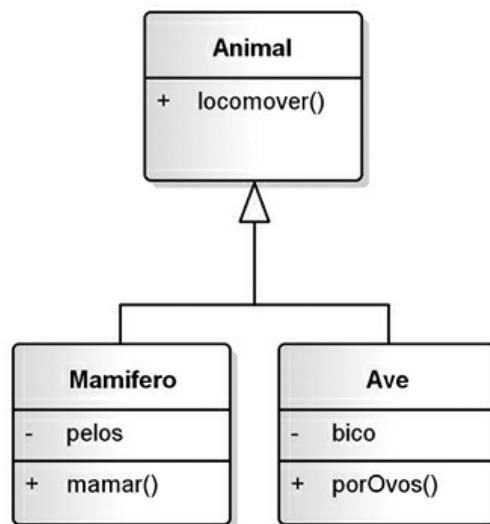
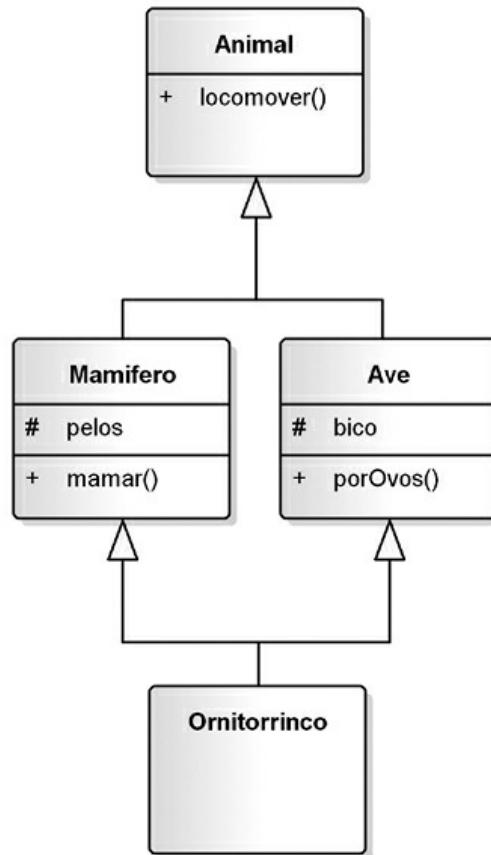


Figura 2.5 – Exemplo de Herança.

Ao observarmos a figura 2.5, percebemos a existência de uma classe geral chamada **Animal** que tem um método chamado **locomover()**. Assim, qualquer instância da classe **Animal** pode se locomover de um lugar para outro. A partir da classe **Animal**, derivamos duas subclasses, **Mamifero** e **Ave**. A classe **Mamifero** tem como atributo a existência de pelos e como método a capacidade de mamar, enquanto a classe **Ave** tem como atributo a existência de um bico e como método a capacidade de pôr ovos, mas ambas têm a capacidade de se locomover, herdada da superclasse **Animal**.

## 2.6.1 Herança Múltipla

Basicamente, a herança múltipla ocorre quando uma subclasse herda características de duas ou mais superclasses. No caso, uma subclasse pode herdar atributos e métodos de diversas superclasses. No entanto, não são todas as linguagens de programação orientadas a objeto que fornecem esse tipo de recurso, sendo este encontrado, por exemplo, na linguagem C++. Veja um exemplo de herança múltipla na figura 2.6.



*Figura 2.6 – Exemplo de Herança Múltipla.*

No exemplo da figura 2.6, aproveitamos o exemplo da figura 2.5, acrescentando uma nova subclasse derivada das classes **Mamifero** e **Ave**, a classe **Ornitorrinco**. Assim, qualquer instância da classe **Ornitorrinco** terá pelos e poderá mamar, peculiaridades herdadas da classe **Mamifero**, e também terá bico e poderá pôr ovos, peculiaridades da classe **Ave**. Observe que a visibilidade dos atributos **pelos** e **bico** é protegida, de maneira que objetos da classe **Ornitorrinco** possam enxergá-los.

## 2.7 Polimorfismo

O conceito de polimorfismo está associado à herança. O polimorfismo trabalha com a redeclaração de métodos previamente herdados por uma classe. Esses métodos, embora semelhantes, diferem de alguma forma da implementação utilizada na superclasse, sendo necessário, portanto, reimplementá-los na subclasse.

Porém, para evitar ter de modificar o código-fonte, inserindo uma chamada a um método com um nome diferente, redeclara-se o método com o mesmo nome declarado na superclasse. Assim, podem existir dois ou mais métodos com a mesma nomenclatura, diferenciando-se na maneira como foram implementados, sendo o sistema responsável por verificar se a classe da instância em questão contém o método declarado nela própria ou se o herda de uma superclasse. A figura 2.7 ilustra um exemplo de polimorfismo.

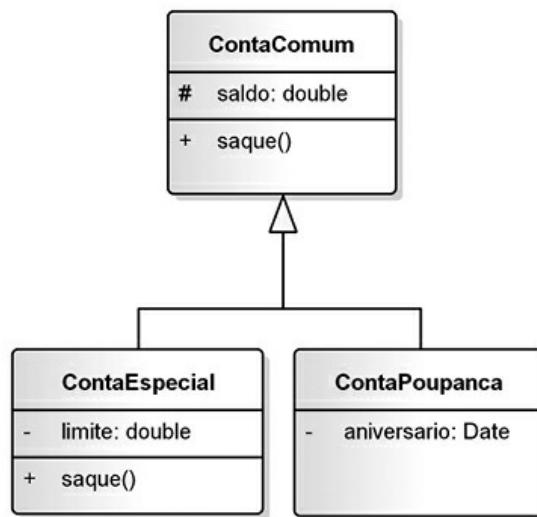


Figura 2.7 – Exemplo de Polimorfismo.

No exemplo apresentado na figura 2.7, utilizamos uma classe geral chamada **ContaComum**. Essa classe tem um atributo chamado “saldo” (com visibilidade protegida para que possa ser acessado pelas subclasses), o qual contém o valor total depositado em uma determinada instância da classe, e um método chamado **saque**. O método **saque** da classe **ContaComum** simplesmente diminui o valor a ser debitado do saldo da conta e, se este não tiver o valor suficiente, a operação deverá ser recusada.

A partir da classe **ContaComum** derivamos uma nova classe chamada **ContaEspecial** que tem um atributo próprio, além dos herdados, chamado “limite”. Esse atributo define o valor extra que pode ser sacado além do valor contido no saldo da conta. Por esse motivo, a classe **ContaEspecial** apresenta uma redefinição do método **saque**, porque o algoritmo do método **saque** da classe **ContaEspecial** não é idêntico ao do método **saque** declarado na classe **ContaComum**, já que é necessário incluir o limite da conta no teste para determinar se o cliente pode retirar ou não o valor solicitado. Porém, o nome do método permanece o mesmo: apenas no momento de executar o método o sistema deverá verificar se a instância que chamou o método pertence à classe **ContaEspecial** ou à **ContaComum**, executando o método definido na classe em questão.

Em uma situação em que existam diversas subclasses que herdem um método de uma superclasse, se uma delas redeclarar esse método, ele só se comportará de maneira diferente nos objetos da classe que o modificou, permanecendo igual à forma como foi implementado na superclasse para os objetos de todas as demais classes. Métodos que são redeclarados em subclasses e apresentam um comportamento diferente do método de mesmo nome contido na superclasse são chamados de métodos polimórficos.

## CAPÍTULO 3

# Diagrama de Casos de Uso

O diagrama de casos de uso procura possibilitar a compreensão do comportamento externo do sistema (em termos de funcionalidades oferecidas por ele) por qualquer pessoa com algum conhecimento sobre o problema enfocado, tentando apresentar o sistema por intermédio de uma perspectiva dos usuários. Esse diagrama costuma ser utilizado, sobretudo, no início da modelagem do sistema, principalmente nas etapas de elicitação e análise de requisitos, embora venha a ser consultado e possivelmente modificado durante todo o processo de engenharia e sirva de base para a modelagem de outros diagramas.

Esse diagrama tem por objetivo apresentar uma visão externa geral das funcionalidades que o sistema deverá oferecer aos usuários, sem se preocupar em profundidade com a questão de como tais funcionalidades serão implementadas. O diagrama de casos de uso é de grande auxílio para a identificação e compreensão dos requisitos funcionais do sistema, ajudando a especificar, visualizar e documentar as funções e serviços do software desejados pelos clientes e stakeholders (usuários com conhecimento sobre como a informação é gerida e modificada em determinados setores da empresa e que possuem interesse no software). O diagrama de casos de uso tenta identificar os tipos de usuários que interagirão com o sistema, quais papéis eles assumirão e quais funções um usuário específico poderá requisitar.

De acordo com diversos autores, casos de uso são de interesse especial na engenharia de requisitos, uma vez que se mostraram valiosos para eliciar, documentar e analisar requisitos, sobretudo os funcionais, ou seja, que identificam as funções que devem ser fornecidas pelo software. Além disso, casos de uso também permitem a rastreabilidade de requisitos, ou seja, qual a origem de um requisito e por que foi considerado importante nas fases de projeto, implementação e verificação e validação.

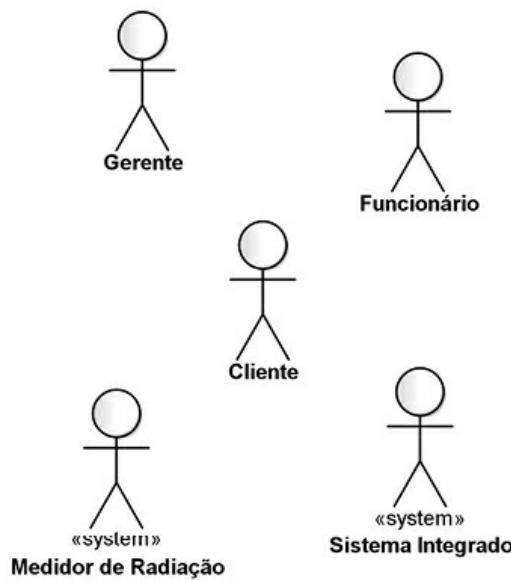
Por utilizar uma linguagem menos formal e apresentar uma visão geral do comportamento do sistema a ser desenvolvido, o diagrama de casos de uso pode e deve ser apresentado durante as reuniões iniciais com os stakeholders para ilustrar as funcionalidades e o comportamento do sistema de informação, facilitar a compreensão dos stakeholders do que se pretende desenvolver e auxiliar na identificação de possíveis falhas de especificação, verificando se os requisitos do sistema foram bem compreendidos. É bastante útil e recomendável que um diagrama de casos de uso seja apresentado aos clientes com um protótipo, o que permitirá que um complemente o outro.

### 3.1 Atores

O diagrama de casos de uso concentra-se em dois itens principais: atores e casos de uso. Os atores costumam representar os papéis desempenhados pelos diversos usuários que poderão utilizar, de alguma maneira, os serviços e funções do sistema. Eventualmente, um ator pode representar algum hardware especial ou mesmo outro software que interaja com o sistema, como no caso de um sistema integrado, por exemplo.

Assim, um ator pode ser qualquer elemento externo que interaja com o software, porém, na maioria das vezes, um ator representará uma pessoa que utilizará o sistema. Esse elemento recebe o nome de ator porque um usuário pode representar mais de um papel no sistema. Por exemplo, um usuário pode se logar em um software com um nível de permissões menor, sendo capaz de utilizar poucas funções do sistema, ou pode se logar com um nível de permissão mais alto e utilizar mais recursos, ou pode ainda se logar como administrador e ter acesso a todas ou à maioria das funcionalidades do sistema. Em cada situação, o papel representado pelo usuário será diferente, por isso o ator não representa um usuário propriamente dito, mas sim um papel que pode vir a ser desempenhado por um ou mais usuários.

Os atores são representados por símbolos de “bonecos magros”, contendo uma breve descrição logo abaixo de seu símbolo que identifica o papel que o ator em questão assume dentro do diagrama. A figura 3.1 apresenta alguns exemplos de atores.



*Figura 3.1 – Exemplos de Atores.*

Nesse exemplo, os atores **Gerente**, **Funcionário** e **Cliente** representam usuários normais, enquanto o ator **Medidor de Radiação** representa um hardware externo que envia informações para o sistema. Já o ator **Sistema Integrado** representa um software que interage de alguma forma com o sistema.

Observe que os atores que representam o hardware externo e o sistema integrado possuem também a palavra “**system**” entre sinais de ‘<’ e ‘>’. Isto é chamado estereótipo e serve para destacar um componente ou associação, atribuindo-lhe características especiais em relação a seus iguais. Nesse caso específico, o estereótipo “**<<system>>**” serve para tornar explícito que os atores em questão referem-se a atores não humanos, ou seja, sistemas de software ou hardware. A utilização desse estereótipo nesse tipo de ator não é obrigatória, apenas recomendada para destacar a sua característica especial que o diferencia dos atores mais comuns. Voltaremos a falar sobre estereótipos com mais detalhe ao longo do livro.

No entanto, é preciso salientar que o uso de atores para representar hardware ou software não é comum, sendo só utilizado em situações em que o hardware ou software desempenha papéis especiais. Neste exemplo, um medidor de radiação desempenha uma função que dificilmente poderia ser executada por um ser humano normal e, ao desempenhá-la, transmite informações importantes para o sistema e executa tarefas ordenadas por

ele.

### **3.2 Como Identificar os Atores?**

Para determinar quais atores farão parte de um modelo de casos de uso, deve-se procurar identificar as entidades externas que interagirão com o sistema, tanto usuários humanos, como também, eventualmente, softwares e/ou hardwares especiais. Também se deve procurar agrupar usuários com características semelhantes, que utilizarão as mesmas funcionalidades no sistema e possuirão os mesmos níveis de permissão, identificando-os como um ator único.

Algumas perguntas podem ser úteis para identificar candidatos a atores, como:

- Que tipos de usuários poderão utilizar o sistema?
- Quais usuários estão interessados ou utilizarão quais funcionalidades e serviços do software?
- Quem fornecerá informações ao sistema?
- Quem utilizará as informações do sistema?
- Quem poderá alterar ou mesmo excluir informações do sistema?
- Existe algum outro software que interagirá com o sistema?
- Existe algum hardware especial (como um robô, por exemplo) que interagirá com o software?

Os candidatos a atores devem ser listados e deve-se tentar atribuir responsabilidades e objetivos a cada um deles, ou seja, metas que cada ator poderia desejar atingir ao utilizar o software. Atores para os quais não é possível atribuir um objetivo facilmente serão atores verdadeiros e deverão ser eliminados.

### **3.3 Casos de Uso**

Os casos de uso são utilizados para capturar os requisitos funcionais do sistema, ou seja, referem-se a serviços, tarefas ou funcionalidades identificados como necessários ao software e que podem ser utilizados de alguma maneira pelos atores que interagem com o sistema. Assim, casos de uso expressam e documentam os comportamentos pretendidos para as

funções do software.

Os casos de uso são representados por elipses contendo dentro de si um texto que descreve a que funcionalidade o caso de uso se refere. Na verdade, não existe um limite determinado para o texto que possa estar contido dentro da elipse, mas, em geral, a descrição de um caso de uso costuma ser bastante sucinta. O texto contido em um caso de uso costuma iniciar com um verbo denotando a ação que será realizada quando de sua execução. A figura 3.2 apresenta um exemplo de caso de uso, representando a abertura de uma conta-corrente.



*Figura 3.2 – Exemplo de Caso de Uso.*

Casos de uso podem ser classificados em casos de uso primários ou secundários. Um caso de uso é considerado primário quando se refere a um processo importante, que enfoca um dos requisitos funcionais do software, como realizar um saque ou emitir um extrato em um sistema de controle bancário. Já um caso de uso secundário se refere a um processo periférico, como a manutenção de um cadastro ou a emissão de um relatório simples. Em situações em que há um grande número de casos de uso, é recomendável limitar a representação de casos de uso secundários, representando-os de maneira geral ou mesmo não os representando de forma alguma, para evitar poluir demais o diagrama.

Os casos de uso costumam ser documentados. A documentação de um caso de uso fornece instruções em linhas gerais de como será seu funcionamento, quais atores poderão utilizá-lo, quais atividades deverão ser executadas pelos atores e pelo sistema, qual evento forçará sua execução e quais suas possíveis restrições, entre outras informações.

Normalmente, as ações contidas na documentação de um caso de uso são descritas em termos gerais, embora nada impeça que o engenheiro de software insira detalhes mais técnicos, até mesmo de implementação na documentação, o que não é recomendado. A documentação deve

representar o comportamento de um caso de uso da maneira o mais completa possível, mas sem detalhes técnicos. Um detalhamento maior será acrescido por outros diagramas em fases posteriores.

### 3.4 Documentação de Casos de Uso

A documentação de um caso de uso costuma descrever, por meio de uma linguagem bastante simples, informações como a função em linhas gerais do caso de uso, quais atores interagem com ele, quais etapas devem ser executadas pelo ator e pelo sistema para que o caso de uso execute sua função, quais parâmetros devem ser fornecidos e quais restrições e validações o caso de uso deve possuir.

Não existe um formato específico de documentação para casos de uso definido pela UML propriamente dita, porém há formatos propostos em diversas literaturas técnicas. O diagrama de casos de uso, todavia, é flexível à forma como se deseja documentá-lo, permitindo que se documente um caso de uso da maneira que se considerar melhor.

Os casos de uso podem também ser documentados por meio de outros diagramas, como o de sequência, de máquina de estados ou de atividade, como será demonstrado em seus respectivos capítulos. A tabela 3.1 apresenta uma sugestão de como documentar o caso de uso representado na figura 3.2.

*Tabela 3.1 – Documentação do Caso de Uso Abertura de Conta*

Nome do Caso de Uso	UC01 – Abrir Conta
Caso de Uso Geral	
Ator Principal	Funcionário
Atores Secundários	Cliente
Resumo	Esse caso de uso descreve as etapas percorridas por um cliente, intermediado por um funcionário, para abrir uma conta-corrente
Pré-condições	O pedido de abertura precisa ter sido previamente aprovado
Pós-condições	É necessário realizar um depósito inicial
<b>Cenário Principal</b>	
Ações do Ator	Ações do Sistema
2. O funcionário informa o CPF ou CNPJ do cliente e consulta seu registro	

	3. Consultar cliente por seu CPF ou CNPJ
4. O cliente informa a senha da conta	5. Abrir conta
6. O cliente fornece um valor a ser depositado	7. Executar caso de uso “Realizar Depósito” para registrar o depósito do cliente 8. Emitir cartão da conta
Restrições/Validações	1. Para abrir uma conta-corrente, é preciso ser maior de idade 2. O valor mínimo de depósito é R\$ 5,00 3. O cliente precisa fornecer algum comprovante de residência
<b>Cenário Alternativo – Manutenção do Cadastro do Cliente</b>	
Ações do Ator	Ações do Sistema
	1. Executar o Caso de Uso “Gerenciar Clientes”, para registrar um novo cliente ou atualizar o cadastro do cliente consultado
<b>Cenário de Exceção – Cliente menor de idade</b>	
Ações do Ator	Ações do Sistema
	1. Comunicar ao cliente que ele não tem a idade mínima para possuir uma conta- corrente 2. Recusar o pedido

Ao seguir o modelo apresentado na tabela 3.1, em primeiro lugar, deve-se fornecer uma descrição (para identificá-lo) para o caso de uso que está sendo documentado, em geral a mesma descrição apresentada externamente no caso de uso. É comum que os casos de uso recebam também um código como UC01, por exemplo.

A informação seguinte talvez aparente ser um pouco mais complexa: o que significa “Caso de Uso Geral”? Como será visto nas próximas seções, pode haver casos de uso gerais e casos de uso especializados, que herdam as características dos casos de uso gerais. Assim, no item caso de uso geral, deve-se informar, quando existir, o nome do caso de uso geral a partir do qual o caso de uso atual foi derivado. Nesse exemplo, não existe um caso de uso geral e, por isso, o campo foi deixado em branco (poderia ter sido completamente eliminado nessa situação). Porém, se o caso de uso em questão representasse a abertura de uma conta especial derivada do caso de uso “Abrir Conta Comum”, o campo caso de uso geral armazenaria o texto “Abrir Conta Comum”, indicando que o caso de uso documentado é uma especialização do caso de uso “Abrir Conta Comum”.

O campo ator principal identifica o ator que mais interage com o caso de uso. Já os atores secundários são aqueles que interagem em um nível menor com o caso de uso. Não é obrigatório declarar um ator secundário, visto que pode existir ou não, e pode não ser importante declará-lo, dependendo da importância de sua participação no processo.

Nesse exemplo, foram identificados dois atores, **Funcionário** e **Cliente**. O ator principal é o **Funcionário**, porque é ele quem interage com a funcionalidade. Já o **Cliente** foi identificado como um ator secundário, porque, embora seja o mais interessado no processo e forneça a maioria das informações, não interage realmente com a funcionalidade. O campo seguinte da documentação desse caso de uso apresenta um breve resumo explicando seu objetivo. Em seguida, são descritas as possíveis pré-condições para que o caso de uso seja executado ou concluído e as possíveis pós-condições, ou seja, tarefas que devem ser realizadas depois que as etapas do caso de uso tiverem sido concluídas. Nesse exemplo, para que a abertura de conta seja realizada, é preciso primeiro, como pré-condição, que o pedido de abertura do cliente seja aprovado e, após a abertura da conta, é necessário depositar algum valor nela, o que caracteriza uma pós-condição.

Em seguida, passa-se ao cenário (também chamado fluxo) principal do caso de uso, que apresenta as ações que devem normalmente ser realizadas quando o serviço representado pelo caso de uso for solicitado. As ações são divididas em ações realizadas pelo ator que interage com o sistema e em ações executadas pelo próprio sistema, numeradas em uma ordem sequencial. Observe que nesse exemplo estabelecemos como primeiro passo do processo a solicitação do cliente ao funcionário para que uma conta seja aberta. Na verdade, esse primeiro passo poderia ser considerado desnecessário e o processo representado pelo caso de uso seria iniciado realmente após essa solicitação. Optamos por inserir esse passo inicial apenas para ilustrar a conversação inicial entre os dois atores interessados no processo em questão.

A documentação de um caso de uso pode também conter cenários (ou fluxos) alternativos. Cenários alternativos, como seu nome indica, podem ser executados ou não, dependendo se uma condição for satisfeita. Nesse exemplo, existe um fluxo alternativo que representa uma situação em que

o cliente não possui cadastro ou este precisa ser atualizado, havendo a necessidade de executar os passos de outro caso de uso para realizar essa manutenção. No cenário principal, muitas vezes se representa um “caminho feliz” onde tudo transcorre conforme o esperado. Neste exemplo, no cenário principal, considera-se que o cliente em questão já possuía registro ao se iniciar o processo de abertura de conta.

Podem existir ainda cenários de exceção que determinam ações que devem ser tomadas em situações em que um cenário principal ou alternativo não pode ser concluído, em razão de alguma regra de negócio ter sido transgredida, por exemplo. Nesse exemplo, há um cenário de exceção que trata uma situação em que a regra, que determina que para abrir uma conta é necessário ser maior de idade, foi quebrada.

Finalmente, é apresentada uma lista de possíveis restrições e validações do caso de uso, com o objetivo de tornar o processo mais consistente. Na documentação desse caso de uso, determinou-se que o cliente precisa ser maior de idade para abrir uma conta-corrente, o que constitui uma restrição. Além disso, fixou-se o valor mínimo para depósito em R\$ 5,00, o que precisa ser validado no processo de abertura de conta, ou seja, o sistema não pode aceitar depósitos com valor inferior a esse no momento em que uma conta é aberta. Essas restrições caracterizam as regras do negócio da empresa, ou seja, regras que determinam as condições para que o processo seja executado.

Alguns autores inserem também, eventualmente, no final da documentação, requisitos não funcionais que especificam condições de usabilidade, desempenho, confiabilidade ou segurança, por exemplo.

### **3.5 Como Identificar os Casos de Uso?**

Para identificar os casos de uso que comporão o modelo, é necessário determinar as funcionalidades necessárias ao software, ou seja, todas as funções e serviços que correspondem aos requisitos funcionais declarados pelos stakeholders como necessários ao sistema.

Na verdade, o processo de identificação de casos de uso se confunde bastante com o de identificação dos atores, já que é possível, e muitas vezes necessário, identificar vários casos de uso que serão utilizados pelos atores identificados.

Assim, uma maneira de auxiliar a identificação das funcionalidades é verificar a lista dos atores que compõem o sistema e quais os objetivos de cada ator. Esses objetivos muitas vezes corresponderão a uma ou mais funcionalidades. Também pode ser útil identificar os eventos externos que afetarão o software. Tais eventos muitas vezes acarretam a execução de uma funcionalidade ou influenciam de alguma forma seu processamento.

Outra maneira de identificar casos de uso é verificar a lista de requisitos funcionais estabelecidos nas conversações com os stakeholders e se cada um deles possui um (ou até mais) caso de uso especificando seu comportamento e, obviamente, qual ator ou atores pode(m) utilizá-lo.

Mas como determinar se os casos de uso foram identificados corretamente? Como vimos na seção anterior, um caso de uso costuma ter uma série de passos ou etapas. Dessa forma, uma maneira de determinar se uma funcionalidade é real, ou seja, se realmente corresponde a um caso de uso, é verificar quais ações seriam realizadas quando o caso de uso fosse executado. Se não for possível identificar essas etapas, provavelmente este não será um caso de uso real.

Da mesma forma, se o número de passos atribuídos a um caso de uso for muito pequeno, deve-se verificar se esse caso de uso não deveria ser englobado por outro, acrescentando seus passos às etapas dele ou mesmo se tornando um cenário alternativo de outro caso de uso. É um pouco comum que iniciantes na criação de modelos de casos de uso confundam cenários alternativos ou mesmo etapas individuais com casos de uso completos, quando, na verdade, seriam apenas uma parte do processo de um caso de uso maior.

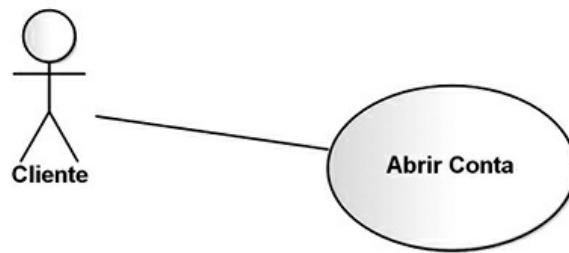
## 3.6 Associações

As associações representam interações ou relacionamentos entre os atores e os casos de uso que fazem parte do diagrama ou os relacionamentos entre os casos de uso e outros casos de uso. Os relacionamentos entre casos de uso recebem nomes especiais, como inclusão, extensão e generalização. Todos esses relacionamentos serão examinados no decorrer das próximas seções.

Uma associação entre um ator e um caso de uso demonstra que o ator utiliza, de alguma maneira, a funcionalidade do sistema representada pelo

caso de uso em questão, seja requisitando a execução dessa função, seja recebendo o resultado produzido por ela a pedido de outro ator.

A associação entre um ator e um caso de uso é representada por uma linha ligando o ator ao caso de uso, podendo ocorrer que as extremidades da linha contenham setas, indicando o sentido em que as informações trafegam, ou seja, se estas são fornecidas pelo ator ao caso de uso, se são transmitidas pelo caso de uso ao ator ou ambos (nesse último caso, a linha não tem setas, significando que as informações são transmitidas nas duas direções). As setas também servem para indicar quem inicia a comunicação. Além disso, uma associação pode ter uma descrição própria quando é necessário esclarecer a natureza da informação que está sendo transmitida ou dar um nome a esta, se isso for necessário. A figura 3.3 representa uma associação entre um ator e um caso de uso.



*Figura 3.3 – Associação entre um Ator e um Caso de Uso.*

Ao examinarmos o exemplo ilustrado pela figura 3.3, percebemos que o ator **Cliente** utiliza, de alguma forma, a funcionalidade de **Abrir Conta**, representada pelo caso de uso, e que a informação referente a esse processo trafega nas duas direções.

### **3.7 Generalização/Especialização**

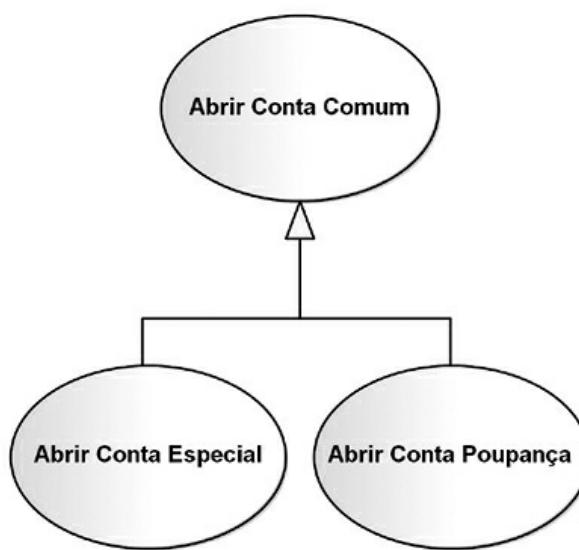
Este relacionamento aplica os princípios de herança da orientação a objetos, permitindo que os passos descritos em um caso de uso sejam herdados por outros casos de uso que especializam o caso de uso original chamado geral.

Assim, é desnecessário colocar a mesma documentação para todos os casos de uso envolvidos, pois toda a estrutura de um caso de uso generalizado é herdada pelos casos de uso especializados. Além disso, os casos de uso especializados herdam também quaisquer possíveis

associações de inclusão ou extensão que o caso de uso geral venha a ter, bem como quaisquer associações com os atores que utilizam o caso de uso geral. A associação de generalização/especialização é representada por uma linha com uma seta mais grossa, que indica qual o caso de uso geral (para o qual a seta aponta) e quais os casos de uso especializados (os que se encontram na outra extremidade da seta, apontando para o caso de uso geral). Um exemplo de generalização/especialização pode ser visto na figura 3.4.

Como se pode perceber no exemplo da figura 3.4, há três opções de abertura de conta muito semelhantes entre si: abertura de conta comum, abertura de conta especial e abertura de conta poupança, cada uma representada por um caso de uso diferente. Os processos de abertura de conta especial e de conta poupança são muito semelhantes ao de abertura de conta comum, mas têm algumas características próprias, o que justifica colocá-las como especializações do caso de uso **Abrir Conta Comum**, detalhando-se as particularidades de cada caso de uso especializado em sua própria documentação.

Nesse exemplo, o caso de uso **Abrir Conta Especial** deve incluir a definição do limite da conta no momento de sua aprovação, enquanto o caso de uso **Abrir Conta Poupança** não apresenta algumas das restrições e validações necessárias à abertura de uma conta-corrente, como a obrigatoriedade de o cliente ser maior de idade, por exemplo.



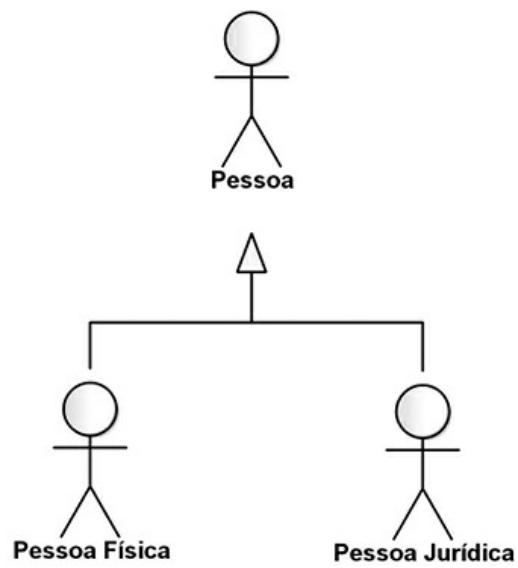
*Figura 3.4 – Generalização/Especialização.*

Em situações em que esse tipo de associação é empregado, a documentação dos casos de uso especializados deve conter o item “Caso de Uso Geral”, onde será especificado a partir de qual caso de uso geral eles foram especializados.

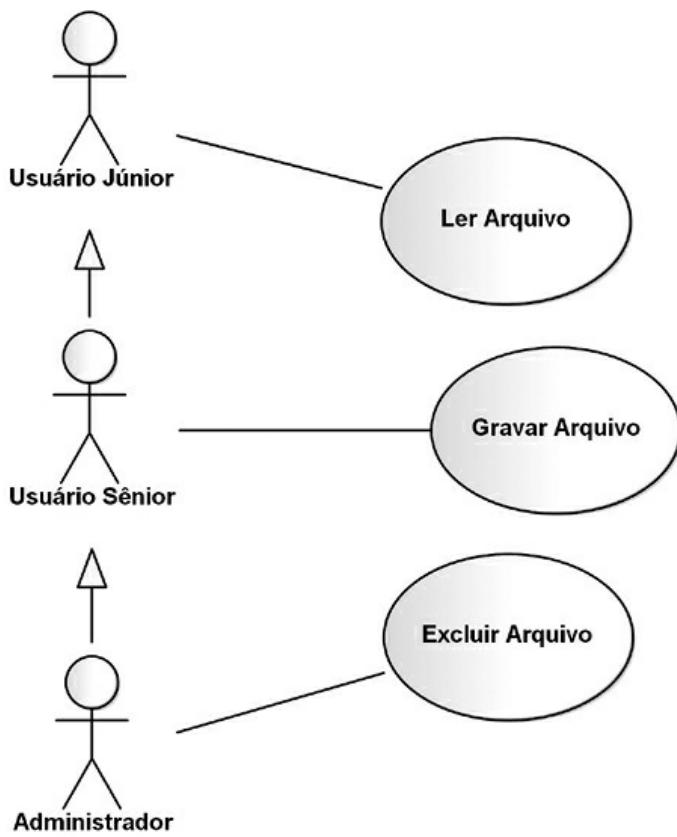
Todavia, alguns autores não recomendam esse tipo de associação para casos de uso, afirmando que isto pode tornar sua compreensão difícil. Recomendamos seu uso com parcimônia, em situações em que possam ser utilizados para evitar a repetição de instruções, por exemplo.

O relacionamento de generalização/especialização também pode ser aplicado sobre atores, o que é até mais comum. A figura 3.5 apresenta um exemplo de generalização/especialização entre atores em que existe um ator geral chamado **Pessoa** e dois atores especializados chamados, respectivamente, **Pessoa Física** e **Pessoa Jurídica**.

Um outro exemplo desse tipo de relacionamento aplicado a atores pode ser visto na figura 3.6, na qual temos três atores, **Usuário Júnior**, **Usuário Sênior** e **Administrador**. Nesse exemplo, o ator **Usuário Júnior** pode apenas executar o caso de uso **Ler Arquivo**; já o ator **Usuário Sênior** pode executar o caso de uso **Ler Arquivo**, por ser uma especialização do ator **Usuário Júnior**, e o caso de uso **Gravar Arquivo**. Observe que não existe uma associação entre o ator **Usuário Sênior** e o caso de uso **Ler Arquivo**, já que isto não é necessário, uma vez que o ator **Usuário Sênior** herda essa associação do ator **Usuário Júnior**. Finalmente, o ator **Administrador** pode executar os casos de uso **Ler Arquivo** e **Gravar Arquivo** herdados do ator **Usuário Sênior**, além do caso de uso **Excluir Arquivo**, que somente ele pode executar. Assim, percebe-se que o relacionamento de generalização/especialização aplicado a atores pode representar níveis de usuários e aproveitar associações já definidas, evitando deixar o diagrama com muitas linhas entrecruzadas.



*Figura 3.5 – Generalização/Especialização com Atores.*



*Figura 3.6 – Generalização/Especialização com Atores e Casos de Uso.*

### 3.8 Inclusão

A associação de inclusão costuma ser utilizada quando existe um cenário, situação ou rotina comum a mais de um caso de uso. Quando isso ocorre, a documentação dessa rotina é colocada em um caso de uso específico para que outros casos de uso utilizem esse serviço, evitando-se descrever uma mesma sequência de passos em vários casos de uso. Os relacionamentos de inclusão indicam uma obrigatoriedade, ou seja, quando um determinado caso de uso tem um relacionamento de inclusão com outro, a execução do primeiro obriga também a execução do segundo. Um relacionamento de inclusão pode ser comparado à chamada de uma sub-rotina ou função, artifício bastante utilizado na maioria das linguagens de programação.

Uma associação de inclusão é representada por uma linha tracejada contendo uma seta em uma de suas extremidades, a qual aponta para o caso de uso incluído no caso de uso posicionado na outra extremidade da linha. As associações de inclusão costumam apresentar também um estereótipo que contém o texto “**include**”, entre dois sinais de menor (<) e dois de maior (>). Um exemplo de inclusão pode ser examinado na figura 3.7.

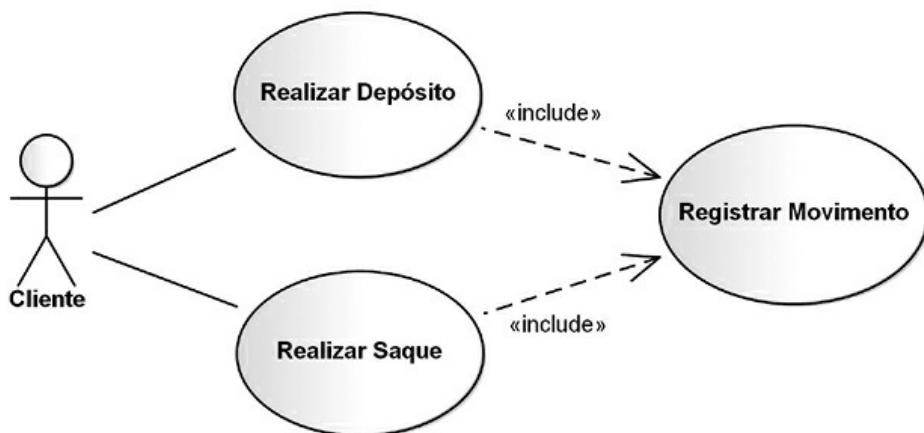


Figura 3.7 – Inclusão.

Ao analisarmos a figura 3.7 percebemos que, sempre que um saque ou depósito ocorrer, deverá ser registrado para fins de histórico bancário. Como as rotinas para registro de um saque ou depósito são extremamente semelhantes, diferenciando-se apenas pelo tipo de movimento, colocou-se a rotina de registro em um caso de uso à parte, chamado **Registrar**

**Movimento.** O caso de uso **Registrar Movimento** será executado obrigatoriamente sempre que os casos de uso **Realizar Depósito** ou **Realizar Saque** forem utilizados. Assim, não é preciso descrever os passos para registrar um movimento nesses casos de uso, pois as etapas estão descritas no caso de uso **Registrar Movimento**.

Outro exemplo de inclusão pode ser visto na figura 3.8, onde modelamos parte de um sistema de livraria virtual em que o cliente pode logar-se, adicionar livros ao carrinho de compras, visualizar o conteúdo do carrinho e concluir o pedido.

Nesse exemplo, o cliente pode solicitar a visualização de seu carrinho de compras sempre que quiser. No entanto, no momento em que decidir concluir o pedido, obrigatoriamente deverá verificar mais uma vez os livros que escolheu e fornecer uma última confirmação. Dessa forma, como o cliente pode executar o processo de visualização de carrinho sempre que quiser, este se caracteriza como um processo independente da conclusão de pedido. Entretanto, como é necessário visualizar o conteúdo do carrinho antes de concluir o pedido, o caso de uso **Concluir Pedido** deverá incluir o caso de uso **Visualizar Carrinho** para não ter que repetir os passos já definidos nesse caso de uso, poupando tempo e facilitando futuras manutenções.

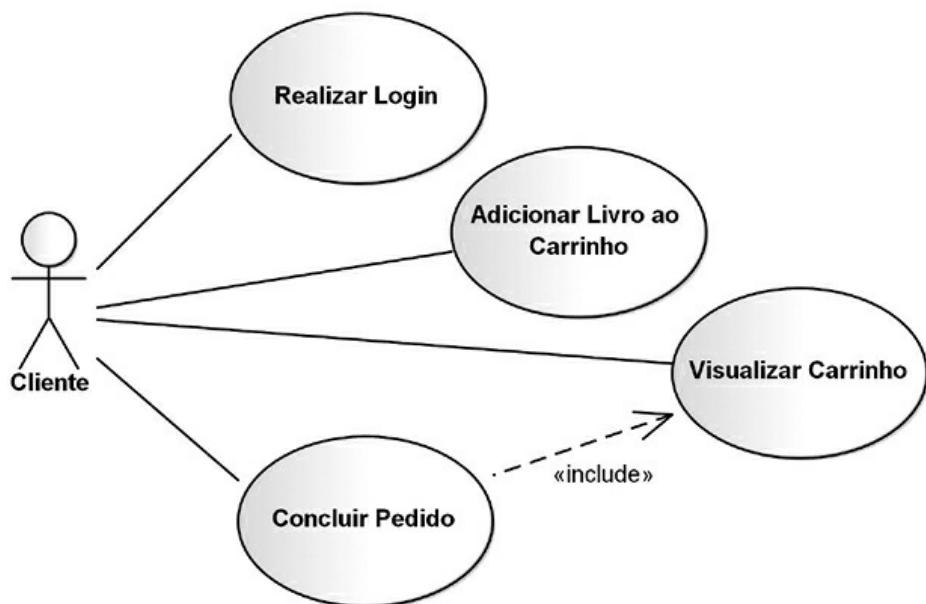


Figura 3.8 – Inclusão.

### 3.9 Extensão

Associações de extensão são utilizadas para descrever cenários opcionais que podem ser estendidos pelos comportamentos de outros casos de uso. Os casos de uso estendidos descrevem cenários que apenas serão executados em situações específicas quando determinadas condições forem satisfeitas. Dessa forma, as associações de extensão indicam a necessidade de um teste no comportamento de um caso de uso para determinar se é necessário executar também o comportamento (os passos) de um caso de uso estendido.

Relacionamentos de extensão representam situações que não ocorrem sempre, o que não significa que sejam incomuns. As associações de extensão têm uma representação muito semelhante às de inclusão, sendo também representadas por uma linha tracejada, diferenciando-se pelo fato de a seta apontar para o caso de uso que utiliza o caso de uso estendido e por haver um estereótipo contendo o texto “**extend**” em vez de “**include**”. Para ajudar a compreender o conceito de extensão, ilustraremos um exemplo bastante simples com um formulário apresentado na figura 3.9.

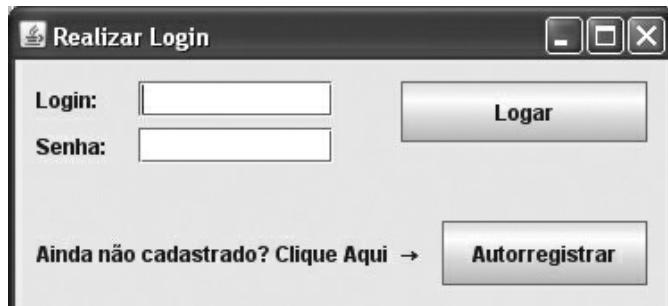


Figura 3.9 – Formulário de Login.

A figura 3.9 enfoca uma situação pela qual a maioria dos leitores provavelmente já passou, representando um formulário de login em que o cliente deverá informar seu nome-login e senha para poder se autenticar no sistema. No cenário ideal desse processo, o cliente informará um nome-login e uma senha válidos e lhe será permitido acessar o software. Contudo, pode acontecer de o cliente estar acessando a esse formulário pela primeira vez e não possuir cadastro no sistema. Ao prever essa possibilidade, o formulário permite que o cliente se registre, apresentando-lhe a opção de pressionar o botão “Autorricular” para se cadastrar no sistema. Obviamente, o cliente só fará isso na primeira vez, seguindo o

processo normal nas vezes seguintes.

Uma vez que o processo de **Autorricular** poderá ser chamado a partir do processo de **Login**, mediante a condição de o cliente não estar cadastrado, podemos representar esse relacionamento por meio de uma associação de extensão, conforme demonstra a figura 3.10.

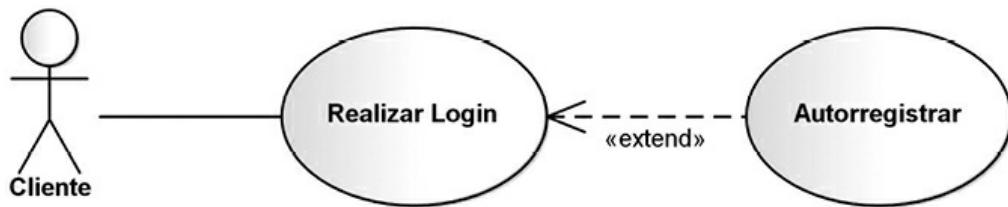
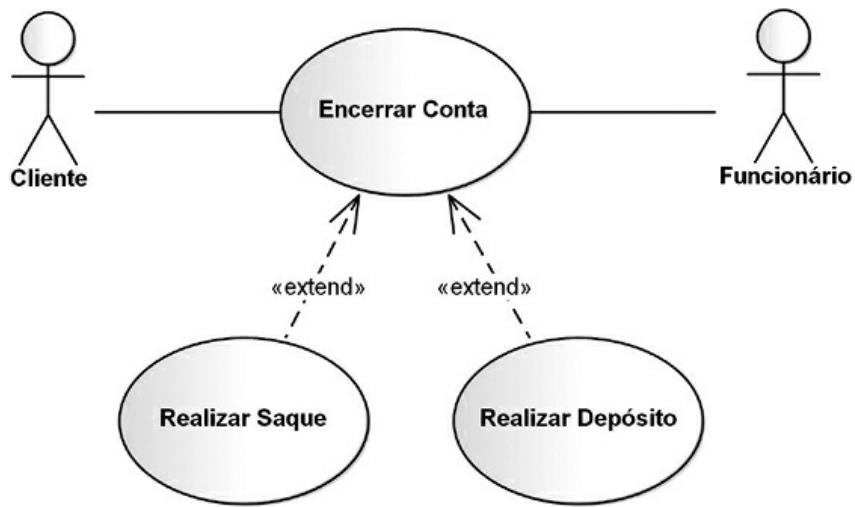


Figura 3.10 – Extensão.

Como podemos perceber, os processos de **Login** e **Autorricular** são representados como casos de uso e há uma associação de extensão entre eles, demonstrando que o caso de uso **Autorricular** poderá ser chamado a partir do caso de uso **Realizar Login**.

Um caso de uso pode ter muitos relacionamentos de extensão, conforme pode ser observado na figura 3.11, onde apresentamos um exemplo de extensão um pouco mais complexo, referente ao processo de encerramento de conta do sistema de controle bancário que temos modelado.

Nesse exemplo, um cliente dirige-se a um funcionário do banco e solicita o encerramento de uma determinada conta. Como sabemos, para encerrar uma conta, é necessário que seu saldo seja igual a zero. O caso de uso **Encerrar Conta** representa a funcionalidade de encerramento de conta utilizada pelo funcionário do banco e pode, eventualmente, fazer uma chamada ao caso de uso **Realizar Saque** – se o saldo da conta estiver positivo – ou ao caso de uso **Realizar Depósito** – se o saldo da conta estiver negativo. Essas associações entre o caso de uso **Encerrar Conta** e os casos de uso **Realizar Saque** e **Realizar Depósito** são associações de extensão porque os casos de uso somente serão chamados pelo caso de uso **Encerrar Conta** se as condições a eles associadas forem verdadeiras e, nessa situação, mesmo quando forem chamados, será somente um deles.



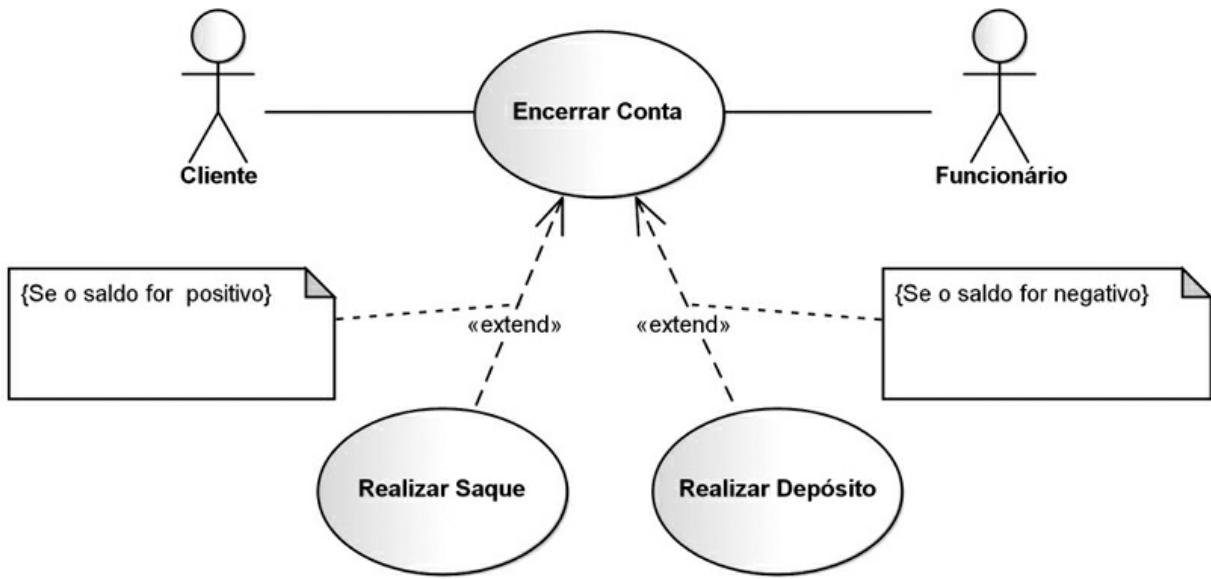
*Figura 3.11 – Extensão.*

### 3.10 Restrições em Associações de Extensão

Restrições são compostas de um texto entre chaves e utilizadas para definir validações, consistências, condições etc., que devem ser aplicadas a um determinado componente ou situação. Ao se tratar de extensões, às vezes nem sempre fica claro qual é a condição para que um caso de uso estendido seja executado. Assim, pode-se acrescentar uma restrição à associação de extensão por meio de uma nota explicativa, determinando a condição para que o caso de uso seja executado, conforme apresentado nas figuras 3.12 e 3.13.



*Figura 3.12 – Associação de Extensão com Restrição – Realizar Login.*



*Figura 3.13 – Associação de Extensão com Restrição – Encerrar Conta.*

No exemplo da figura 3.12, acrescentou-se uma restrição à associação de extensão para determinar a condição que necessita ser satisfeita para que o caso de uso **Autorregarstrar** seja executado, ou seja, se o cliente ainda não estiver registrado. Aqui, empregamos uma nota explicativa que é utilizada para incluir comentários ou restrições sobre um componente ou relacionamento. A seta tracejada que a une ao componente é chamada âncora.

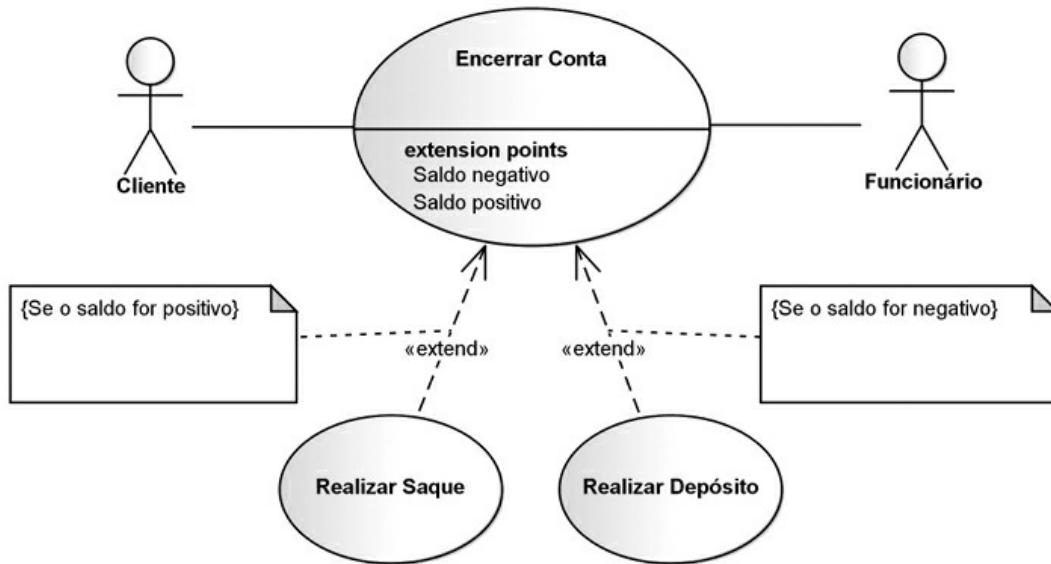
Já no exemplo da figura 3.13, o caso de uso **Realizar Saque** só será executado se o saldo da conta a ser encerrada for positivo e o caso de uso **Realizar Depósito** somente será executado se o saldo da conta for negativo. Essas restrições nada mais são do que regras de negócio referentes ao processo de encerramento de conta.

### 3.11 Pontos de Extensão

Um ponto de extensão identifica um ponto no comportamento de um caso de uso a partir do qual esse comportamento poderá ser estendido pelo comportamento de outro caso de uso, se a condição para que isso ocorra for satisfeita, conforme mostra a figura 3.14.

Aqui, modificamos o exemplo apresentado na figura 3.13, inserindo os pontos de extensão **Saldo positivo** e **Saldo negativo**, a partir dos quais os

casos de uso **Realizar Saque** e **Realizar Depósito** poderão ser estendidos. As associações de extensão deverão estar de acordo com a documentação do caso de uso, conforme demonstra a tabela 3.2, onde os cenários alternativos **Saldo positivo** e **Saldo negativo** fazem referência aos casos de uso **Realizar Saque** e **Realizar Depósito**.



*Figura 3.14 – Exemplo de Ponto de Extensão.*

*Tabela 3.2 – Documentação do Caso de Uso Encerrar Conta*

Nome do Caso de Uso	UC10 – Encerrar Conta
Ator Principal	Funcionário
Atores Secundários	Clientes
Resumo	Este caso de uso descreve as etapas necessárias para que um cliente encerre uma conta
Pré-condições	É necessário existir uma conta ativa
Pós-condições	
<b>Cenário Principal</b>	
<b>Ações do Ator</b>	
1. O cliente solicita o encerramento da conta fornecendo o número da conta em questão	
2. O funcionário solicita a emissão do saldo da conta	
	3. Executar caso de uso Emitir Saldo
	4. Encerrar a conta
<b>Ações do Sistema</b>	
	1. A conta só pode ser encerrada pelo titular

Restrições/Validações	2. A conta só pode ser encerrada se o seu saldo estiver zerado
<b>Cenário Alternativo I – Saldo Positivo</b>	
Ações do Ator	Ações do Sistema
	1. Executar Caso de Uso Realizar Saque
<b>Cenário Alternativo II – Saldo Negativo</b>	
Ações do Ator	Ações do Sistema
	1. Executar Caso de Uso Realizar Depósito
<b>Cenário Alternativo III – Manutenção do Cadastro do Cliente</b>	
Ações do Ator	Ações do Sistema
	1. Se for a única conta do cliente, será preciso atualizar seu cadastro, tornando-o inativo – Executar Caso de Uso Gerenciar Clientes

### 3.12 Multiplicidade no Diagrama de Casos de Uso

A multiplicidade em uma associação entre um ator e um caso de uso basicamente especifica o número de vezes que um ator pode utilizar um determinado caso de uso. A figura 3.15 apresenta um exemplo de multiplicidade em associações entre atores e casos de uso.



Figura 3.15 – Exemplos de Multiplicidade em Associações entre Atores e Casos de Uso.

Ao examinarmos esse exemplo, percebemos que um ator **Sócio** utiliza o caso de uso **Cadastrar Sócio** somente uma vez, enquanto o ator **Funcionário** pode utilizá-lo muitas vezes. Da mesma forma, percebemos que esse caso de uso só pode ser utilizado por um sócio e um funcionário por vez.

### 3.13 Estereótipos

Os estereótipos possibilitam certo grau de extensibilidade aos componentes ou associações da UML, além de permitir a identificação de componentes ou associações que, embora semelhantes aos outros, tenham

alguma característica que os diferencie, dando-lhes mais destaque no diagrama. Estereótipos podem atribuir funções extras a um componente, permitindo que este possa ser utilizado para modelar situações diferentes daquelas para as quais foi originalmente projetado. Existe uma grande quantidade de estereótipos que podem ser aplicados aos mais diversos componentes da UML e o engenheiro de software pode ele próprio criar seus estereótipos se assim desejar, por meio de perfis, por exemplo, como será explicado neste livro.

Como exemplo, poderíamos querer deixar bem claro que o caso de uso **Abrir Conta** refere-se a um processo. Assim, poderíamos atribuir-lhe o estereótipo <<process>>, cujo objetivo é deixar explícito que o caso de uso em questão refere-se a um processo. Esse tipo de estereótipo é apresentado na parte superior do componente, acima da descrição de seu nome. Existem alguns estereótipos que simplesmente apresentam um texto entre sinais de maior e menor sobre o componente, chamados estereótipos de texto, e outros que modificam o desenho-padrão do componente, chamados de estereótipos gráficos. A figura 3.16 ilustra um exemplo de estereótipo de texto. Exemplos de estereótipos gráficos serão apresentados no capítulo 4, sobre o diagrama de classes.



Figura 3.16 – Estereótipo de texto.

### 3.14 Fronteira de Sistema

Uma fronteira de sistema identifica um classificador que contém um conjunto de casos de uso. Permite identificar um subsistema ou mesmo um sistema completo, além de destacar o que está contido no sistema e o que não está. Atores são externos ao sistema, enquanto casos de uso são internos. Uma fronteira de sistema é representada por um retângulo envolvendo os casos de uso nela contidos, além de um título que a descreve. Ao longo deste capítulo, serão apresentados diversos exemplos contendo fronteiras de sistema.

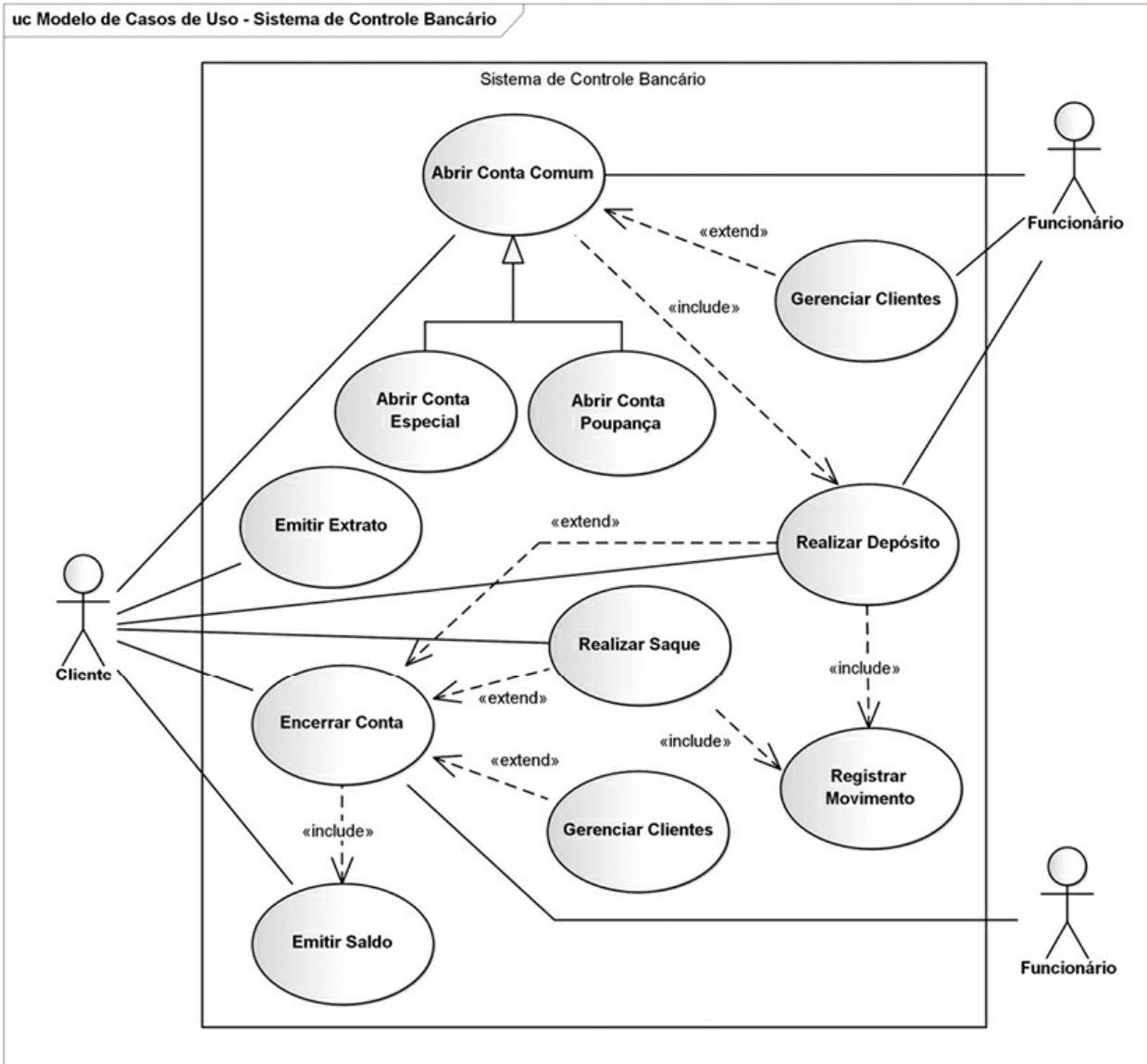
### **3.15 Exemplo de Diagrama de Casos de Uso – Sistema de Controle Bancário**

A figura 3.17 apresenta o diagrama de casos de uso referente a um sistema de controle bancário. Esse sistema permite que seus clientes, tanto pessoas físicas como jurídicas, possuam, abram e encerrem contas, que podem ser comuns, especiais ou poupança, bem como depositem ou saquem valores nessas contas (esses movimentos precisam ser registrados) e emitam saldos ou extratos relativos a uma determinada conta. Essas últimas quatro funcionalidades podem ser realizadas diretamente em um caixa eletrônico pelos clientes, porém, para abrir ou encerrar uma conta no banco, um cliente necessitará interagir com um funcionário do banco, que poderá, ainda, realizar alguma manutenção em seu cadastro, ou seja, registrá-lo ou alterar seus dados.

Pode-se perceber que um retângulo intitulado **Sistema de Controle Bancário** envolve todos os casos de uso do diagrama. Esse retângulo representa a fronteira do sistema, cujo objetivo, como já foi dito, é separar os atores externos das funcionalidades do software. Os atores, externos ao sistema, representam os clientes e funcionários do banco.

No sistema representado pelo diagrama, primeiramente um cliente, representado aqui por um ator, solicita a abertura de uma conta, a qual pode ser uma conta comum, que não permite a retirada de mais dinheiro do que está depositado; uma conta especial, que permite o saque extra até um determinado limite; ou uma conta poupança, que rende juros enquanto o dinheiro depositado permanecer sem ser movimentado.

Como os processos de abertura para cada tipo de conta têm características muito semelhantes entre si, com poucas distinções, resolveu-se colocar o caso de uso **Abrir Conta Comum** como uma generalização (embora seja efetivamente utilizado) e os outros dois tipos de abertura como especializações do primeiro, detalhando-se em sua documentação as características particulares que elas têm em relação ao caso de uso geral **Abrir Conta Comum**. Uma alternativa seria definir um caso de uso geral, denominado simplesmente **Abrir Conta**, que não seria utilizado diretamente, e derivar, a partir daí, os três tipos de abertura de conta possíveis.



*Figura 3.17 – Diagrama de Casos de Uso – Sistema de Controle Bancário.*

Podemos perceber que existe uma associação do tipo extensão entre o caso de uso **Abrir Conta Comum** (herdada também por suas especializações) e o caso de uso **Gerenciar Clientes**, cuja função é permitir o registro de novos clientes ou a alteração dos dados de clientes já registrados. Embora o caso de uso **Gerenciar Clientes** possa ser utilizado independentemente pelos funcionários do banco, a criação de uma conta bancária normalmente implica o registro do novo cliente ou, se este já estiver cadastrado, uma possível atualização. Dessa forma, como nem sempre é necessário registrar ou atualizar o cliente, o comportamento deste caso de uso só será incluído no comportamento do caso de uso **Abrir**

**Conta Comum** se o cliente que deseja abrir a conta não estiver registrado ou precisar atualizar seus dados.

Há também uma associação do tipo inclusão entre o caso de uso **Abrir Conta Comum** (igualmente herdado por suas especializações) e o caso de uso **Realizar Depósito**, porque é obrigatório depositar algum valor no momento em que o processo de abertura da conta for concluído, o que exige uma associação do tipo inclusão com o caso de uso **Realizar Depósito**. Assim, sempre que uma conta for aberta, as instruções contidas no caso de uso **Realizar Depósito** serão igualmente executadas.

Um cliente pode eventualmente querer encerrar uma conta, porém, antes de efetuar o encerramento, algumas operações devem ser levadas a efeito. Em primeiro lugar, é preciso verificar o saldo da conta para determinar se o banco precisa devolver algum dinheiro ao cliente ou, caso a conta seja especial e esteja negativa, se o cliente precisa depositar algum dinheiro para encerrá-la. A verificação do saldo é obrigatória, por isso utiliza-se uma associação do tipo inclusão entre o caso de uso **Encerrar Conta** e o caso de uso **Emitir Saldo**. Se o saldo estiver positivo, deve-se realizar um saque por meio do caso de uso **Realizar Saque**. Se o saldo estiver negativo, deve-se realizar um depósito por intermédio do caso de uso **Realizar Depósito**. Como não é possível saber se será necessário sacar ou depositar algum valor, as associações entre os casos de uso **Encerrar Conta**, **Realizar Saque** e **Realizar Depósito** serão representadas por extensões.

Esse diagrama representa, ainda, uma última associação de extensão entre o caso de uso **Encerrar Conta** e o caso de uso **Gerenciar Cliente**. Essa associação é necessária pela possibilidade de a conta a ser encerrada constituir-se na única conta ativa do cliente, o que determina a manutenção do registro dele, identificando-o como inativo, pois, uma vez cliente, seu registro não pode ser excluído do sistema, o mesmo ocorrendo com quaisquer contas que um dia ele possa ter tido na instituição bancária. Esta pode ser definida como encerrada, mas jamais excluída.

Existem ainda os casos de uso **Emitir Saldo** e **Emitir Extrato**, que são serviços que podem ser utilizados diretamente pelo cliente por meio de um caixa eletrônico, sem a intermediação de um funcionário. Como são serviços simples, não necessitam de interações com outros casos de uso do diagrama, embora o caso de uso **Emitir Saldo** seja utilizado pelo caso de

uso **Encerrar Conta**, por meio de uma associação de inclusão, conforme já descrito.

Finalmente, temos os casos de uso **Realizar Saque** e **Realizar Depósito**, que já foram citados, por poderem ser também requisitados pelo caso de uso **Encerrar Conta**. Normalmente, no entanto, tais serviços, da mesma maneira que os casos de uso **Saldo** e **Extrato**, serão utilizados diretamente pelo cliente. Contudo, esses serviços necessitam registrar o movimento realizado, razão pela qual ambos obrigatoriamente utilizam o caso de uso **Registrar Movimento**, que, para fins de histórico bancário, registra qualquer saque ou depósito porventura realizado em uma conta. Por ser obrigatória sua utilização, a associação entre os casos de uso **Realizar Saque** e **Realizar Depósito** com o caso de uso **Registrar Movimento** precisa ser de inclusão.

O exemplo do sistema de controle bancário é relativamente complexo. Ao utilizarmos diversos tipos de associações, procuramos ilustrar uma situação em que diversos recursos desse diagrama pudessem ser aplicados. Todavia, deve-se ter em mente que o diagrama de casos de uso, embora importante e frequentemente indispensável, possui um enfoque mais geral e tem por objetivo oferecer uma visão externa do sistema ao usuário, apresentando as funcionalidades do software e quem está autorizado a utilizá-las. É preciso destacar que não é obrigatória a aplicação de associações de inclusão, extensão ou especialização, visto que só devem ser utilizadas quando isto for considerado necessário, de acordo com as necessidades do software que está sendo modelado. Assim, deve-se procurar evitar desenvolver diagramas de casos de uso complexos sempre que possível. Ao longo deste capítulo e nas soluções dos exercícios propostos, apresentaremos novos exemplos, algumas vezes menos complexos que o ilustrado aqui.

### 3.16 Documentação do Diagrama de Casos de Uso do Sistema de Controle Bancário

Nesta seção, daremos uma sugestão de como documentar um diagrama de casos de uso. Os atores e casos de uso aqui documentados referem-se ao diagrama de casos de uso do sistema de controle bancário apresentado na figura 3.17, descrita na seção 3.15. Os casos de uso **Abrir Conta** e **Encerrar**

Conta foram apresentados anteriormente nas seções anteriores.

### 3.16.1 Atores que Interagem com o Sistema

- **Cliente** – Este ator representa as pessoas físicas ou jurídicas que mantêm ou mantiveram contas na instituição bancária, com direito a utilizar serviços como saque, depósito, saldo ou extrato enquanto mantiveram(em) contas ativas.
- **Funcionário** – Este ator representa os funcionários contratados para atender presencialmente os clientes da instituição. São basicamente os caixas do banco.

### 3.16.2 Documentação do Caso de Uso Abrir Conta Especial

*Tabela 3.3 – Documentação do Caso de Uso Abrir Conta Especial*

Nome do Caso de Uso	UC02 – Abrir Conta Especial
Caso de Uso Geral	Abrir Conta Comum
Autor Principal	Funcionário
Atores Secundários	Clientes
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para abrir uma conta-corrente especial com intermediação de um funcionário
Pré-condições	O pedido de abertura precisa ser aprovado
Pós-condições	É necessário realizar um depósito inicial
Cenário Principal	
Idênticas às do caso de uso Abrir Conta Comum, exceto por fornecer comprovante de estar empregado e de que seu salário é superior a R\$ 1.500,00	Idênticas às do caso de uso Abrir Conta Comum, exceto por incluir o limite do cheque especial ao abrir a conta
Restrições/Validações	<ol style="list-style-type: none"><li>1. Para abrir uma conta-corrente especial, é preciso ser maior de idade</li><li>2. É necessário comprovar estar empregado e o salário tem que ser superior a R\$ 1.500,00</li><li>3. O valor mínimo de depósito inicial é R\$ 50,00</li></ol>

### 3.16.3 Documentação do Caso de Uso Abrir Conta Poupança

*Tabela 3.4 – Documentação do Caso de Uso Abrir Conta Poupança*

Nome do Caso de Uso	UC03 – Abrir Conta Poupança
---------------------	-----------------------------

Caso de Uso Geral	Abrir Conta Comum
Ator Principal	Funcionário
Atores Secundários	Clientes
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para abrir uma conta poupança com intermediação de um funcionário
Pré-condições	
Pós-condições	
<b>Cenário Principal</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
Idênticas às do caso de uso Abrir Conta Idênticas às do caso de uso Abrir Conta Comum, exceto por não solicitar ao cliente exceto por não ser obrigatória a realização de um depósito	
Restrições/Validações	

### 3.16.4 Documentação do Caso de Uso Gerenciar Clientes

*Tabela 3.5 – Documentação do Caso de Uso Gerenciar Clientes*

<b>Nome do Caso de Uso</b>		<b>UC04 – Gerenciar Clientes</b>
Ator Principal	Funcionário	
Atores Secundários		
Resumo	Este caso de uso descreve as possíveis atividades de manutenção do cadastro de clientes, ou seja, permite incluir, alterar ou consultar clientes. Um cliente não pode ser excluído apenas se tornando inativo	
Pré-condições	<b>Cenário Principal</b>	
Pós-condições	<b>Ações do Ator</b>	
1. Informar o CPF ou CNPJ do cliente	<ol style="list-style-type: none"> <li>2. Validar CPF/CNPJ</li> <li>3. Consultar o cliente por seu CPF ou CNPJ</li> <li>4. Apresentar dados do cliente</li> </ol>	
Restrições/Validações	<p>1. O CPF ou CNPJ precisa ser validado</p> <p style="text-align: center;"><b>Cenário de Exceção I – CPF/CNPJ inválido</b></p>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>	
	<ol style="list-style-type: none"> <li>1. Informar que o número fornecido não condiz com um CPF/CNPJ válido</li> <li>2. Solicitar um novo número</li> </ol>	
	<b>Cenário de Exceção II – Cliente não encontrado</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>	

	1. Informar que o CPF/CNPJ fornecido não corresponde a nenhum cliente registrado
<b>Cenário Alternativo I – Incluir Cliente</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
1. Informar os dados cadastrais do cliente	
	2. Registrar cliente
Restrições/Validações	1. Os campos nome, endereço e data de nascimento são obrigatórios
<b>Cenário Alternativo II – Alterar Cliente</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
1. Alterar os dados cadastrais do cliente	
	2. Registrar alterações no cadastro do cliente
Restrições/Validações	1. Os campos nome, endereço e data de nascimento são obrigatórios

### 3.16.5 Documentação do Caso de Uso Realizar Depósito

*Tabela 3.6 – Documentação do Caso de Uso Realizar Depósito*

Nome do Caso de Uso	UC05 – Realizar Depósito
Ator Principal	Funcionário
Atores Secundários	Clientes
Resumo	Descreve os passos necessários para um cliente depositar algum valor em uma determinada conta
Pré-condições	
Pós-condições	
<b>Cenário Principal</b>	
Ações do Ator	Ações do Sistema
1. Informar o número da conta	
	2. Verificar se a conta existe
3. Fornecer o valor a ser depositado	
	4. Somar o valor depositado ao saldo da conta
	5. Executar caso de uso “Registrar Movimento”
Restrições/Validações	1. A conta precisa existir e estar ativa
<b>Cenário de Exceção – Conta não encontrada</b>	
Ações do Ator	Ações do Sistema
	1. Comunicar ao cliente que o número da conta informada não foi encontrado

### 3.16.6 Documentação do Caso de Uso Emitir Saldo

*Tabela 3.7 – Documentação do Caso de Uso Emitir Saldo*

Nome do Caso de Uso		UC06 – Emitir Saldo
Autor Principal	Cliente	
Atores Secundários		
Resumo	Descreve os passos necessários para um cliente obter o saldo referente a uma determinada conta	
Pré-condições		
Pós-condições		
Cenário Principal		
Ações do Ator	Ações do Sistema	
1. Informar o número da conta	2. Verificar a existência da conta	
	3. Solicitar a senha da conta	
4. Informar a senha	5. Verificar se a senha está correta	
	6. Emitir o saldo	
Restrições/Validações	1. A conta precisa existir e estar ativa	
	2. A senha precisa estar correta	
Cenário de Exceção I – Conta não encontrada		
Ações do Ator	Ações do Sistema	
	1. Comunicar ao cliente que o número da conta informada não foi encontrado	
Cenário de Exceção II – Senha inválida		
Ações do Ator	Ações do Sistema	
	1. Comunicar ao cliente que a senha fornecida não combina com a senha da conta	

### 3.16.7 Documentação do Caso de Uso Emitir Extrato

*Tabela 3.8 – Documentação do Caso de Uso Emitir Extrato*

Nome do Caso de Uso		UC07 – Emitir Extrato
Autor Principal	Cliente	
Atores Secundários		
Resumo	Descreve os passos necessários para um cliente obter o extrato referente a uma determinada conta em um dado período	
Pré-condições		
Pós-condições		

Cenário Principal	
Ações do Ator	Ações do Sistema
1. Informar o número da conta	2. Verificar se a conta existe e está ativa 3. Solicitar a senha
4. Informar a senha	5. Verificar se a senha está correta 6. Solicitar períodos
7. Informar o período inicial e o final do extrato	8. Listar os movimentos da conta dentro do período informado
Restrições/Validações	1. A conta precisa existir e estar ativa 2. A senha precisa estar correta 3. Os períodos precisam estar corretos e o período inicial não pode ser maior que o final
Cenário de Exceção I – Conta não encontrada	
Ações do Ator	Ações do Sistema
	1. Comunicar ao cliente que o número da conta informada não foi encontrado
Cenário de Exceção II – Senha inválida	
Ações do Ator	Ações do Sistema
	1. Comunicar ao cliente que a senha fornecida não combina com a senha da conta
Cenário de Exceção III – Períodos inválidos	
Ações do Ator	Ações do Sistema
	1. Informar ao cliente que as datas fornecidas são inválidas

### 3.16.8 Documentação do Caso de Uso Realizar Saque

*Tabela 3.9 – Documentação do Caso de Uso Realizar Saque*

Nome do Caso de Uso	UC08 – Realizar Saque
Ator Principal	Cliente
Atores Secundários	
Resumo	Descreve os passos necessários para um cliente sacar algum valor de uma determinada conta
Pré-condições	
Pós-condições	
Cenário Principal	

Ações do Ator	Ações do Sistema
1. Informar o número da conta	2. Verificar se a conta existe 3. Solicitar a senha
4. Informar a senha	5. Verificar se a senha está correta 6. Solicitar o valor a sacar
7. Informar o valor a ser retirado	8. Diminuir o valor a sacar do saldo da conta 9. Entregar a importância solicitada ao cliente 10. Executar caso de uso "Registrar Movimento"
Restrições/Validações	1. A conta precisa existir e estar ativa 2. A senha precisa estar correta 3. Se for uma conta comum ou conta poupança, o valor solicitado precisará ser igual ou inferior ao saldo da conta 4. Se for uma conta especial, o valor solicitado precisará ser igual ou inferior ao saldo da conta somado ao seu limite
<b>Cenário de Exceção I – Conta não encontrada</b>	
Ações do Ator	Ações do Sistema
	1. Comunicar ao cliente que o número da conta informada não foi encontrado
<b>Cenário de Exceção II – Senha inválida</b>	
Ações do Ator	Ações do Sistema
	1. Comunicar ao cliente que a senha fornecida não combina com a senha da conta
<b>Cenário de Exceção III – Saldo insuficiente</b>	
Ações do Ator	Ações do Sistema
	1. Emitir uma mensagem informando que o saldo é insuficiente e recusar o pedido

### 3.16.9 Documentação do Caso de Uso Registrar Movimento

*Tabela 3.10 – Documentação do Caso de Uso Registrar Movimento*

Nome do Caso de Uso	UC09 – Registrar Movimento
Ator Principal	Cliente
Atores Secundários	
Resumo	Descreve os passos necessários para registrar um movimento referente a um saque ou a um depósito

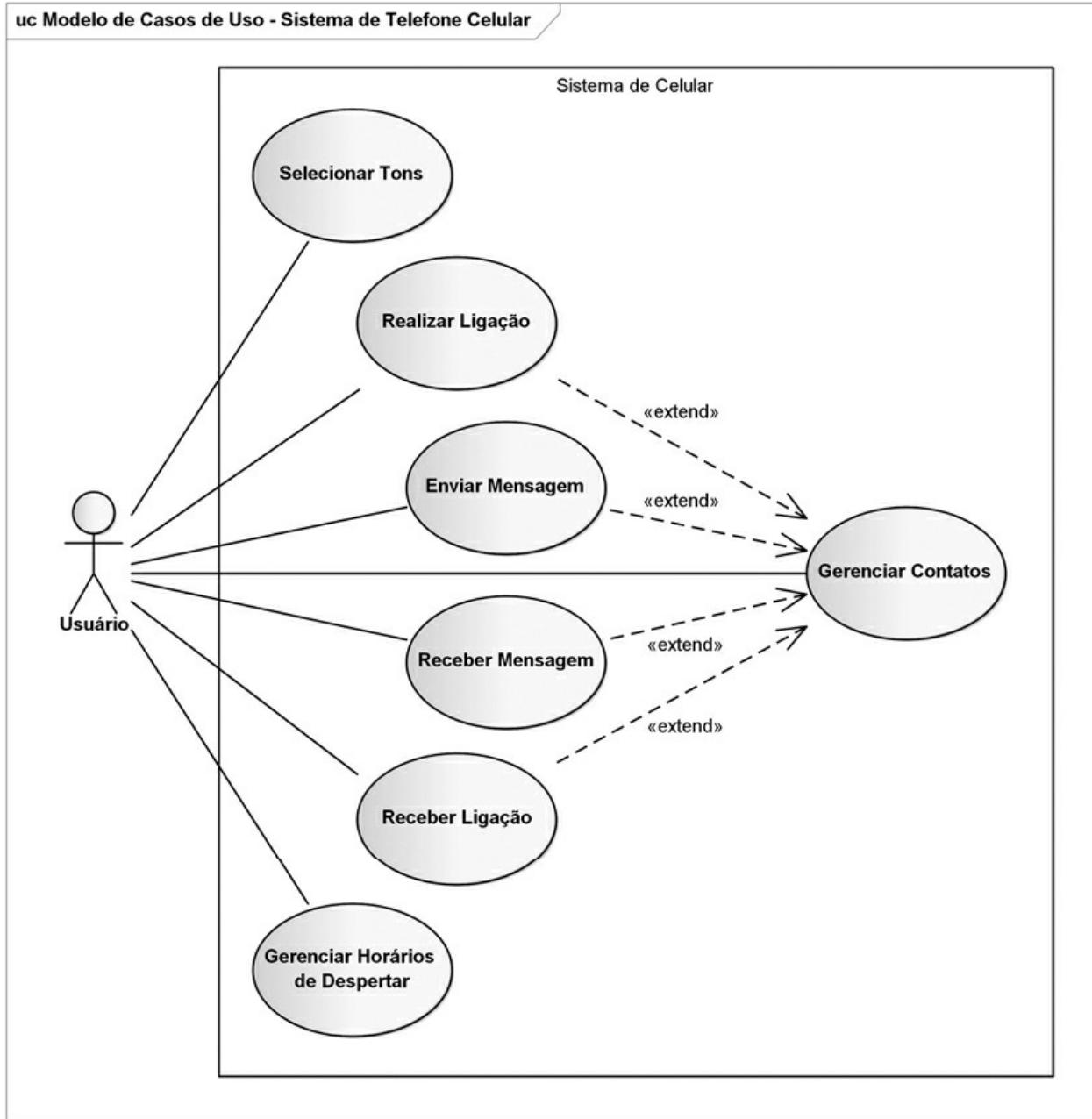
Pré-condições		
Pós-condições		
	Cenário Principal	
Ações do Ator	Ações do Sistema	
	1. Receber o número da conta movimentada, o tipo do movimento (se é depósito ou saque), a data e o valor movimentado 2. Registrar o movimento	
Restrições/Validações		

### 3.17 Exemplo de Diagrama de Casos de Uso – Sistema de Telefone Celular

Nesta seção, apresentaremos o diagrama de casos de uso para um sistema simples de telefone celular (Figura 3.18), levando em consideração os seguintes requisitos:

- O celular oferece o serviço de realizar chamadas, no qual o usuário deve informar um telefone para que o celular ligue. O celular deve registrar as últimas chamadas.
- Semelhante ao serviço de chamadas, o telefone oferece o serviço de mensagens, em que o usuário deve informar o número de telefone para o qual deseja enviar a mensagem. O celular deve igualmente registrar as últimas mensagens.
- O aparelho oferece o serviço de agenda, por meio do qual é possível cadastrar os diversos contatos do usuário. Cada contato armazena o nome do contato e seu telefone. Caso o usuário consulte um telefone já existente, ele poderá ligar para esse contato ou enviar uma mensagem. O sistema deve guardar as últimas ligações feitas, bem como as últimas mensagens enviadas.
- O celular oferece também o serviço de recebimento de chamadas. O sistema deve avisar o recebimento de uma chamada por meio do toque de uma música e o usuário pode aceitar a chamada ou não. As últimas ligações também devem ser gravadas.
- Da mesma forma, o sistema deve oferecer o serviço de recebimento de mensagens, devendo também registrar as últimas mensagens recebidas.
- O celular oferece ainda o serviço de despertador, no qual o usuário pode cadastrar e/ou ativar um ou mais horários para despertar.
- Finalmente, o sistema oferece o serviço de tons, no qual o usuário pode

selecionar, entre muitas músicas possíveis, a que mais lhe agrada para avisá-lo do recebimento de uma chamada ou mensagem, ou para despertá-lo.



*Figura 3.18 – Diagrama de Casos de Uso – Sistema de Telefone Celular.*

Nesta solução, o único ator representado é o usuário, uma vez que somente ele interage com o aparelho, não havendo interação com nenhum outro ator. A seguir, detalharemos os casos de uso identificados:

- **Realizar Ligação** – Este caso de uso representa o processo, bem simples,

aliás, pelo qual o usuário faz uma ligação para um número de telefone, bastando para isso informar o número do aparelho e pressionar o botão para chamá-lo. O sistema deverá gravar cada ligação efetuada, registrando o número chamado e se ela foi atendida ou não. Se o limite máximo de ligações registradas for atingido, o sistema deverá excluir a mais antiga.

- **Enviar Mensagem** – Este caso de uso representa os passos necessários para que o usuário envie uma mensagem para outro telefone celular. Esse processo é quase tão simples quanto o anterior, no qual o usuário deve informar o número para o qual deseja enviar a mensagem, digitá-la e pressionar o botão para enviá-la. O celular deverá também registrar essa mensagem e, da mesma forma que no processo anterior, se o limite máximo de mensagens registradas for atingido, o sistema deverá excluir a mais antiga.
- **Receber Mensagem** – Este caso de uso representa o processo pelo qual o usuário recebe uma mensagem, cujo evento deverá ser também notificado pelo aparelho e registrado por ele, excluindo a mensagem mais antiga se o número máximo de mensagens recebidas já tiver sido atingido.
- **Receber Ligação** – Este caso de uso representa o processo pelo qual o usuário recebe uma ligação, ou seja, quando alguém, por meio de outro aparelho, liga para o celular do usuário. Quando esse evento ocorrer, o aparelho deverá avisar o usuário de que ele está recebendo uma ligação, por meio de uma música, por exemplo. Cabe ao usuário optar por atender à ligação ou não, de qualquer forma, o celular deverá registrá-la, identificando se foi atendida ou recusada.
- **Gerenciar Contatos** – Este caso de uso representa os passos que o usuário deve percorrer para inserir um novo contato na agenda, consultar um contato já existente, alterar ou mesmo excluir um contato. Existem associações de extensão entre os casos de uso **Realizar Ligação**, **Enviar Mensagem**, **Receber Mensagem** e **Receber Ligação**, uma vez que é possível registrar um novo contato por meio desses processos.
- **Gerenciar Horários de Despertar** – Este caso de uso representa os passos necessários para que o usuário insira, altere ou exclua horários em que o celular deverá despertá-lo.

- **Selecionar Tons** – Finalmente, este caso de uso representa o processo por meio do qual o usuário seleciona os tons de música que mais lhe agradam ouvir quando recebe uma ligação, uma mensagem ou quando o despertador toca. O usuário deve selecionar qual dessas opções deseja alterar o tom de música e escolher em uma lista a música que considerar mais adequada.

### 3.17.1 Documentação do Caso de Uso Realizar Ligação

*Tabela 3.11 – Documentação do Caso de Uso Realizar Ligação*

Nome do Caso de Uso	UC01 – Realizar Ligação
Atores Principal	Usuário
Atores Secundários	
Resumo	Descreve os passos necessários para que um usuário de celular possa fazer uma ligação para um determinado número
Pré-condições	
Pós-Condições	
	<b>Cenário Alternativo I – Número do Contato Conhecido</b>
Ações do Ator	Ações do Sistema
1. Solicitar a lista de contatos	2. Apresentar todos os contatos registrados em ordem alfabética
3. Selecionar contato	4. Consultar e apresentar o número do contato selecionado
	<b>Cenário Alternativo II – Número não Registrado</b>
Ações do Ator	Ações do Sistema
1. Informar o número a discar	
	<b>Cenário Principal</b>
Ações do Ator	Ações do Sistema
1. Confirmar o número da ligação	2. Estabelecer ligação
Restrições/Validações	

### 3.18 Exemplo de Diagrama de Casos de Uso – Sistema de Biblioteca

Nesta seção, apresentamos o diagrama de casos de uso para um sistema de biblioteca, considerando que uma biblioteca pode possuir muitos

exemplares de um determinado livro e os livros possuídos pela biblioteca podem pertencer aos mais diversos gêneros. Sendo assim, conclui-se ser necessário existir um módulo para manutenção do cadastro de livros. Além disso, uma vez que é útil poder pesquisar livros por gênero ou por determinado autor, torna-se também necessária a existência de um módulo para manutenção de gêneros e de outro para manutenção do cadastro de autores. Da mesma forma, só é permitido locar exemplares para sócios registrados, sendo, portanto, necessária a existência de um módulo para a manutenção do cadastro de sócios da biblioteca.

O processo principal desse sistema é a locação de exemplares de livros propriamente dita, na qual o sócio se identifica e informa os exemplares que deseja tomar emprestado. Se o registro do sócio existir, ele não possuir empréstimos em atraso e o limite máximo de exemplares que ele pode manter emprestado não for atingido, a locação será autorizada. Outros processos importantes referem-se à devolução de exemplares, quando um sócio devolve livros locados anteriormente; à renovação de exemplares, quando um sócio renova o empréstimo de um ou mais exemplares; e à reserva de exemplares para futura locação, quando estes forem devolvidos.

Finalmente, existem ainda processos importantes para a biblioteca de cunho gerencial e estatístico: a emissão dos autores e dos livros mais locados, para se ter uma ideia da preferência dos sócios. A figura 3.19 apresenta o diagrama de casos de uso para esse sistema.

Esse diagrama representa cada processo de manutenção de cadastro como um caso de uso e o identifica como **Gerenciar Sócios** ou **Gerenciar Livros**. Na verdade, estes são casos de uso secundários, que servem como apoio aos casos de uso primários representados aqui basicamente pelos casos de uso **Locar Exemplares**, **Devolver Exemplares**, **Renovar Exemplares** e **Reservar Exemplares**, que representam os processos de locação de exemplares de livros por um sócio e de sua posterior devolução, bem como a renovação do empréstimo de exemplares e a reserva de exemplares indisponíveis em um determinado momento.



*Figura 3.19 – Diagrama de Casos de Uso – Sistema de Biblioteca.*

O caso de uso **Emitir Livros Mais Locados** também é secundário, uma vez que consiste basicamente em um relatório que apresenta os livros locados pela ordem de maior preferência. Em sistemas mais complexos em que haja muitos módulos de manutenção e/ou relatórios de pequena importância ou complexidade, não é muito recomendado inserir um caso de uso para cada um desses módulos, porque o diagrama poderá se tornar muito poluído, ou seja, difícil de ler, contendo muitas informações irrelevantes e escondendo as funcionalidades realmente importantes. Em tais situações, é possível encontrar um caso de uso genérico, descrito como “Gerenciar Cadastros” ou “Emitir Relatórios”, por exemplo, ou mesmo

não haver nenhuma referência aos casos de uso secundários.

Observe que o ator sócio somente interage com o caso de uso **Locar Exemplares**, uma vez que ele é o ator mais interessado no processo, identificando-se e informando quais exemplares deseja locar; com o caso de uso **Devolver Exemplares**, por meio do qual o sócio retorna à biblioteca os livros que ele tomou emprestado; com o caso de uso **Renovar Exemplares**, por meio do qual um sócio renova o empréstimo de um ou mais exemplares; e com o caso de uso **Reservar Exemplares**, por meio do qual um sócio reserva um livro cujos exemplares estejam locados. Os outros casos de uso são acessados somente pelo funcionário. Poder-se-ia argumentar que o caso de uso **Gerenciar Sócios** poderia ter interação também com o sócio, já que fornece as informações ao bibliotecário para a manutenção do seu cadastro. Todavia, uma vez que essa interação é mínima, consideramos que o único ator que realmente interage com esse módulo é o bibliotecário.

O leitor talvez questione onde é feita a verificação para determinar se o sócio já se encontra cadastrado ou se possui locações pendentes. Todas essas etapas estão contidas no próprio caso de uso **Locar Exemplares**, já que não é necessário criar um caso de uso exclusivo para qualquer uma dessas validações, sendo perfeitamente possível defini-las na documentação do caso de uso.

### 3.18.1 Documentação do Caso de Uso Locar Exemplares

*Tabela 3.12 – Documentação do Caso de Uso Locar Exemplares*

Nome do Caso de Uso	UC01 – Locar Exemplares
Ator Principal	Bibliotecário
Atores Secundários	Sócios
Resumo	Descreve os passos necessários para que o sócio possa realizar um empréstimo na biblioteca
Pré-condições	1. Limite de exemplares não pode ser excedido 2. Não pode haver empréstimos em atraso
Pós-condições	
Cenário Principal	
Ações do Ator	Ações do Sistema
1. Informar o sócio	2. Consultar o sócio e o tipo de sócio

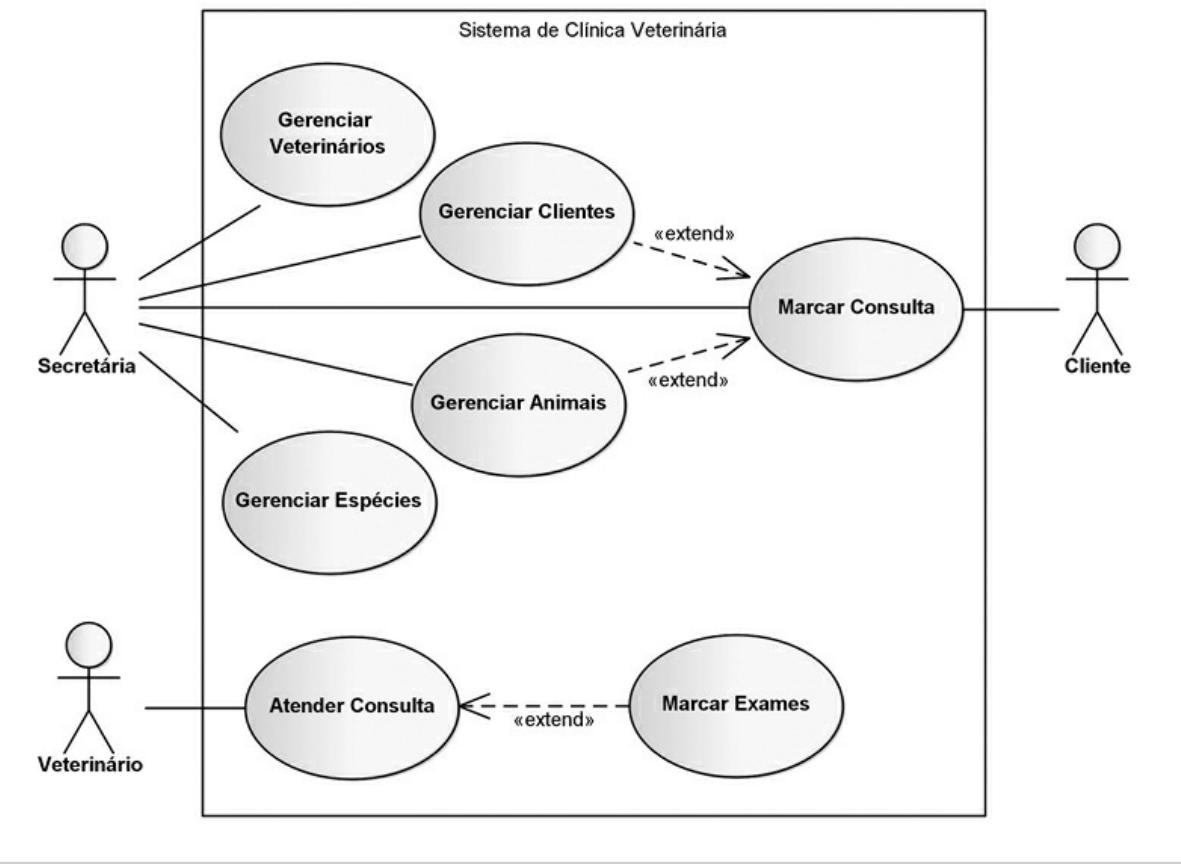
	3. Verificar se há empréstimos em atraso
4. Consultar livros desejados	5. Consultar dados do livro e se há algum exemplar disponível
6. Selecionar livros disponíveis e confirmar	7. Registrar empréstimo 8. Alterar situação dos exemplares para emprestados
<b>Cenário Alternativo I – Empréstimos em Atraso ou Número Máximo de Exemplares para Locação Excedido</b>	
Ações do Ator	Ações do Sistema
Restrições/Validações	Recusar empréstimo

### 3.19 Exemplo de Diagrama de Casos de Uso – Sistema de Clínica Veterinária

Neste exemplo, o sistema de clínica veterinária a ser modelado se comporta da seguinte maneira:

- Os clientes primeiramente marcam consultas com a secretaria, fornecendo informações pessoais e as dos animais que desejam tratar. Se o cliente ou o animal ainda não estiver cadastrado no sistema ou existir algum dado que precise ser atualizado, a secretaria deverá atualizar o cadastro.
- Em cada sessão de tratamento (uma sessão equivale a uma consulta), o cliente deve informar os sintomas aparentes do animal, os quais devem ser registrados. O tratamento pode ser encerrado em apenas uma consulta, quando se tratar de algo simples, ou arrastar-se por muitas sessões, dependendo do diagnóstico do médico-veterinário.
- Durante a consulta, o veterinário pode marcar exames para o animal, a serem trazidos na sessão seguinte. O pedido dos exames e seus resultados devem ser registrados no histórico de tratamento do animal. Após cada sessão, o histórico da consulta deve ser atualizado.
- É responsabilidade da secretaria manter atualizados os cadastros de clientes, animais, médicos e espécies. A figura 3.20 apresenta a solução para esse sistema na forma de um diagrama de casos de uso.

uc Modelo de Casos de Uso - Sistema de Clínica Veterinária



*Figura 3.20 – Diagrama de Casos de Uso – Sistema de Clínica Veterinária.*  
Os atores que compõem esse diagrama são descritos a seguir:

- **Cliente** – Este ator representa uma pessoa física que possui um ou mais animais que alguma vez foram tratados pela clínica.
- **Secretária** – A descrição deste ator é autoexplicativa. Ele representa os funcionários da clínica responsáveis por marcar consultas e gerenciar a maioria dos cadastros da empresa.
- **Veterinário** – Este ator também é autoexplicativo, representando os médicos-veterinários da clínica que atendem os animais.

Os casos de uso que compõem o sistema são os seguintes:

- **Marcar Consulta** – Este caso de uso representa as etapas necessárias para que um cliente possa agendar uma consulta para um determinado animal. Nesse caso de uso, interagem os atores **Cliente** e **Secretária**. A partir desse caso de uso, observe que podem ser executados os casos de

uso secundários **Gerenciar Clientes** e **Gerenciar Animais**, pela possibilidade de que seja necessário registrar um novo cliente ou animal ou de que seus dados precisem ser atualizados. Observe que o ator **Secretária** pode utilizar esses dois últimos casos de uso, independentemente do caso de uso **Marcar Consulta**, como é possível verificar por meio da associação direta entre o ator **Secretária** e esses casos de uso.

- **Gerenciar Veterinários e Gerenciar Espécies** – Estes dois casos de uso secundários são bastante simples, representando os módulos de cadastro dos veterinários que trabalham na clínica, bem como as espécies de animais tratados na veterinária.
- **Atender Consulta** – Este caso de uso representa o registro de atendimento de uma consulta pelo médico-veterinário responsável. Observe que inserimos um relacionamento de extensão com o caso de uso **Marcar Exames**, já que eventualmente o veterinário pode pedir ao cliente para realizar exames no animal em questão. Na verdade, a documentação desse último caso de uso poderia estar no próprio caso de uso **Atender Consulta**. Preferimos, no entanto, representá-lo de forma separada para tornar mais clara a compreensão do diagrama. Obviamente, os resultados dos exames seriam verificados e registrados na consulta seguinte.

### 3.19.1 Documentação do Caso de Uso Atender à Consulta

*Tabela 3.13 – Documentação do Caso de Uso Atender à Consulta*

Nome do Caso de Uso	UC01 – Atender à Consulta
Ator Principal	Veterinário
Atores Secundários	
Resumo	Descreve os passos necessários para que um veterinário atenda a uma consulta
Pré-condições	
Pós-condições	
Cenário Principal	
Ações do Ator	Ações do Sistema
2. Selecionar consulta	<ol style="list-style-type: none"> <li>1. Apresentar as consultas do dia</li> <li>3. Consultar tratamento relativo à consulta e os dados do animal</li> </ol>

- e do cliente
4. Listar os possíveis exames solicitados anteriormente
  5. Relatar o que foi feito durante a consulta

6. Atualizar consulta

**Cenário Alternativo I – Solicitação de Exames**

Ações do Ator	Ações do Sistema
	Executar caso de uso Marcar Exames
Restrições/Validações	

### **3.20 Exemplo de Diagrama de Casos de Uso – Sistema de Controle de Advocacia**

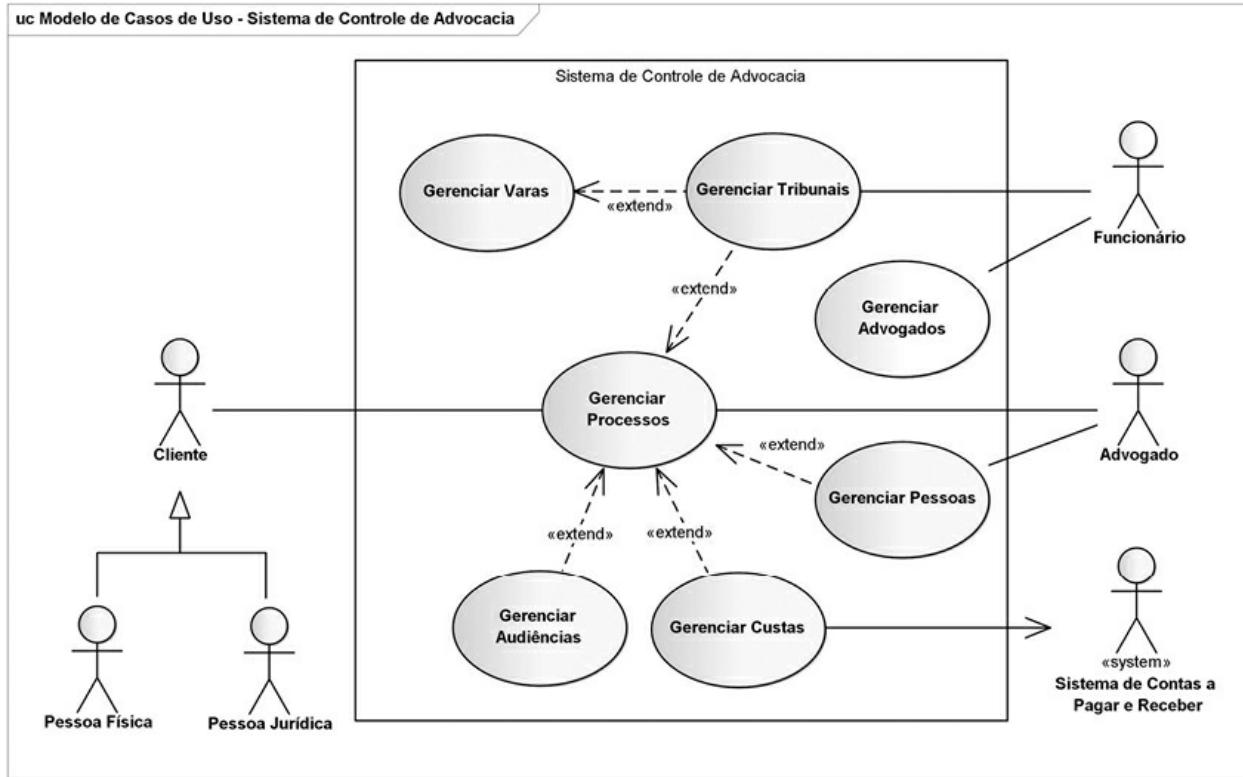
A seguir, apresentaremos o diagrama de casos de uso para um sistema de controle de processos jurídicos, de acordo com as seguintes afirmações:

- O escritório de advocacia possui muitos clientes, que podem ser tanto pessoas físicas como jurídicas, que contratam o escritório para defendê-las ou processar outra pessoa. Quando uma pessoa procura o escritório, se ainda não estiver cadastrada, um funcionário deverá registrar seus dados pessoais.
- Em seguida, o cliente deve fornecer informações a respeito do processo que deseja que o escritório move contra alguém ou sobre o processo contra o qual ele precisa ser defendido. Obviamente, o processo precisa ser registrado e receberá diversas adições enquanto estiver em andamento. O cliente deve fornecer também informações sobre a parte contrária (pessoa física ou jurídica que está processando ou sendo processada pelo cliente), que deverá também ser registrada, caso ainda não esteja. Observe que uma mesma pessoa física ou jurídica pode ser tanto um cliente como uma parte contrária, em períodos diferentes, obviamente.
- Um processo é gerenciado por um determinado advogado do escritório (eventualmente pode haver mais de um advogado associado a um processo). O escritório possui muitos advogados, portanto precisa manter um cadastro atualizado com os dados de todos os advogados que trabalham ou já trabalharam para o escritório.
- Um processo deve tramitar em um determinado tribunal e em uma

determinada vara. No entanto, um tribunal pode julgar muitos processos e uma vara pode ter diversos processos tramitando nela. Um tribunal pode ter inúmeras varas, porém um processo julgado por um determinado tribunal só pode tramitar em uma das varas pertencentes a este. O escritório pode achar necessário emitir relatórios de todos os processos em andamento, em um determinado tribunal, e tramitando em uma determinada vara.

- Cada processo tem, no mínimo, uma audiência, e cada audiência relativa a um determinado processo deve conter sua data e a recomendação do tribunal. Para fins de histórico do processo, cada audiência deve ser registrada.
- Um processo pode gerar custas (despesas com fotocópias, viagens, gastos em geral). Cada custa deve ser armazenada de forma a ser cobrada da parte contrária, caso o processo seja ganho.
- Esse sistema deve estar integrado a um sistema de contas a pagar e receber; cada custa gera uma conta a pagar. Caso o processo seja ganho, gerará uma ou mais contas a receber, dependendo da negociação com a parte contrária.

A figura 3.21 apresenta uma solução para o problema enunciado por meio de um diagrama de casos de uso.



*Figura 3.21 – Diagrama de Casos de Uso – Sistema de Controle de Advocacia.*

Esse modelo de casos de uso é manipulado pelos seguintes atores:

- **Cliente** – Ator que representa as pessoas físicas ou jurídicas que solicitam que o escritório de advocacia as defenda ou que processe outra pessoa. Observe que existem duas especializações a partir do ator **Cliente**, chamadas **Pessoa Física** e **Pessoa Jurídica**, indicando que o cliente pode tanto constituir-se de uma pessoa física quanto de uma pessoa jurídica.
- **Advogado** – Este ator representa os advogados do escritório de advocacia responsáveis pelos diversos processos movidos ou defendidos pelo escritório de advocacia.
- **Funcionário** – Este ator representa os diversos funcionários do escritório responsáveis por interagir com alguns processos simples do sistema.
- **Sistema de Contas a Pagar e Receber** – Este ator representa um sistema integrado ao de controle de processos, conforme demonstra seu estereótipo <<system>>, para onde são enviadas as custas geradas por

um determinado processo.

A seguir, descreveremos os casos de uso que compõem esse diagrama. O leitor notará que são basicamente cadastros simples, sendo o único com alguma complexidade o caso de uso **Gerenciar Processos**:

- **Gerenciar Processos** – Este é o principal caso de uso desse sistema, no qual são manipulados os processos em andamento do escritório. Além do ator **Advogado**, note que o ator **Cliente** também interage com este caso de uso, não diretamente, é claro, mas precisa fornecer informações sobre o processo ao advogado. A esse caso de uso estão ligados praticamente todos os demais casos de uso do diagrama.
- **Gerenciar Pessoas** – Este caso de uso representa o módulo de manutenção de pessoas que fazem ou já fizeram parte de algum processo, seja como clientes, seja como partes contrárias. O **Advogado** pode manter o registro de qualquer pessoa a qualquer momento. No entanto, esse caso de uso está ligado ao caso de uso **Gerenciar Processos**, pois pode ser chamado a partir dele sempre que um cliente ou parte contrária ainda não estiver registrado ou seus dados necessitarem de atualização. Como a chamada a esse caso de uso só ocorre nas situações citadas, a associação entre esses casos de uso é classificada como uma extensão.
- **Gerenciar Advogados** – Este caso de uso representa o módulo de manutenção dos advogados que trabalham ou já trabalharam no escritório. Este módulo é normalmente gerido pelo ator **Funcionário**.
- **Gerenciar Tribunais e Gerenciar Varas** – Estes casos de uso representam os módulos de manutenção de tribunais e varas. O caso de uso **Gerenciar Varas** só pode ser chamado a partir do caso de uso **Gerenciar Tribunais**. Na verdade, o caso de uso **Gerenciar Varas** poderia estar contido no caso de uso **Gerenciar Tribunais**, porém resolvemos apresentá-lo separadamente, para facilitar a compreensão do diagrama. Observe que o caso de uso **Gerenciar Tribunais** pode tanto ser utilizado de forma independente quanto ser chamado a partir do caso de uso **Gerenciar Processos**, na eventual possibilidade de ser necessária alguma alteração do tribunal ou vara onde o processo for tramitar.

- **Gerenciar Audiências** – Conforme foi dito anteriormente, um processo tem, no mínimo, uma audiência, mas pode transcorrer ao longo de diversas delas. Essas audiências precisam ser registradas. Assim, criamos um caso de uso responsável pela manutenção das audiências de cada processo. A associação de extensão entre os casos de uso **Gerenciar Audiências** e **Gerenciar Processos** indica que o caso de uso apenas será executado eventualmente a partir do processo ao qual ele pertence, ou seja, sempre que uma nova audiência for marcada, esta deverá ser registrada, e sempre que uma audiência tiver transcorrido, esta deverá ser atualizada com a recomendação do tribunal.
- **Gerenciar Custas** – Um processo normalmente envolve despesas conhecidas como custas. Cada possível custa de um determinado processo precisa ser registrada. Observe que, da mesma forma que o caso de uso **Gerenciar Audiências**, o caso de uso **Gerenciar Custas** só poderá ser chamado a partir do caso de uso **Gerenciar Processos**, e mesmo assim somente quando houver necessidade. Por esse motivo, novamente utilizamos um relacionamento de extensão entre os dois casos de uso. Observe que outro ator interage com o caso de uso em questão, o ator denominado **Sistema de Contas a Pagar e Receber**. Esse ator é um sistema de software, como demonstra seu estereótipo <<system>>, e representa um sistema integrado ao sistema de advocacia. Ao registrar custas, o sistema de advocacia as transmite ao sistema de contas a pagar e receber, que registra cada custa como uma conta a receber. É importante notar a direção da seta da associação em questão, que demonstra que o ator não envia informações, mas somente as recebe.

### 3.20.1 Documentação do Caso de Uso Gerenciar Processos

*Tabela 3.14 – Documentação do Caso de Uso Gerenciar Processos*

Nome do Caso de Uso	UC01 – Gerenciar Processos
Atores Principal	Advogado
Atores Secundários	
Resumo	Descreve os passos necessários para que um advogado registre um novo processo ou atualize um processo já existente
Pré-condições	

Pós-condições		Cenário Principal	
Ações do Ator	Ações do Sistema	Cenário Alternativo I – Incluir novo Processo	
1. Selecionar cliente, parte contrária, advogado(s) que se envolverão no processo e tribunal onde o processo será julgado			1. Apresentar as informações necessárias, a saber: listar pessoas, advogados, tribunais e processos cadastrados
2. Listar as varas associadas ao tribunal selecionado			2. Listar as varas associadas ao tribunal selecionado
3. Selecionar a vara do processo, informar o relato inicial do processo e confirmar			3. Selecionar a vara do processo, informar o relato inicial do processo e confirmar
4. Registrar processo			4. Registrar processo
5. Registrar os advogados envolvidos no processo			5. Registrar os advogados envolvidos no processo
Cenário Alternativo II – Atualizar Processo			
Ações do Ator	Ações do Sistema		
1. Selecionar processo		2. Carregar informações do processo	
3. Informar alterações no processo e confirmar		4. Atualizar processo	
Cenário Alternativo III – Atualizar Audiências			
Ações do Ator	Ações do Sistema		
	Executar caso de uso Gerenciar Audiências		
Cenário Alternativo IV – Atualizar Custas			
Ações do Ator	Ações do Sistema		
	Executar caso de uso Gerenciar Custas		
Restrições/Validações			

## 3.21 Exercícios Propostos

### 3.21.1 Sistema de Controle de Cinema

Desenvolva o diagrama de casos de uso para um sistema de controle de cinema, sabendo que:

- Um cinema pode ter muitas salas, sendo necessário, portanto, registrar informações a respeito de cada uma, como sua capacidade, ou seja, o número de assentos disponíveis.

- O cinema apresenta muitos filmes. Um filme tem informações como título e duração. Assim, sempre que um filme for apresentado, deve-se registrá-lo também.
- Um mesmo filme pode ser apresentado em diferentes salas e em horários diversos. Cada apresentação em uma determinada sala e horário é chamada sessão. Um filme apresentado em uma sessão tem um conjunto máximo de ingressos, determinado pela capacidade da sala.
- Os clientes do cinema podem comprar ingressos para assistir a uma sessão. O funcionário deve intermediar a compra do ingresso. Um ingresso deve conter informações como o tipo de ingresso (meia-entrada ou ingresso inteiro) adquirido, a hora em que o ingresso foi comprado e o valor pago. Além disso, um cliente só pode comprar ingressos para sessões ainda não encerradas.

### **3.21.2 Sistema de Controle de Clube Social**

Desenvolva um modelo de casos de uso para um sistema de controle de clube social de acordo com os seguintes requisitos:

- Para ingressar no clube, é necessário apresentar uma solicitação a ser avaliada por uma comissão nomeada pelo clube.
- Em caso de aprovação, o candidato pode associar-se ao clube. Opcionalmente, caso possua dependentes, poderá associá-los também, o que obviamente aumentará o valor da mensalidade a ser paga.
- Uma vez sendo sócio do clube, deverá pagar uma mensalidade para poder frequentá-lo.
- As mensalidades são geradas pelo clube, levando em consideração a categoria do sócio e o número de seus dependentes.

### **3.21.3 Sistema de Locação de Veículos**

Desenvolva o diagrama de casos de uso para um sistema de controle de aluguel de veículos, levando em consideração os seguintes requisitos:

- A empresa tem uma grande frota de carros de passeio, os quais apresentam diferentes marcas e modelos. Eventualmente, um carro pode ser retirado da frota por acidente grave ou simplesmente por ter sido considerado velho demais para o padrão da empresa, sendo vendido. Da

mesma forma, a empresa eventualmente renova a frota, sendo necessário, portanto, sempre manter o cadastro de veículos da empresa.

- Os clientes dirigem-se à empresa e solicitam o aluguel de carros. No entanto, primeiramente, é necessário cadastrá-los, caso ainda não possuam cadastro ou seus dados tenham sido alterados.
- Depois de ter se identificado/cadastrado, o cliente escolherá o carro que deseja alugar (o valor da locação varia de acordo com o ano, a marca e o modelo do automóvel). Durante o processo de locação, o cliente deve informar por quanto tempo utilizará o carro, para qual finalidade e por onde desejará trafegar, já que essas informações também influenciam o preço da locação. Antes de liberar o veículo, a empresa exige que o cliente forneça um valor superior ao estabelecido na análise da locação, a título de caução. Caso o cliente não utilize todo o valor da caução até o momento da devolução do veículo, o valor restante lhe será devolvido.
- Quando o cliente devolve o carro, deve-se definir o automóvel como devolvido, registrar a data e hora da devolução e a quilometragem em que se encontra, bem como verificar se o automóvel se encontra nas mesmas condições em que foi locado. Caso o cliente tenha ocupado o carro por mais tempo que o combinado, deverá pagar o aluguel referente ao tempo extra em que permaneceu com o veículo. Da mesma maneira, o cliente deverá pagar por qualquer dano sofrido pelo veículo quando este se encontrava locado. Por outro lado, o cliente pode ser resarcido de parte do valor que pagou caso o custo do tempo em que esteve de posse do veículo seja inferior ao valor previamente fornecido.

### **3.21.4 Sistema para Controle de Leilão Via Internet**

Desenvolva um modelo de casos de uso para um sistema de leilão via internet, de acordo com os seguintes requisitos:

- Pode haver diversos participantes em cada leilão, interessados em adquirir os itens ofertados. Cada participante deve se logar no sistema, e, caso ainda não esteja cadastrado, deverá se registrar.
- Um participante pode dar quantos lances quiser durante a realização do leilão, mas não é obrigado a dar lance algum. Antes de fazer quaisquer ofertas, ele precisará se logar no sistema. Além disso, deve haver um

leilão em andamento.

- Pode haver diversos leilões programados. É responsabilidade do leiloeiro gerenciar cada leilão e os itens arrolados para serem ofertados em cada um deles.
- É também responsabilidade do leiloeiro iniciar um determinado leilão no horário estabelecido.
- Durante um leilão é ofertado cada um dos itens arrolados, que pode ou não receber lances.
- Sempre que um lance suplantar o lance anterior, o sistema deverá anunciar-lo. Caso tenha havido algum lance e se o tempo para lances tiver acabado, o sistema deverá declarar qual o vencedor.

### **3.21.5 Sistema de Controle de Hotelaria**

Desenvolva o diagrama de casos de uso para um sistema de controle de hotelaria, sabendo que:

- Os quartos podem ser alugados no momento em que o hóspede chega ao hotel (desde que existam vagas) ou reservados via internet.
- Caso seja a primeira vez que alugue quartos ou seus dados tenham mudado, o hóspede deverá ser cadastrado antes de finalizar o aluguel do quarto.
- Além do aluguel do quarto, o hotel oferece diversos serviços, como restaurante, lavar e/ou passar roupas etc. Obviamente, qualquer um deles, se solicitado, será cobrado na fatura final.
- O hóspede pode também consumir os produtos contidos no frigobar, que também são cobrados pelo hotel.
- As diárias vencem ao meio-dia. A política do hotel exige que as diárias sejam quitadas semanalmente. Quando o cliente for quitar a fatura, quitará não somente as diárias do(s) quarto(s) que alugou, mas também qualquer serviço que tenha solicitado e os itens consumidos no frigobar.
- Depois de quitar a fatura, o hóspede pode permanecer no hotel ou encerrar sua estada.
- Quando for encerrar sua estada, o hóspede deverá pagar quaisquer serviços e/ou diárias ainda não pagas.

### **3.21.6 Sistema de Controle de Imobiliária**

Desenvolva um modelo de casos de uso para um sistema de controle de imobiliária, levando em consideração que:

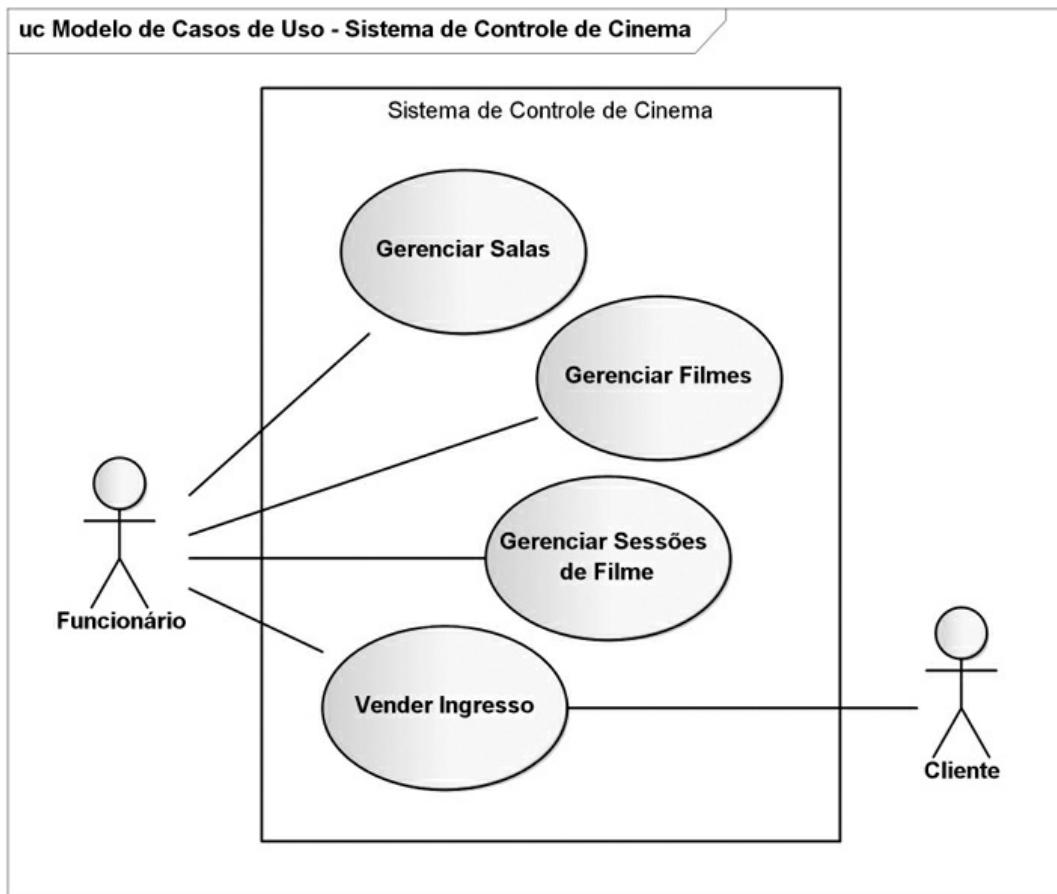
- A imobiliária negocia com muitos donos de imóveis, que desejam vendê-los ou alugá-los. Os imóveis podem ser de diversos tipos, como casas, apartamentos, chácaras ou terrenos. A imobiliária considera dono a pessoa física ou jurídica possuidora de um ou mais imóveis.
- Para que um imóvel possa ser vendido ou alugado, é necessário que o dono do imóvel preencha um contrato de intermediação de compra ou aluguel com a imobiliária. Nesse contrato, a imobiliária se compromete em oferecer o imóvel em questão a interessados e intermediar o processo de venda/aluguel.
- A imobiliária também se compromete a pagar ao dono do imóvel os valores relativos à transação de compra ou aluguel, descontadas a comissão da imobiliária, possíveis taxas governamentais ou quaisquer outras taxas pendentes, como gastos da imobiliária no reparo do imóvel, por exemplo.
- A empresa possui muitos corretores trabalhando para ela. É responsabilidade deles atender os clientes da imobiliária e intermediar a venda ou aluguel de imóveis. A imobiliária considera clientes as pessoas físicas ou jurídicas que compram ou alugam imóveis oferecidos pela empresa, bem como os donos desses imóveis.
- A imobiliária precisa manter registros de quem era o dono de um imóvel vendido e quem o comprou. Da mesma forma, a empresa precisa saber quem é o dono de um determinado imóvel que está sendo alugado, quem já o alugou (incluindo o início e o fim da locação e os valores pagos durante a locação) e quem o aluga atualmente ou, ainda, se o imóvel encontra-se sem locador.

### **3.22 Resolução dos Exercícios**

#### **3.22.1 Resolução do Exercício Sistema de Controle de Cinema**

A figura 3.22 apresenta a solução desse exercício. Os atores identificados foram:

- **Funcionário** – A função desse ator é óbvia. Ele representa os funcionários que trabalham no cinema e são responsáveis por vender ingressos e realizar a manutenção nos cadastros.
- **Cliente** – Este ator representa as pessoas que comprarão ingressos para assistir às sessões de filmes oferecidas pelo cinema.



*Figura 3.22 – Diagrama de Casos de Uso – Sistema de Controle de Cinema.*  
A seguir, descreveremos os casos de uso que compõem esse diagrama:

- **Gerenciar Salas** – Este é um caso de uso secundário que se refere ao processo de manutenção do cadastro de salas pertencentes ao cinema.
- **Gerenciar Filmes** – Este caso de uso, também secundário, refere-se ao processo de manutenção do cadastro de filmes apresentados pelo cinema.
- **Gerenciar Sessões de Filmes** – Este caso de uso, igualmente secundário, representa a manutenção do cadastro de sessões, em que são definidos os filmes que serão apresentados, em quais salas e em que datas e

horários.

- **Vender Ingresso** – Este é o principal caso de uso desse sistema, sendo o único primário, e apresenta os passos necessários para que o funcionário venda um ingresso para uma sessão a um cliente.

Nesse modelo, poder-se-ia acrescentar mais casos de uso secundários, para manter o cadastro de gêneros dos filmes ou dos atores principais, mas não achamos necessário esse detalhamento por justamente se tratar de casos de uso secundários sem muito impacto no sistema.

Como forma de ilustrar o processo de venda de ingressos, apresentamos na tabela 3.15 a documentação do caso de uso **Vender Ingresso**.

*Tabela 3.15 – Documentação do Caso de Uso Vender Ingresso*

Nome do Caso de Uso	UC01 – Vender Ingresso
Ator Principal	Funcionário
Atores Secundários	Clientes
Resumo	Este caso de uso descreve as etapas percorridas por um funcionário para emitir um ou mais ingressos para uma sessão de cinema
Pré-condições	Só podem ser apresentadas as sessões ainda em aberto e com assentos disponíveis
Pós-condições	
Cenário Principal	
Ações do Ator	Ações do Sistema
1. Selecionar opção Venda de Ingresso	2 Apresentar sessões disponíveis
3. Selecionar a sessão desejada	4. Apresentar os assentos disponíveis da sessão selecionada
5. Selecionar os assentos desejados e se cada ingresso é inteiro ou meia-entrada	6. Calcular o valor dos ingressos
7. Selecionar a forma de pagamento e autorizar a emissão dos ingressos	8. Emitir ingressos
Restrições/Validações	1. Os clientes só podem comprar ingressos compatíveis com sua idade
	2. Os tipos de ingressos disponíveis são ingressos inteiros, que pagam

o valor integral; e as meias-entradas, que pagam metade do valor – só menores de idade têm direito a meias-entradas

#### Cenário de Exceção – Idade Incompatível com a Sessão Escolhida

##### Ações do Ator

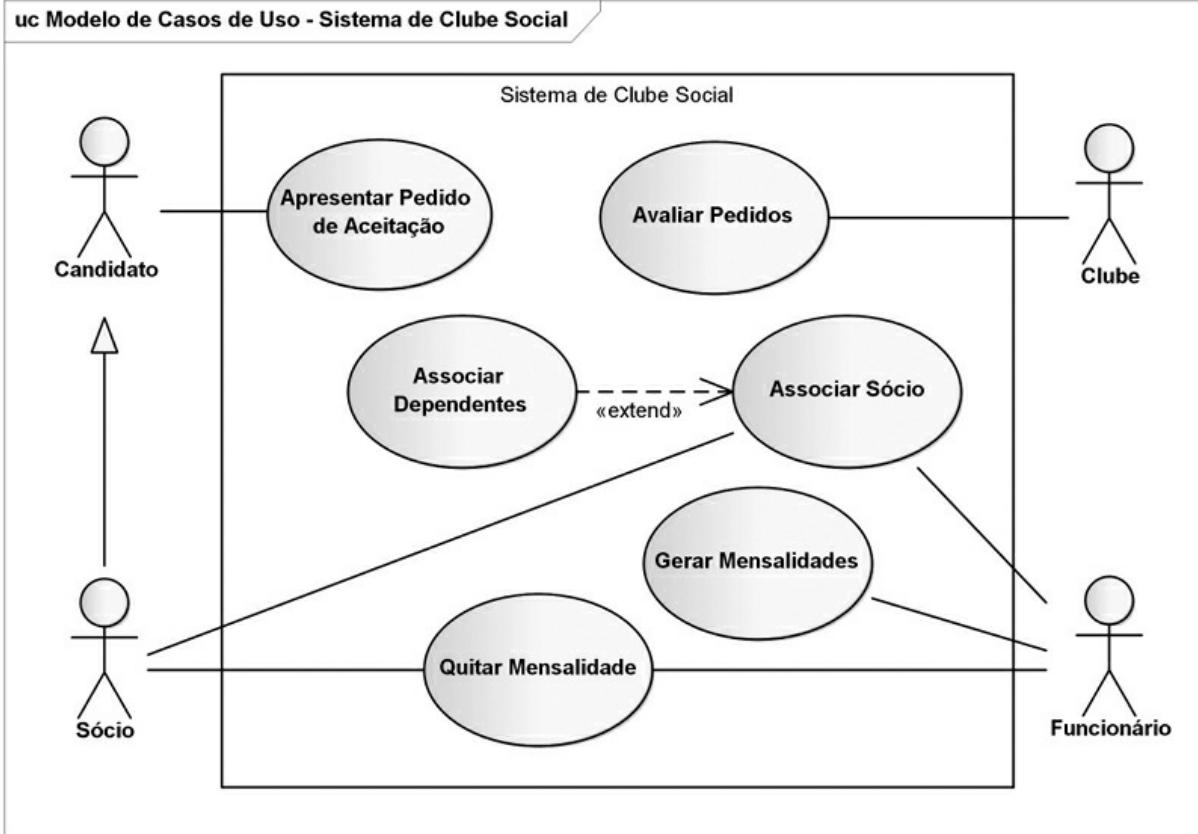
##### Ações do Sistema

1. Comunicar ao cliente que ele não possui idade suficiente para assistir à sessão

### 3.22.2 Resolução do Exercício Sistema de Controle de Clube Social

A figura 3.23 apresenta a solução do exercício de clube social. Os atores identificados foram:

- **Candidato** – Este ator representa uma pessoa interessada em associar-se ao clube, mas cujo pedido precisa ser aprovado.
- **Sócio** – Este ator, derivado a partir do ator **Candidato**, representa o sócio propriamente dito, depois de seu cadastro ter sido aprovado.
- **Clube** – Este ator representa os membros responsáveis por avaliar os pedidos de ingresso no clube de candidatos a sócio.
- **Funcionário** – Este ator representa os funcionários que interagem com os sócios do clube e manipulam funções do sistema.



*Figura 3.23 – Diagrama de Casos de Uso – Sistema de Clube Social.*  
A seguir, descreveremos os casos de uso que compõem esse diagrama:

- **Apresentar Pedido de Aceitação** – Este caso de uso refere-se ao processo por meio do qual um candidato a sócio apresenta seu pedido para ser aceito como sócio do clube.
- **Avaliar Pedidos** – Este caso de uso representa o processo por meio do qual cada um dos pedidos de aceitação pendentes é avaliado pelo clube.
- **Associar Sócio** – Uma vez tendo seu pedido aprovado, este caso de uso detalha os passos necessários para que um funcionário do clube associe um novo sócio. Observe que há uma interação entre o ator **Sócio** com esse caso de uso, uma vez que ele fornece os dados necessários para que a associação seja realizada. Observe ainda que existe uma associação de extensão entre esse caso de uso e o caso de uso **Associar Dependentes**, em que são detalhadas as etapas pelas quais o sócio poderá associar seus possíveis dependentes. Levando-se em consideração que esse caso de uso é opcional, já que o sócio não necessariamente terá dependentes, essa associação caracteriza-se por ser uma extensão.

- **Gerar Mensalidades** – Este caso de uso representa o processo para gerar as mensalidades dos sócios do clube e é manipulado exclusivamente por um funcionário.
- **Quitar Mensalidade** – Este é o processo pelo qual o sócio interage com um funcionário para quitar uma ou mais mensalidades.

A tabela 3.16 apresenta a documentação do processo de Quitar Mensalidade.

*Tabela 3.16 – Documentação do Caso de Uso Quitar Mensalidade*

Nome do Caso de Uso	UC01 – Quitar Mensalidade
Ator Principal	Funcionário
Atores Secundários	Sócios
Resumo	Este caso de uso descreve as etapas percorridas por um sócio para quitar uma ou mais mensalidades
Pré-condições	O cartão do sócio precisa ser válido
Pós-condições	
Cenário Principal	
Ações do Ator	Ações do Sistema
1. O sócio informa ao funcionário que deseja quitar mensalidades	
2. O funcionário seleciona a opção quitar mensalidades	3. Solicitar número do cartão do sócio
4. Informar cartão	5. Consultar sócio
	6. Consultar mensalidade(s) a pagar
	7. Apresentar mensalidade(s) a pagar
8. O sócio seleciona a(s) mensalidade(s) a pagar	
9. O funcionário solicita a quitação da(s) mensalidade(s) selecionada(s)	10. Quitar mensalidade(s)
	11. Emitir recibo de quitação
Cenário Alternativo – Mensalidade em Atraso	
Ações do Ator	Ações do Sistema
	1. Calcular juros da(s) mensalidade(s)
	2. Atualizar valor a pagar
Cenário de Exceção – Cartão Inválido	
Ações do Ator	Ações do Sistema

Restrições/Validações

1. Caso o cartão não seja válido, emitir mensagem de erro

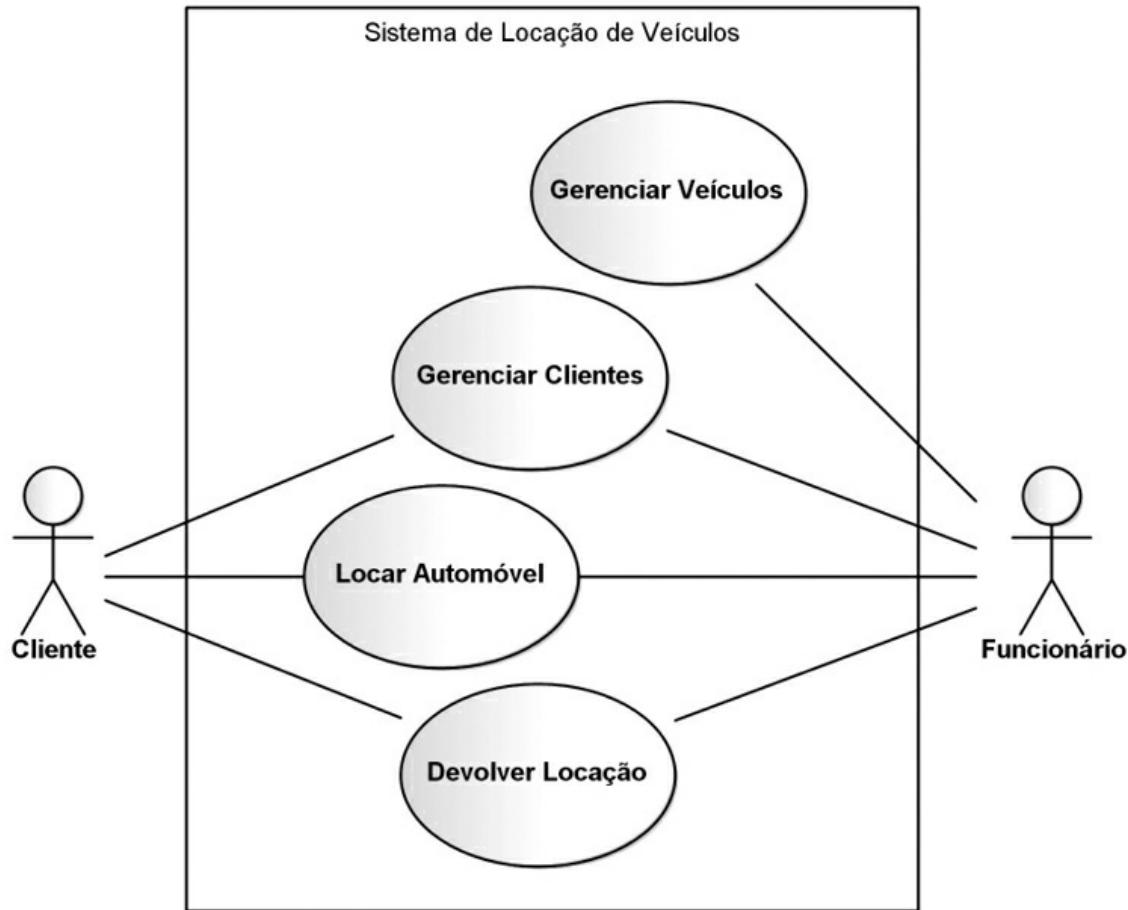
O cartão do sócio precisa ser válido

### 3.22.3 Resolução do Exercício Sistema de Locação de Veículos

A figura 3.24 apresenta a solução do exercício de sistema de locação de veículos. Os atores identificados foram:

- **Cliente** – Este ator representa os clientes que desejam locar veículos na locadora. Esse ator interage com todos os casos de uso, uma vez que o funcionário precisa de informações do cliente para utilizá-los, sendo a única exceção o caso de uso **Gerenciar Veículos**, manipulado exclusivamente pelo funcionário.
- **Funcionário** – Este ator representa os funcionários que atendem os clientes da empresa.

**uc Modelo de Casos de Uso - Sistema de Locação de Veículos**



*Figura 3.24 – Diagrama de Casos de Uso – Sistema de Controle de Aluguel de Carros.*

A seguir, descreveremos os casos de uso que compõem esse diagrama:

- **Gerenciar Veículos** – Este é um caso de uso secundário que representa o processo de manutenção do cadastro de veículos da empresa.
- **Gerenciar Clientes** – Este é também um processo secundário e representa a manutenção do cadastro de clientes. Sempre que um novo cliente solicitar a locação de um veículo, se este ainda não estiver registrado ou seus dados tiverem sido alterados, o funcionário deverá executar esse caso de uso.
- **Locar Automóvel** – Este caso de uso identifica as etapas necessárias para que um cliente consiga locar um automóvel. É necessário que ele

selecione o veículo que deseja locar e informe por quanto tempo deseja locá-lo, bem como para qual finalidade, além de fornecer um valor de caução para poder alugar o automóvel.

- **Devolver Locação** – Este caso de uso identifica os passos que serão executados quando o usuário devolver o veículo, onde serão registradas a data e a hora de devolução do automóvel, sua quilometragem e se este se encontra nas mesmas condições de quando foi alugado. Nesse processo, o cliente pode ter que pagar o aluguel referente ao período extra que ocupou o veículo ou qualquer dano ou multa sofrida enquanto o utilizava. Por outro lado, ele pode vir a ser resarcido de parte do valor que pagou se tiver ocupado o automóvel por menos tempo.

A tabela 3.17 apresenta a documentação do processo de **Locar Automóvel**.

*Tabela 3.17 – Documentação do Caso de Uso Locar Automóvel*

Nome do Caso de Uso		UC01 – Locar Automóvel
Ator Principal		Funcionário
Atores Secundários		Clientes
Resumo		Este caso de uso descreve as etapas percorridas por um funcionário para realizar a locação de um automóvel
Pré-condições		O cliente precisa estar cadastrado
Pós-condições		Receber valor de caução
Cenário Principal		
Ações do Ator	Ações do Sistema	
1. Selecionar opção de locação de veículos	2. Carregar clientes 3. Carregar veículos disponíveis	
4. Selecionar cliente	6. Apresentar detalhes do automóvel	
5. Selecionar veículo	7. Informar por quanto tempo o veículo será locado e a finalidade da locação (passeio ou negócios)	
	8. Calcular valores da locação e da caução	
9. Fornecer valor da caução	10. Registrar locação 1. O cliente precisa ser maior de idade	

Restrições/Validações	2. O cliente precisa possuir uma CNH válida
<b>Cenário de Exceção I – Cliente Menor de Idade</b>	
Ações do Ator	Ações do Sistema
	1. Informar ao cliente que este não possui idade para locar um veículo 2. Recusar locação
<b>Cenário de Exceção II – CNH inválida</b>	
Ações do Ator	Ações do Sistema
	1. Informar ao cliente que sua CNH não é válida 2. Recusar locação

### 3.22.4 Resolução do Exercício Sistema para Controle de Leilão Via Internet

A figura 3.25 apresenta a solução do exercício referente ao sistema de leilão via internet. Os atores identificados foram:

- **Participante** – Este ator representa as pessoas interessadas em participar do leilão, registrando-se previamente antes de este iniciar, e que, se assim desejarem, podem dar lances e eventualmente arrematar algum item ofertado.
- **Leiloeiro** – Este ator representa o funcionário responsável por manipular o cadastro de leilões, bem como registrar os itens a serem leiloados, determinando seus lances mínimos. Além disso, esse ator representa o funcionário que coordenará o leilão, dando-lhe início, oferecendo os itens, controlando os lances e os participantes que os fizerem, bem como anunciando os itens que forem arrematados e por quem. É também responsabilidade desse ator encerrar o leilão.

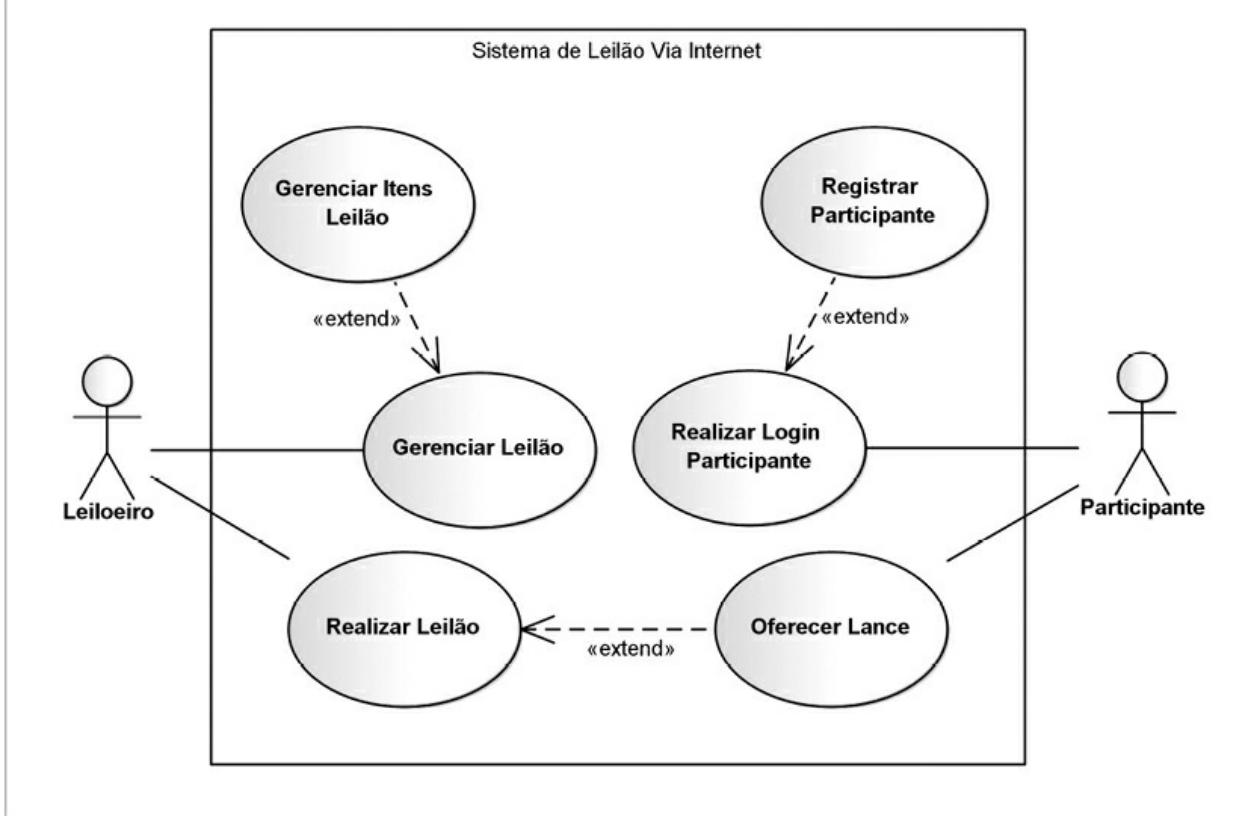


Figura 3.25 – Diagrama de Casos de Uso – Sistema de Controle de Leilão Via Internet.

A seguir, descreveremos os casos de uso que compõem esse diagrama:

- **Gerenciar Leilão** – Este é um caso de uso secundário que representa o processo para a manutenção dos leilões agendados, podendo-se incluir um novo leilão ou modificar a data de início de um leilão já registrado, por exemplo. Observe que há um relacionamento de extensão entre esse caso de uso e o caso de uso **Gerenciar Itens Leilão**, uma vez que é necessário, primeiro, consultar um leilão para, depois, dar manutenção aos itens a serem leiloados nele.
- **Realizar Login Participante** – Este é o caso de uso que estabelece as etapas para que um participante logue-se no sistema e possa participar do leilão.
- **Registrar Participante** – Este é um caso de uso secundário que define os passos percorridos para que o cliente se registre, caso não possua um nome-login nem uma senha para participar de um leilão.

- **Realizar Leilão** – Este é um caso de uso primário que representa as etapas percorridas pelo leiloeiro para abrir o leilão e ofertar seus itens. Observe que há uma associação de extensão com o caso de uso **Oferecer Lance**, já que um participante pode ou não dar lances para um determinado item.
- **Oferecer Lance** – Este caso de uso primário representa o processo por meio do qual um participante pode ofertar um lance para um determinado item em leilão.

A tabela 3.18 apresenta a documentação do processo de Realizar Leilão.

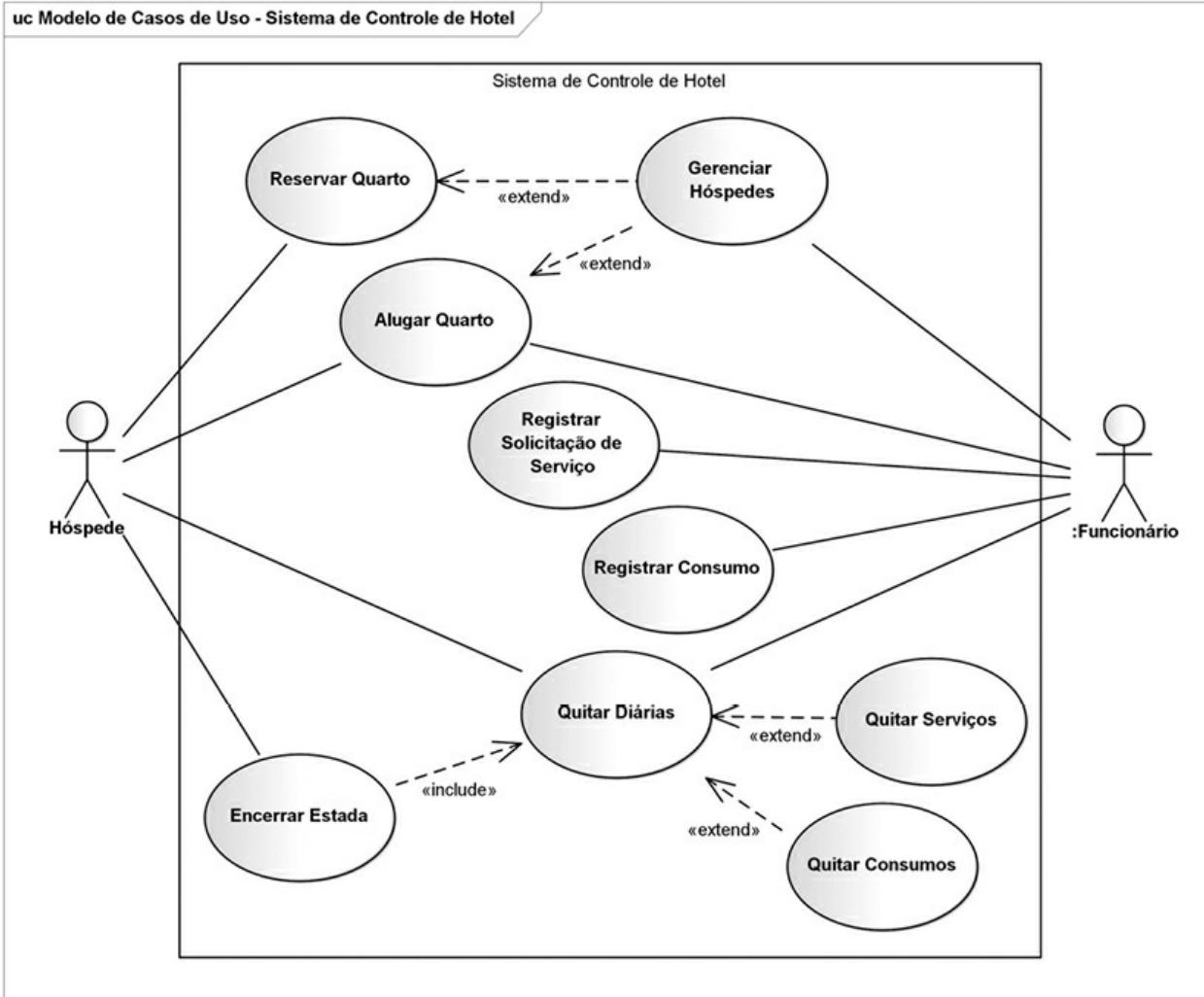
*Tabela 3.18 – Documentação do Caso de Uso Realizar Leilão*

Nome do Caso de Uso	UC01 – Realizar Leilão	
Ações do Ator	Ações do Sistema	
Ator Principal	Leiloeiro	
Resumo	Este caso de uso descreve as etapas percorridas por um leiloeiro para realizar um leilão	
Pré-condições	Deve haver ao menos um leilão não encerrado	
Pós-condições		
<b>Cenário Principal</b>		
1. O leiloeiro seleciona a opção realizar leilão	2. Apresentar leilões em aberto	
3. O leiloeiro seleciona e inicia um leilão	4. Apresentar itens do leilão	
<b>Cenário Alternativo I – Selecionar e Anunciar Itens</b>		
1. O leiloeiro escolhe um item e o anuncia no sistema	2. Apresentar o item a todos os participantes do leilão	
<b>Cenário Alternativo II – Oferta de Lance Recebida</b>		
1. Anunciar o lance recebido, atualizando o valor atual ofertado	2. Registrar item como arrematado, informando o valor pago e o participante que o comprou	
<b>Cenário Alternativo III – Tempo de Ofertas Encerrado com ao Menos um Lance</b>		
1. Anunciar o vencedor que ofereceu o lance mais alto	2. Registrar item como arrematado, informando o valor pago e o participante que o comprou	

<b>Cenário Alternativo IV – Tempo de Ofertas Encerrado sem Lances</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
	1. Encerrar a oferta do item 2. Registrar o item como não arrematado
<b>Cenário Alternativo V – Leilão Encerrado</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
1. Selecionar o encerramento do leilão	2. Registrar leilão como encerrado
<b>Cenário de Exceção – Não há participantes logados</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
Restrições/Validações	1. Informar ao leiloeiro que não é possível iniciar o leilão 1. Caso um item não receba nenhum lance, não será arrematado 2. Os participantes não são obrigados a realizar lances

### **3.22.5 Resolução do Exercício Sistema de Controle de Hotelaria**

A figura 3.26 apresenta a solução do exercício referente ao sistema de controle de hotelaria.



*Figura 3.26 – Diagrama de Casos de Uso – Sistema de Controle de Hotelaria.*  
Os atores identificados foram:

- **Hóspede** – Este ator representa, como o próprio nome indica, os hóspedes que se hospedam no hotel.
- **Funcionário** – Este ator representa os funcionários responsáveis por reservar e alugar quartos, bem como manter o cadastro de hóspedes, realizar serviços e quitar diárias.

A seguir, descreveremos os casos de uso que compõem este diagrama:

- **Reservar Quarto** – Este caso de uso representa o processo pelo qual um hóspede reserva um ou mais quartos no hotel.
- **Alugar Quarto** – Este caso de uso representa o processo pelo qual um hóspede aluga um ou mais quartos no hotel.

- **Gerenciar Hóspedes** – Este caso de uso representa as etapas necessárias para que um funcionário efetue a manutenção do cadastro de hóspedes. Observe que existe uma associação de extensão entre esse caso de uso e os casos de uso **Reservar Quarto** e **Alugar Quarto**, uma vez que caso o hóspede não possua cadastro no hotel ou seus dados tenham sofrido alguma alteração desde a última reserva ou aluguel, será necessário registrar ou alterar seu cadastro.
- **Registrar Solicitação de Serviço** – Este caso de uso apresenta as etapas necessárias para que um funcionário registre a solicitação de algum dos serviços oferecidos pelo hotel por um hóspede.
- **Registrar Consumo** – Este caso de uso representa o processo por meio do qual são registrados os possíveis consumos de frigobar por um hóspede.
- **Quitar Diárias** – As regras do hotel exigem que as diárias sejam pagas semanalmente. Esse caso de uso apresenta os passos percorridos por um hóspede para quitar suas diárias. A quitação das diárias não necessariamente significa o encerramento da estada do hóspede. Na quitação das diárias, é necessário quitar também quaisquer serviços solicitados ou consumo do frigobar. Por esse motivo, existe a associação de extensão entre esse caso de uso e os casos de uso **Quitar Serviços** e **Quitar Consumos**, em que estão detalhados os passos para que o hóspede possa saldar essas dívidas.
- **Encerrar Estada** – Este caso de uso representa o processo pelo qual um hóspede encerra sua estada no estabelecimento. Observe que há um relacionamento de inclusão com o caso de uso **Quitar Diárias**, uma vez que, ao encerrar sua estada, o hóspede deverá quitar as diárias ainda não pagas.

A tabela 3.19 apresenta a documentação do processo de **Quitar Diárias**.

*Tabela 3.19 – Documentação do Caso de Uso Quitar Diárias*

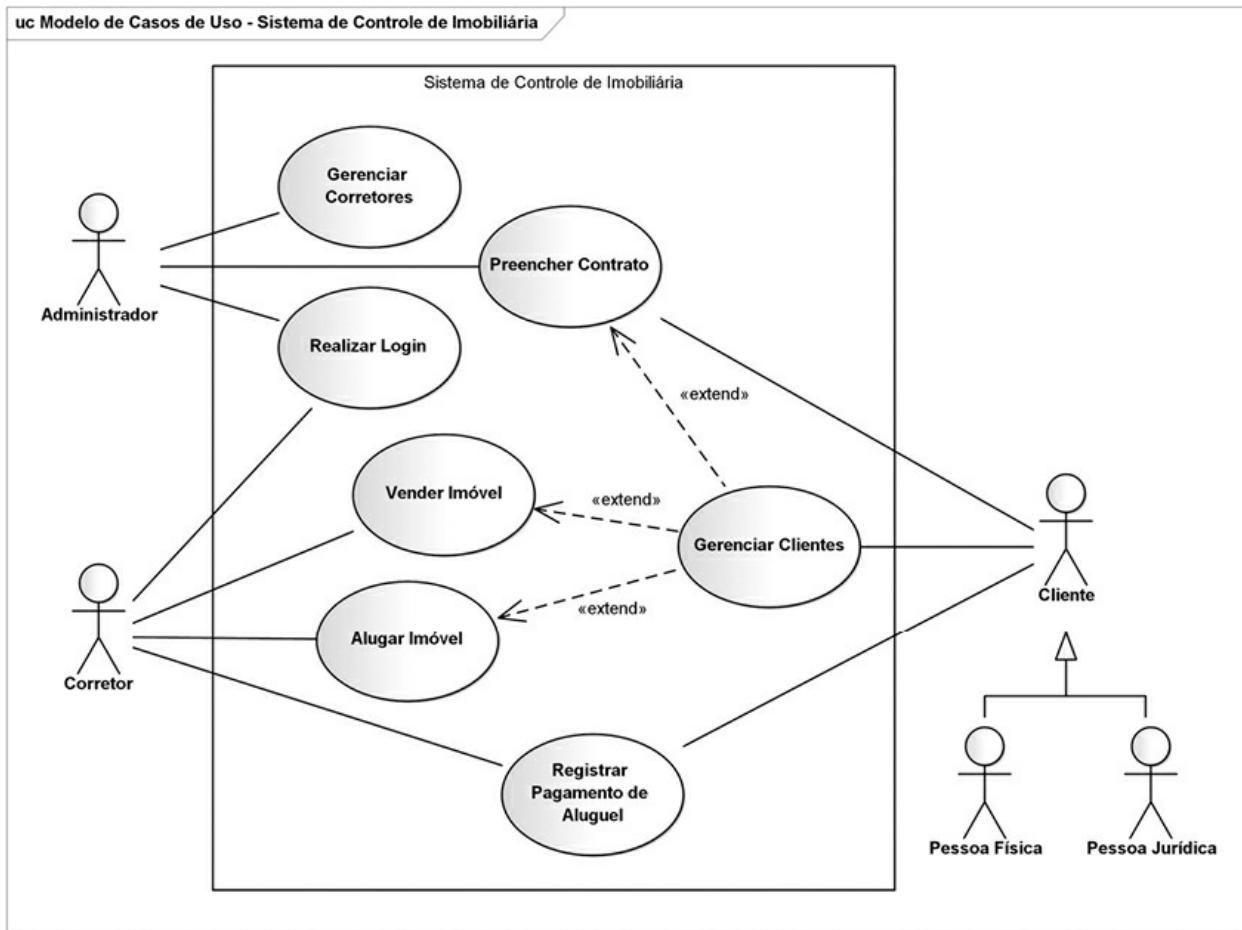
Nome do Caso de Uso	UC01 – Quitar Diárias
Ator Principal	Funcionário
Atores Secundários	Hóspedes
Resumo	Este caso de uso descreve as etapas percorridas por um funcionário durante o processo de quitação de diárias por um hóspede
Pré-condições	

Pós-condições		Cenário Principal	
Ações do Ator	Ações do Sistema	Ações do Ator	Ações do Sistema
1. Informar hóspede e solicitar a quitação de diárias	2. Selecionar e apresentar diárias	3. Selecionar quitar diárias	4. Quitar diárias
<b>Cenário Alternativo I – Quitar Serviços</b>		<b>Cenário Alternativo II – Quitar Consumo</b>	
Ações do Ator	Ações do Sistema	Ações do Ator	Ações do Sistema
1. Caso tenham sido solicitados serviços, executar o caso de uso Quitar Serviços		1. Caso tenha havido consumo de itens do frigobar, executar o caso de uso Quitar Consumo	
Restrições/Validações			

### 3.22.6 Resolução do Exercício Sistema de Controle de Imobiliária

A figura 3.27 apresenta a solução do exercício referente ao sistema de controle de imobiliária. Os atores identificados foram:

- **Administrador** – Este ator representa o funcionário responsável por gerenciar a corretora de imóveis.
- **Corretor** – Este ator representa os corretores que trabalham para a imobiliária, responsáveis por intermediar os processos de aluguel e venda de imóveis, bem como o pagamento de aluguéis.
- **Cliente** – Este ator representa os clientes da imobiliária que podem tanto ser donos de imóveis sob aluguel ou venda como pessoas que compram ou alugam esses imóveis. Observe que existem duas especializações a partir do ator Cliente, chamadas Pessoa Física e Pessoa Jurídica, indicando que o cliente pode tanto constituir-se de uma pessoa física quanto de uma pessoa jurídica.



*Figura 3.27 – Diagrama de Casos de Uso – Sistema de Controle de Imobiliária.*

A seguir, descreveremos os casos de uso que compõem esse diagrama:

- **Gerenciar Corretores** – Este caso de uso representa o cadastro simples por meio do qual são registrados os dados dos corretores que trabalham para a corretora.
- **Preencher Contrato** – Este caso de uso representa o processo por meio do qual um contrato de venda ou locação é firmado entre o dono do imóvel e a locadora.
- **Realizar Login** – Este caso de uso representa o processo por meio do qual o administrador ou os corretores se logam no sistema.
- **Vender Imóvel** – Este caso de uso representa o processo por meio do qual um imóvel à venda passa a ser definido como tendo sido vendido. Esse processo inclui o pagamento ao dono do imóvel do valor pago, descontada a comissão da imobiliária.

- **Alugar Imóvel** – Este caso de uso representa o processo por meio do qual um imóvel posto para alugar passa a ser definido como tendo sido locado. Esse processo inclui o pagamento de uma comissão ao corretor que intermediou o aluguel.
- **Registrar Pagamento de Aluguel** – Este caso de uso representa o processo por meio do qual um cliente paga um ou mais meses de aluguel. O processo inclui o depósito do aluguel ao dono do imóvel, descontando a comissão da imobiliária.
- **Gerenciar Clientes** – Este caso de uso representa o processo para registrar ou alterar o cadastro dos clientes da imobiliária, que podem tanto ser donos de imóveis como pessoas que compram ou alugam imóveis. Os clientes da imobiliária podem ser tanto pessoas físicas como jurídicas.

As tabelas 3.20 a 3.23 apresentam a documentação dos casos de uso primários desse modelo.

*Tabela 3.20 – Documentação do Caso de Uso Preencher Contrato*

Nome do Caso de Uso		UC01 – Preencher Contrato
Ator Principal		Administrador
Atores Secundários		Donos dos Imóveis
Resumo		Este caso de uso descreve as etapas percorridas pelo administrador durante o preenchimento do contrato de aluguel ou venda de um imóvel
Pré-condições		O administrador precisar estar logado no sistema
Pós-condições		
Cenário Principal		
Ações do Ator	Ações do Sistema	
1. Informar se o contrato é de aluguel ou venda de um imóvel 2. Informar o tipo do imóvel (apartamento, casa, terreno etc.) 3. Informar o endereço do imóvel 4. Informar o dono do imóvel 5. Informar o valor do aluguel/venda do imóvel		

6. Confirmar dados do contrato	7. Registrar imóvel 8. Registrar contrato
<b>Cenário Alternativo – Dono do Imóvel não Registrado</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
Executar caso de uso Gerenciar Clientes	
<b>Restrições/Validações</b>	
<i>Tabela 3.21 – Documentação do Caso de Uso Vender Imóvel</i>	
<b>Nome do Caso de Uso</b>	<b>Vender Imóvel</b>
Ator Principal	Corretor
Atores Secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um corretor durante o processo de venda de um imóvel
Pré-condições	O corretor precisar estar logado no sistema
Pós-condições	
<b>Cenário Principal</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
1. Selecionar o tipo de imóvel	2. Apresentar os imóveis do tipo escolhido ofertados para venda
3. Selecionar o imóvel a ser vendido	4. Apresentar dados do imóvel 5. Listar pessoas registradas
6. Selecionar comprador	
7. Informar os valores relativos a taxas decorrentes da transação	
8. Informar o valor efetivamente pago e confirmar	9. Calcular o percentual da comissão da imobiliária e do corretor 10. Registrar venda e taxas pagas
<b>Cenário Alternativo – Comprador não Registrado</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
Executar caso de uso Gerenciar Clientes	
<b>Restrições/Validações</b>	

*Tabela 3.22 – Documentação do Caso de Uso Alugar Imóvel*

<b>Nome do Caso de Uso</b>	<b>Alugar Imóvel</b>
Ator Principal	Corretor

<b>Atores Secundários</b>	
Resumo	Este caso de uso descreve as etapas percorridas por um corretor durante o processo de locação de um imóvel
Pré-condições	O corretor precisar estar logado no sistema
Pós-condições	
<b>Cenário Principal</b>	
Ações do Ator	Ações do Sistema
1. Selecionar o tipo de imóvel	2. Apresentar os imóveis do tipo escolhido ofertados para locação
3. Selecionar o imóvel a ser alugado	4. Apresentar dados do imóvel 5. Listar pessoas registradas
6. Informar locador	
7. Informar o valor da locação	
8. Informar os dados dos fiadores e confirmar	9. Calcular o valor da comissão do corretor 10. Registrar locação
<b>Cenário Alternativo I – Não Há Fiadores</b>	
Ações do Ator	Ações do Sistema
1. Estabelecer valor da caução e confirmar	2. Registrar locação
<b>Cenário Alternativo II – Locador não Registrado</b>	
Ações do Ator	Ações do Sistema
Executar caso de uso Gerenciar Clientes	
<b>Restrições/Validações</b>	

*Tabela 3.23 – Documentação do Caso de Uso Registrar Pagamento de Aluguel*

Nome do Caso de Uso	Registrar Pagamento de Aluguel
Autor Principal	Corretor
Atores Secundários	Locadores
Resumo	Este caso de uso descreve as etapas percorridas por um corretor para registrar o pagamento de aluguel de um imóvel
Pré-condições	O corretor precisar estar logado no sistema
Pós-condições	

<b>Cenário Principal</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
	1, Listar os locatários ativos da imobiliária
2. Selecionar o locatário	3. Apresentar os imóveis alugados pelo locatário
4. Selecionar o imóvel locado	5. Apresentar dados do imóvel e listar meses de aluguel a pagar
6. Selecionar mês(es) de aluguel a quitar	7. Apresentar dados do aluguel
8. Marcar como quitado	9. Calcular comissão imobiliária 10. Registrar pagamento
<b>Cenário Alternativo I – Há Meses de Aluguel em Atraso</b>	
<b>Ações do Ator</b>	<b>Ações do Sistema</b>
	1. Calcular a multa por atraso
	2. Apresentar o(s) mês(es) de aluguel com o valor recalculado em cor de destaque
<b>Restrições/Validações</b>	

## CAPÍTULO 4

# Diagrama de Classes

O diagrama de classes é um dos mais importantes e utilizados da UML. Seu principal enfoque está em permitir a visualização das classes que compõem o sistema com seus respectivos atributos e métodos, bem como em demonstrar como as classes do diagrama se relacionam, complementam e transmitem informações entre si. Esse diagrama apresenta uma visão estática de como as classes estão organizadas, preocupando-se em como definir a estrutura lógica delas. O diagrama de classes serve ainda como apoio para a construção da maioria dos outros diagramas da linguagem UML.

Basicamente, o diagrama de classes é composto de suas classes e associações existentes entre elas, ou seja, os relacionamentos entre as classes. Alguns métodos de desenvolvimento de software, como o Processo Unificado, recomendam que se utilize o diagrama de classes ainda durante a fase de análise, produzindo-se um modelo conceitual a respeito das informações necessárias ao software. No modelo conceitual, o engenheiro preocupa-se apenas em representar as informações que o software necessitará, em termos de classes e seus atributos, bem como as associações entre as classes, não modelando características como os métodos que as classes poderão conter nessa etapa (os métodos já fazem parte de “como” o software será desenvolvido). Somente na fase de projeto toma-se o modelo conceitual do diagrama de classes e produz-se o modelo de domínio, que já enfoca a solução do problema. Os métodos necessários às classes são descobertos a partir da modelagem de diagramas de interação, como o diagrama de sequência, que será estudado nos próximos capítulos.

Assim, pode-se perceber que pode haver vários diagramas ou modelos de classe associados a um mesmo projeto de software com enfoques diferentes. Pode-se produzir um modelo conceitual e mais tarde evoluí-lo

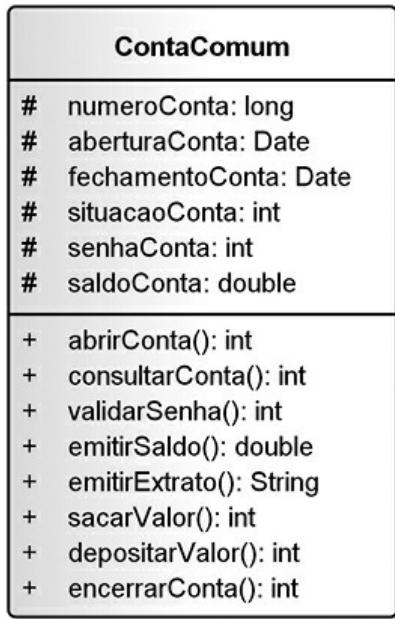
para um modelo de domínio; pode-se produzir um modelo de classes representando o mapeamento das classes conceituais em um modelo lógico de banco de dados relacional; pode-se criar um modelo de classes para representar o modelo navegacional de um sistema para web, entre outras possibilidades. Além disso, um projeto pode ser dividido em subsistemas e cada um deles pode possuir seus diagramas de classes particulares. Essas aplicações do diagrama de classes dependerão do enfoque e do objetivo com que será aplicado. Todavia, é preciso destacar que não é obrigatório produzir nenhum desses artefatos. Recomendamos, porém, que sejam produzidos ao menos um modelo conceitual e um modelo de domínio para todo projeto de software.

É importante destacar que alguns autores se referem ao que chamamos de modelo conceitual como “modelo de domínio” ou “modelo de análise” e ao que chamamos de modelo de domínio como “modelo de projeto”, mas preferimos a nomenclatura utilizada aqui por acreditarmos ser mais lógica.

## 4.1 Atributos e Métodos

Classes costumam ter atributos que, como já explicado no capítulo 2, armazenam os dados dos objetos da classe. Classes também costumam possuir métodos, também chamados operações, que são as funções que uma instância da classe pode executar. Os valores dos atributos podem variar de uma instância para outra, ao passo que os métodos são idênticos para todas as instâncias de uma classe específica.

Embora os métodos sejam declarados no diagrama de classes, identificando os possíveis parâmetros que são por eles recebidos e os possíveis valores por eles retornados, o diagrama de classes não se preocupa em definir as etapas que tais métodos deverão percorrer quando forem chamados, sendo essa uma função atribuída a outros diagramas, como o diagrama de atividade, que será analisado nos capítulos seguintes. A figura 4.1 apresenta um exemplo de classe contendo atributos e métodos.



*Figura 4.1 – Classe.*

Como foi explicado no capítulo 2, uma classe, na linguagem UML, é representada como um retângulo com até três divisões descritas a seguir:

- A primeira contém a descrição ou nome da classe, que, nesse exemplo, é **ContaComum**.
- A segunda armazena os atributos e seus tipos de dados (o formato que os dados devem ter para serem armazenados em um atributo). No exemplo da figura 4.1, a classe **ContaComum** contém os atributos **numeroConta**, do tipo **long**; **aberturaConta** e **fechamentoConta**, do tipo **Date** (este não é um tipo primitivo e se refere a uma classe); **situacaoConta** e **senhaConta**, do tipo **int**; e **saldoConta**, do tipo **double**.
- Finalmente, a terceira divisão lista os métodos da classe. Nesse exemplo, a classe **ContaComum** contém os métodos **abrirConta**, **consultarConta**, **validarSenha**, **emitirSaldo**, **emitirExtrato**, **sacarValor**, **depositarValor** e **encerrarConta**.

Os símbolos de sustenido (#) e mais (+) na frente dos atributos e métodos representam a visibilidade destes, o que determina quais objetos de quais classes podem utilizar o atributo ou o método em questão. Os tipos de visibilidade foram explicados no capítulo 2.

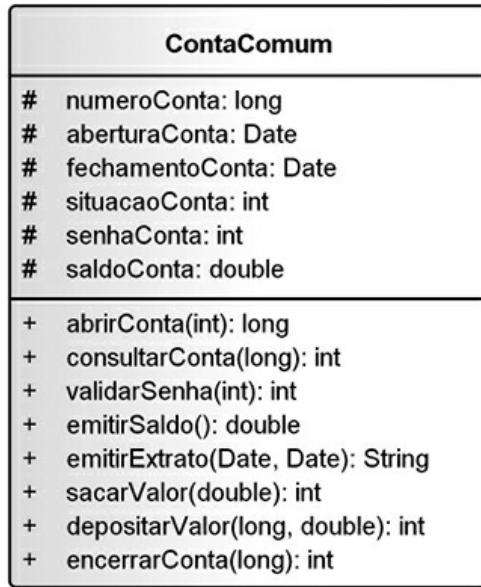
Não é realmente obrigatório que uma classe apresente as três divisões, pois pode haver classes que não tenham atributos nem contenham métodos, ou pode acontecer ainda que seus atributos e métodos não precisem ser apresentados no diagrama, já que é recomendado apresentar apenas atributos relevantes ao diagrama para evitar, por exemplo, tornar o diagrama muito poluído. Assim, é possível encontrar classes com somente duas divisões ou mesmo com apenas uma, no caso aquela que contém a descrição da classe, porque esta é a única obrigatória.

Métodos podem receber valores como parâmetros e retornar valores que podem ser o resultado produzido pela execução do método ou simplesmente um valor representado se o método foi realizado com sucesso ou não, por exemplo. Nessa classe podemos perceber que o retorno do método `abrirConta` é um `long`, que conterá o número da nova conta gerada. Por sua vez, o retorno dos métodos `consultarConta`, `validarSenha`, `sacarValor`, `depositarValor` e `encerrarConta` é um inteiro (`int`), e esse valor é utilizado para determinar se o método foi concluído com sucesso ou não. Por padrão, o retorno 1 significa verdadeiro (ou sucesso) e 0, falso (ou fracasso).

Já os métodos `emitirSaldo` e `emitirExtrato` retornam, respectivamente, um `double` e uma `String`, que contêm o resultado da execução desses métodos. Para o método `abrirConta`, se o valor retornado for igual a 1, significará que o método foi concluído com sucesso e que uma nova conta foi aberta. Já se o retorno for igual a zero, saberemos que o método não foi concluído da forma esperada e que algum problema ocorreu.

Na figura 4.1, os métodos foram apresentados sem o detalhamento de quais argumentos (parâmetros) eles deveriam receber (a lista de argumentos de um método com seu valor de retorno é chamada de assinatura da operação). Esse detalhamento é opcional, o que foi feito propositadamente para explicar os diagramas que serão apresentados no decorrer do capítulo. Alguns métodos podem ter muitos parâmetros e o detalhamento destes em um diagrama que contenha muitas classes tornará esse diagrama muito extenso e será difícil visualizá-lo claramente como um todo, porque as classes ficarão largas. Assim, é melhor, quando se trata de apresentar um diagrama de classes composto de muitas classes, apresentar somente o nome dos métodos da classe, sem especificar os argumentos que

ele receberá. O detalhamento dos métodos de cada classe poderá ser feito individualmente em um diagrama separado, conforme apresentado na figura 4.2.



*Figura 4.2 – Detalhamento das Assinaturas das Operações.*

Ao examinarmos a figura 4.2, podemos perceber que os métodos **abrirConta** e **validarSenha** recebem um inteiro como parâmetro; o método **consultarConta**, um **long**; **sacarValor**, um **double**; e **depositarValor**, um **long** e um **double**. Já os outros métodos não recebem argumento algum. A partir do modelo da classe **contaComum** apresentada na figura 4.2, é possível gerar o código correspondente a ele. A seguir, será apresentado o código correspondente a essa classe implementado em Java:

```
public class ContaComum {  
    protected long numeroConta;  
    protected Date aberturaConta;  
    protected Date fechamentoConta;  
    protected int situacaoConta;  
    protected int senhaConta;  
    protected double saldoConta;  
  
    public ContaComum(){  
    }  
}
```

```

public long abrirConta(int senha){
    return 0;
}

public int consultarConta(long numeroConta){
    return 0;
}

public int validarSenha(int senha){
    return 0;
}

public double emitirSaldo(){
    return 0;
}

public String emitirExtrato(Date dataInicial, Date dataFinal){
    return "";
}

public int sacarValor(double valor){
    return 0;
}

public int depositarValor(long numeroConta, double valor){
    return 0;
}

public int encerrarConta(){
    return 0;
}

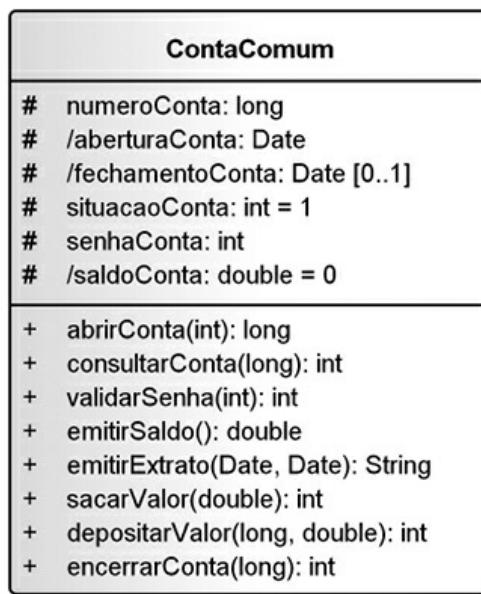
```

Obviamente, esse código é apenas um esqueleto da classe. Como se pode perceber, não há nenhum detalhamento de como os métodos desempenharão sua função. Além disso, será preciso criar uma classe chamada **Date** para que o compilador reconheça os atributos desse tipo. Observe que o valor de retorno em Java é declarado antes do nome do método, enquanto na UML é definido depois do nome do método e dos parâmetros que ele receberá.

Os atributos de uma classe podem ainda ter características extras, entre as

quais podemos citar valores iniciais, multiplicidade e se o atributo é derivado, ou seja, se seus valores são produzidos por meio de algum tipo de cálculo, conforme apresentado na figura 4.3.

Ao estudarmos esse exemplo, podemos verificar que o valor inicial dos atributos **situacaoConta** e **saldoConta** será, respectivamente, 1 e 0 quando da instanciação de um objeto dessa classe durante a abertura de uma nova conta. Dessa forma, sempre que uma nova conta for aberta, sua situação inicial terá o valor 1 (está ativa) e o seu saldo permanecerá com valor 0 até que um depósito seja realizado.



*Figura 4.3 – Detalhamento dos Atributos.*

Pode-se perceber também que o atributo **fechamentoConta**, após a definição de seu tipo (**Date**), contém os valores [0..1]. Isso é chamado multiplicidade e, nesse contexto, significa que existirão, no mínimo, nenhuma (0) e, no máximo, uma (1) data de encerramento para a conta, uma vez que a conta pode estar aberta ainda, portanto não poderá ter uma data de encerramento e, se tiver sido encerrada, não poderá ter mais do que uma.

Finalmente, os atributos **aberturaConta**, **encerramentoConta** e **saldoConta** têm uma barra (/) antes de seus nomes, significando que os valores desses atributos sofrem algum tipo de cálculo. No caso das datas, quando for realizada a operação de abertura de conta, o valor da data de

abertura será tomado da data do sistema, o mesmo ocorrendo com o valor da data de encerramento quando do encerramento da conta. A rigor, não necessariamente isso poderia ser considerado um cálculo, e, sim, uma simples atribuição. Alguns poderiam considerar isso como o valor inicial dos atributos. No entanto, principalmente no caso da data de encerramento, que será deixada indefinida até que a conta seja encerrada, isso não seria verdadeiro, e o valor desse atributo seria definido em uma operação posterior à criação do objeto. Já no caso do saldo, precisa ser recalculado sempre que uma operação de saque ou depósito for realizada.

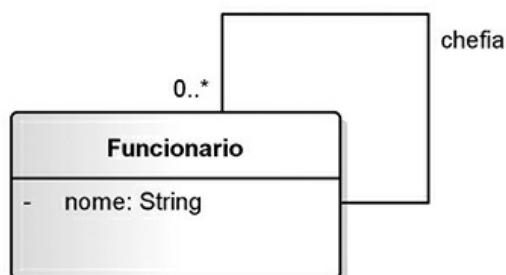
## 4.2 Relacionamentos ou Associações

As classes costumam ter relacionamentos entre si, chamados associações, que permitem que elas compartilhem informações entre si e colaborem para a execução dos processos executados pelo sistema. Uma associação descreve um vínculo que ocorre normalmente entre os objetos de uma ou mais classes.

As associações são representadas por linhas ligando as classes envolvidas. Tais linhas podem ter nomes ou títulos para auxiliar a compreensão do tipo de vínculo estabelecido entre os objetos das classes envolvidas nas associações. Há outras informações que uma associação poderá conter, conforme será visto ao longo do capítulo, na medida em que serão apresentadas as diversas formas de relacionamento possíveis em um diagrama de classes.

### 4.2.1 Associação Unária ou Reflexiva

Este tipo de associação ocorre quando existe um relacionamento de um objeto de uma classe com objetos da mesma classe. Um exemplo de associação unária pode ser observado na figura 4.4.



#### *Figura 4.4 – Associação Unária.*

Ao examinarmos a figura 4.4, é possível perceber que a única classe do exemplo tem o nome **Funcionario** e possui, como único atributo, o nome do funcionário. Observe que uma linha intitulada “chefia” parte da classe **Funcionario** e atinge a própria classe. Essa linha representa a associação existente entre os objetos dessa classe. O título ou nome da associação não é realmente obrigatório, mas seu uso é recomendado para auxiliar a compreender o que a associação representa.

Normalmente é escolhido um verbo ou uma expressão verbal como título da associação. Neste caso, foi escolhido o termo “chefia”, porque, nesse exemplo, um funcionário pode ser chefe de outros funcionários. O chefe de um funcionário também é, por sua vez, um funcionário da empresa, portanto também se constitui em uma instância da classe **Funcionario**. Logo, a associação denominada “chefia” indica uma possível relação entre uma ou mais instâncias da classe **Funcionario** com outras instâncias da própria classe **Funcionario**, ou seja, essa associação determina se um funcionário pode ou não chefiar outros funcionários.

Observe que existe outra informação na associação, além de seu próprio nome, representada pelo valor “0..\*”. Essa informação é conhecida como multiplicidade, a qual procura determinar o número mínimo e o máximo de objetos envolvidos em cada extremidade da associação, além de permitir especificar o nível de dependência de um objeto para com os outros envolvidos na associação.

No caso apresentado na figura 4.4, a multiplicidade 0..\* demonstra que um determinado funcionário pode chefiar nenhum (0) ou muitos (\*) funcionários, ou seja, um funcionário pode não chefiar ninguém ou pode chefiar um ou mais funcionários. Percebe-se que só existe multiplicidade em uma das extremidades. Nesse caso, entende-se que a multiplicidade é 1 porque, por default, quando não existe multiplicidade explícita, assume-se que a multiplicidade é “1..1” (no mínimo, 1 e, no máximo, 1) ou simplesmente “1”, significando que um e somente um objeto dessa extremidade da associação relaciona-se com os objetos da outra extremidade. Nesse exemplo, significa que um funcionário pode ou não chefiar outros funcionários, mas um funcionário tem um e apenas um funcionário como chefe imediato. Nada impede, todavia, que se defina

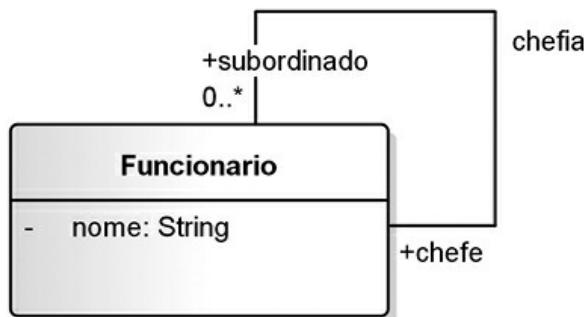
explicitamente a multiplicidade 1 na associação, para tornar mais clara ou explícita a multiplicidade, se assim se desejar.

A tabela 4.1 demonstra alguns dos diversos valores de multiplicidade que podem ser utilizados em uma associação.

*Tabela 4.1 – Exemplos de multiplicidade*

Multiplicidade	Significado
0..1	No mínimo, zero (nenhum) e, no máximo, um. Indica que os objetos das classes associadas não precisam obrigatoriamente estar relacionados, mas se houver relacionamento, indicará que apenas uma instância da classe relaciona-se com as instâncias da outra classe (ou da outra extremidade da associação, se esta for unária).
1..1	Um e somente um. Indica que apenas um objeto da classe relaciona-se com os objetos da outra classe.
0..*	No mínimo, nenhum e, no máximo, muitos. Indica que pode ou não haver instâncias da classe participando do relacionamento.
*	Muitos. Indica que muitos objetos da classe estão envolvidos na associação.
1..*	No mínimo, um e, no máximo, muitos. Indica que há pelo menos um objeto envolvido no relacionamento, podendo haver muitos objetos envolvidos.
3..5	No mínimo, três e, no máximo, cinco. Estabelece que existem pelo menos três instâncias envolvidas no relacionamento, mas podem ser quatro ou cinco as instâncias envolvidas, mas não mais do que isso.

Outra informação que pode ser considerada útil é a definição de papéis, visto que se trata de uma informação extra na associação que pode ajudar a explicar a função de um objeto (o papel que este representa) dentro da associação. Um exemplo do uso de papéis pode ser visto na figura 4.5.



*Figura 4.5 – Associação Contendo Papéis.*

Nesse exemplo, percebemos claramente qual a função de cada objeto envolvido na associação: o objeto na extremidade cuja multiplicidade é 1 executa o papel de chefe, enquanto os objetos na extremidade de

multiplicidade muitos interpretam o papel de subordinados. Como se pode perceber, os papéis desse exemplo têm visibilidade pública, mas podem ter visibilidades protegidas ou privadas. O uso de papéis pode facilitar a compreensão da associação existente, mas nem sempre é necessário e seu uso excessivo pode deixar os diagramas visualmente muito poluídos.

Na verdade, a classe **Funcionario** é uma classe persistente (embora não a tenhamos identificado explicitamente como tal), ou seja, suas instâncias deverão ser preservadas de alguma maneira em disco. No entanto, não é necessário definir atributos do tipo chave primária ou chave estrangeira como no modelo entidade-relacionamento. Como regra geral, cada objeto de uma classe, ao ser instanciado, já tem um código interno implícito, não sendo necessário declarar um atributo exclusivamente para isso. Ao final do capítulo, trataremos da questão de persistência e mapeamento de classes em tabelas em bancos de dados relacionais.

## 4.2.2 Associação Binária

Associações binárias ocorrem quando são identificados relacionamentos entre objetos de duas classes distintas. Em geral, esse tipo de associação é o mais comumente encontrado. A figura 4.6 demonstra um exemplo de associação binária.

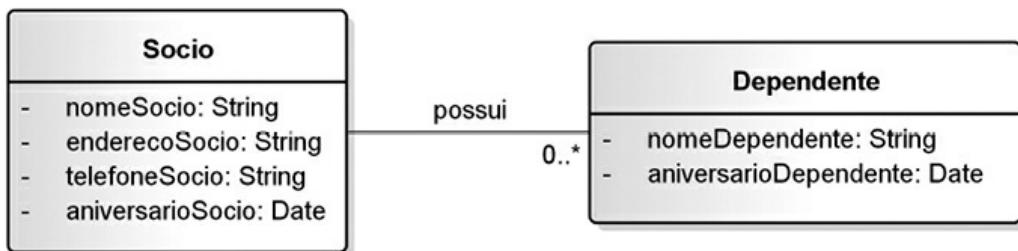


Figura 4.6 – Associação Binária.

Como podemos verificar na figura, um objeto da classe **Socio** pode relacionar-se ou não com instâncias da classe **Dependente**, conforme demonstra a multiplicidade **0..\***, ao passo que se existir um objeto da classe **Dependente**, terá de se relacionar obrigatoriamente com um objeto da classe **Socio**, pois, como não foi definida a multiplicidade na extremidade da classe **Socio**, isto significa que esta é **1..1**, ou seja, um objeto da classe **Dependente** precisa relacionar-se exclusivamente com um objeto da classe

## Socio.

Poderíamos acrescentar outras informações a essa associação, como definir a navegabilidade dela, conforme demonstrado na figura 4.7.

A navegabilidade é representada mais comumente por uma seta em um dos fins da associação, embora seja possível representá-la nos dois sentidos, se isso for considerado necessário. Quando há navegabilidade unilateral, ela determina que os objetos da classe para onde a seta aponta não têm conhecimento dos objetos aos quais estão associados na outra extremidade da associação e, no momento da implementação, pode ser necessário criar um atributo na classe não navegável que retenha uma referência aos objetos da classe navegável (como um vetor, por exemplo).

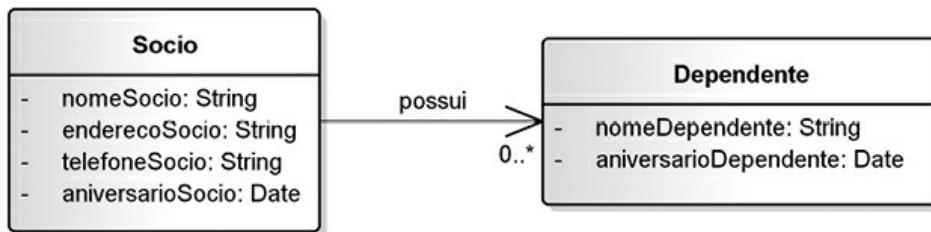


Figura 4.7 – Associação binária com Navegabilidade.

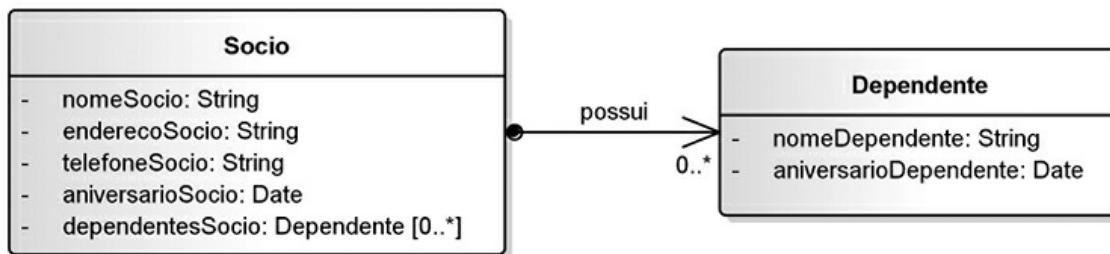
A navegabilidade também determina o sentido em que os métodos poderão ser disparados. Nesse exemplo, um objeto da classe **Socio** poderá disparar métodos em objetos da classe **Dependente**, mas a recíproca não é verdadeira: um objeto da classe **Dependente** não poderá disparar métodos em um objeto da classe **Socio**. A navegabilidade não é obrigatória, mesmo porque, se não houver setas, significará que as informações podem trafegar entre os objetos de todas as classes da associação.

A navegabilidade é uma informação que costuma ser acrescentada na fase de projeto, porém é possível, durante a fase de análise, definir a direção de leitura da associação, representada, em algumas ferramentas CASE, como um triângulo em forma de seta ao lado do nome da associação; em outras ferramentas, porém, não há diferenciação entre a seta de direção de leitura e a de navegabilidade.

Contudo, a direção de leitura e a navegabilidade não são exatamente a mesma coisa. O sentido de leitura tem o objetivo de facilitar a compreensão da associação, enquanto a navegabilidade define em que

sentido os métodos poderão ser disparados. No exemplo da figura 4.7, o uso da direção de leitura nos transmitiria a informação de que a leitura da associação deveria ser feita da seguinte forma: “Uma instância da classe **Socio** possui, no mínimo, nenhuma instância e, no máximo, muitas instâncias da classe **Dependente** e uma instância da classe **Dependente** é possuída por uma e somente uma instância da classe **Socio**”.

Outra informação que pode também ser encontrada em uma associação, muitas vezes atrelada à naveabilidade, é a que define se o fim de uma associação é possuído por uma classe (class-owned end). Um fim de associação possuído é representado por um pequeno ponto no fim da associação e significa que o fim dessa associação é possuído pela classe como um atributo que, em geral, será uma referência aos objetos da outra extremidade (fim) da associação. A figura 4.8 apresenta um exemplo de fim de associação possuída.



*Figura 4.8 – Fim de Associação Possuída (Class-Owned End).*

Nesse exemplo, o ponto encostado à classe **Socio** determina que o fim da associação pertence a essa classe. Sendo assim, acrescentou-se um atributo chamado **dependentesSocio** que poderá conter nenhuma (**0**) ou muitas (**\***) referências a objetos da classe **Dependente**. Isto não era necessariamente obrigatório, já que a própria representação gráfica do fim de associação possuído já determinaria isso. Contudo, neste exemplo, optamos por representar explicitamente esse atribuído com o objetivo de ilustrar que, quando a classe for implementada, deverá possuir um atributo para se referenciar a objetos da outra classe à qual está associada.

### 4.2.3 Associação Ternária ou N-ária

Associações ternárias ou n-árias conectam objetos de mais de duas classes. São representadas por um losango para onde convergem todas as ligações

da associação. A figura 4.9 apresenta um exemplo de associação ternária.

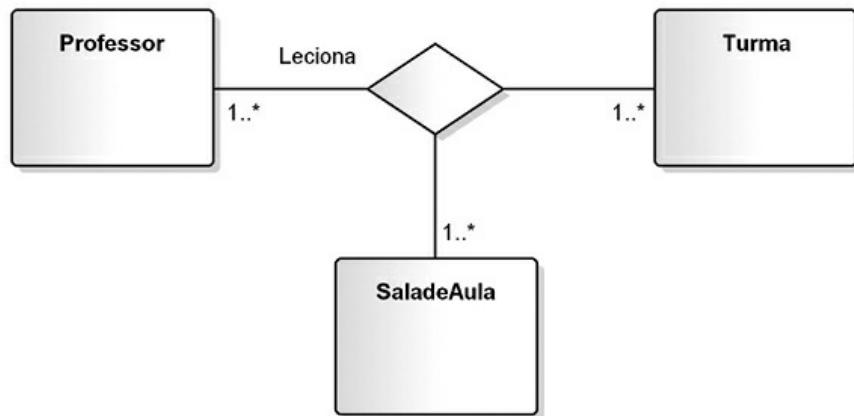


Figura 4.9 – Associação Ternária.

Nessa ilustração, identificamos uma associação que demonstra um fato corriqueiro na maioria das universidades, em que um professor pode lecionar para muitas turmas, uma turma pode ter muitos professores e utilizar muitas salas de aula e um professor, ao lecionar para uma turma específica, pode utilizar mais de uma sala de aula (ele pode ministrar aulas teóricas em salas comuns ou práticas em laboratório).

Assim, podemos ler a associação apresentada na figura 4.9 da seguinte forma: “Um professor leciona para, no mínimo, uma turma e, no máximo, para muitas, uma turma tem, no mínimo, um professor e, no máximo, muitos lecionando para ela e um professor, ao lecionar para uma determinada turma, utiliza, no mínimo, uma sala de aula e, no máximo, muitas”. As associações ternárias são úteis para demonstrar associações complexas. No entanto, convém evitar utilizá-las, pois sua leitura é, por vezes, difícil de ser interpretada, todavia seu uso pode ser inevitável em algumas situações.

#### 4.2.4 Agregação

Agregação é um tipo especial de associação em que se tenta demonstrar que as informações de um objeto (objeto-todo) são complementadas pelas informações contidas em um ou mais objetos no outro fim da associação (chamados objetos-parte). Esse tipo de associação tenta demonstrar uma relação todo/parte entre os objetos associados. O símbolo de agregação difere do de associação por conter um losango no fim da associação que

contém os objetos-todo. A figura 4.10 demonstra um exemplo de agregação.

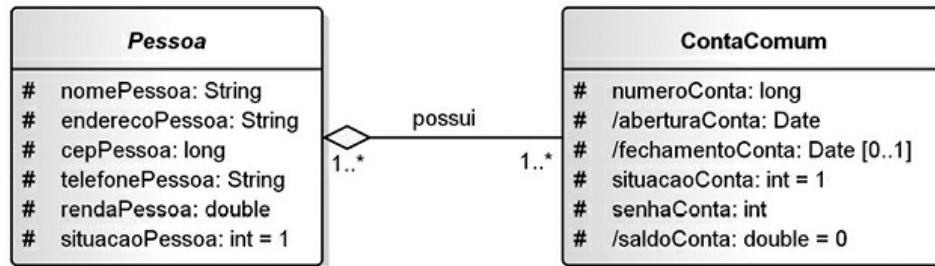


Figura 4.10 – Agregação.

Esse exemplo demonstra uma associação de agregação existente entre uma classe **Pessoa** e uma classe **ContaComum**, o que determina que os objetos da classe **Pessoa** são objetos-todo que precisam ter suas informações complementadas pelos objetos da classe **ContaComum**, que, nessa associação, são objetos-parte. Dessa maneira, sempre que uma pessoa for consultada, além das informações pessoais, serão apresentadas todas as contas que ela possui.

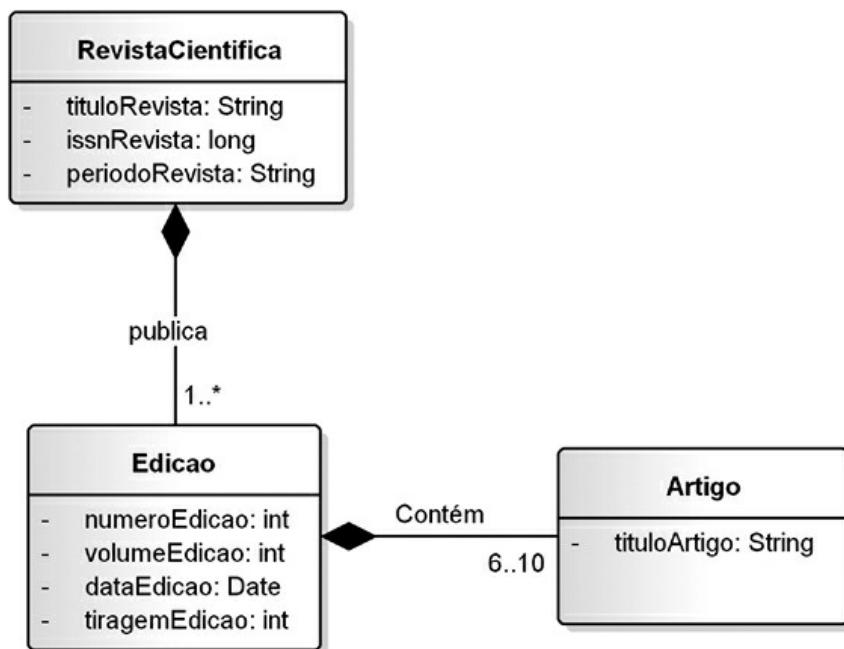
Ao observarmos as multiplicidades dessa associação, percebemos que, da mesma forma que uma pessoa pode possuir muitas contas, uma conta pode ser possuída por muitas pessoas, como no caso de uma conta conjunta. Isso é característico das agregações nas quais os objetos-parte podem ser compartilhados por mais de um objeto-todo.

A associação de agregação pode, em alguns casos, ser substituída por uma associação binária simples, dependendo da visão de quem faz a modelagem. A função principal de uma associação do tipo agregação é identificar a obrigatoriedade de uma complementação das informações de um objeto-todo por seus objetos-parte, quando este for consultado. Em uma associação binária, todavia, essa obrigatoriedade não está explícita.

## 4.2.5 Composição

Uma associação do tipo composição constitui-se em uma variação da agregação, onde é apresentado um vínculo mais forte entre os objetos-todo e os objetos-parte, procurando demonstrar que os objetos-parte têm de estar associados a um único objeto-todo. Em uma composição, os objetos-parte não podem ser destruídos por um objeto diferente do objeto-todo ao

qual estão relacionados. O símbolo de composição diferencia-se graficamente do símbolo de agregação por utilizar um losango preenchido. Da mesma forma que na agregação, o losango deve ficar ao lado do objeto-todo. A figura 4.11 apresenta um exemplo de composição.



*Figura 4.11 – Composição.*

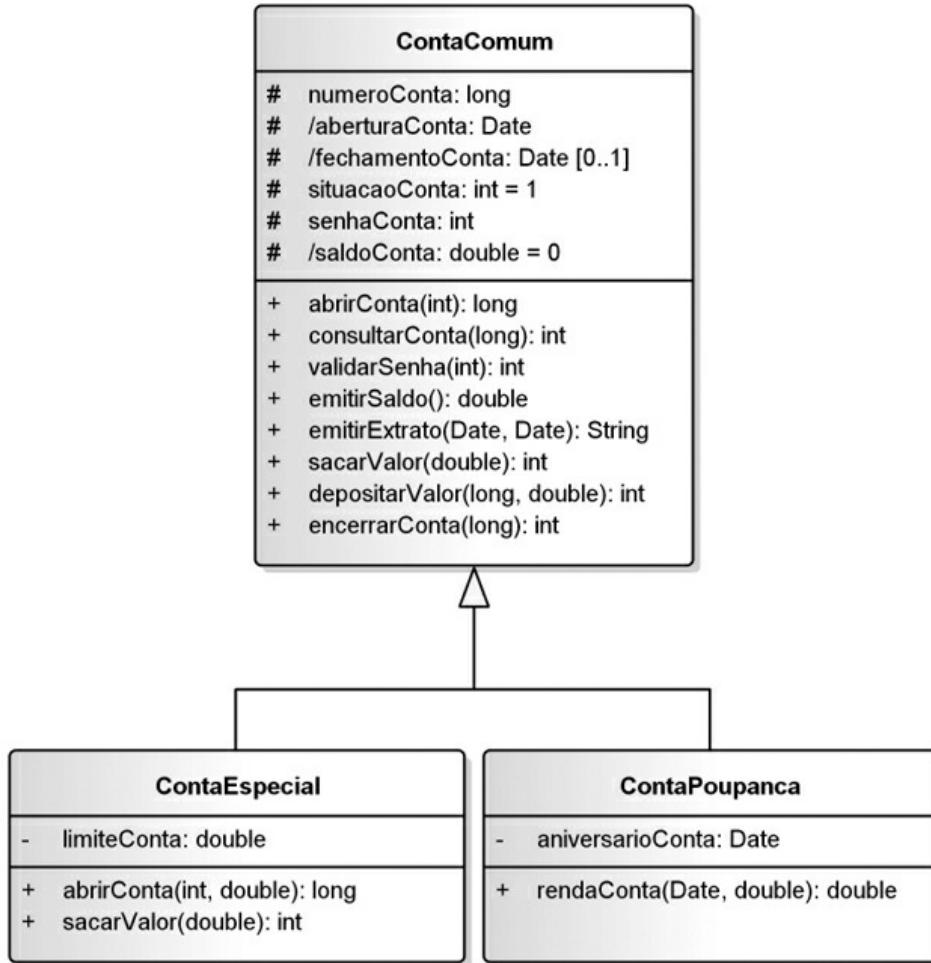
Ao observar a figura, é possível perceber que um objeto da classe **RevistaCientifica** refere-se a, no mínimo, um objeto da classe **Edicao**, podendo se referir a muitos objetos dessa classe, e cada instância da classe **Edicao** relaciona-se única e exclusivamente a uma instância específica da classe **RevistaCientifica**, não podendo relacionar-se a nenhuma outra.

Ainda nesse exemplo, percebemos que um objeto da classe **Edicao** deve se relacionar a, no mínimo, seis objetos da classe **Artigo**, podendo se relacionar com até 10 objetos da já citada classe. Esse tipo de informação torna-se útil como documentação e serve como forma de validação, que impede que uma revista seja publicada sem ter, no mínimo, seis artigos ou mais de 10. No entanto, um objeto da classe **Artigo** refere-se unicamente a um objeto da classe **Edicao**. Isso é também uma forma de documentação, pois uma edição de uma revista científica só deve publicar trabalhos inéditos. Assim, é lógico que não é possível a um mesmo objeto da classe **Artigo** relacionar-se a mais de um objeto da classe **Edicao**.

## **4.2.6 Generalização/Especialização**

Este é um tipo especial de relacionamento, similar à associação de mesmo nome utilizada no diagrama de casos de uso. O objetivo dessa associação é representar a ocorrência de herança entre as classes, identificando as classes-mãe (ou superclasses), chamadas gerais, e classes-filhas (ou subclasses), chamadas especializadas, demonstrando a hierarquia entre as classes e, possivelmente, métodos polimórficos nas classes especializadas.

Esse tipo de relacionamento permite representar classes derivadas a partir de classes mais antigas e, ao mesmo tempo que as novas classes herdam todos os atributos e métodos das classes das quais foram derivadas, é possível adicionar novos atributos e/ou métodos a essas classes, dessa forma especializando-as. Isso permite maior rapidez no desenvolvimento, uma vez que não é necessário adicionar os atributos e métodos já existentes às classes anteriores, apenas os novos atributos ou métodos, o que também impede que erros de codificação sejam cometidos desnecessariamente. Além disso, métodos podem ser redeclarados em uma classe especializada, com o mesmo nome, mas comportando-se de forma diferente, não sendo, portanto, necessário modificar o código-fonte do sistema em relação às chamadas de métodos das classes especializadas, pois o nome do método não mudou, somente foi redeclarado em uma classe especializada e só se comportará de maneira diferente quando for chamado por objetos dessa classe. A figura 4.12 apresenta um exemplo de generalização/especialização. O símbolo de generalização/especialização é o mesmo do diagrama de casos de uso.



*Figura 4.12 – Generalização/Especialização no Diagrama de Classes.*

No exemplo da figura 4.12, tomamos a classe **ContaComum** apresentada na figura 4.3, tornando-a uma classe geral (embora não seja uma classe abstrata), e derivamos duas classes especializadas a partir dela, as classes **ContaEspecial** e **ContaPoupanca**, que herdam suas características.

Além dos atributos e métodos herdados, a classe **ContaEspecial** contém ainda o atributo **limiteConta**, que determina quanto o cliente pode sacar além de seu saldo, e os métodos **abrirConta** e **sacarValor**. Esses métodos são uma redeclaração dos métodos **abrirConta** e **sacarValor** da classe **ContaComum**, pois estes precisam incluir o limite da conta. Já a classe **ContaPoupanca** contém, além dos atributos e métodos herdados, o atributo **aniversarioConta**, que define a data em que a conta renderá juros, e o método **rendaConta**, cuja função é calcular os rendimentos a serem acrescidos ao saldo da conta sempre que esta fizer aniversário.

O leitor poderia achar necessário criar uma classe **Conta** que fosse realmente geral, que não tivesse instâncias e servisse apenas para definir os atributos e métodos comuns a todas as contas, ou seja, uma classe abstrata e, a partir desta, derivar as classes **ContaComum**, **ContaEspecial** e **ContaPoupanca**. Isso não estaria errado, apenas não consideramos necessário fazê-lo, por acreditarmos que todas as características de uma conta comum sejam também contidas nas outras classes. Assim, além de definir as características de uma conta comum, a classe pode servir também como classe geral, a partir da qual possam ser derivadas as classes **ContaEspecial** e **ContaPoupanca**. No caso de realmente ser identificada uma classe abstrata, por convenção, seu nome deve ser escrito em itálico.

#### 4.2.7 Classe Associativa

Classes associativas são necessárias nos casos em que existem atributos relacionados à associação que não podem ser armazenados por nenhuma das classes envolvidas. As classes associativas costumam ser utilizadas principalmente em associações que apresentem multiplicidade muitos (\*) em todas as suas extremidades, porém pode ocorrer que sejam utilizadas em outras situações. A figura 4.13 apresenta um exemplo de classe associativa.

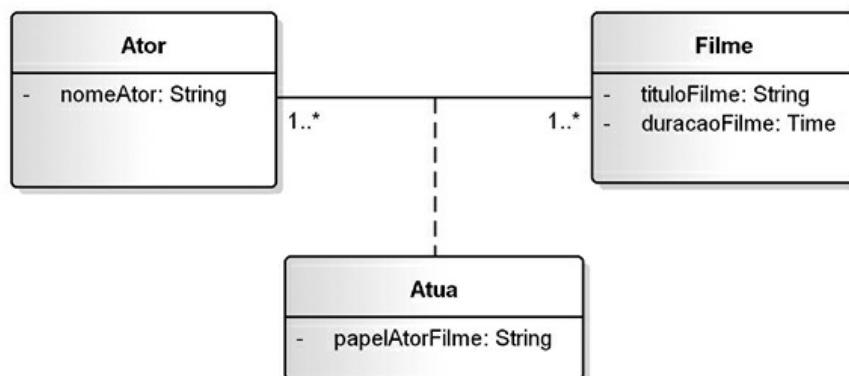


Figura 4.13 – Classe Associativa.

Nesse exemplo, uma instância da classe **Autor** pode se relacionar com muitas instâncias da classe **Filme**, e uma instância da classe **Filme** pode se relacionar com muitas instâncias da classe **Autor**, ou seja, um autor pode atuar em muitos filmes, e um filme pode ter muitos atores atuando nele.

Ocorre que existe a necessidade de saber qual o papel interpretado por um ator em um determinado filme, mas onde armazenar essa informação?

Se fôssemos armazenar todos os papéis que um ator pudesse vir a interpretar, seria necessário incluir na classe Ator atributos que identificassem cada papel que um determinado ator já interpretou e a qual filme cada papel se referiria, bem como estabelecer alguma forma de ligação entre essas duas informações. Além desta não ser a modelagem mais adequada (para dizer o mínimo), tornaria praticamente desnecessária a existência de uma classe Filme e tornaria complexo pesquisar todos os atores que participaram de um determinado filme. Já se fôssemos criar atributos para armazenar cada papel pertencente a um filme específico, seria necessário criar atributos para identificar também o ator que o interpretou e ligar essas duas informações de alguma forma, o que tornaria desnecessário identificar uma classe para armazenar atores. Esse tipo de abordagem (bastante inadequada) também impediria, ou pelo menos tornaria bastante difícil, pesquisar todos os papéis que um determinado ator já interpretou. Para solucionar esse problema, é mais correto criar uma classe para guardar essa informação, representada aqui pela classe **Atua**, que conterá o atributo **papelAtorFilme**, referindo-se este exclusivamente a um ator específico atuando em um determinado filme. Uma classe associativa pode perfeitamente ter métodos também, se isto for considerado necessário.

Todavia, classes associativas são válidas somente quando existe um único objeto relacionado a duas instâncias associadas. No caso deste exemplo, um ator que atue em um filme terá um único papel. Caso um ator interpretasse dois papéis em um mesmo filme, o uso da classe associativa não seria o mais adequado, sendo necessário inserir uma classe normal atuando como classe intermediária da associação, conforme demonstra a figura 4.14.



Figura 4.14 – Classe Intermediária.

Na figura 4.14, utilizamos o mesmo exemplo da figura 4.13, substituindo a classe associativa por uma classe intermediária, representando uma situação em que um ator poderia atuar em muitos filmes e um filme poderia ter muitos atores, diferindo-se da situação anterior por permitir que um ator interprete mais de um papel no mesmo filme.

#### 4.2.8 Associação Qualificada

Quando existe um atributo único em uma classe, é possível criar uma associação qualificada, que é uma forma de identificar individualmente um objeto dentro de uma coleção (uma coleção pode ser definida como um conjunto de instâncias associadas a outro objeto). O qualificador de uma associação é representado por um pequeno retângulo ligado ao final da associação, conforme demonstra a figura 4.15.

Neste exemplo, o qualificador é o atributo `numeroConta`, que pode identificar unicamente um objeto da classe `Conta`. Uma associação qualificada reduz a multiplicidade de uma associação 1 para muitos para uma associação 1 para 1 de forma a identificar e associar um único objeto a outro.

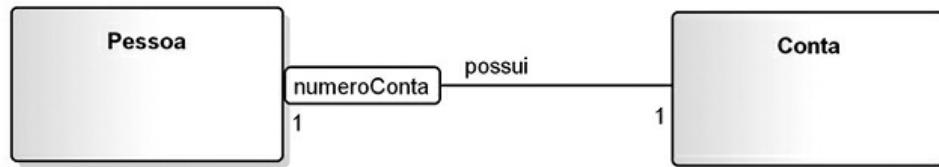


Figura 4.15 – Exemplo de Associação Qualificada.

#### 4.2.9 Dependência

Este relacionamento, como o próprio nome indica, identifica certo grau de dependência de um elemento (neste diagrama, normalmente uma classe) em relação à outro. O relacionamento de dependência é representado por uma linha tracejada entre dois elementos, contendo uma seta apontando para o elemento do qual o elemento posicionado na outra extremidade do relacionamento é dependente.

Uma dependência significa um relacionamento fornecedor/cliente entre elementos do modelo em que a modificação de um fornecedor pode impactar de alguma forma os elementos de modelo clientes. O

relacionamento de dependência possui muitas especializações utilizadas como estereótipos tanto no diagrama de classes como também em diversos outros diagramas, como **usage** no próprio diagrama de classes, **include** ou **extend** no diagrama de casos de uso, **instanceOf** no diagrama de objetos ou **merge** no diagrama de pacotes.

Por exemplo, **usage (uso)** é um tipo de dependência no qual um elemento necessita que um ou mais elementos o complementem de alguma forma, como em uma operação, por exemplo. **Usage** não especifica como o cliente utiliza o fornecedor, mas apenas deixa claro que é utilizado. A figura 4.16 apresenta um exemplo de dependência com o estereótipo **usage**.

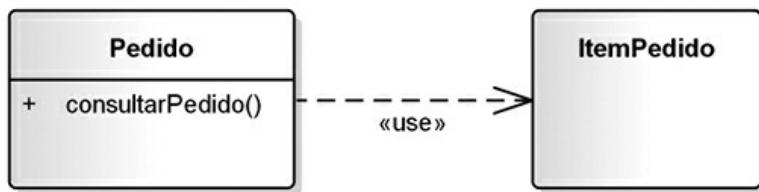


Figura 4.16 – Dependência com Estereótipo Usage.

Neste exemplo, podemos perceber que um objeto da classe **Pedido** precisa ser complementado de alguma maneira por objetos da classe **ItemPedido**, quando for executada a operação **consultarPedido**, ou seja, as informações contidas em um objeto **Pedido** precisam ser acrescidas pelas informações contidas nos objetos **ItemPedido** a ele associados.

## 4.2.10 Realização

Uma realização é um tipo de dependência especializada que mistura características dos relacionamentos de generalização e dependência, sendo usada para identificar classes responsáveis por executar funções para outras classes, muitas vezes classes de interface, que serão mais bem explicadas a seguir. Esse tipo de relacionamento herda o comportamento de uma classe, mas não sua estrutura.

O relacionamento de realização é representado por uma linha tracejada contendo uma seta vazia que aponta para a classe, que tem uma ou mais funções que devem ser realizadas por outra, enquanto na outra extremidade da linha é definida a classe que realiza esse comportamento. A

associação de realização pode ser comparada à palavra-chave **implements** da linguagem Java, que determina que uma classe implementará os métodos definidos em outra classe, ou seja, realizará o comportamento de outra classe. A figura 4.17 apresenta um exemplo de relacionamento de dependência e realização.

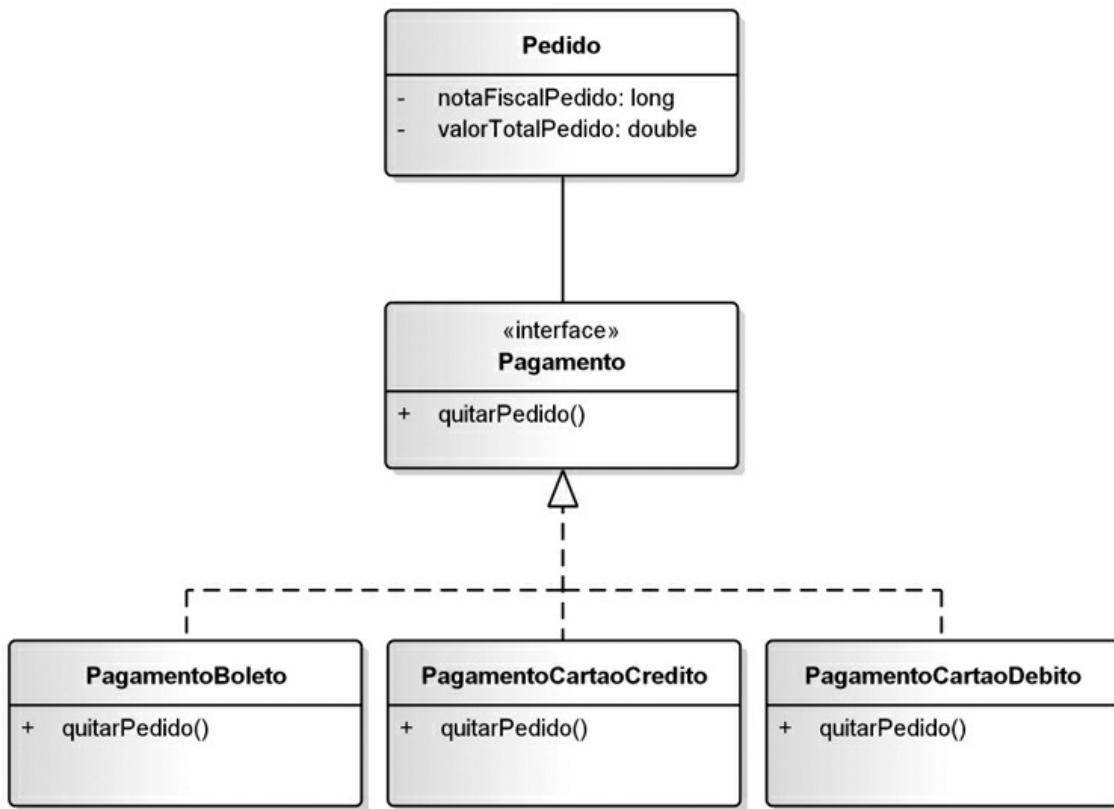


Figura 4.17 – Relacionamento de Dependência e Realização.

Ao observarmos essa figura, percebemos que a classe **PlacaMae** tem um relacionamento de dependência com a classe **iMonitor** e a classe **PlacaMae** utiliza de alguma forma essa interface, conforme demonstra o estereótipo “**use**” no relacionamento de dependência. Já a classe **Monitor**, por sua vez, tem um relacionamento de realização com a classe **iMonitor**, o que determina que a classe **Monitor** implementa os serviços oferecidos pela classe **iMonitor**. Observe que a classe **iMonitor** tem um estereótipo <<interface>> para destacar sua função.

Uma interface, na orientação a objetos, é como um contrato entre uma classe e o mundo exterior (não confundir com interface humano computador). As interfaces são formadas pela declaração de um ou mais métodos que não possuem corpo. Quando uma classe implementa uma interface, compromete-se a fornecer o comportamento oferecido por essa interface. As operações específicas a cada um desses métodos são realizadas pela classe que implementa a interface, o que é representado pela associação de realização. A figura 4.18 apresenta um exemplo um pouco mais prático de aplicação da associação de realização.

Neste exemplo, podemos perceber que há uma classe **Pedido** associada a uma classe de interface, denominada **Pagamento**, que representa o processo de pagamento de um pedido. Observe que a classe possui um método chamado **quitarPedido**. O pagamento pode ser realizado de três maneiras: pelo pagamento de um boleto, por meio de cartão de crédito ou de débito.



*Figura 4.18 – Relacionamento de Realização.*

Posto que uma classe de interface representa apenas um contrato cujas operações deverão ser implementadas por outras classes, representamos neste exemplo três outras classes: **PagamentoBoleto**, **PagamentoCartaoCredito** e **PagamentoCartaoDebito**, para representar cada uma das formas de pagamento possíveis. Em seguida, associamos essas novas classes à classe de interface **Pagamento** por meio de associações de realização, o que significa que o método `quitarPedido` será implementado em cada uma delas, uma vez que cada forma de pagamento possui nuances particulares.

### 4.3 Portas

Uma porta é uma característica estrutural de um classificador que especifica uma interação distinta entre o classificador e seu ambiente ou entre o classificador (em termos de seu comportamento) e suas partes internas. Uma porta pode especificar os serviços que um classificador fornece para seu ambiente, bem como os serviços que o classificador espera

de seu ambiente. Uma porta representa, portanto, um ponto de comunicação.

Portas e partes internas serão mais bem detalhadas e exemplificadas nos capítulos referentes ao diagrama de estrutura composta e ao diagrama de componentes. É necessário, porém, introduzir esse conceito neste momento por ser necessário ao tópico seguinte. Portas são representadas como pequenos quadrados colocados sobre a borda da classe, conforme mostra a figura 4.19.

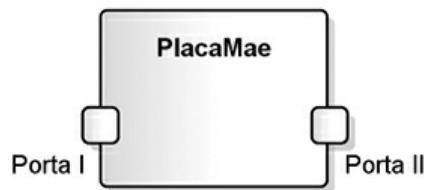


Figura 4.19 – Exemplo de Portas.

Neste exemplo, utilizamos novamente a classe **PlacaMae**, acrescentando-lhe duas portas, chamadas **Porta I** e **Porta II**, embora não seja obrigatório definir uma nomenclatura para elas. Essa figura servirá de base para os próximos exemplos.

## 4.4 Interfaces

A partir da versão 2.0 da UML, passou-se a oferecer símbolos alternativos para representar as interfaces, embora o estereótipo **interface** ainda seja perfeitamente válido. Essencialmente, as interfaces podem ser de dois tipos: fornecidas ou requeridas.

### 4.4.1 Interfaces Fornecidas

Uma interface fornecida descreve um serviço implementado por uma classe. O conjunto de interfaces implementadas por uma classe forma suas interfaces fornecidas e representa o conjunto de serviços que a classe oferece a seus clientes. Ao implementar uma interface, uma classe suporta o conjunto de características contidas por esta e obedece às suas restrições. As interfaces fornecidas são representadas por um círculo fechado ligado à classe por uma linha sólida. É necessário definir uma porta de comunicação entre a interface e a classe. Uma interface fornecida

corresponde à associação de realização explicada anteriormente. A figura 4.20 apresenta um exemplo de interface fornecida.

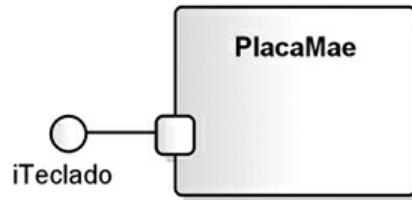


Figura 4.20 – Exemplo de Interface Fornecida.

Nesse exemplo, continuamos utilizando a classe chamada **PlacaMae**, que representa a placa principal de um computador. Essa classe tem uma interface fornecida chamada **iTeclado**, que representa a interface física entre uma placa-mãe e um teclado, onde a placa-mãe (na verdade, um de seus componentes, o processador) é responsável por interpretar as teclas pressionadas no teclado.

#### 4.4.2 .Interfaces Requeridas

Este tipo de interface descreve os serviços que outras classes devem fornecer a uma determinada classe, que não precisa ter conhecimento de quais classes implementarão esses serviços. As interfaces requeridas são representadas por um semicírculo fechado ligado a uma classe por uma linha sólida. Da mesma forma que nas interfaces fornecidas, é preciso definir uma porta de comunicação entre a interface e a classe. Uma interface requerida corresponde a uma associação de dependência do tipo **usage**. A figura 4.21 apresenta um exemplo de interface requerida.

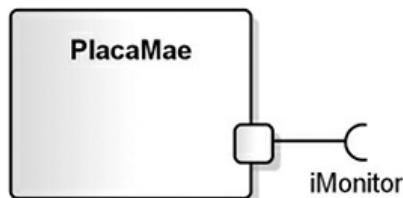


Figura 4.21 – Exemplo de Interface Requerida.

Aqui, utilizamos novamente a classe **PlacaMae**, dessa vez contendo uma interface requerida chamada **iMonitor**, que representa a interface física entre uma placa-mãe e um monitor, em que o monitor deve apresentar no

vídeo as imagens e os símbolos solicitados pelo processador. É comum que uma interface fornecida em uma classe seja uma interface requerida em outra, podendo facilmente ocorrer de ambas as interfaces surgirem juntas, como demonstra a figura 4.22.

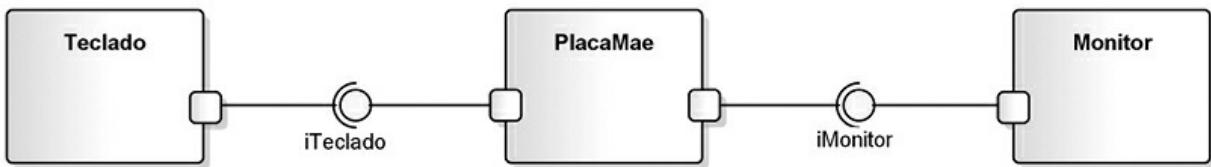


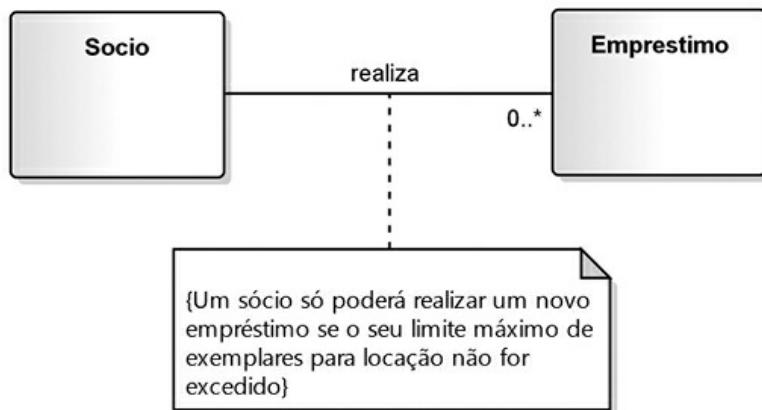
Figura 4.22 – Exemplo de Interface Fornecida e Requerida.

No exemplo da figura 4.22, existem duas interfaces. A primeira representa a interface entre as classes **Teclado** e **PlacaMae**, chamada **iTeclado**, e a segunda representa a interface entre as classes **PlacaMae** e **Monitor**, chamada **iMonitor**. Observe que **iTeclado** é uma interface contida tanto pela classe **Teclado** quanto pela **PlacaMae**, porém é uma interface requerida para a primeira e uma interface fornecida para a segunda. O mesmo ocorre com a interface **iMonitor** entre as classes **PlacaMae** e **Monitor**, sendo esta requerida pela primeira e fornecida pela última. Esses exemplos de interfaces serão ainda trabalhados nos capítulos sobre os diagramas de sequência, componentes e estruturas compostas, dentro do escopo de cada diagrama.

É importante notar que as interfaces requeridas e fornecidas podem, às vezes, ser substituídas pelos relacionamentos de dependência e realização já existentes nas versões anteriores, conforme demonstrado na figura 4.17, que apresenta o mesmo exemplo.

## 4.5 Restrições

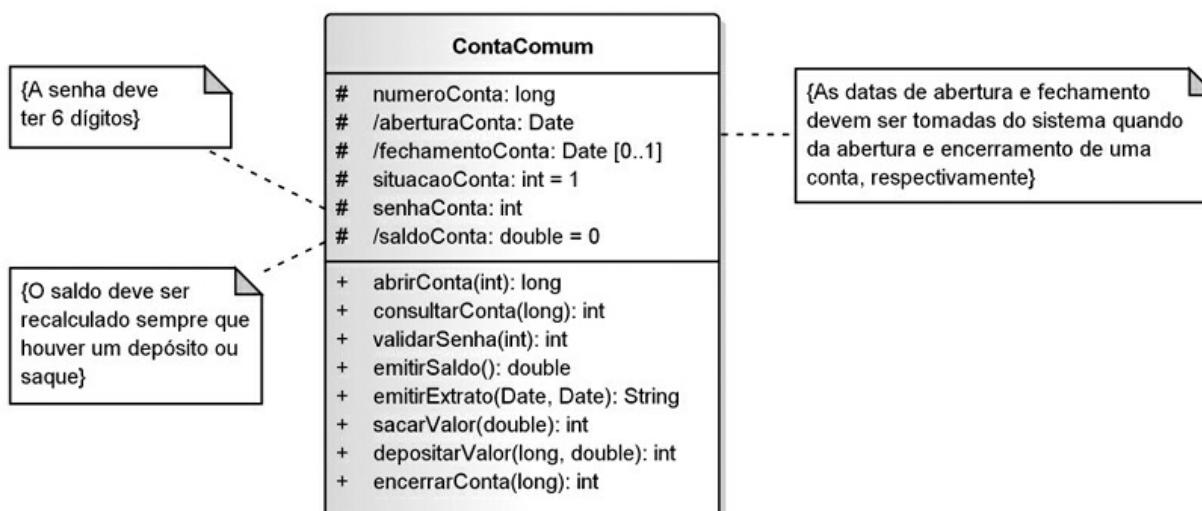
Restrições constituem-se em informações extras que definem condições a serem validadas durante a implementação dos métodos de uma classe, das associações entre as classes ou de seus atributos. As restrições são representadas por textos limitados por chaves. Restrições podem ser usadas para detalhar requisitos não funcionais, incluindo regras do negócio. A figura 4.23 apresenta um exemplo de restrição.



*Figura 4.23 – Restrição em uma Associação.*

Nesse exemplo, utiliza-se uma restrição para determinar que um sócio só poderá realizar um empréstimo se o limite máximo de exemplares que ele pode tomar emprestado não tiver sido excedido. Esse exemplo enfoca uma restrição que representa uma regra do negócio, ou seja, uma norma que deve ser obedecida quando da execução de um processo. Observe que esta é uma restrição da associação **realiza** e está ligada a ela por meio de uma nota. Não é obrigatório inserir restrições por meio de notas, pois o leitor encontrará exemplos de restrições inseridas diretamente no diagrama. Consideramos, contudo, que essa forma é mais organizada e mais bem apresentada.

Restrições podem ser aplicadas também para validar um atributo ou método de uma classe específica, como no exemplo apresentado na figura 4.24.



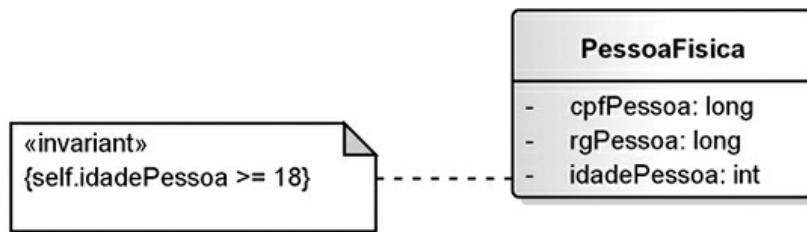
*Figura 4.24 – Exemplo de Restrição em Atributos.*

No exemplo da figura 4.24, uma nota informa a necessidade de que o atributo **senha** tenha seis dígitos, restringindo o uso de senhas com um número de dígitos diferentes. Outras notas informam que as datas de abertura e encerramento de conta devem ser tomadas do sistema quando da execução dessas operações e o saldo da conta deve ser recalculado sempre que um depósito ou saque for realizado.

#### **4.5.1 .Restrições em OCL (Object Constraint Language)**

Nos dois exemplos anteriores, descrevemos restrições de maneira informal. Embora fáceis de compreender, as restrições apresentadas eram basicamente regras de negócio escritas em português e não seguiam um formato de representação específico. Contudo, existe uma linguagem utilizada especificamente para definir restrições, semelhante a uma linguagem de programação, chamada OCL (Object Constraint Language ou Linguagem de Restrição de Objeto). Entre outros objetivos, essa linguagem procura fornecer um maior formalismo na declaração de restrições, procurando, assim, evitar a ambiguidade em suas representações.

Suponhamos, por exemplo, que em uma classe **PessoaFísica** queiramos deixar explícito que nenhum objeto dessa classe poderá conter um valor inferior a 18 em seu atributo **idade**. Em OCL, poderíamos representar isso da seguinte maneira, ilustrada pela figura 4.25.

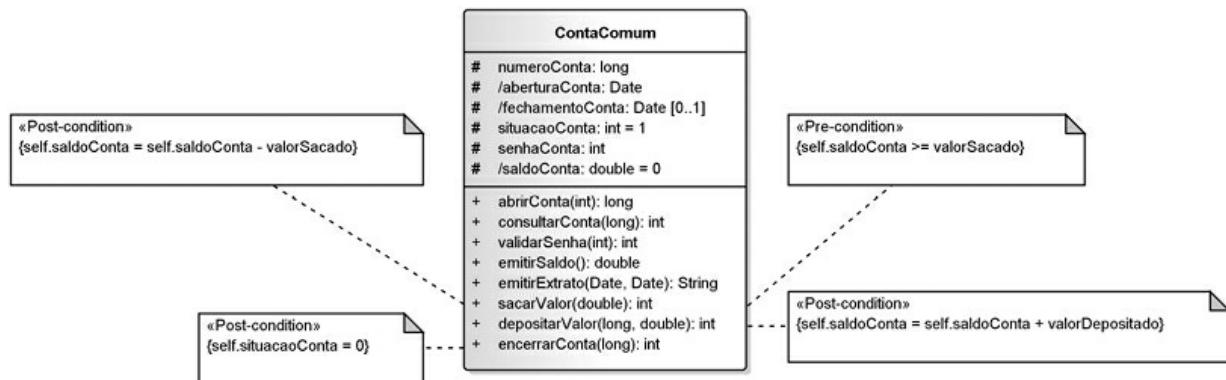


*Figura 4.25 – Exemplo de Restrição em Atributo Utilizando a Linguagem OCL.*

Ao examinarmos essa figura, percebemos que há uma restrição associada à classe **PessoaFísica**. Essa restrição contém algumas informações que explanaremos a seguir. Primeiramente, observe que a restrição possui um

estereótipo denominado **<<invariant>>**, o qual passa a informação de que a restrição refere-se a qualquer objeto da classe, ou seja, é invariante. A expressão seguinte inicia com a palavra **self**, que representa uma instância da classe PessoaFisica, a partir da qual se inicia a avaliar a expressão – muitas vezes essa palavra pode ser suprimida, desde que o contexto da expressão esteja claro (contexto na OCL refere-se a um tipo específico, como uma classe, associação, classe associativa, interface etc.). O restante da expressão, **idadePessoa >= 18**, compõe a restrição propriamente dita, que torna explícito que todo objeto da classe PessoaFisica precisa obrigatoriamente possuir idade superior ou igual a 18 anos.

Existem outros estereótipos que podem ser aplicados a restrições OCL, como **<<pre-condition>>** e **<<post-condition>>**, que se referem a restrições que precisam ser aplicadas antes ou depois que um método for executado. A figura 4.26 apresenta exemplos desse tipo de restrição.



*Figura 4.26 – Exemplo de Restrição em Métodos Utilizando a Linguagem OCL.*

Nesse exemplo são apresentadas quatro restrições aplicadas a métodos da classe **ContaComum**, duas associadas ao método **sacarValor**, uma associada ao método **depositarValor** e a última associada ao método **encerrarConta**. A primeira restrição do método **sacarValor** é uma restrição do tipo **<<Pre-condition>>**, que determina que a condição por ela explicitada deve ser satisfeita para que o método seja executado. No caso, a condição é a de que o saldo da conta seja igual ou superior ao valor que se deseja sacar. O método **sacarValor** possui também uma restrição do tipo **<<Post-condition>>**, que representa uma ação que precisa ser realizada ao final da execução do método. No caso, o valor sacado deve ser

diminuído do saldo da conta.

Por sua vez, o método `depositarValor` possui uma restrição <<Post-condition>> que determina que se deve somar o valor depositado ao saldo da conta. Finalmente, o método `encerrarConta` possui também uma restrição do tipo <<Post-condition>>, que especifica que, ao encerrar uma conta, deve-se atribuir 0 ao atributo situação da conta. Na verdade, essas pós-condições representam passos que fazem parte desses métodos, porém essas restrições servem para especificar condições que precisam ser satisfeitas pela execução desses métodos.

Restrições OCL também podem ser aplicadas em associações, como podemos observar na figura 4.27, onde há uma associação unária 1 para 1 em uma classe **Pessoa**. Nessa associação, uma pessoa só pode ser cônjuge de uma outra pessoa e, em um extremo da associação, um dos objetos desempenha o papel de marido, enquanto, no outro extremo, outro objeto desempenha o papel de esposa. A restrição aplicada a essa associação tem por objetivo determinar que o objeto que desempenha o papel de marido deve ser diferente do objeto que desempenha o papel de esposa, já que uma pessoa não pode ser cônjuge de si mesma.

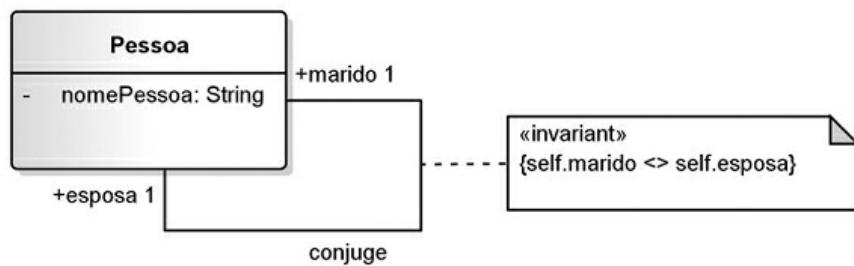
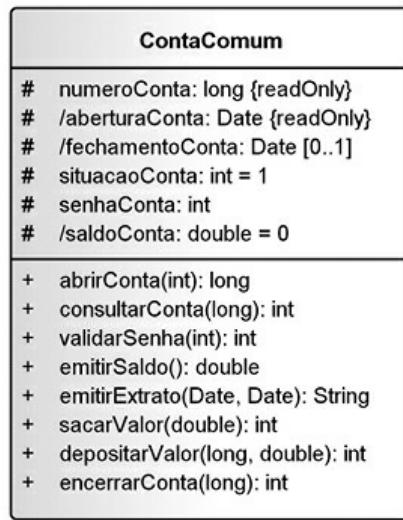


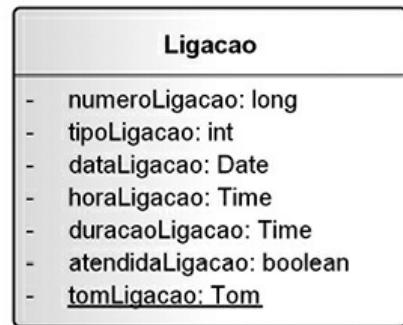
Figura 4.27 – Exemplo de Restrição OCL em Associação.

Há, ainda, outros tipos de restrições que podem ser aplicados a atributos. Por exemplo, pode-se definir um atributo como `{readOnly}` (somente leitura), determinando que seu valor só pode ser lido e não modificado. Quando um atributo é somente leitura, a restrição `{readOnly}` deve constar ao lado da declaração do atributo, como pode ser percebido na figura 4.28, onde se definiu que os atributos `numeroConta` e `aberturaConta` da classe **ContaComum** só podem ser lidos e não modificados.



*Figura 4.28 – Exemplo de Restrição readOnly em Atributos.*

Também é possível determinar que um atributo é estático (**Static**). Atributos estáticos são aqueles cujos valores são idênticos para todos os objetos de uma classe, ou seja, é um atributo pertencente à classe propriamente dita e não aos seus objetos. Essa restrição pode ser igualmente aplicada a operações (métodos). Um atributo estático é apresentado sublinhado, como pode ser visto na figura 4.29.

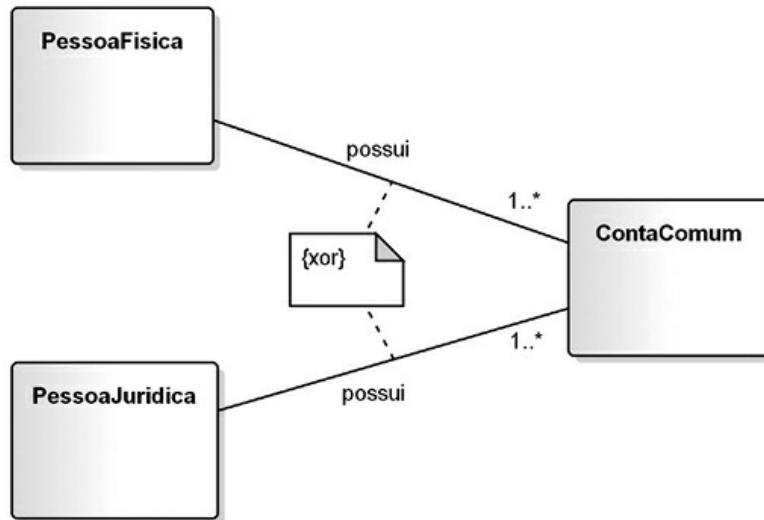


*Figura 4.29 – Exemplo de Atributo Estático.*

Nesse exemplo, representamos uma classe para um sistema de telefone celular que representa as ligações recebidas ou enviadas pelo aparelho. É preciso registrar todas as ligações recebidas ou feitas, armazenando informações como o número do telefone, o tipo de ligação (se foi feita ou recebida), a data e a hora da ligação, a duração da ligação e se esta foi atendida ou não. Além disso, sempre que uma chamada é recebida, uma música deve tocar. Essa música pode ser escolhida pelo usuário, mas será

sempre a mesma para todas as ligações, não havendo uma música particular para uma ligação específica. Assim, identificou-se o atributo **tomLigacao** como um atributo estático da classe, que não muda de objeto para objeto. O tipo do atributo “**Tom**” refere-se a outra classe, como se pode notar por sua inicial maiúscula. Esse exemplo será expandido para um modelo mais completo em um exemplo posterior.

Restrições podem também ser utilizadas para representar o “ou” exclusivo (xor), quando instâncias de duas ou mais classes podem se relacionar com instâncias de outra classe específica, mas somente uma instância de uma das classes pode se relacionar com uma instância da classe em questão, em detrimento das outras. Um exemplo desse tipo de restrição é apresentado na figura 4.30.

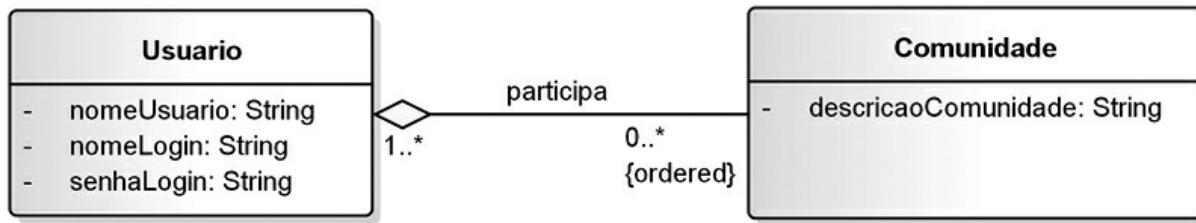


*Figura 4.30 – Restrição com Ou Exclusivo.*

Nesse exemplo, uma conta comum pode pertencer a uma pessoa física ou jurídica, mas uma determinada conta comum só pode pertencer a uma única pessoa, nunca a ambas. É possível também encontrar classes com restrições abaixo ou ao lado de seus nomes, determinando que a classe possui uma função específica. Embora isso normalmente seja uma função realizada pelos estereótipos, é possível encontrar classes com restrições abaixo ou ao lado de seus nomes, informando que a classe em questão é uma classe persistente, por exemplo.

Outro exemplo do uso de restrições pode ser aplicado à definição de coleções ordenadas, conforme demonstra a figura 4.31. Como foi dito

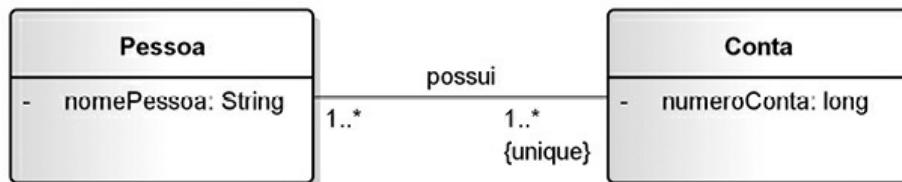
anteriormente, uma coleção pode ser definida como um conjunto de instâncias associadas a outro objeto.



*Figura 4.31 – Exemplo de Coleção Ordenada.*

Esse exemplo baseia-se em uma rede social, onde um usuário poderá participar de muitas comunidades. Sempre que um usuário acessar sua conta, será apresentado um determinado número das comunidades de que ele participa. Aqui, definimos que o conjunto de comunidades de cada usuário deve ter algum tipo de ordenação (alfabética, por exemplo), conforme demonstra a restrição **{ordered}** abaixo da multiplicidade  $0..^*$  na associação **participa**. Nesse exemplo, existe um grande número de comunidades e um determinado usuário pode participar de um determinado número delas. O conjunto de comunidades associadas a um usuário (não importa que possam estar associadas a outros usuários, como demonstra a associação de agregação) é uma coleção.

Outra restrição que pode ser aplicada a uma coleção é a restrição **unique**, que determina que um elemento na coleção não pode se repetir. A figura 4.32 apresenta um exemplo de restrição **{unique}** que determina que uma conta-corrente não pode se repetir.



*Figura 4.32 – Restrição Unique.*

Para que um objeto da conta-corrente seja único, é necessário que ao menos um de seus atributos armazene um valor que não se repita em nenhum outro objeto da classe. No caso de uma conta-corrente, o número da conta poderia desempenhar essa função, uma vez que nenhum objeto

da classe conta poderia armazenar um número de conta igual.

Essas restrições podem também ser aplicadas a atributos individuais, como demonstra a figura 4.33.

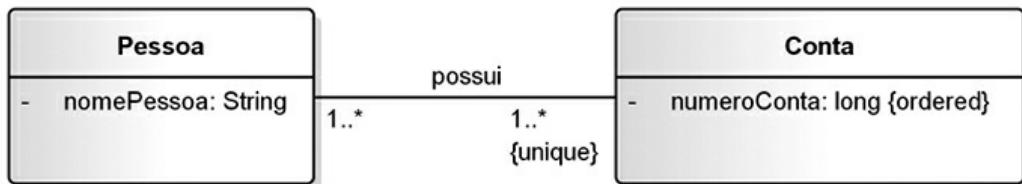


Figura 4.33 – Restrição Ordenada (Ordered) Aplicada a um Atributo.

Neste exemplo, além de os elementos da coleção não poderem se repetir, acrescentamos a informação de que estão ordenados pelo número da conta. Unidas, as restrições **Ordered** e **Unique** podem ser usadas para especificar os quatro tipos possíveis em uma coleção. Assim, se uma coleção não é ordenada, mas é única, é uma coleção do tipo **Set**; se for ordenada e única, será uma coleção do tipo **OrderedSet**; se não for ordenada nem única, será uma **Bag**; se for ordenada, mas não única, será uma **Sequence**. A figura 4.34 apresenta um exemplo de restrição **Sequence** aplicada ao atributo **nomePessoa** da classe **Pessoa**, significando que este está ordenado, mas que duplicatas são permitidas, podendo existir duas pessoas com os nomes exatamente iguais.

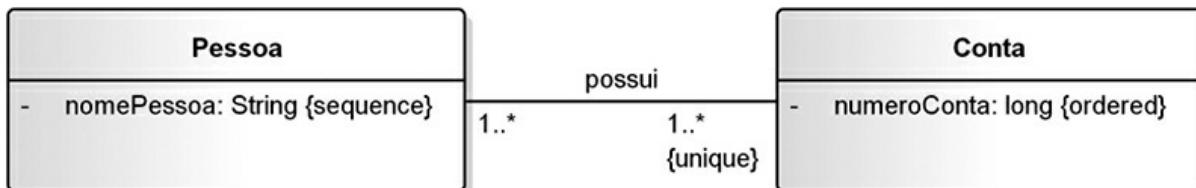
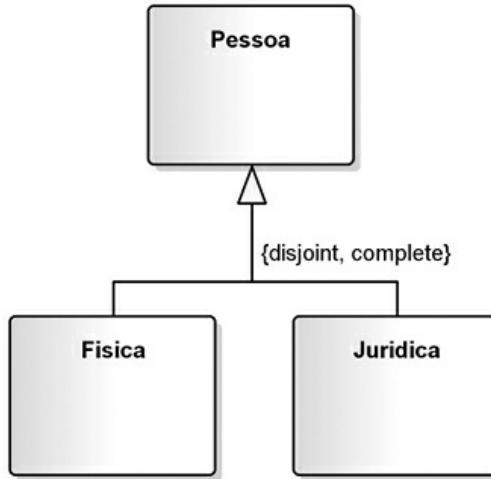


Figura 4.34 – Restrição Sequence Aplicada a um Atributo.

As restrições podem ainda ser utilizadas para definir melhor a semântica de classes especializadas derivadas de classes gerais. As restrições predefinidas para classes especializadas são:

- **Completa (complete)** – Quando todas as subclasses possíveis foram derivadas da classe geral.
- **Incompleta (incomplete)** – Quando ainda é possível derivar novas subclasses.

- **Separada (disjoint)** – Quando as subclasses são mutuamente exclusivas, ou seja, no momento em que uma instância pertence a uma subclasse, não poderá de forma alguma pertencer a nenhuma das outras subclasses derivadas. A figura 4.35 fornece um exemplo de classes especializadas separadas e completas.

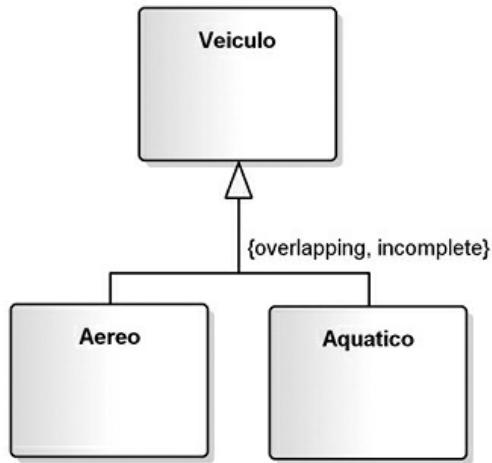


*Figura 4.35 – Restrição Separada e Completa.*

Nesse exemplo, foram derivadas duas subclasses a partir da classe **Pessoa**, as classes **Fisica** e **Juridica**. Dentro do escopo em que estão inseridas, não existem mais classes que possam ser derivadas da classe **Pessoa**, por isso a restrição entre as classes generalizadas é completa. Além disso, no momento em que uma pessoa for física, não poderá ser jurídica, e vice-versa. Portanto, a especialização também é separada (disjoint).

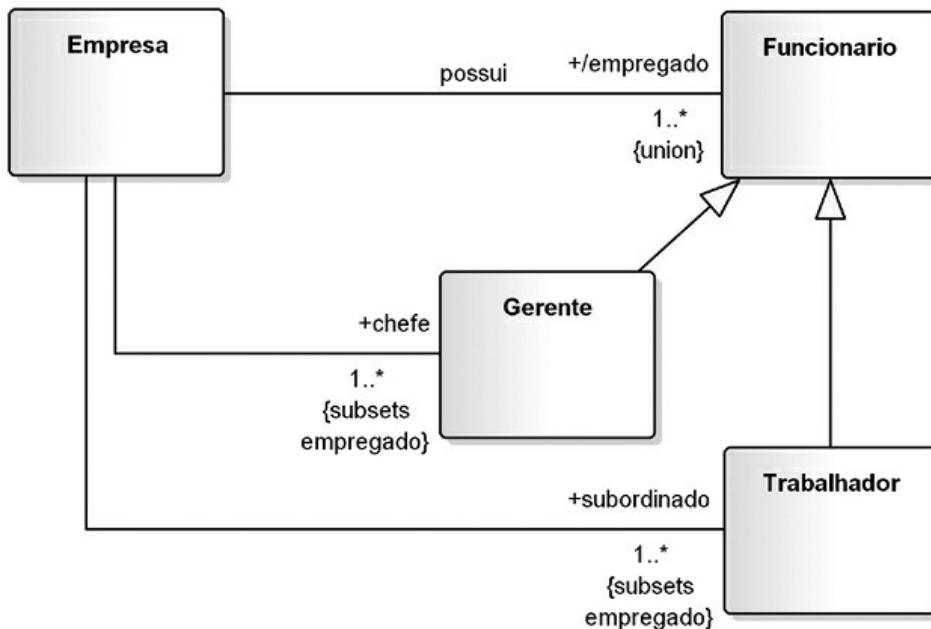
- **Sobreposta (overlapping)** – Quando o fato de pertencer a uma subclasse não impede que pertença a outras. A figura 4.36 apresenta um exemplo de restrição sobreposta e incompleta para classes especializadas.

A figura 4.36 apresenta um exemplo, no qual, a partir da classe **Veiculo**, derivaram-se duas subclasses **Aereo** e **Aquatico**. Como poderíamos ainda derivar uma classe para veículos terrestres, colocou-se uma restrição incompleta e, como pode ocorrer de um veículo ser tanto aéreo quanto aquático, como é o caso do hidroavião, acrescentamos, ainda, a restrição de sobreposta (overlapping) à especialização. Como é possível perceber, as restrições podem ser combinadas, sendo separadas por vírgulas.



*Figura 4.36 – Restrição Sobreposta e Incompleta.*

Outras restrições que podem ser aplicadas sobre especializações são as restrições `{union}` e `{subsets}`, que demonstram que uma associação pode conter a soma (ou união) dos elementos de outras classes derivadas da classe envolvida na associação. A figura 4.37 apresenta um exemplo de situação em que essas restrições podem ser aplicadas.



*Figura 4.37 – Restrições `{union}` e `{subsets}`.*

Neste exemplo, uma empresa pode possuir muitos funcionários, representados pela classe `Funcionario`, e o papel dos objetos envolvidos nessa classe é o de empregado, no entanto há ainda mais informações. O

papel desses objetos é calculado (derivado), conforme demonstra a barra (/) ao lado do papel, e esse cálculo deve ser obtido por meio da união da soma de objetos contidos em outras classes, conforme demonstra a restrição {union}.

Pode-se perceber ainda que a classe Funcionario possui duas especializações: Gerente e Trabalhador. Percebe-se também que existem associações entre a classe Empresa e essas duas classes especializadas e que os objetos da classe Gerente envolvidos na associação desempenham o papel de chefe e os da classe Trabalhador, o papel de subordinado. Observa-se que essas associações possuem a restrição {subsets empregado} e empregado é o papel dos objetos da classe Funcionario envolvidos na associação com a classe Empresa. Isso significa que os objetos que desempenham o papel de chefe ou subordinado são subconjuntos dos objetos que desempenham o papel de empregado.

Apesar de sua utilidade óbvia, devemos evitar o uso de muitas restrições em um diagrama de classes, porque seu uso excessivo pode tornar o diagrama muito poluído e difícil de ler. Devem-se inserir restrições no diagrama de classes somente quando forem realmente importantes, caso contrário as restrições deverão ser detalhadas em outros diagramas que enfoquem os processos nos quais elas são necessárias.

## 4.6 Estereótipos do Diagrama de Classes

Os estereótipos foram explicados no final do capítulo 3 e brevemente abordados ao longo deste capítulo. Como foi dito anteriormente, são uma característica poderosa utilizada em todos os diagramas da UML e, no diagrama de classes, costumam exercer um papel muito importante. Assim, nesta seção, detalharemos alguns dos estereótipos considerados mais importantes.

Conforme explanado no capítulo 3, estereótipos são utilizados para especializar um determinado componente, como uma classe, por exemplo, e atribuir-lhe características extras além daquelas comuns àquele tipo de componente. Assim, um componente estereotipado é um componente que pode apresentar um comportamento diferente de seus pares. Existem diversos estereótipos predefinidos na linguagem UML, todavia a linguagem permite a criação de estereótipos particulares por parte de quem a for-

utilizar. Nas seções seguintes, apresentaremos alguns dos estereótipos utilizados na modelagem classes.

### 4.6.1 Estereótipo <<enumeration>>

Uma enumeração é um tipo de dado cujos valores são enumerados no modelo como literais de enumeração. Basicamente, essa classe lista todos os valores válidos que um tipo de dados pode assumir e, embora não costume possuir associações, é geralmente colocada próxima das classes que utilizam o tipo de dados cujos literais são por ela enumerados. Na figura 4.38, apresentamos um exemplo de enumeração que representa todas as possíveis situações que um exemplar de livro em um sistema de controle de biblioteca pode assumir.

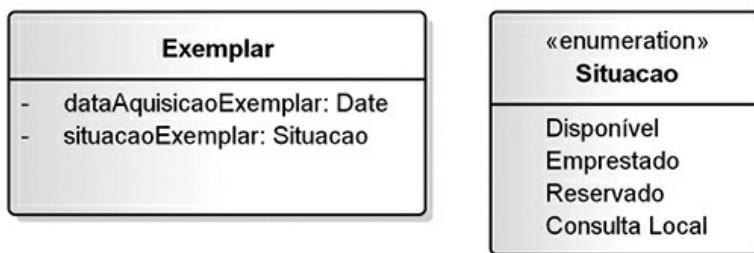


Figura 4.38 – Exemplo de Enumeração.

Ao examinar esse exemplo, percebemos que contém duas classes: a primeira é uma classe normal, chamada **Exemplar**, que representa os exemplares de um determinado livro, enquanto a segunda é uma classe de enumeração, conforme demonstra o seu estereótipo <<enumeration>>, denominada **Situacao**. Os valores válidos da classe de enumeração são “Disponível”, ou seja, o exemplar pode ser locado, “Emprestado”, “Reservado” e “Consulta Local”, o que significa que o exemplar não pode ser locado, apenas consultado dentro das dependências da biblioteca. Observe que a classe **Exemplar** contém dois atributos: o primeiro armazena a data em que o exemplar foi adquirido pela biblioteca, porém o segundo, chamado **situacaoExemplar**, é do tipo **Situacao**, ou seja, só pode conter um dos valores especificados como literais da classe **Situacao**.

### 4.6.2 Estereótipos para Projeto Navegacional

Existem diversos outros tipos de estereótipos com as mais diversas funções,

sendo possível utilizar alguns deles para representar o projeto navegacional de um site, por exemplo. Para isso, é preciso utilizar estereótipos como <<server page>>, <<client page>> e <<form>>. A figura 4.39 ilustra um exemplo de classe com estereótipo <<server page>>.



*Figura 4.39 – Classe com estereótipo <<server page>>.*

O estereótipo <<server page>> é um estereótipo gráfico que apresenta um símbolo de engrenagem ao lado do nome da classe. Esse estereótipo representa uma página web que possui scripts executados pelo servidor. A figura 4.40 apresenta um exemplo de estereótipo <<client page>>.



*Figura 4.40 – Classe com estereótipo <<client page>>.*

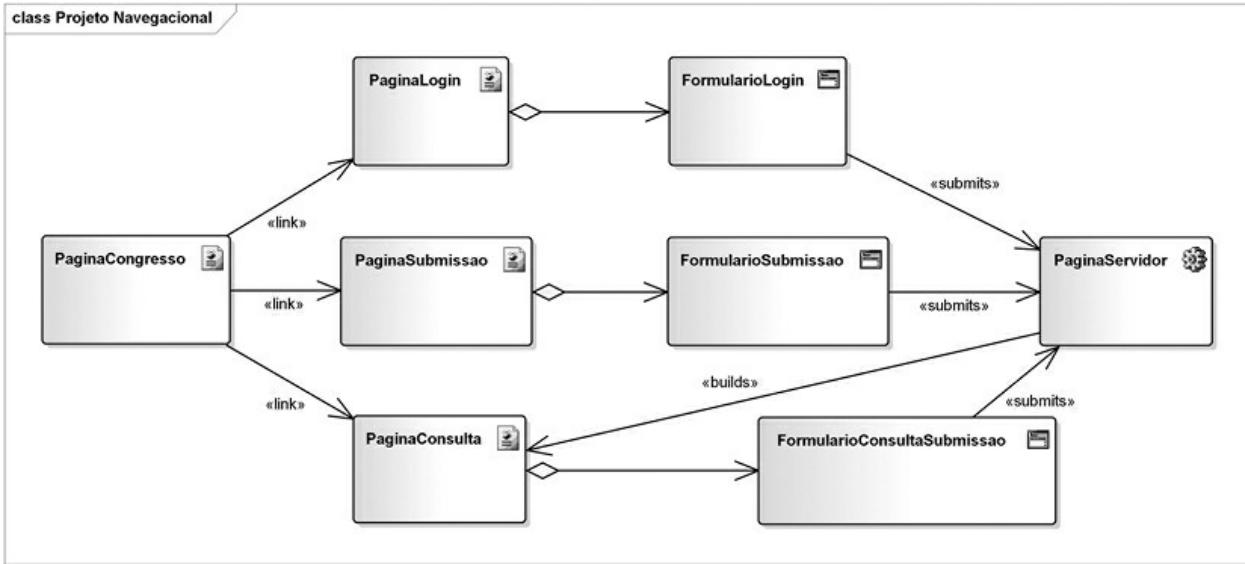
O estereótipo <<client page>> representa uma página HTML carregada pelo navegador do usuário, sendo também um estereótipo gráfico que apresenta o símbolo de uma página web ao lado do nome da classe. A figura 4.41 demonstra um exemplo de estereótipo <<form>>.



*Figura 4.41 – Classe com estereótipo <<form>>.*

O estereótipo <<form>> representa um formulário. Em um projeto navegacional, representa uma classe que contém um conjunto de campos que fazem parte de uma página. Este é um estereótipo gráfico que apresenta o desenho de um formulário ao lado da classe. Na figura 4.42,

apresentamos um exemplo de como utilizar esses estereótipos.



*Figura 4.42 – Projeto Navegacional Utilizando Estereótipos.*

Esse exemplo enfoca uma página de submissões de trabalhos para um evento científico, na qual os autores submetem seus trabalhos para avaliação, e se, porventura, forem aprovados, estes serão publicados nos anais do evento. Dessa forma, existe uma página principal que oferece as alternativas de logar no sistema, submeter um trabalho ou verificar a situação de trabalhos já submetidos.

A página principal é representada pela classe **PaginaCongresso**, detentora do estereótipo **<<client page>>**, que contém uma associação binária com as classes **PaginaLogin**, **PaginaSubmissao** e **PaginaConsulta**, que contêm o mesmo estereótipo. Observe que essa associação tem um estereótipo **<<link>>**, representando o vínculo entre as páginas da web que tais classes representam. Uma instância da classe **PaginaCongresso** nada mais é do que a página propriamente dita carregada na máquina de um usuário.

Cada uma dessas três páginas contém uma associação de agregação com uma classe detentora do estereótipo **<<form>>**, o que determina que a informação da classe **PaginaLogin** precisa ser completada pela informação contida na classe **FormularioLogin**, que a informação da classe **PaginaSubmissao** precisa ser completada pela informação contida na classe **FormularioSubmissao** e que a informação da classe

**PaginaConsulta** precisa ser complementada pela informação contida na classe **FormulárioConsultaSubmissao**. Essas classes com estereótipo <<form>> representam os formulários que deverão ser apresentados quando as páginas representadas pelas classes de estereótipo <<client page>> forem carregadas.

Observe que essas três classes com estereótipo <<form>> contêm uma associação binária com a classe **PaginaServidor**, detentora do estereótipo <<server page>>, e tais associações têm o estereótipo <<submits>> indicando que os valores de seus campos serão submetidos à classe **PaginaServidor**, que processará essas informações de alguma forma ou as repassará a outras classes, se for necessário. Observe ainda que a classe **PaginaServidor** tem também uma associação binária com a classe **PaginaConsulta** e essa associação tem o estereótipo <<builds>>, significando que a **PaginaServidor** pode mandar reconstruir essa página atualizando suas informações. A naveabilidade apresentada no exemplo demonstra o sentido em que trafegam as informações.

Existem vários outros estereótipos que podem ser aplicados a projetos navegacionais, como <<asp page>>, <<jsp page>>, <<servlet>>, <<web page>> etc. Tais estereótipos não fazem parte da UML oficial ainda, tendo sido introduzidos por meio de um perfil UML, que permite adaptar a UML a um domínio para o qual não foi projetada inicialmente. No entanto, esse perfil tem sido amplamente aceito e é suportado por diversas ferramentas CASE. Ao longo deste livro, falaremos mais sobre perfis e de como adaptar a UML a outros domínios.

Há também três estereótipos predefinidos na linguagem UML, bastante utilizados nos diagramas de classes e de sequência que merecem destaque e serão estudados nas subseções seguintes: os estereótipos <<boundary>>, <<control>> e <<entity>>.

#### 4.6.3 Estereótipo <<boundary>>

O estereótipo <<boundary>>, também conhecido como estereótipo de fronteira, identifica uma classe que serve de comunicação entre os atores externos e o sistema propriamente dito. Muitas vezes, uma classe <<boundary>> é associada à própria interface do sistema. Uma classe do tipo <<boundary>> geralmente necessita interagir com outra classe do

tipo <<control>>, que será explicada na seção seguinte.

O uso de classes <<boundary>> é particularmente útil quando se deseja utilizar o padrão MVC (Model-View-Controller ou Controlador-Visão-Modelo). Nesse padrão, a interface com o usuário (ou visão) é uma camada que possui pouco processamento próprio, concentrando-se em repassar os eventos que os usuários causam nela (como o pressionamento de um botão) e apresentando possíveis mudanças que tais eventos acarretam no modelo, quando isto for solicitado por uma classe controladora.

Pode haver uma classe <<boundary>> para cada caso de uso. Na verdade, pode haver várias. Esse padrão permite a existência de múltiplas interfaces distintas para um mesmo processo, possibilitando que um sistema seja acessado em plataformas e dispositivos diferentes. Esta é, aliás, uma das características que tornam esse padrão popular.

Ao se adotar esse padrão, um ator só pode interagir com objetos desse tipo de classe, e as instâncias de classes <<boundary>>, por sua vez, só podem interagir com atores e instâncias de classes controladoras (classes que possuem o estereótipo <<control>>), como veremos a seguir.

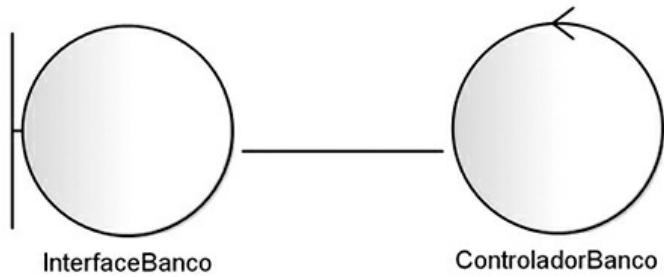
#### 4.6.4 Estereótipo <<control>>

O estereótipo <<control>> identifica classes cujos objetos servem de intermédio entre os objetos das classes <<boundary>> e os objetos de classes de entidade, que serão explicadas a seguir. Os objetos <<control>>, que compõem a camada de controle no padrão MVC, são responsáveis por interpretar os eventos ocorridos sobre os objetos <<boundary>>, como os movimentos do mouse ou o pressionamento de um botão, e executar ações em resposta, como o disparo de métodos sobre objetos das classes de entidade que compõem a camada de modelo.

Objetos de classes <<control>> controlam o processo representado por um caso de uso, interpretando os eventos que ocorrem sobre a interface, tornando as ações necessárias e solicitando a execução de métodos às instâncias das classes de entidade, quando for preciso, além de solicitar que a interface (formada por objetos boundary) apresente os resultados, quando for necessário.

Assim, pode haver um controlador para cada caso de uso ou até mais de um, se isto for considerado necessário. Alternativamente, pode existir

apenas um controlador para todo o sistema, chamado controlador de fachada, quando este for considerado suficiente para controlar todos os processos do sistema, porém essa estratégia pode sobrecarregar a classe controladora. Pode haver ainda um controlador de fachada que desempenha uma função geral e diversos controladores subordinados a ele. Objetos de classes controladoras podem interagir com objetos de classes de fronteira, objetos de classes de entidade e com outros objetos de classes controladoras, mas não com atores. A figura 4.43 demonstra um exemplo de classes com estereótipos <<boundary>> e <<control>>.



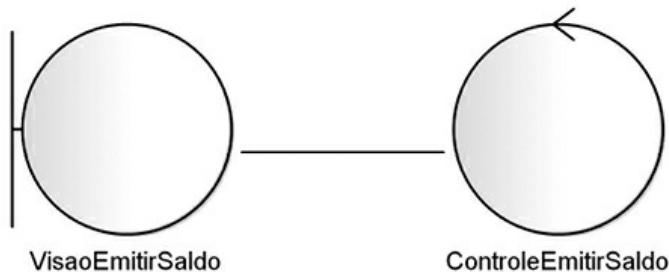
*Figura 4.43 – Classes <<boundary>> e <<control>>.*

Conforme apresentado na figura 4.43, a classe **InterfaceBanco** representa a interface do sistema e seus componentes serão basicamente rótulos (labels) contendo textos, botões e caixas de edição, além do próprio formulário, ou seja, essa classe representa uma interface entre os usuários externos e o sistema. Neste exemplo, utilizamos uma classe de fronteira genérica, que não está particularmente ligada a nenhum processo específico, sendo, portanto, meramente ilustrativa.

Os eventos que ocorrem sobre um objeto dessa classe de fronteira são repassados para um objeto da classe **ControladorBanco** que interpretará esses eventos e, se necessário, os repassará para os objetos da camada de modelo, normalmente na forma de chamada de métodos. Novamente, essa classe controladora é uma classe genérica que não está ligada a nenhum caso de uso particular, podendo ser considerada uma Classe Controladora de Fachada. Cumpre destacar que esses estereótipos são estereótipos gráficos que modificam o desenho-padrão das classes às quais foram aplicados.

Na figura 4.44 é apresentado um exemplo um pouco menos genérico, contendo classes de fronteira e de controle projetadas para manipular o

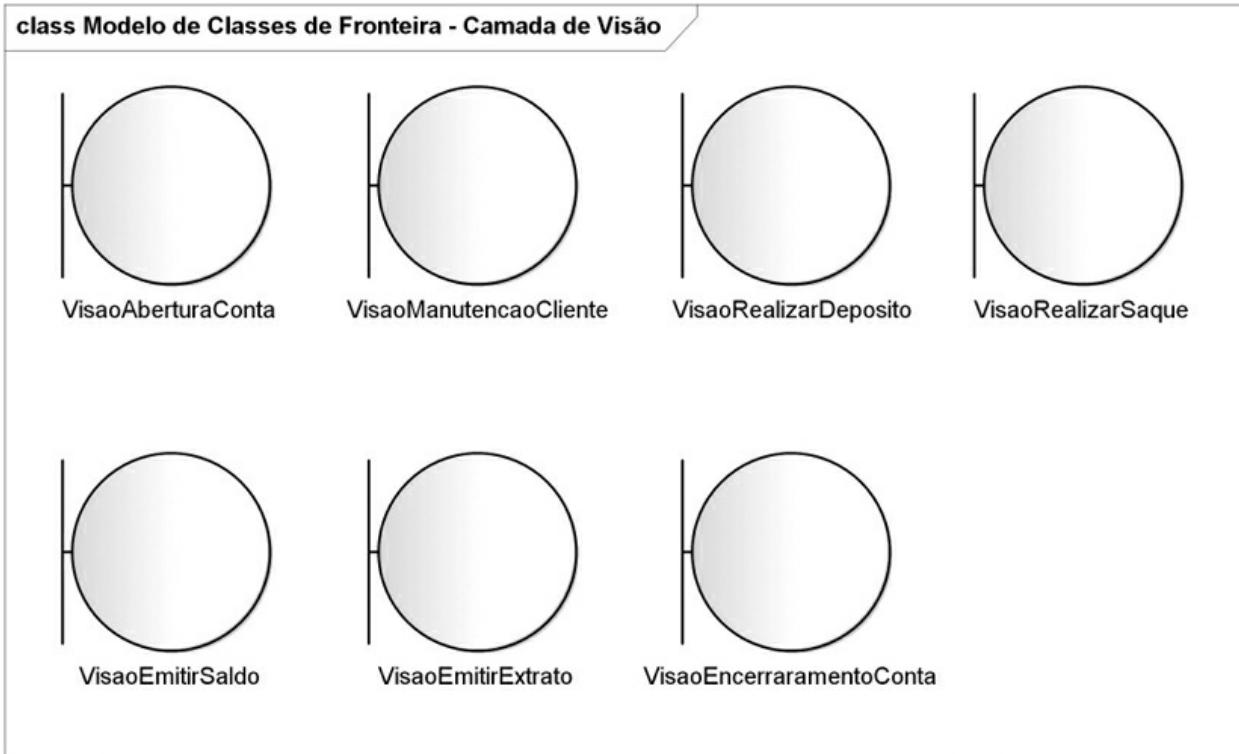
processo de emissão de saldo de uma conta no sistema de controle bancário.



*Figura 4.44 – Classes de Fronteira e Controle para o Processo de Emissão de Saldo.*

Nesse exemplo, a classe **VisaoEmitirSaldo**, conforme demonstra seu estereótipo, representa a interface entre o usuário e o processo para emitir o saldo de uma conta no banco, enquanto a classe **ControleEmitirSaldo** é uma classe controladora, como pode ser observado por seu estereótipo, e tem a função de controlar o processo de emissão de saldo, conforme descrito no caso de uso correspondente. Essa controladora deverá interagir também com objetos de classes de entidade, que serão explicados na seção 4.6.5.

Aqui, demonstramos uma associação entre as classes de fronteira e controle, todavia é comum que as classes de fronteira e controle sejam modeladas em diagramas separados, uma vez que fazem parte de camadas diferentes no padrão MVC. As figuras 4.45 e 4.46 apresentam os diagramas de classes representando as camadas de visão e de controle para o sistema de controle bancário.



*Figura 4.45 – Modelo de Classes de Fronteira – Camada de Visão.*

Nessa figura, identificamos sete classes, uma para cada um dos processos identificados no modelo de casos de uso apresentado no capítulo 3. O processo de abertura de conta é representado por apenas uma classe de fronteira, visto que acreditamos que as aberturas de contas comuns, especiais ou poupança podem ser englobadas em uma única classe de fronteira, com opções para definir o tipo de conta que se está abrindo e, por meio destas, habilitando-se ou desabilitando-se campos. No entanto, nada impediria que houvesse três classes de fronteira distintas. Não há uma classe de fronteira para o processo de registrar movimento porque este é incluído nos processos de depósito e saque e porque suas etapas são principalmente internas. A figura 4.46 ilustra um modelo de classes contendo as classes de controle do sistema bancário.

Como o leitor pode observar, as classes de controle aqui representadas são bastante semelhantes às classes do modelo de visão anteriormente apresentadas, uma vez que cada controladora é responsável por coordenar um dos processos do sistema de controle bancário. Poderíamos ter acrescentado uma controladora de fachada geral para gerenciar as outras controladoras, mas não achamos isso necessário nessa situação.

É importante destacar que classes de fronteira e de controle podem conter métodos, no entanto, neste livro, nos concentraremos somente nos métodos da camada de modelo.

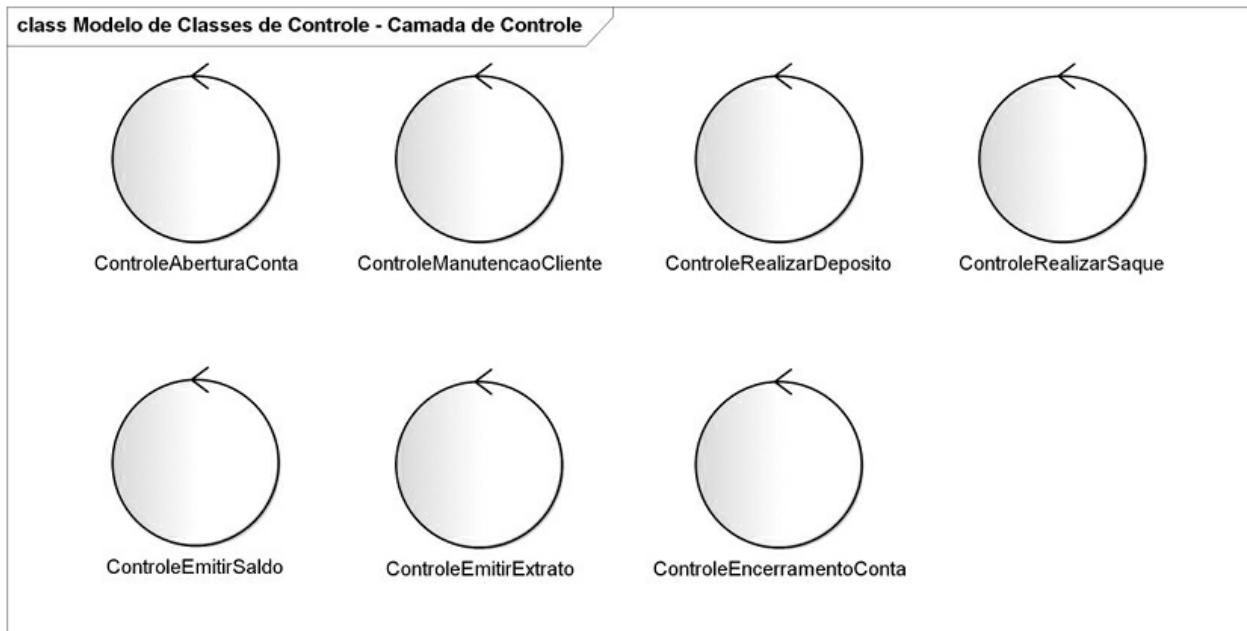
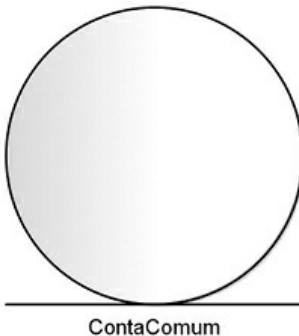


Figura 4.46 – *Modelo de Classes de Controle – Camada de Controle.*

#### 4.6.5 Estereótipo <<entity>>

O estereótipo <<entity>> tem por objetivo tornar explícito que uma classe é uma entidade, ou seja, a classe contém informações recebidas e armazenadas pelo sistema ou geradas por meio deste. Essas informações referem-se ao contexto do problema que o software pretende solucionar. Classes com o estereótipo <<entity>> também fornecem a informação de que normalmente haverá muitos objetos dessas classes e que estes, possivelmente, terão um período de vida longo, isto é, existe a possibilidade de que os objetos dessas classes precisem ser persistidos, ou seja, preservados fisicamente de alguma maneira.

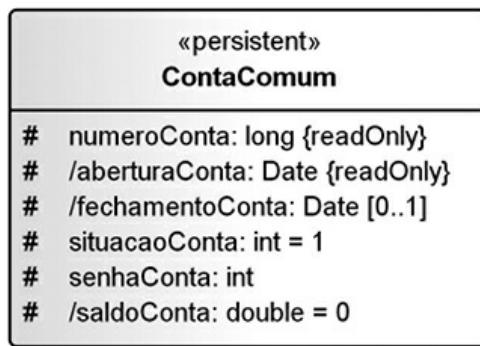
Frequentemente, costuma-se confundir classes detentoras do estereótipo <<entity>> com classes persistentes. Embora muitas delas realmente o sejam, isso não é uma regra. Uma classe do tipo <<entity>> pode ser transitória, isto é, seus objetos conservam suas informações somente em memória, não sendo necessário preservá-los permanentemente. Na figura 4.47, apresentamos um exemplo de classe com o estereótipo <<entity>>.



*Figura 4.47 – Classe ContaComum com o estereótipo <<entity>>.*

Nesse exemplo, deixamos explícito que a classe **ContaComum** é uma entidade, ou seja, que armazena informações referentes ao problema que o sistema no qual está inserida procura solucionar. Observe que esse estereótipo é um estereótipo gráfico, ou seja, modifica o desenho-padrão da classe. Um dos problemas de se utilizar esse estereótipo é que ele esconde os atributos e métodos da classe. Por outro lado, isso pode ser vantajoso em diagramas grandes, onde pode ser útil não apresentar essas características.

O fato de essa classe ser uma classe de entidade não significa obrigatoriamente que é uma classe persistente. Apesar de muitas vezes uma classe de entidade ser persistente, isso não é uma regra, visto que uma classe de entidade pode eventualmente ser transitória. Para deixar claro que uma classe é persistente, pode-se utilizar um estereótipo específico para isso, como o estereótipo <<persistent>>, conforme demonstrado na figura 4.48.



*Figura 4.48 – Classe ContaComum com o estereótipo <<persistent>>.*

Nesse exemplo, utilizamos a mesma classe da figura anterior, mas

utilizamos o estereótipo <<persistent>> para tornar explícito que a classe tem obrigatoriamente de preservar fisicamente, de alguma forma, suas instâncias. Diferentemente do estereótipo de entidade, este não é um estereótipo gráfico nem modifica o desenho-padrão da classe.

O padrão MVC costuma representar as classes com o estereótipo <<entity>> em uma camada específica, chamada Camada de Modelo. Nesse padrão, objetos de classes com o estereótipo <<entity>> não costumam interagir diretamente com objetos de classes de fronteira, pois isso é feito por intermédio de objetos de classes controladoras. É função das controladoras, representadas na camada de controle, solicitar a execução de métodos por instâncias de classes de entidade em resposta aos eventos ocorridos nas instâncias das classes de fronteira, representadas na camada de visão. Da mesma forma, os objetos controladores, de acordo com os resultados obtidos pelos métodos executados pelas classes de entidade, decidem quando os dados apresentados pelos objetos de fronteira devem ser atualizados. Na seção 4.7, serão apresentados exemplos de modelos de classes de entidade.

Aqui, é importante uma observação: o leitor poderá encontrar modelos em que os métodos referentes à lógica do negócio são colocados nas classes controladoras. Nessa estratégia, as classes de entidade se resumiriam a conter somente seus atributos e métodos dos tipos get e set. Esses tipos de métodos (getters e setters) não devem ser representados em modelos UML porque raramente acrescentam alguma informação importante ao modelo, já que sua função é retornar um valor para um atributo (get) e definir um valor para um atributo (set). Dessa forma, quando se adota essa estratégia, a camada de modelo muitas vezes contém apenas os atributos das classes, como se fosse um modelo conceitual.

Neste livro, não adotamos essa estratégia, visto que tentamos seguir a definição mais clássica do padrão MVC, conforme já explicamos. Consideramos a abordagem de colocar métodos referentes à lógica de negócios nas classes de controle inadequada e incondizente com o padrão MVC. Nesse ponto, estamos de acordo com Martin Fowler, que afirma que isso torna as classes de entidade “anêmicas”, uma vez que são desprovidas de comportamentos, o que é contrário ao projeto orientado a objetos de combinar dados e processos. Além disso, essa abordagem anula muitos dos

benefícios de uma camada de modelo e impede a aplicação de técnicas orientadas a objetos que permitem organizar uma lógica complexa.

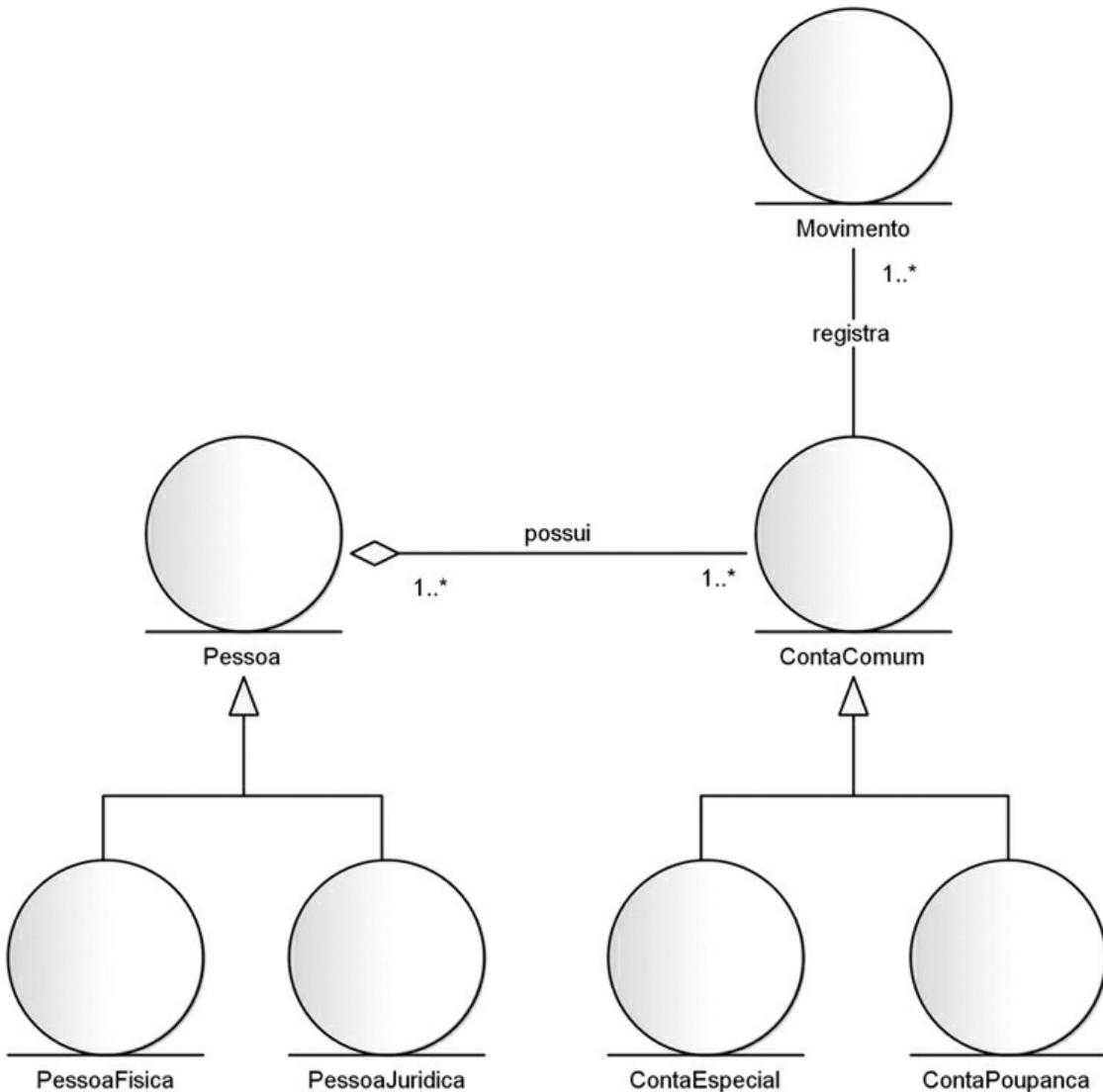
Ao utilizar o padrão MVC, adotamos a diretriz que as classes controladoras na camada de controle devem ser magras e as classes de entidade na camada de modelo devem ser gordas, no sentido que devem conter o comportamento relativo à lógica do negócio. Contudo, nada impede que o leitor adote outra abordagem, porém, se quiser utilizar os modelos descritos neste livro, terá que transferir métodos da camada de modelo para a camada de controle, o que pode ser um pouco difícil em algumas situações e necessitar de certas adaptações.

## **4.7 Exemplo de Diagrama de Classes (Modelo Conceitual) – Sistema de Controle Bancário**

Nesta seção, modelaremos o modelo conceitual do diagrama de classes para o sistema de controle bancário iniciado no capítulo 3. Conforme foi explicado, o modelo conceitual é produzido durante a fase de análise de requisitos e refere-se ao domínio do problema, enquanto o modelo de domínio é produzido durante a fase de projeto e refere-se ao domínio da solução. O modelo conceitual apresenta somente as informações que o sistema necessitará, enquanto o modelo de domínio toma o modelo conceitual e detalha questões como métodos, navegabilidade e até mesmo pode inserir novas classes, se isso for considerado necessário.

Ao longo deste estudo de caso, apresentaremos os dois modelos, enquanto nos exemplos seguintes apresentaremos somente o modelo de domínio. É importante destacar que o modelo de domínio somente deve ser desenvolvido na fase de projeto e deve ser modelado com o detalhamento dos casos de uso por meio de diagramas de interação, como o de sequência, onde se percebe quais métodos serão necessários em cada processo, enriquecendo-se, assim, o modelo de domínio.

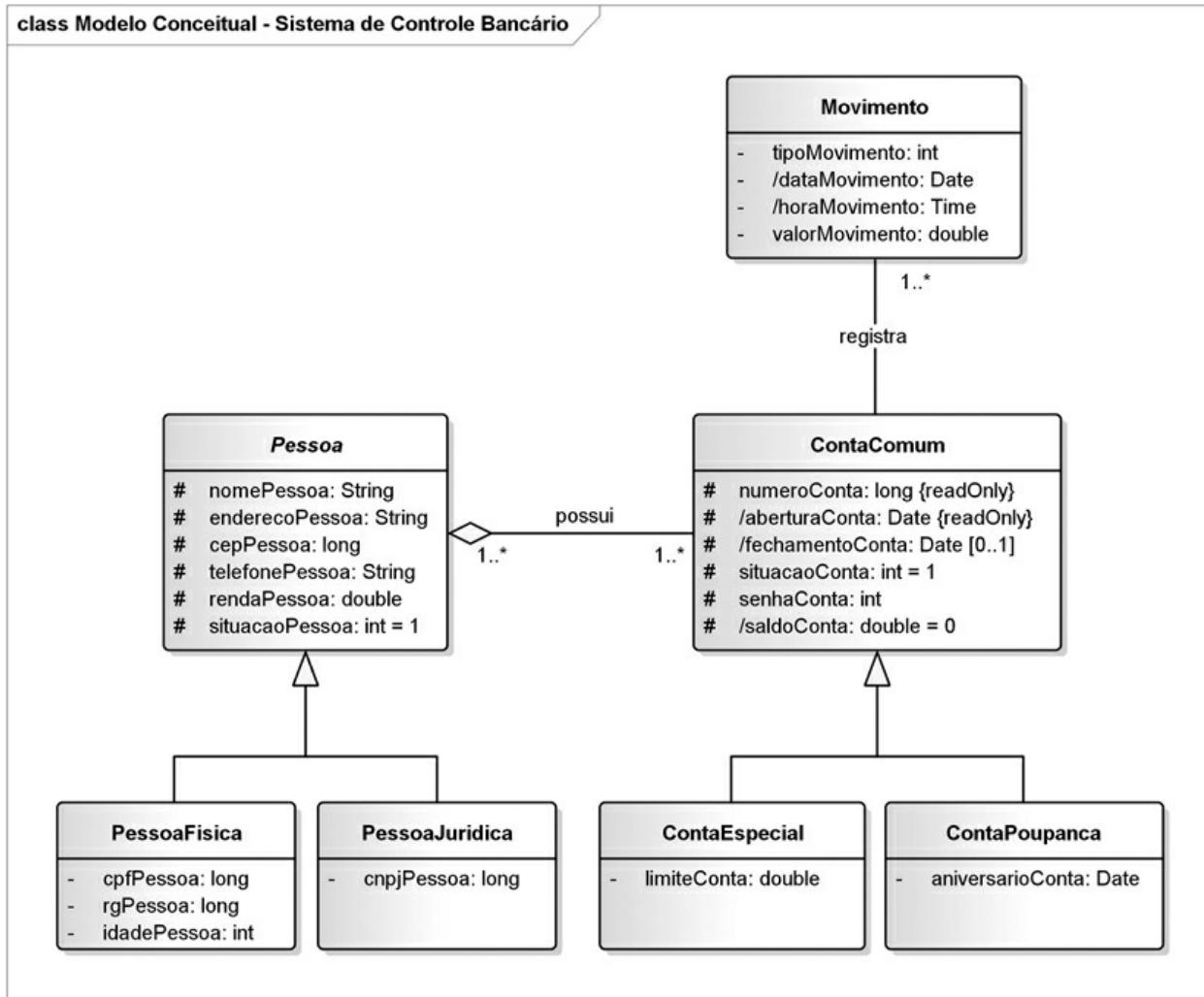
Cumpre ainda destacar que apesar de haver recursos já descritos como agregação, composição, generalização/especialização ou classes associativas, não é obrigatória sua utilização em todo diagrama de classes. Esses recursos deverão ser utilizados eventualmente, quando for necessário. A figura 4.49 ilustra o modelo conceitual para o sistema de controle bancário.



*Figura 4.49 – Diagrama de Classes para o Sistema de Controle Bancário – Modelo Conceitual – Uso do Estereótipo Entity.*

Nesta figura, as classes foram representadas com o estereótipo <<entity>>, que, como foi dito, é um estereótipo gráfico que modifica o desenho-padrão do componente. A vantagem de utilizar esse estereótipo é a de declarar explicitamente que se trata de uma classe de entidade e, em modelos grandes, diminuir o espaço do diagrama. Contudo, preferimos utilizar a imagem-padrão de uma classe porque assim podem ser vistos os atributos e métodos que ela contém. Nos próximos modelos, utilizaremos

a imagem-padrão para poder ilustrar os atributos e métodos de cada classe. A figura 4.50 apresenta o mesmo modelo, porém sem utilizar estereótipos de entidade.



*Figura 4.50 – Diagrama de Classes para o Sistema de Controle Bancário – Modelo Conceitual.*

Neste modelo, foram identificadas as seguintes classes:

- **ContaComum** – Essa classe foi parcialmente descrita nas seções anteriores. Seu objetivo é armazenar as contas-correntes comuns, sem limite de cheque especial, gerenciadas pelo banco. Nesse tipo de conta, não é possível realizar saques além do valor real depositado. A classe **ContaComum** está associada a duas especializações ou subclasses que herdam seus atributos e métodos, representadas pelas classes **ContaEspecial** e **ContaPoupanca**.

A classe **ContaComum** tem os atributos número da conta (**numeroConta**) do tipo **long**, data da abertura (**aberturaConta**) e data de encerramento (**fechamentoConta**) do tipo **Date**, **situacaoConta** (se está ativa ou inativa), **senhaConta** do tipo **int** e **saldoConta** do tipo **double**. Observe que os atributos da classe **ContaComum** são todos protegidos, o que os torna visíveis às suas subclasses. Os atributos dessa classe têm ainda características extras, como a definição de que **aberturaConta**, **fechamentoConta** e **saldoConta** são atributos que receberão algum tipo de cálculo (ou atribuição), que o atributo **fechamentoConta** não necessariamente precisará ser informado e que os valores iniciais de **situacaoConta** e **saldoConta** serão, respectivamente, 1 (ao abrir uma conta, ela se tornará ativa) e 0 (enquanto nenhum depósito for realizado).

É importante destacar que o tipo **Date** refere-se a uma classe e se esta não for implementada pela linguagem adotada para a codificação do sistema, deverá ser criada.

Nas seções anteriores, apresentamos exemplos de restrições que poderiam ser aplicadas aos atributos dessa classe, mas com o objetivo de não deixar o diagrama muito poluído, com grande quantidade de informações, preferimos detalhar esse tipo de informação em diagramas separados, quando necessário. Poderíamos acrescentar, ainda, as restrições **readOnly** aos atributos **numeroConta** e **aberturaConta**, uma vez que, após a criação, o número da conta e sua data de abertura não podem ser modificados.

- **ContaEspecial** – A classe **ContaEspecial** representa as contas que permitem ao correntista sacar valores superiores a seu saldo, até um limite estabelecido. Essa classe tem um atributo particular, além dos herdados da classe **ContaComum**, chamado **limiteConta** do tipo **double**, que representa o limite máximo que o correntista pode retirar além do saldo de sua conta.
- **ContaPoupanca** – A classe **ContaPoupanca**, como o próprio nome indica, representa as contas de poupança mantidas pela instituição bancária. Essa classe tem um único atributo particular, além dos herdados, chamado **aniversarioConta** do tipo **Date**, que representa a

data em que o valor depositado na conta renderá juros, que não é necessariamente equivalente à data de abertura, uma vez que as contas de poupança não exigem um depósito quando de sua abertura.

- **Pessoa** – A classe **Pessoa** armazena as informações gerais dos clientes. Tanto pessoas físicas como jurídicas podem possuir contas na instituição bancária. A classe **Pessoa** tem uma associação do tipo agregação com a classe **ContaComum**, o que significa que uma ou mais instâncias dessa classe ou de suas subclasses complementam as informações armazenadas por instâncias da classe **Pessoa**. Assim, sempre que uma determinada pessoa for consultada, as informações sobre suas contas deverão ser também apresentadas.

Observe que a multiplicidade da associação **possui** existente entre as classes **Pessoa** e **ContaComum** é muitos (\*) em ambas as extremidades, significando que uma pessoa pode possuir, no mínimo, uma e, no máximo, muitas contas e uma conta pode ser possuída por uma ou mais pessoas, como no caso de contas conjuntas. O leitor pode se perguntar se não haveria necessidade de inserir uma classe associativa ou intermediária entre essa associação, porém isso somente seria necessário se houvesse um atributo específico relativo a uma determinada pessoa possuindo uma determinada conta que não pudesse ser armazenado nem na classe **Pessoa** nem na classe **ContaComum**.

É preciso notar que a classe **Pessoa** é uma superclasse abstrata (por isso seu nome está em itálico), que não tem instâncias e, portanto, não interage realmente com a classe **ContaComum** nem com suas especializações. As classes que interagem com a classe **ContaComum** são as subclasses **PessoaFisica** e **PessoaJuridica**, que herdam todos os atributos da classe **Pessoa** e ainda têm seus próprios atributos particulares.

Os atributos da classe **Pessoa** são nome, endereço e telefone do tipo **String**, CEP do tipo **long**, renda do tipo **double** e situação (se está ativa – possui contas ainda não encerradas – ou inativa) do tipo **int**.

- **PessoaFisica** – A classe **PessoaFisica** é uma subclass derivada da classe **Pessoa**, representando as pessoas físicas que possuem ou possuíram contas ativas na instituição bancária. Essa classe herda todos

os atributos de sua superclasse, tendo os atributos extras CPF e carteira de identidade (RG) do tipo `long` e idade do tipo `int`.

- **PessoaJurídica** – Essa classe também é uma subclasse derivada da classe **Pessoa**, representando as pessoas jurídicas que possuem ou possuíram contas ativas na instituição. Da mesma maneira, a classe **PessoaJurídica** herda todos os atributos da classe **Pessoa**, possuindo somente o atributo particular CNPJ da empresa do tipo `long`.
- **Movimento** – Essa classe é responsável por armazenar as transações ocorridas nas contas, ou seja, todos os saques e depósitos. A classe **Movimento** tem como atributos o tipo de movimento (convencionamos atribuir 0 para saque e 1 para depósito) do tipo `int`, a data do movimento (**dataMovimento**) do tipo **Date**, a hora do movimento (**horaMovimento**) do tipo **Time** e o valor movimentado do tipo `double`. Observe que o tipo **Time**, da mesma forma que o **Date**, refere-se a classes e não a tipos primitivos, portanto se essas classes não forem implementadas pela linguagem, terão que ser igualmente codificadas. A classe **Movimento** tem uma associação binária simples com a classe **ContaComum**. A multiplicidade dessa associação demonstra que uma conta terá, no mínimo, um movimento (após a abertura, é obrigatório depositar algum valor) e, no máximo, muitos. Essa associação não caracteriza uma composição porque as informações da classe **Movimento** não complementam as informações da classe **Conta**, quando esta for consultada.

## 4.8 Como Identificar Classes

O leitor pode se perguntar como essas classes foram identificadas e de que maneira se concluiu que seriam as classes corretas para constar no modelo conceitual. Isso é um pouco influenciado pela própria experiência da equipe, todavia existem algumas técnicas para identificar as “classes candidatas”, ou seja, as classes que têm probabilidade de compor o modelo. A estratégia mais comum é examinar a descrição dos requisitos, podendo estas estar contidas em documentos de requisitos ou na documentação dos casos de uso, por exemplo.

Nessa análise textual, costuma-se procurar por substantivos e verbos (ou

descrições de ações). Os substantivos podem representar as classes candidatas ou seus atributos (é necessário verificar os possíveis sinônimos de um substantivo e agrupá-los em uma classe ou atributo único), enquanto os verbos podem identificar as operações válidas para uma determinada classe ou associações entre as classes.

Também é útil procurar descrições de restrições ou condições nos substantivos e verbos que foram identificados anteriormente. Os requisitos não funcionais costumam conter esse tipo de informação e são úteis para identificar as possíveis restrições que deveriam ser aplicadas às classes, seus atributos, operações e associações.

Dessa maneira, para cada substantivo identificado na documentação de requisitos, deve-se tentar representar uma classe correspondente no modelo conceitual ou, eventualmente, um atributo contido em uma das classes do modelo. Em seguida, para cada descrição de ação, deve-se procurar identificar um comportamento ou combinação de comportamentos associado a uma classe. Verbos normalmente identificam ações, contudo, eventualmente, podem identificar uma associação entre as classes. É necessário certificar-se também de que as operações (comportamentos) de cada classes recebam os dados corretos para que o comportamento seja executado a contento. Em geral, esses dados serão atributos que deverão estar contidos em uma classe. É importante destacar que esse processo pode (e deve) passar por refinamentos, em que o modelo inicial deve ser analisado, corrigido (se necessário) e melhorado.

Assim, se analisarmos o item 3.15 no capítulo 3, poderemos identificar alguns substantivos, como clientes, pessoas físicas, pessoas jurídicas, contas comuns, contas especiais, contas de poupança e movimentos.

Para desenvolver o modelo conceitual apresentado na figura 4.50, consideramos que os termos “clientes” e “pessoas” são sinônimos e preferimos adotar a terminologia genérica “Pessoa” para identificar uma classe, enquanto Pessoa Física e Pessoa Jurídica tornaram-se especializações dessa classe. Cada um dos tipos de conta (comum, especial e poupança) foi identificado como uma classe, porém as duas últimas foram representadas como especializações da classe **ContaComum**. Também **Movimento** tornou-se uma classe e foi associada à classe **ContaComum** e, automaticamente, também às suas especializações, por meio da associação

de herança.

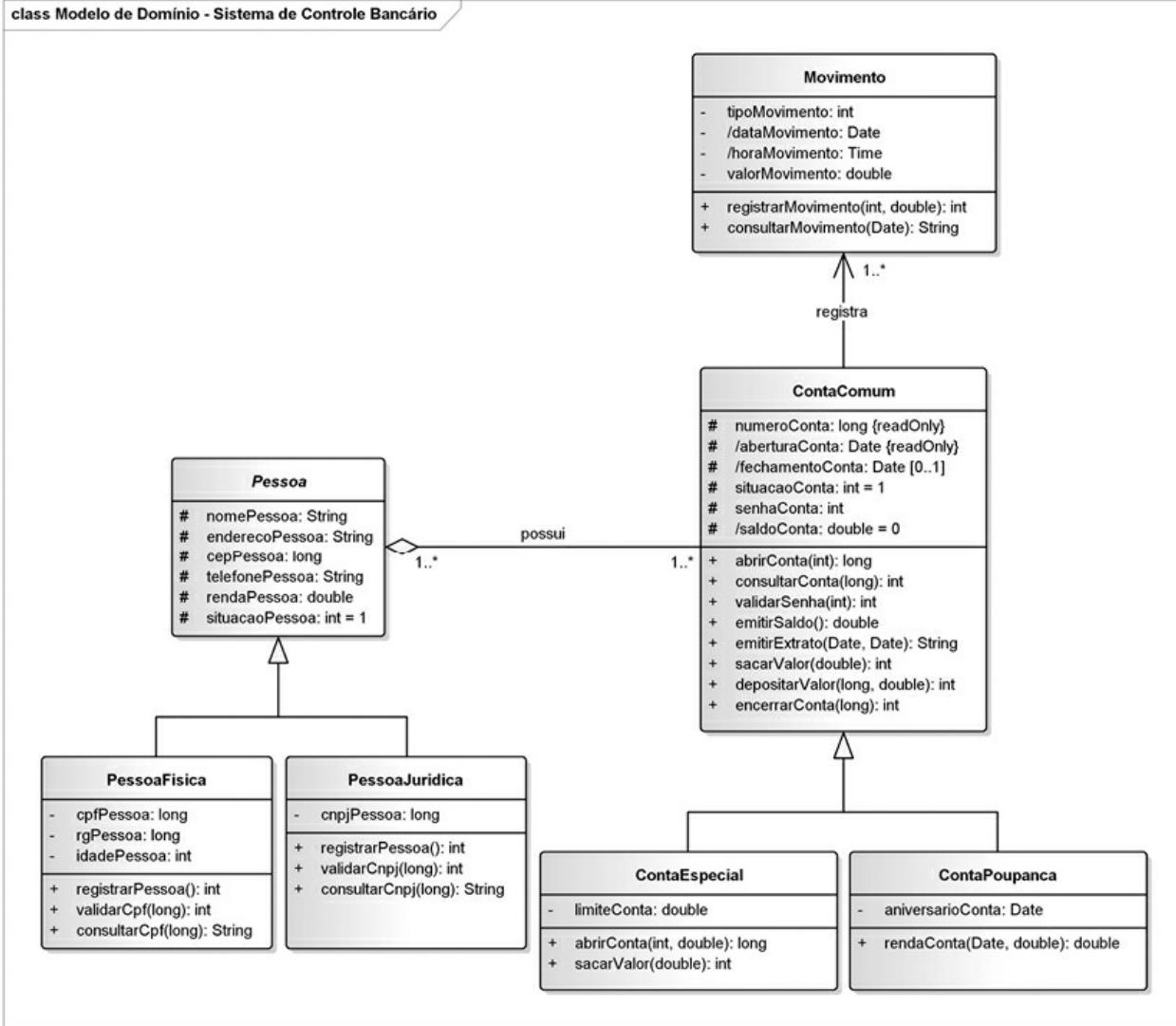
Nesse modelo, como dito anteriormente, não são identificados métodos, porque estes fazem parte da solução e só são representados na fase de projeto. Assim, os verbos não foram analisados com muita profundidade nessa fase e apenas serviram para auxiliar a identificar as associações “**registra**” entre as classes **ContaComum** e **Movimento** e “**possui**” entre as classes **Pessoa** e **ContaComum**. Os verbos foram melhor analisados durante a fase de projeto, na qual foi produzido o modelo de domínio apresentado a seguir.

Obviamente, o modelo explicado nesta seção já passou por alguns refinamentos e o diagrama apresentado é a versão final do modelo conceitual para este estudo de caso.

## 4.9 Exemplo de Modelo de Domínio

A seguir, apresentaremos o modelo de domínio desse sistema, no qual detalhamos os métodos necessários às classes identificadas no modelo conceitual. Lembramos que o modelo de domínio costuma ser produzido somente na fase de projeto e seus métodos são identificados por meio da modelagem de diagramas de interação correspondentes aos processos representados no modelo de casos de uso.

A figura 4.51 apresenta o mesmo diagrama de classes para o sistema de controle bancário explicado anteriormente, mas, desta vez, o diagrama representa o modelo de domínio e enfoca a solução do problema a ser resolvido pelo sistema. Como podemos observar, foram acrescentados métodos e naveabilidade a esse modelo.



*Figura 4.51 – Diagrama de Classes para o Sistema de Controle Bancário – Modelo de Domínio.*

É importante destacar que todas as outras classes do diagrama, com exceção da classe **Pessoa** (que é uma classe abstrata, como demonstra seu nome em itálico), são classes de entidade, mas optamos por não utilizar o estereótipo <<entity>> porque este, por ser gráfico, modifica o desenho-padrão da classe e esconde seus atributos e métodos, o que atrapalharia a explicação desse diagrama. A seguir, detalharemos os métodos identificados para cada classe, que foram descobertos quando do detalhamento dos casos de uso por meio de diagramas de sequência e comunicação que serão explanados nos próximos capítulos.

- **ContaComum** – Os métodos identificados para essa classe foram:

Método	Descrição
<b>abrirConta</b>	A função desse método é abrir uma nova conta, onde um novo objeto dessa classe será instanciado, de forma que esse método agirá quase como um método construtor. Esse método recebe como parâmetro somente a senha da conta, a ser definida pelo cliente, uma vez que o número da conta é gerado automaticamente e retornado pelo método. A data de abertura é tomada do sistema, a de encerramento é deixada indefinida e a situação e o saldo têm valores iniciais predefinidos. O retorno do método é o próprio número da conta, se o método for concluído com sucesso, ou zero, se ocorrer algum erro durante sua execução.
<b>consultarConta</b>	Tem por objetivo descobrir se uma determinada conta existe. Recebe um <b>long</b> que armazena o número da conta e retorna um <b>inteiro</b> para determinar se a conta foi encontrada (1) ou não (0).
<b>validarSenha</b>	Determina se a senha informada é válida. O método recebe a senha como parâmetro e retorna um inteiro determinando se a senha é a correta (1) ou não (0).
<b>emitirSaldo</b>	Retorna o valor contido na conta, retornando um valor <b>double</b> que armazena o saldo da conta. Como esse método só pode ser chamado após o método <b>Consulta</b> , não é preciso informar o número da conta, visto que já foi informada.
<b>emitirExtrato</b>	Retorna os movimentos realizados na conta em um determinado período. Recebe como parâmetro as datas iniciais e finais do extrato. O método tem a necessidade de solicitar a execução de outro método sobre a classe <b>Movimento</b> .
<b>sacarValor</b>	Diminui o valor solicitado para saque do saldo da conta. Recebe como parâmetro o valor a ser retirado do saldo e retorna verdadeiro, se foi possível realizar a operação, ou falso, em caso contrário. O retorno falso pode ocorrer por a conta não ter saldo suficiente para o valor solicitado. Como o método <b>sacarValor</b> só poderá ser chamado após a conta ter sido consultada e sua senha, validada, não é necessário receber o número da conta, pois este já foi informado no momento da consulta da conta.
<b>depositarValor</b>	Soma o valor fornecido pelo cliente ao saldo de uma conta. Recebe como parâmetro o número da conta (o depósito pode ser relativo à outra conta) e o valor a ser depositado. Tanto esse método como o <b>sacarValor</b> disparam um método na classe <b>Movimento</b> para registrar o movimento realizado sobre a conta.
<b>encerrarConta</b>	Encerra uma conta já existente, tornando-a inativa. Não é um método destrutor, pois apenas altera o valor do atributo <b>situacao</b> para 0, indicando que a conta se encontra inativa. Possivelmente, faz o mesmo com o cliente se esta for a única conta por ele possuída. Uma conta não pode ser excluída: é preciso mantê-la para fins de histórico bancário. Esse método não recebe parâmetros, já que para ser encerrada a conta precisa ser, primeiro, consultada por meio do método <b>consultarConta</b> e, depois

disso, o número da conta já estará armazenado na memória. Esse método retorna 1, caso a conta tenha sido encerrada com sucesso, ou 0, caso não tenha sido possível encerrá-la.

O leitor talvez se pergunte o porquê de o método `abrirConta` ser considerado um método construtor. Na verdade, recomenda-se que os métodos construtores e destrutores não sejam representados no diagrama de classes porque a forma como são implementados varia muito entre as linguagens de programação. Basicamente, um método criador aloca memória para uma instância de uma determinada classe e, depois, determina valores para seus atributos, enquanto um método destrutor libera a memória utilizada por uma determinada instância. Ao seguir esse raciocínio, o método `abrirConta` pode perfeitamente implementar a rotina para alocar memória para uma nova instância da classe `ContaComum` ou chamar um método para isso antes de finalizar a abertura da conta. Na verdade, esse método não seria totalmente obrigatório, já que a própria classe controladora poderia chamar um método construtor, mas optamos por defini-lo por considerarmos esse comportamento importante para o sistema.

- **ContaEspecial** – A classe `ContaEspecial` herda todos os métodos da classe `ContaComum`, mas tem ainda três métodos particulares, a saber:

Método	Descrição
<code>abrirConta</code>	Este é uma redeclaração do método <code>abrirConta</code> da classe <code>ContaComum</code> . O método precisa ser redeclarado pela necessidade de se incluir o atributo <code>limite</code> , inexistente na classe <code>ContaComum</code> , no momento da abertura da conta. Essa redeclaração do método <code>abrirConta</code> caracteriza um exemplo de polimorfismo, já que a classe redeclarada tem o mesmo nome, mas algumas diferenças em sua implementação.
<code>sacarValor</code>	O segundo método da classe <code>ContaEspecial</code> também é uma redeclaração do mesmo método existente na classe <code>ContaComum</code> . Esse método precisa ser redeclarado para incluir o uso do atributo <code>limite</code> , pois um correntista que possua uma conta especial pode sacar valores superiores a seu saldo real até o limite estabelecido pelo atributo de mesmo nome. Tal redeclaração caracteriza outro exemplo de polimorfismo, uma vez que a chamada do método permanece a mesma, diferenciando-se na forma como o método é implementado.

- **ContaPoupança** – Da mesma forma que a classe `ContaEspecial`, essa classe herda todos os métodos da classe `ContaComum`, tendo somente um

método extra:

Método	Descrição
<b>rendaConta</b>	Recebe como parâmetros a data atual e o percentual de juros que as contas com aniversário no dia receberão. O método aplica-se a todas as instâncias cujo dia da data de aniversário seja igual ao dia da data atual.

- **PessoaFisica** – Essa classe tem os seguintes métodos:

Método	Descrição
<b>registrarPessoa</b>	É responsável por instanciar um novo objeto da classe. Uma vez que esse método recebe como parâmetro todos os atributos da classe, com exceção da situação, cujo valor inicial é 1 (ativo), optamos por não detalhar a assinatura do método para não deixar a classe larga demais, o que atrapalharia a visualização da figura.
<b>validarCPF</b>	É utilizado para determinar se o CPF informado é válido. Esse método é chamado quando do registro de uma nova pessoa física e recebe como parâmetro um <b>long</b> que armazenará o valor do CPF a validar.
<b>consultarCpf</b>	Permite consultar uma pessoa por seu CPF. Se o CPF for encontrado, retornará uma String com os dados do cliente.

- **PessoaJuridica** – Os métodos dessa classe são descritos a seguir:

Método	Descrição
<b>registrarPessoa</b>	Tem a mesma função do método de mesmo nome explicado na classe anterior, diferenciando-se por não registrar o CPF nem o RG da pessoa, e sim o CNPJ da empresa.
<b>validarCnpj</b>	É utilizado para determinar se o CNPJ informado é válido. Esse método é chamado quando do registro de uma nova pessoa jurídica e recebe como parâmetro um <b>long</b> que armazenará o valor do CNPJ a validar.
<b>consultarCnpj</b>	Permite consultar uma pessoa jurídica por seu CNPJ. Se o CNPJ for encontrado, retornará uma String contendo os dados do cliente.

- **Movimento** – Essa classe tem os métodos explanados a seguir:

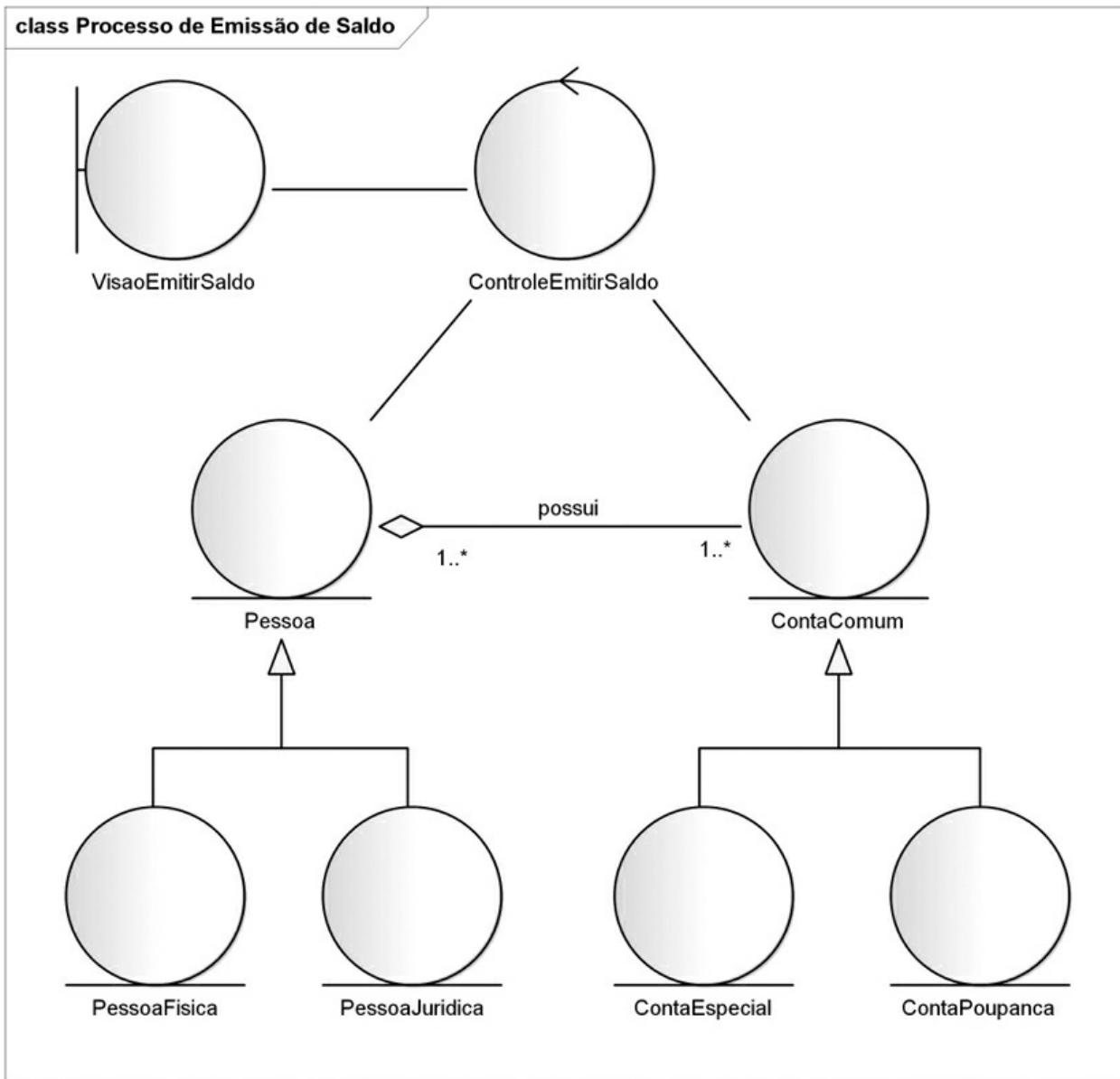
Método	Descrição
<b>registrarMovimento</b>	É responsável por registrar cada movimento ocorrido em alguma conta. Esse método recebe como parâmetros o tipo de movimento (se foi saque ou depósito) do tipo <b>int</b> e o valor do movimento do tipo <b>double</b> , pois a data e a hora do movimento são capturadas diretamente do sistema. O método retornará 1 se conseguir registrar o movimento, caso contrário, 0. Esse método é chamado pelos métodos <b>sacarValor</b> e <b>depositarValor</b> declarados na classe <b>ContaComum</b> .
<b>consultarMovimento</b>	É utilizado para consultar todos os movimentos pertencentes a uma determinada data, recebe como parâmetro a data da consulta e retorna uma <b>String</b> contendo os valores dos atributos de cada

objeto.

Aqui, é necessário fazer uma observação principalmente em relação aos exemplos seguintes. A rigor, uma classe deve ter um **get** e um **set** para cada um de seus atributos. No entanto, torna-se inviável e enfadonho criar esses métodos para cada classe; se a classe tiver muitos atributos, tornar-se-á muito grande e, por conseguinte, ficará difícil ler a representação do diagrama como um todo. Por esse motivo, recomenda-se não representar esse tipo de método em diagramas de classe, mesmo porque a informação que esse tipo de método acrescenta é pífia. Assim, optamos por utilizar métodos genéricos como registrar ou consultar, para instanciar ou retornar todos os valores de uma classe com o objetivo de simplificar o modelo, o que não impede que o leitor proceda de forma diferente, se assim achar necessário.

Também é possível criar diagramas referentes a um caso de uso específico, contendo somente as classes de fronteira, controle e entidade envolvidas no processo representado pelo caso de uso. No estudo de caso do sistema de controle bancário existem poucos exemplos que possam ser aplicados nesse sentido, uma vez que a maioria dos processos envolve praticamente todas as classes, no entanto podemos ilustrar essa prática por meio do processo emitir saldo, que não envolve objetos da classe **Movimento**, conforme demonstra a figura 4.52.

No exemplo apresentado na figura 4.52, ilustramos as classes envolvidas no processo para emitir o saldo de uma conta. Pode-se perceber que a classe **VisaoEmitirSaldo** é a classe de fronteira que representa a interface com o usuário (pode haver mais de uma interface para o mesmo processo para englobar diversas dispositivos e formas de acesso para o mesmo processo) e a classe **ControleEmitirSaldo** (ou melhor, suas instâncias) é responsável por controlar o processo para emissão do saldo de uma conta. As classes de entidade aqui presentes foram explicadas anteriormente.



*Figura 4.52 – Diagrama de Classes para o Processo de Emissão de Saldo – Sistema de Controle Bancário.*

Nos modelos seguintes, apresentaremos somente modelos de domínio contendo classes de entidade, ou seja, as classes contidas na camada de modelo do padrão MVC. Consideraremos que as classes de fronteira e controle estão definidas em suas respectivas camadas.

## 4.10 Exemplo de Diagrama de Classes – Sistema de Telefone Celular

Nesta seção, apresentaremos o diagrama de classes referente ao modelo de domínio para o sistema de telefone celular, cuja modelagem foi iniciada no

capítulo sobre o diagrama de casos de uso. A figura 4.53 apresenta o modelo de domínio desse sistema.

A seguir, descreveremos as classes de entidade que compõem esse diagrama, com seus atributos, métodos e associações.

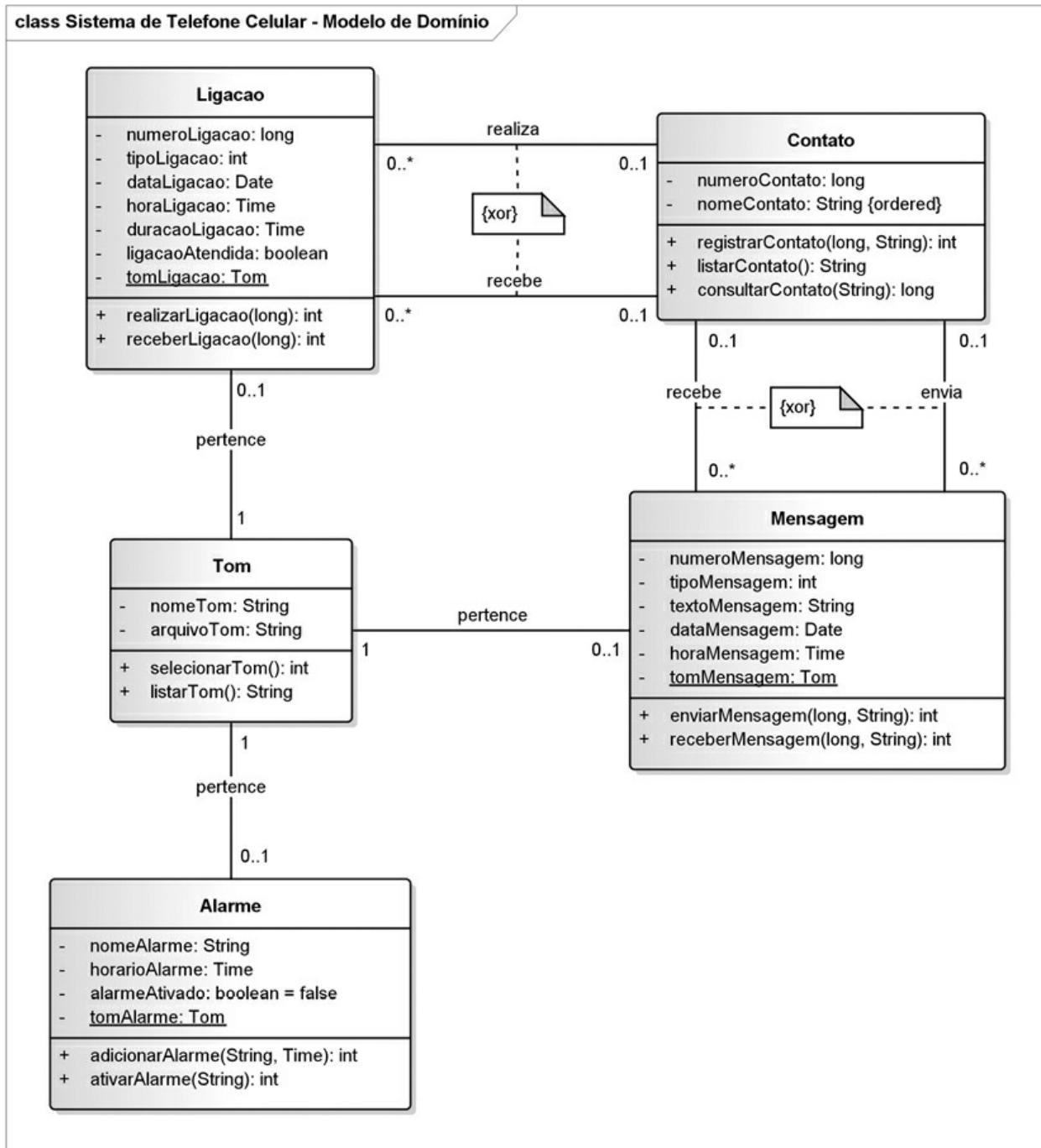


Figura 4.53 – Diagrama de Classes para o Sistema de Telefone Celular – Modelo de Domínio.

O leitor notará que incluímos as assinaturas dos métodos das classes, uma vez que o diagrama não é muito grande e o número de parâmetros dos métodos é pequeno, o que permite esse detalhamento.

- **Contato** – Essa classe é autoexplicativa, sendo seu objetivo armazenar as informações dos contatos que o usuário mantém em seu celular. A classe tem os atributos número do contato do tipo `long` e nome do contato do tipo `String`. Observe que esse último atributo possui a restrição `{ordered}`, ou seja, a coleção está organizada em ordem alfabética.

A classe apresenta ainda os seguintes métodos:

Método	Descrição
<code>registrarContato</code>	Tem como função instanciar um novo objeto da classe <code>Contato</code> . Retorna um inteiro que representará verdadeiro (1), se a operação for realizada com sucesso, ou falso (0), caso contrário.
<code>listarContato</code>	Executa a tarefa de listar os contatos armazenados no aparelho. O método é chamado enquanto houver contatos, apresentando o nome de cada contato na tela do aparelho, para que o usuário possa visualizá-los.
<code>consultarContato</code>	Tem por objetivo consultar um contato a partir da listagem fornecida após a execução do método <code>visualizarContato</code> . O método receberá como parâmetro o nome do contato a consultar e retornará o número correspondente.

- **Ligacao** – Essa classe armazena as ligações realizadas e recebidas pelo usuário. Tem como atributos o número da ligação do tipo `long`, o tipo da ligação (se foi realizada ou recebida) do tipo `int`, a data da ligação do tipo `Date`, a hora e a duração da ligação do tipo `Time`, se a ligação foi atendida (verdadeiro) ou não (falso), do tipo `boolean` e a definição do tom que será tocado quando uma ligação for recebida do tipo `Tom`, ou seja, esse atributo guarda um objeto da classe `Tom`, que será explicada a seguir, e refere-se ao som a ser executado quando uma ligação for recebida. Observe que esse último atributo é um atributo estático, conforme demonstra seu sublinhado, ou seja, esse é um atributo pertencente à classe e não a seus objetos. A classe contém ainda os seguintes métodos:

Método	Descrição
	Tem a função de fazer uma chamada para outro telefone, recebendo como parâmetro o número a discar e retornando verdadeiro, se a ligação foi realizada, ou falso, caso contrário. O método deve ainda instanciar um

**realizarLigacao** objeto da classe **Ligacao** e o tipo da ligação será 1, que, por convenção, identifica uma chamada enviada. A data e a hora serão tomadas do sistema, a duração será contada enquanto a ligação durar e o atributo **ligacaoAtendida** receberá verdadeiro, se a ligação for atendida, ou falso, caso contrário.

**receberLigacao** Executa a tarefa de receber uma ligação, recebendo como parâmetro o número do aparelho que está solicitando atendimento. Esse método também deverá instanciar um objeto da classe **Ligacao**, sendo a ligação atendida ou não. Se o usuário atender a chamada, o atributo **atendidaLigacao** receberá o valor verdadeiro, caso contrário, falso. O valor dos outros atributos será idêntico ao método anterior, excetuando-se o tipo, que receberá o valor 2.

Observe que a classe **Ligacao** tem duas associações binárias com a classe **Contato**. A primeira associação, cujo nome é **realiza**, indica uma situação em que um contato previamente cadastrado liga para o aparelho. Note que um contato pode nunca ligar para o usuário ou pode fazer muitas ligações, e uma ligação pode não ser realizada por nenhum contato (o número pode ser desconhecido).

A segunda associação refere-se a uma situação em que, a partir da consulta de um contato, o usuário do aparelho faz uma chamada para este, passando como parâmetro o número consultado. Da mesma maneira que na associação anterior, um contato pode receber muitas ligações ou nunca receber nenhuma, e uma ligação pode não ser recebida por nenhum contato cadastrado. Observe ainda que existe uma restrição entre as duas associações, do tipo ou exclusivo (**xor**), ou seja, um contato não pode fazer e receber uma mesma ligação.

Há, ainda, uma terceira associação com a classe **Tom** explicitando que um determinado tom deve estar associado às ligações recebidas pelo aparelho.

- **Mensagem** – Essa classe armazena as mensagens enviadas e recebidas pelo usuário. Tem como atributos o número do aparelho para o qual a mensagem foi enviada, do tipo **long**; o tipo da mensagem (se foi enviada ou recebida), do tipo **int**; o texto da mensagem, do tipo **String**; a data da mensagem, do tipo **Date**; a hora da mensagem, do tipo **Time**; e o som da mensagem, do tipo **Tom**. Essa classe contém ainda os métodos **enviarMensagem** e **receberMensagem**, bastante semelhantes aos

métodos declarados na classe **Ligacao**, diferenciando-se principalmente por, além do parâmetro do número do telefone, receberem ainda a mensagem a enviar ou receber.

Observe que a classe **Mensagem** tem duas associações com a classe **Contato**, que representam as situações em que um contato envia ou recebe uma mensagem, podendo ocorrer de a mensagem não ser enviada ou recebida de nenhum contato conhecido ou estar associada a um único contato. Além disso, uma mensagem não pode ser enviada e recebida ao mesmo tempo pelo destinatário, como demonstra a restrição do tipo ou exclusivo. Há ainda uma associação com a classe **Tom** que determina que uma instância dessa classe precisa estar associada à classe **Mensagem**.

- **Alarme** – Essa classe armazena os horários em que o usuário deseja ser despertado. Tem como atributos o nome do alarme, do tipo **String**; o horário que o alarme deverá despertar, do tipo **Time**; e o atributo **alarmeAtivado**, que determina se o alarme em questão está ativado (deve despertar) ou não, do tipo **boolean**. O valor inicial para esse último atributo é falso. Como em outras classes, há ainda um atributo que define a música que deverá tocar quando o alarme despertar, do tipo **Tom**. A classe contém os seguintes métodos:

Método	Descrição
<b>adicionarAlarme</b>	Instancia um novo objeto da classe <b>Alarme</b> , recebendo como parâmetros o nome do alarme e seu horário, e o atributo <b>ativadoAlarme</b> será iniciado com falso e não precisa ser passado como parâmetro. Esse método retorna verdadeiro, se a operação for realizada com sucesso, e falso, caso contrário.
<b>ativarAlarme</b>	Recebe o nome do alarme a ativar (ou desativar, caso já esteja ativado) como parâmetro, modificando o valor do atributo <b>ativadoAlarme</b> para verdadeiro, se estiver falso, ou falso, caso contrário.

Pode-se notar que as classes **Ligacao**, **Mensagem** e **Alarme** têm um atributo muito semelhante que representa o tom a ser tocado quando uma ligação ou mensagem for recebida ou quando chegar a hora de o alarme despertar. Deve-se observar também que esse é um atributo estático, ou seja, um atributo que não pertence aos objetos, mas à classe propriamente dita, uma vez que sempre que uma ligação ou mensagem

for recebida, a mesma música deverá tocar. O atributo `tom` é uma `String` porque se refere ao nome do tom que deverá ser tocado e o aparelho tem um conjunto de tons que podem ser selecionados, mas somente um para as chamadas recebidas, outro para as mensagens recebidas e outro para o despertar do alarme.

- **Tom** – Essa classe armazena os tons disponíveis no aparelho. Seus atributos são o nome do tom e o nome do arquivo que o celular deverá procurar para executar a música, ambos do tipo `String`. Os métodos dessa classe são `selecionarTom`, que apresenta ao usuário uma lista na qual este deverá escolher entre ligação, mensagem ou alarme. Se o proprietário do aparelho escolher uma das opções, o método chamará o método `listarTom` para visualizar os tons disponíveis e permitir que o usuário selecione um deles para a função previamente escolhida. Esta classe possui associações com as classes `Ligacao`, `Mensagem` e `Alarme` por meio das quais são definidos os sons que serão executados quando do recebimento de uma ligação, mensagem ou quando o alarme for ativado. Por meio dessas associações, pode-se verificar que um objeto da classe Tom pode estar associado à classe `Ligacao` ou à classe `Mensagem` ou à classe `Alarme`.

## 4.11 Exemplo de Diagrama de Classes – Sistema de Biblioteca

Nesta seção, apresentaremos o modelo de domínio representado pelo diagrama de classes do sistema de controle de biblioteca que iniciamos a modelar no capítulo sobre o diagrama de casos de uso, conforme ilustrado pela figura 4.54.

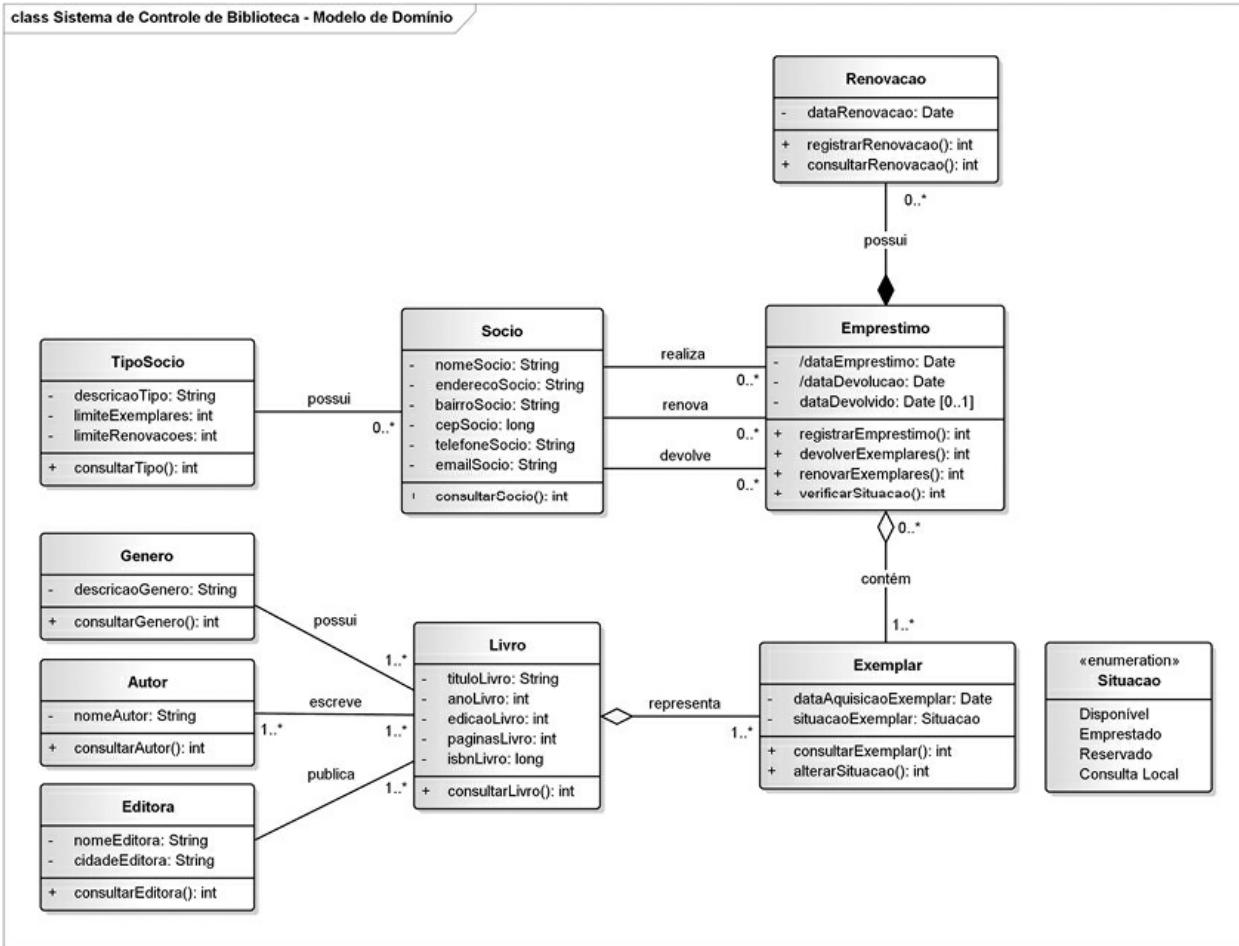


Figura 4.54 – Diagrama de Classes para o Sistema de Biblioteca – Modelo de Domínio.

Nesse modelo, explicitamos somente os métodos que consideramos importantes e são utilizados pelos casos de uso primários. Não consideramos necessário representar métodos cuja única função seria registrar ou excluir objetos em classes simples. Detalharemos a seguir as classes de entidade identificadas nesse modelo.

- **TipoSocio** – Essa classe representa os tipos de sócio aceitos pela biblioteca, como professores, alunos, funcionários e pessoas da comunidade externa. Essa classe é importante porque, além de determinar que tipo de sócio está realizando um empréstimo, estabelece qual o limite máximo de exemplares um sócio de um determinado tipo pode tomar emprestado por meio do atributo **limiteExemplares**, do tipo inteiro. Quando esse limite for atingido, o sócio não poderá locar mais livros até que devolva algum exemplar que tenha tomado

emprestado anteriormente. A classe também estabelece o número máximo de vezes que um sócio pode renovar um empréstimo, por meio do atributo `limiteRenovacoes`, do tipo inteiro. Essa classe poderia ser uma enumeração, porém preferimos deixá-la como uma classe de entidade para permitir que novos tipos de sócio sejam cadastrados sem que seja necessário dar manutenção ao sistema. A classe `TipoSocio` também contém um método para consultar um tipo de sócio.

- **Socio** – Essa classe armazena as informações dos sócios que podem retirar livros emprestados da biblioteca. Seus atributos e métodos são autoexplicativos. Observe que uma instância da classe `Socio` possui três associações com a classe `Emprestimo`, que informam que um sócio pode realizar nenhum ou muitos empréstimos, bem como devolver ou renovar nenhum ou vários empréstimos.
- **Gênero** – Essa classe tem como função armazenar os possíveis gêneros de livros possuídos pela biblioteca, como aventura, ficção científica, época, romance, terror, história, geografia, matemática, biologia, física etc. Essa classe tem como atributo a descrição do gênero do tipo `String`. Também representamos aqui um método para consultar um determinado gênero.
- **Autor** – Essa classe tem como função armazenar os autores que escreveram os livros possuídos pela biblioteca. Como pode se perceber pelas multiplicidades da associação entre esta classe e a classe Livro, um autor pode escrever muitos livros e um livro pode ser escrito por muitos autores. Esta classe tem como atributo o nome do autor do tipo `String` e um método para consultar um autor específico.
- **Editora** – Essa classe tem como função armazenar as editoras que publicaram os livros possuídos pela biblioteca. Essa classe possui os atributos para conter o nome e a cidade da editora do tipo `String`, além de um método para consultar uma editora específica.
- **Livro** – Esta classe armazena as informações relativas aos livros possuídos pela biblioteca. Seus atributos são título do tipo `String`, ano, edição, número de páginas do tipo `int` e ISBN do tipo `long`. Há, ainda, um método para consultar um determinado livro. Observe que a classe possui associações com diversas outras classes do modelo. Além da

associação já explicada com a classe **Autor**, pode-se perceber que uma instância da classe **Livro** deve estar associada a uma instância da classe **Genero** e a uma instância da classe **Editora**, posto que um livro precisa pertencer a um gênero e ser publicado por uma editora. Há, ainda, uma associação de agregação com a classe **Exemplar** que demonstra que as informações de um livro precisam ser complementadas pelas informações dos exemplares a ele associados.

- **Exemplar** – Essa classe armazena as informações referentes a cada exemplar de um livro. A classe **Livro** armazena as informações dos livros possuídos pela biblioteca, mas a classe **Exemplar** refere-se às cópias físicas existentes na biblioteca para empréstimo. Os atributos desta classe são a data em que o exemplar foi adquirido, do tipo **Date**, e a situação do exemplar, do tipo **Situacao**. **Situacao** é uma classe de enumeração cujos literais definem se o exemplar se encontra disponível, está emprestado, está reservado ou só pode ser consultado dentro da biblioteca. Esta classe possui ainda os métodos **consultarExemplar** e **alterarSituacao**: o primeiro serve para consultar um exemplar específico e o último, para alterar a situação de um determinado exemplar. A classe **Exemplar** possui ainda uma associação com a classe **Livro**, uma vez que cada exemplar representa fisicamente um livro, todavia um livro pode ser representado por muitos exemplares.
- **Emprestimo** – Essa classe tem por função registrar os empréstimos de exemplares feitos pelos sócios da biblioteca. Os atributos relevantes para essa classe são a data em que o empréstimo foi realizado, a data prevista de devolução e a possível data em que o empréstimo foi realmente devolvido, todos do tipo **Date**. Note que os dois primeiros atributos são derivados, uma vez que a primeira data é tomada do sistema e a segunda, calculada com base na primeira. Observe ainda que o atributo que representa a data em que o empréstimo foi efetivamente devolvido pode ter, no mínimo, nenhum e, no máximo, um valor, uma vez que essa data fica em aberto até todos os exemplares associados ao empréstimo terem sido devolvidos. A classe tem ainda os métodos **registrarEmprestimo**, por meio do qual um novo empréstimo é gerado e os exemplares envolvidos marcados como emprestados;

`devolverExemplares`, responsável por definir que exemplares do empréstimo foram devolvidos e, se todos foram devolvidos, que o empréstimo foi encerrado; `renovarExemplares`, que permite renovar o empréstimo de um ou mais exemplares; `verificarSituacao`, que verifica a atual situação do empréstimo, se está dentro do prazo, atrasado ou já foi devolvido.

Observe que a classe `Emprestimo` tem uma associação de agregação com a classe `Exemplar`, o que determina que os objetos da classe `Exemplar` são objetos-parte que complementam a informação dos objetos-todo da classe `Emprestimo`. Isto significa que ao consultar um empréstimo, deve-se apresentar também as informações dos exemplares locados nele. Observe ainda que um empréstimo pode conter, no mínimo, um e, no máximo, muitos exemplares e um objeto `Exemplar` pode estar relacionado a muitos objetos da classe `Emprestimo`.

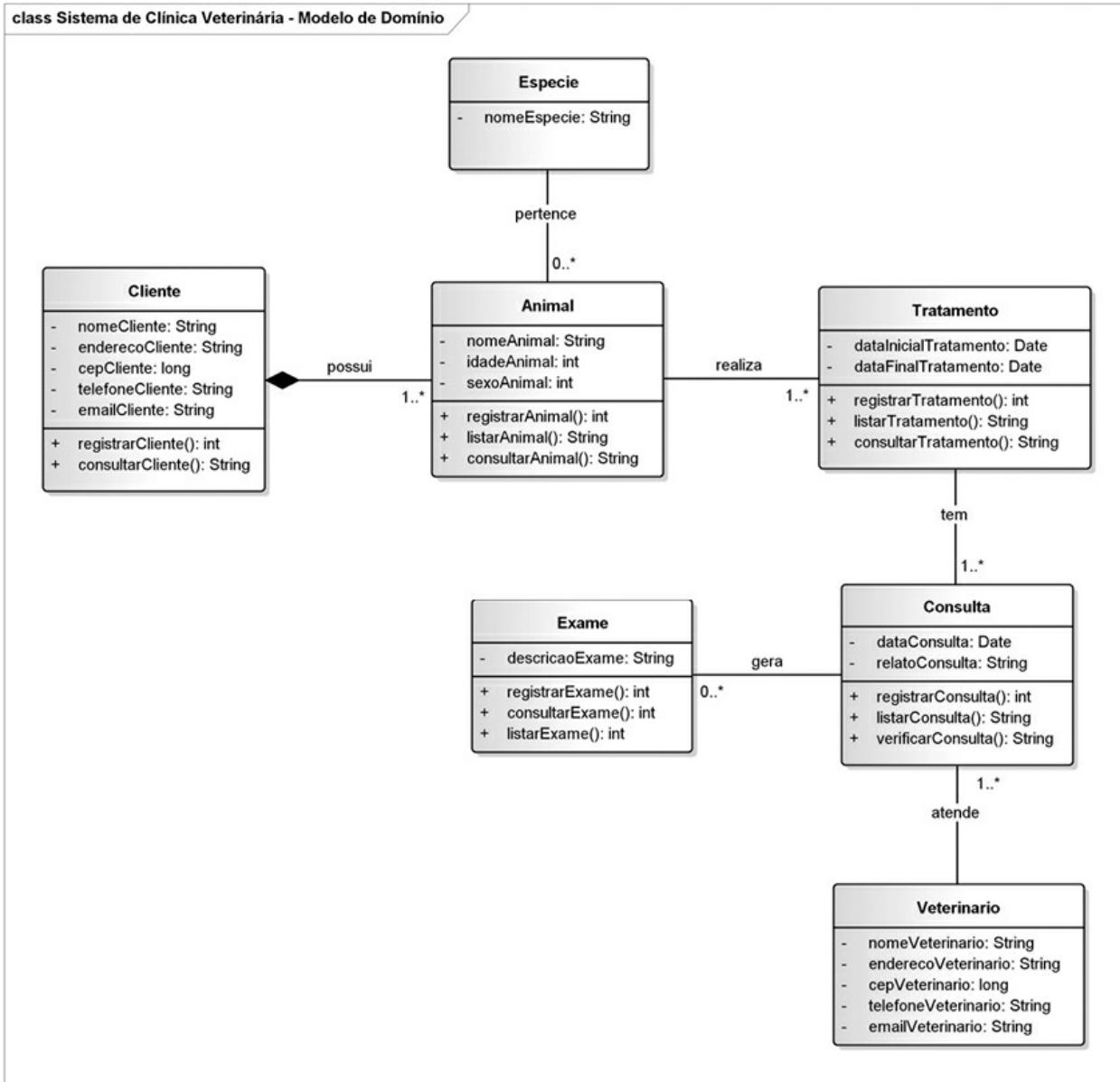
- **Renovacao** – Esta classe tem a função de armazenar as datas em que um empréstimo foi renovado. Possui o atributo `dataRenovacao` do tipo `Date` e os métodos `registrarRenovacao` e `consultarRenovacao` que são autoexplicativos. Como se pode perceber pela associação de composição entre esta classe e a classe `Emprestimo`, um empréstimo pode ser renovado muitas vezes, porém a renovação não é obrigatória, assim um empréstimo pode nunca ser renovado. Além disso, a composição demonstra que as informações do empréstimo precisam ser complementadas pelas possíveis datas em que o empréstimo foi renovado. A quantidade de vezes que um empréstimo pode ser renovado dependerá do tipo de sócio que fez o empréstimo.

## 4.12 Exemplo de Diagrama de Classes – Sistema de Clínica Veterinária

Nesta seção, apresentaremos o diagrama de classes referente ao modelo de domínio para o sistema de clínica veterinária iniciado no capítulo sobre o diagrama de casos de uso. A figura 4.55 apresenta o modelo de domínio desse sistema.

Detalharemos a seguir as classes de entidade identificadas nesse modelo. O leitor notará que declaramos somente os métodos considerados importantes nesse modelo.

- **Especie** – Essa classe representa as diversas espécies de animais tratadas na clínica e tem como atributo o nome da espécie do tipo **String**. Uma instância da classe **Especie** pode estar associada a muitas instâncias da classe **Animal**, porém pode ocorrer de nenhuma instância da classe **Animal** relacionar-se com uma determinada instância da classe **Espécie**, conforme demonstra a associação denominada **pertence** entre elas.
- **Cliente** – Classe de fácil compreensão cujo objetivo é representar os clientes da clínica. Seus atributos são autoexplicativos. O método **consultarCliente** tem como objetivo consultar as informações de um cliente. Já o método **registrarCliente** tem a função de registrar um novo cliente. Observe que um cliente precisa ter suas informações complementadas pelos animais que ele possui, como demonstra a associação de composição entre as classes **Cliente** e **Animal**. É importante relembrar que, a rigor, seria necessário definir um método **get** e um método **set** para cada atributo da classe, no entanto muitas vezes é inviável proceder dessa forma, posto que pode haver muitos atributos, o que deixaria a classe muito grande, além do que esses métodos são um tanto óbvios e repetitivos, portanto desnecessários representar. Cumpre lembrar também que não é recomendado representar métodos construtores na UML, uma vez que variam muito entre as linguagens orientadas a objetos. Assim, pelos motivos enunciados anteriormente, preferimos definir métodos gerais para registrar e consultar clientes internamente. Durante a implementação, os métodos de registro podem ser traduzidos em métodos construtores ou chamar um método construtor; os métodos de consulta podem ser desdobrados em vários métodos **get**, por exemplo.



*Figura 4.55 – Diagrama de Classes para o Sistema de Clínica Veterinária – Modelo de Domínio.*

- **Animal** – Essa classe também é fácil de compreender, sendo responsável por armazenar as informações dos animais já tratados na clínica. Da mesma forma que na classe **Cliente**, seus atributos são autoexplicativos. O método **registrarAnimal** permite registrar ou atualizar um novo objeto da classe **Animal**, já o método **listarAnimal** retorna todos os animais que um determinado cliente possui e o método **consultaAnimal** permite consultar um dos animais listados. Conforme é possível notar ao examinarmos as associações dessa classe, um animal

deve pertencer a um único dono e a uma única espécie, embora um cliente possa ter muitos animais e uma espécie possa referir-se a muitos animais. Além disso, um animal, para estar registrado na clínica, deve ter iniciado ao menos um tratamento, podendo estar associado a muitos deles.

- **Tratamento** – Essa classe representa os diversos tratamentos pelos quais passa ou passou um determinado animal. Para estar registrado nesse sistema, uma instância da classe **Animal** deve estar associada a, no mínimo, uma instância da classe **Tratamento**. Um tratamento tem como atributos as datas de seu início e término do tipo **Date**. O método **listarTratamento** permite visualizar todos os tratamentos já realizados por um determinado animal, o método **consultarTratamento** permite consultar um determinado tratamento e o método **registrarTratamento**, abrir um novo tratamento.
- **Consulta** – Essa classe representa cada uma das consultas pelas quais passa um animal durante um tratamento. Um tratamento associa-se a, no mínimo, uma consulta. Cada instância da classe **Consulta** tem como atributos a data em que a consulta foi realizada, do tipo **Date**, e o relato do que ocorreu durante a consulta, do tipo **String**. Os métodos dessa classe são semelhantes aos da anterior, em que o método **listarConsulta** permite listar todas as consultas de um determinado tratamento, o método **verificarConsulta** permite consultar uma consulta específica e o método **registrarConsulta**, marcar uma nova consulta ou atualizar uma consulta após esta ter sido atendida por um veterinário.
- **Veterinario** – As instâncias dessa classe armazenam informações referentes a cada um dos veterinários que trabalham na clínica. Observe que uma instância da classe **Veterinario** pode se relacionar a muitas instâncias da classe **Consulta**, mas uma instância da classe **Consulta** associa-se a somente uma instância da classe **Veterinario**. Os atributos dessa classe são autoexplicativos.
- **Exame** – As instâncias dessa classe armazenam os possíveis exames marcados em uma determinada consulta. Por esse motivo, uma instância da classe **Exame** estará sempre associada a uma instância da classe

**Consulta**, porém uma instância da classe **Consulta** pode se associar a nenhuma (caso o veterinário não marque nenhum exame) ou a muitas instâncias da classe **Exame**. O atributo desta classe armazena a descrição de cada exame solicitado. Os métodos desta classe são autoexplicativos.

## 4.13 Exemplo de Diagrama de Classes – Sistema de Controle de Advocacia

Nessa seção, apresentaremos o diagrama de classes referente ao modelo de domínio para o sistema de controle de advocacia que foi iniciado no capítulo sobre o diagrama de casos de uso. A figura 4.56 apresenta o modelo de domínio desse sistema.

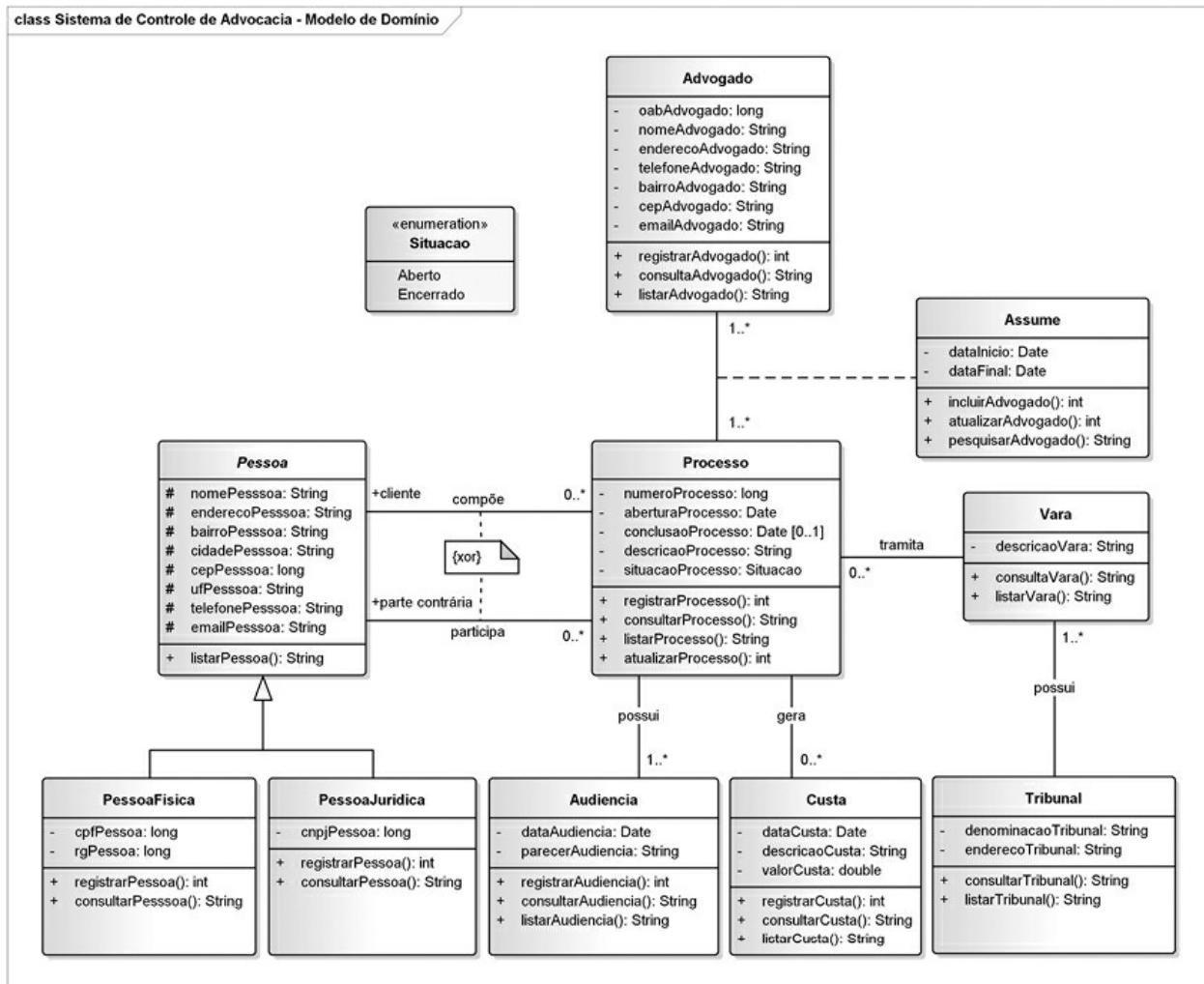


Figura 4.56 – Diagrama de Classes para o Sistema de Controle de Advocacia – Modelo de Domínio.

Detalharemos a seguir as classes de entidade identificadas nesse modelo.

- **Pessoa** – Essa classe representa as pessoas que alguma vez participaram de algum processo como clientes ou partes contrárias. Observe que esta classe é uma classe abstrata, conforme demonstra o texto em itálico de seu nome, cujo único objetivo é armazenar os atributos comuns às duas classes que são especializadas a partir desta, as classes **PessoaFísica** e **PessoaJurídica**, indicando que um cliente ou parte contrária pode tanto constituir-se em uma pessoa física quanto em jurídica. Note ainda que há duas associações desta classe com a classe **Processo**. Ao examinarmos essas associações, percebemos que tanto uma instância da classe **PessoaFísica** quanto da **PessoaJurídica** só pode associar-se como cliente ou parte contrária de uma instância da classe **Processo**, conforme demonstra a restrição “**xor**” (ou exclusivo) entre as associações e os papéis assumidos pelos objetos nessas associações. Além disso, uma instância de qualquer dessas duas classes pode nunca ter se associado como cliente ou parte contrária, conforme demonstra a multiplicidade dessas associações. Os atributos dessa classe são autoexplicativos. O método **listarPessoa** serve para retornar todas as pessoas registradas e é herdado pelas subclasses da classe **Pessoa**.
- **PessoaFísica** – Essa classe armazena os atributos e métodos particulares das pessoas físicas que são clientes ou estão sendo processadas pelo escritório. Seus atributos são o CPF e a carteira de identidade da pessoa, ambos do tipo **long**, e seus métodos **registrarPessoa** e **consultarPessoa** têm por objetivo registrar uma nova pessoa física e consultar as informações de uma pessoa, respectivamente.
- **PessoaJurídica** – Essa classe armazena os atributos e métodos particulares das pessoas jurídicas que são clientes ou estão sendo processadas pelo escritório. Seu atributo é o CNPJ da pessoa, do tipo **long**. Nesta classe, igualmente os seus métodos **registrarPessoa** e **consultarPessoa** têm como objetivo registrar uma nova pessoa jurídica e consultar as informações de uma pessoa, respectivamente. Optamos por declarar esses dois métodos em ambas as classes por a consulta ser realizada por atributos diferentes (CPF ou CNPJ) e por terem de registrar igualmente diferentes atributos particulares.

- **Advogado** – Essa classe representa todos os advogados que trabalham ou trabalharam no escritório. Seus atributos são autoexplicativos e seus métodos `registrarAdvogado`, `consultaAdvogado` e `listarAdvogado` servem, respectivamente, para registrar um novo advogado ou alterar os dados de um advogado já cadastrado, consultar um advogado específico e listar todos os advogados registrados.
- **Processo** – Esta é a principal classe desse diagrama e representa todos os processos já concluídos ou em andamento do escritório. Como o número do processo (do tipo `long`) é um valor informado externamente, precisa ser declarado explicitamente na classe. Além deste, os atributos dessa classe armazenam a data em que o processo foi aberto, a data de seu possível término, ambos do tipo `Date`, uma descrição do processo do tipo `String` e a situação do processo que é do tipo `Situacao`, que é uma classe de enumeração contendo dois literais que representam as situações possíveis de um processo, ou seja, se ele se encontra em andamento ou já foi encerrado. A classe **Processo** contém ainda os seguintes métodos:

Método	Descrição
<code>registrarProcesso</code>	Permite a geração de um novo processo.
<code>consultarProcesso</code>	Consulta um processo específico. Esse método recebe como parâmetro o número do processo que se deseja consultar e retorna uma <code>String</code> com os dados do processo, se este tiver sido encontrado.
<code>listarProcesso</code>	Retorna todos os processos registrados no sistema.
<code>atualizarProcesso</code>	Atualiza os dados de um determinado processo, retornando 1 (verdadeiro), se a atualização for bem-sucedida, ou 0 (falso), caso contrário.

- **Assume** – Essa classe, como o leitor pode notar, é uma classe associativa, uma vez que o advogado pode assumir muitos processos e um processo pode ter mais de um advogado responsável por ele, conforme demonstra a multiplicidade. Essa classe registra a data inicial em que um advogado assumiu um determinado processo até a possível data final que ele deixou o processo, seja por este ter sido concluído, seja por outro motivo qualquer. Os métodos `registrarAdvogado`, `atualizarAdvogado` e `pesquisarAdvogado` servem para associar um novo advogado a um processo, para atualizar a situação de um advogado, se um advogado

deixar um processo, e para pesquisar os advogados que participam de um determinado processo, respectivamente.

- **Audiencia** – Essa classe armazena todas as audiências por que passa um determinado processo. Uma instância da classe **Processo** deve estar associada a, no mínimo, uma instância da classe **Audiencia**. No entanto, uma instância da classe **Audiencia** só pode se associar a uma única instância da classe **Processo**. Os atributos dessa classe contêm a data em que ocorreu a audiência e o parecer do tribunal. Essa classe possui ainda os seguintes métodos:

Método	Descrição
<code>registrarAudiencia</code>	Permite o registro de uma nova audiência.
<code>consultarAudiencia</code>	Permite a consulta de uma determinada audiência.
<code>listarAudiencia</code>	É utilizado para retornar todas as audiências de um determinado processo.

Poderíamos pensar em transformar a associação binária simples entre as classes **Audiencia** e **Processo** em uma composição, mas não consideramos necessário, ao consultar um processo, que todas as suas audiências sejam também apresentadas obrigatoriamente.

- **Custas** – Essa classe representa as possíveis custas de um processo. Cada instância dessa classe deve se relacionar com uma instância da classe **Processo**, mas uma instância da classe **Processo** pode se associar com nenhuma ou muitas instâncias da classe **Custas**. Seus atributos são a data em que a custa foi contraída, do tipo **Date**, a descrição da custa, do tipo **String**, e o valor da custa, do tipo **double**. Os métodos desta classe são autoexplicativos.
- **Vara** – Um processo tramita em uma determinada vara, mas uma vara pode ter muitos processos em tramitação, conforme demonstra a multiplicidade da associação entre essas classes. Acreditamos que os atributos e métodos desta classe são autoexplicativos.
- **Tribunal** – Finalmente, essa classe armazena os tribunais aos quais as varas onde tramitam os processos estão vinculadas. Uma vara pertence a um único tribunal, mas a um tribunal podem pertencer muitas varas, como demonstra a multiplicidade da associação entre essas classes. A classe Tribunal tem como atributos o nome e o endereço do tribunal, do

tipo `String`, e o método para consultar um tribunal (`consultarTribunal`) e para listar todos os tribunais registrados (`listarTribunal`).

## 4.14 Persistência

Em muitas situações, pode ser necessário preservar de maneira permanente os objetos de uma classe, ou seja, a classe em questão precisa ser persistente, o que significa que seus objetos devem ser gravados em disco de alguma maneira. Deve ficar claro, no entanto, que nem toda classe é persistente, sendo muitas vezes desnecessário preservar suas instâncias. Esse tipo de classe é chamada transiente.

Classes de entidade normalmente são persistentes, enquanto classes de fronteira e de controle normalmente são transientes. Todavia, como foi dito anteriormente, nem toda classe de entidade é obrigatoriamente persistente, assim, pode ser necessário identificar quais classes são persistentes (ou quais são transientes). É preciso definir ainda se todos os atributos de uma classe deverão ser persistidos, pois pode haver atributos transientes (resultados de cálculos, por exemplo). Essas identificações podem ser feitas por meio do emprego de estereótipos e/ou restrições, como foi demonstrado em seções anteriores.

Classes de entidade apresentam algumas semelhanças com o conceito de entidade do antigo modelo Entidade-Relacionamento utilizado para definir modelos lógicos de bancos de dados relacionais – a semelhança de nomenclatura foi intencional. Vale lembrar que uma entidade no modelo E-R não se refere obrigatoriamente a uma tabela em um banco de dados, podendo referenciar-se a um repositório de informações armazenado em memória e não em disco.

Na verdade, o diagrama de classes foi intencionalmente projetado para ser uma evolução do modelo Entidade-Relacionamento, sendo assim também pode ser utilizado para modelar a estrutura lógica de um banco de dados. O leitor familiarizado com o uso do modelo Entidade-Relacionamento pode ter percebido, ao longo deste capítulo, muitos conceitos semelhantes entre esse modelo e o diagrama de classes, embora obviamente o diagrama de classes possua muito mais recursos e sua ênfase esteja focada na definição lógica das classes.

Além disso, diferentemente do modelo E-R, que é relacional, o diagrama de classes é orientado a objetos, portanto, além de definir os atributos das classes, também define as operações que podem ser executadas sobre esses atributos. Dessa forma, os conceitos de entidade no diagrama de classes e no modelo E-R não são idênticos e uma classe, mesmo de entidade, não corresponde a uma tabela – pode ser mapeada em uma tabela, o que não é a mesma coisa.

Muitas vezes, uma classe persistente terá seu equivalente na forma de uma tabela relacional. Nesses casos, cada atributo será uma coluna na tabela e cada objeto tornar-se-á uma linha. Porém, isso não é uma regra: uma classe pode ser dividida em mais de uma tabela ou uma tabela pode representar muitas classes – isso é particularmente comum quando se mapeiam hierarquias de classes definidas por meio de herança. Trataremos disso na seção 4.15.

## 4.15 Mapeamento de Classes em Tabelas

A maneira como as instâncias de uma classe serão preservadas dependerá da forma como o sistema for implementado, podendo estas ser atualizadas em disco, à medida que o sistema for sendo manipulado, ou mantidas em memória e registradas no disco apenas quando do encerramento do sistema. Normalmente – mas não necessariamente –, os objetos de classes são persistidos por meio de um SGBD (Sistema Gerenciador de Banco de Dados), um software responsável por gerir bancos de dados.

Lembramos aqui que SGBD e banco de dados são conceitos diferentes. Bancos de dados existiam muito antes de o primeiro computador ter sido concebido. Um banco de dados é qualquer repositório de informações. Há milhares de anos, os sumérios possuíam bancos de dados na forma de tabuletas de argila. Dessa forma, um conjunto de arquivos (ou apenas um) contendo informações já se caracteriza em um banco de dados. Já um SGBD é um software capaz de organizar e gerenciar bancos de dados estruturados em diversos formatos, existindo atualmente vários no mercado, como PostgreSQL, Oracle ou MySQL.

Quando se utiliza um SGBD orientado a objetos, não é necessário preocupar-se tanto com o mapeamento desses objetos. Porém, o modelo relacional ainda é – e provavelmente continuará sendo por muito tempo –

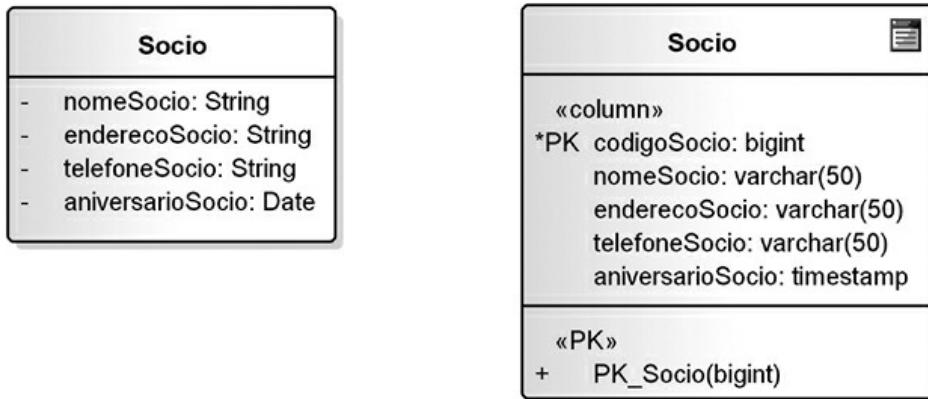
muito popular na área de banco de dados, uma vez que se provou eficiente e confiável ao longo de décadas. A orientação a objetos é uma alternativa melhor em termos de programação em memória, porém, com relação à persistência, o modelo relacional parece ser o mais adequado ou, pelo menos, o mais utilizado.

Atualmente, existem diversos frameworks de persistência, também chamados ORMs – Object Relational Mappers (Mapeadores Relacionais de Objeto), que se responsabilizam por persistir os objetos das classes. Entre eles, pode-se citar, por exemplo, o Hibernate, o Entity Framework, o Torque ou o Castor, entre outros. Entretanto, em nível de projeto, é necessário saber quais tabelas correspondem a quais classes, de modo a saber quais repositórios lógicos armazenam as informações do software. Assim, descreveremos como mapear classes em tabelas relacionais nas próximas seções.

#### 4.15.1 Estereótipo Table

Nos casos mais simples, uma classe persistente será representada por uma tabela. Nesses casos, será apenas preciso definir uma chave primária para a tabela em questão, podendo-se criar um atributo exclusivamente para isso ou utilizar um atributo já existente. A figura 4.57 apresenta o mapeamento de uma classe denominada **Socio** em uma tabela.

Nesse exemplo, a classe **Socio** está representada à esquerda na figura, enquanto a tabela em que ela foi mapeada está à direita. Como podemos perceber, a tabela **Socio** é uma classe contendo um estereótipo gráfico. O perfil de modelagem de dados representa uma tabela como uma classe estereotipada, ou seja, uma classe com o estereótipo <<table>>.



*Figura 4.57 – Mapeamento de uma Classe em Tabela.*

Ao examinarmos a tabela mapeada, percebemos que suas colunas são quase idênticas aos atributos da classe **Socio**, acrescentando-se somente uma coluna extra, denominada **codigoSocio**, do tipo **bigint** (aqui aplicamos os tipos utilizados pelo SGBD PostgreSQL, em que o tipo **bigint** é equivalente ao tipo **long**, utilizado em outros SGBDs como o Oracle) para servir de chave primária, conforme demonstra o texto PK (Primary Key, ou Chave Primária) ao lado da coluna. Uma chave primária é composta de uma ou mais colunas de uma tabela cujo(s) valor ou valores serve(m) para identificar uma linha específica da tabela em questão.

Foi necessário incluir uma nova coluna para servir de chave primária porque nenhum dos atributos da classe **Socio** poderia assumir essa função, já que uma chave primária precisa ser única, ou seja, seu valor não pode se repetir em nenhuma linha. Observe que a tabela possui uma divisão contendo o estereótipo **<<column>>** para identificar as colunas da tabela e outra divisão contendo o estereótipo **<<PK>>** para identificar a chave primária.

## 4.15.2 Associações e Chaves Estrangeiras

### *Associação de 1 para 1*

Embora isso não seja muito comum, pode acontecer de um objeto em uma das extremidades de uma associação referir-se a somente um objeto da outra extremidade, e esse objeto, por sua vez, também estar associado unicamente ao mesmo objeto da outra extremidade, ou seja, o objeto **obj1** está associado somente ao objeto **obj2** e o **obj2** está associado somente ao

## obj1.

Muitas vezes, uma associação de 1 para 1 denota um erro de modelagem, o que normalmente demonstra que uma das classes envolvidas na associação não deveria existir e que seus atributos deveriam ser transferidos para a outra classe da associação, deixando a própria associação de existir também. Porém, esse tipo de associação pode ocorrer, como é demonstrado na figura 4.58.

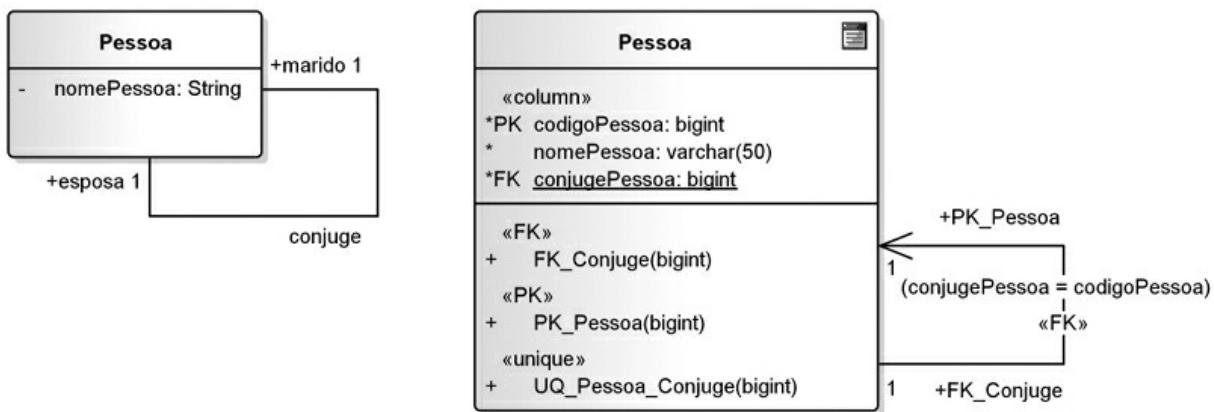


Figura 4.58 – Mapeamento de uma Classe com Associação 1 para 1.

Neste exemplo, identificamos uma situação em que existe uma classe **Pessoa** (à esquerda na figura) que tem uma associação unária denominada **conjuge**, com multiplicidade 1 em seus dois extremos. Percebemos ainda que um objeto dessa classe envolvido na associação interpreta o papel de marido e deve estar associado a um único objeto da mesma classe interpretando o papel de esposa. Essa associação pode ser lida da seguinte forma: um marido é cônjuge de uma e somente uma esposa e uma esposa é cônjuge de um e somente um marido.

Em termos relacionais, quando existe uma associação de 1 para 1, deve-se adicionar uma chave estrangeira em uma das tabelas envolvidas na associação para referenciar a chave primária da tabela localizada na outra extremidade da associação. Uma chave estrangeira é uma coluna ou conjunto de colunas que armazena um valor por meio do qual podemos identificar, de forma única, uma linha em uma tabela associada, por meio da comparação do valor da chave estrangeira com o valor da chave primária da tabela pesquisada.

Dessa forma, para mapearmos a classe pessoa, criamos uma tabela de

mesmo nome (à direita na figura). Ao examinarmos a tabela **Pessoa**, perceberemos que foi incluída uma coluna denominada **codigoPessoa**, para servir de chave primária, e foi criada outra coluna chamada **conjugePessoa** para servir como chave estrangeira, conforme demonstra o texto FK (Foreign Key, ou Chave Estrangeira) ao lado do nome da coluna.

Nesse caso específico, a chave estrangeira identificará uma linha da própria tabela **Pessoa**, posto que se está mapeando um relacionamento unário, por isso essa coluna deverá armazenar o valor de um código de pessoa (obviamente diferente do código da pessoa em questão – uma pessoa não pode ser casada consigo mesma). Além disso, como o campo **conjugePessoa** é único (seu valor não pode se repetir em outra linha), é apresentado sublinhado. Observe que na terceira divisão da tabela, além da definição da chave primária por meio do estereótipo <<PK>>, foi definida uma chave estrangeira chamada **FK\_Conjuge** e detalhou-se quais colunas são únicas, como demonstra o estereótipo <<unique>>. Observe ainda que existe uma associação unária na tabela **Pessoa** com associação 1 para 1, em que, para que se possa identificar o cônjuge de uma pessoa, é necessário que o valor da chave estrangeira seja igual ao da chave primária que identifica uma pessoa específica.

#### *Associação de 1 para Muitos*

Quando existe uma associação de 1 para muitos (\*) entre uma ou mais tabelas, deve-se adicionar uma chave estrangeira na extremidade muitos (\*) da associação para se referenciar à chave primária da tabela da outra extremidade. Um exemplo disso é apresentado na figura 4.59.

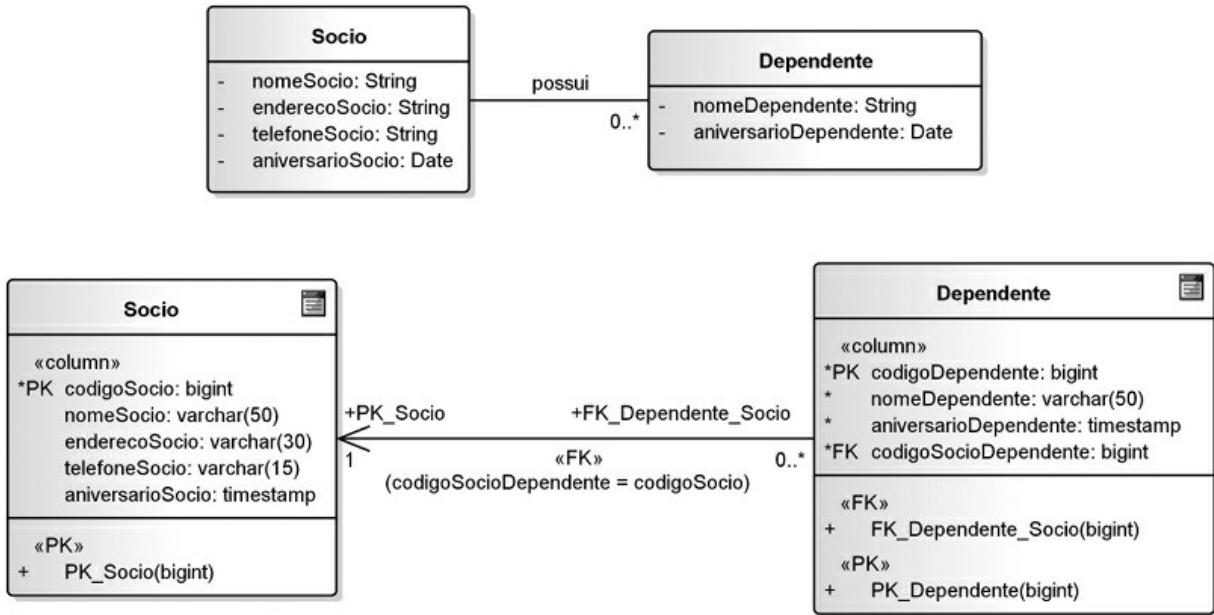


Figura 4.59 – Mapeamento de Classes com Associação 1 para Muitos.

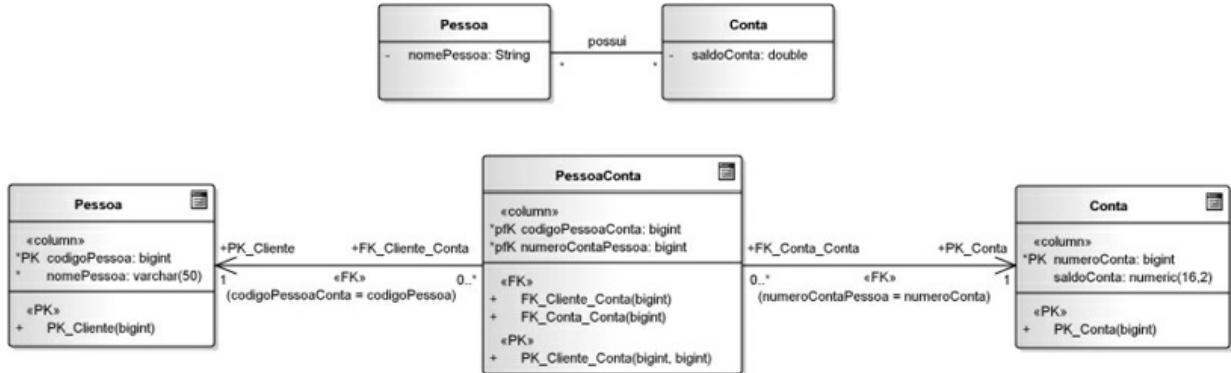
Neste exemplo, identificamos uma situação em que existe uma associação binária simples entre as classes **Socio** e **Dependente**. Essa associação foi explicada no início do capítulo. As classes **Socio** e **Dependente** foram mapeadas em tabelas equivalentes, sendo acrescentada em cada uma delas uma coluna para identificar a chave primária da tabela. Observe que na tabela **Dependente** acrescentou-se ainda uma coluna para servir de chave estrangeira, cuja função é identificar um sócio específico. Observe que essa coluna não é única, porque mais de um dependente pode estar associado a um mesmo sócio.

Deve-se notar que a mesma associação existente entre as classes **Socio** e **Dependente** existe também entre as tabelas, mantendo-se a mesma multiplicidade e, devido a esta ser muitos na extremidade da tabela **Dependente**, força a criação da chave estrangeira nessa tabela. Observe que o relacionamento dessa associação é feito entre a chave estrangeira da tabela **Dependente** e a chave primária da tabela **Socio**. Para que se possa descobrir o sócio ao qual o dependente está associado, é necessário que o valor da chave estrangeira da linha do dependente em questão seja igual ao da chave primária de uma linha da tabela **Socio**.

### Associação de Muitos para Muitos

Quando ocorre uma associação de muitos para muitos, deve-se criar uma

tabela intermediária contendo duas chaves estrangeiras, cada uma relacionada à chave primária de uma das classes com a qual se relaciona. Essas chaves estrangeiras podem constituir a própria chave primária da tabela intermediária ou pode-se criar uma chave primária própria. A figura 4.60 apresenta um exemplo que ilustra uma situação como essa.



*Figura 4.60 – Mapeamento de Classes com Associação Muitos para Muitos.*

Esse exemplo foi adaptado do modelo conceitual do sistema de controle bancário apresentado anteriormente neste capítulo, em que uma pessoa pode possuir muitas contas e uma conta pode pertencer a mais de uma pessoa.

Uma associação muitos para muitos é perfeitamente aceitável dentro da orientação a objetos, desde que não haja atributos particulares relativos a um objeto específico associado com outro objeto específico, ou seja, atributos ligados à associação, caso em que é necessário incluir uma classe associativa ou intermediária, como já foi exposto. Em um modelo relacional, porém, uma associação muitos para muitos exige a criação de uma tabela intermediária posicionada entre as tabelas envolvidas na associação.

Ao observarmos a figura, percebemos que sua parte superior apresenta as classes **Pessoa** e **Conta**, ligadas por uma associação binária com multiplicidade muitos nas duas extremidades. Já na parte inferior da figura são apresentadas as tabelas mapeadas a partir dessas classes. O leitor perceberá que existem três tabelas para apenas duas classes, o que é necessário porque não é possível inserir uma coluna para referenciar cada conta possuída por uma pessoa, tampouco é possível acrescentar uma coluna para referenciar cada pessoa que possui uma determinada conta.

Dessa forma, é necessário criar uma tabela intermediária para ligar cada pessoa a cada conta por ela possuída.

O leitor perceberá que existe uma tabela equivalente à classe **Pessoa** e outra à classe **Conta**, acrescentando-se somente uma chave primária para cada uma delas. Porém, existe uma tabela intermediária entre essas duas tabelas chamada **PessoaConta**, contendo dois atributos denominados **codigoPessoa** e **numeroContaPessoa**. Esses atributos fazem às vezes tanto de chave primária da tabela como de chaves estrangeiras para pesquisar, respectivamente, uma pessoa e uma conta em particular, como demonstra o texto **pFK** na frente do nome das duas colunas. Observe que na terceira divisão da tabela está discriminado que esta contém duas chaves estrangeiras e uma chave primária composta de dois valores.

O leitor notará que a tabela **PessoaConta** está relacionada tanto à tabela **Pessoa** como à tabela **Conta** e o relacionamento entre as tabelas **PessoaConta** e **Pessoa** é feito por meio da comparação entre a chave primária **codigoPessoa** da tabela **Pessoa** e a chave estrangeira **codigoPessoaConta** da tabela **PessoaConta**, sendo necessário que os valores de ambas sejam iguais para se descobrir a qual pessoa pertence uma determinada conta. O mesmo ocorre entre as tabelas **PessoaConta** e **Conta**, onde o relacionamento é feito entre a chave primária **numeroConta** da tabela **Conta** e a chave estrangeira **numeroContaPessoa** da tabela **PessoaConta**.

Quando já existe uma classe associativa ou intermediária, normalmente se gera uma tabela para essa classe, contendo seus atributos. Em seguida, cria-se uma chave estrangeira para relacionar-se com cada chave primária das tabelas com a qual essa tabela se relaciona.

### *Associações Ternárias*

No caso de associações ternárias, utiliza-se um procedimento semelhante à situação de associações muitos para muitos, criando-se uma tabela para representar a associação ternária e uma chave estrangeira para esta se relacionar com cada tabela associada. A figura 4.61 apresenta um exemplo de mapeamento de associação ternária.

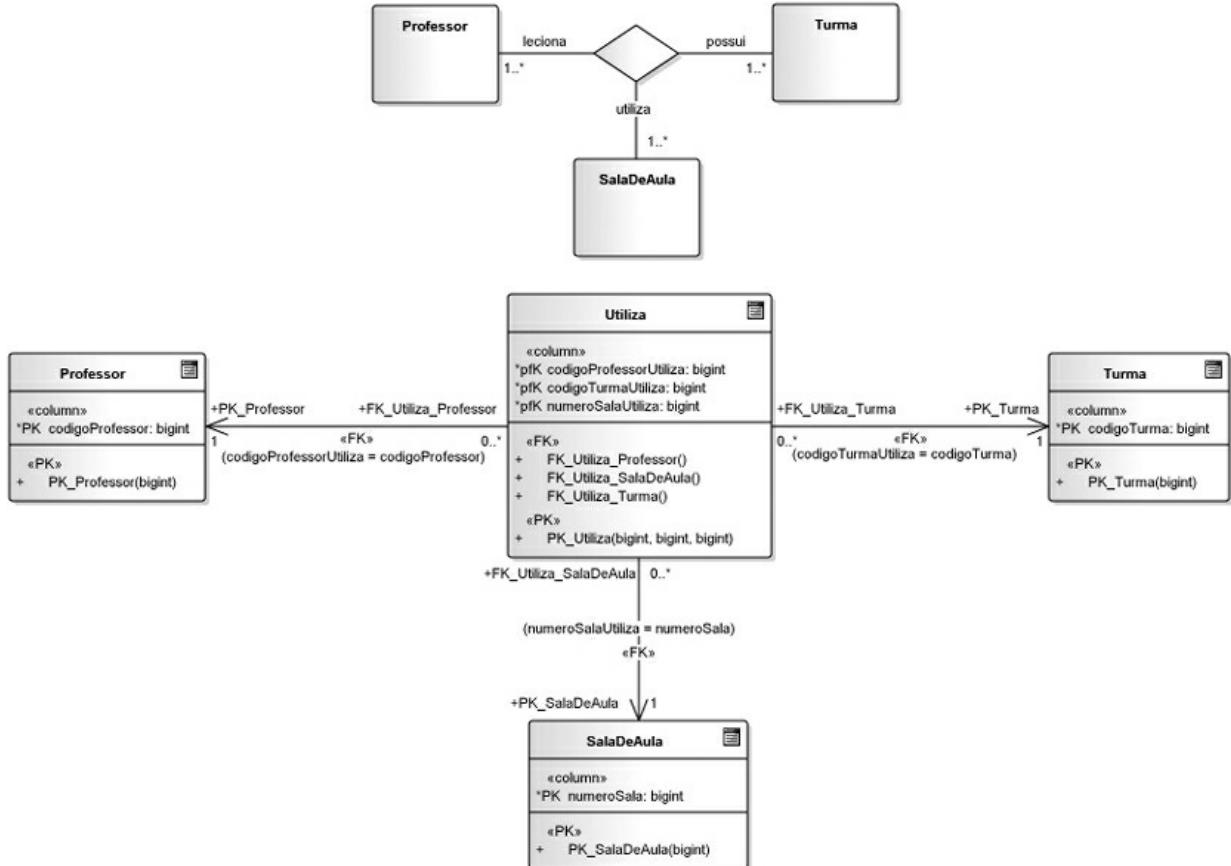


Figura 4.61 – Mapeamento de Associação Ternária.

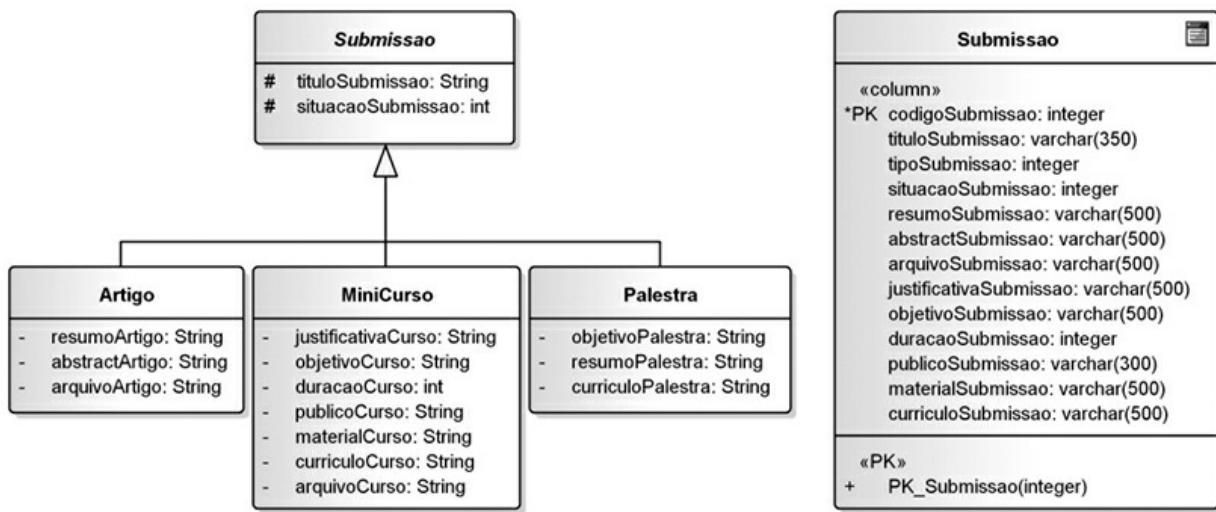
Aqui, tomamos o exemplo de associação ternária apresentado no início deste capítulo e mapeamos essa associação. O leitor pode perceber que cada classe foi mapeada em uma tabela e foi criada uma tabela intermediária mapeando a associação ternária em si, criando-se uma chave estrangeira para ligá-la a cada classe. Observe que a chave primária é composta das três colunas da tabela.

### Herança

Quando se utilizam associações do tipo generalização/especialização, o processo de representação de tabelas relacionais é um pouco mais complexo. Existem basicamente três procedimentos possíveis.

Na primeira estratégia, toda a hierarquia de classes é representada por uma única tabela no banco de dados, que deve incluir uma coluna para identificar o tipo do objeto representado por cada linha. As desvantagens dessa estratégia consistem na ausência de normalização dos dados e na possibilidade de existir muitos campos nulos em diversas linhas da tabela.

A figura 4.62 apresenta um exemplo dessa estratégia.



*Figura 4.62 – Mapeamento de Herança – Primeira Estratégia.*

Aqui, identificamos uma situação em que existe uma classe abstrata denominada **Submissao**, que representa um conjunto de trabalhos submetidos a um evento científico. Como existem três tipos de submissões possíveis, referentes a artigos, propostas de minicursos ou de palestras, definiu-se a classe geral **Submissao** contendo os atributos comuns a todos os tipos de submissão e derivaram-se três classes especializadas dela, contendo cada uma atributos particulares.

O leitor notará que ao lado dessa hierarquia de classes existe uma tabela denominada **Submissao**, que foi mapeada da referida hierarquia. Como se aplicou nesse caso a primeira estratégia, existe uma tabela única para representar todas as classes da hierarquia, contendo colunas equivalentes a todos os atributos identificados nas classes da referida hierarquia, além de uma coluna extra para identificar o tipo de submissão referente a cada linha da tabela, bem como uma coluna para servir de chave primária. Obviamente muitas colunas deverão ser deixadas em branco em cada registro da tabela.

Na segunda estratégia, cria-se uma tabela para cada classe concreta existente. Isso leva à redundância de dados, já que quaisquer atributos definidos em uma classe abstrata na hierarquia devem ser criados em todas as tabelas que mapeiam classes-filhas desta. Na figura 4.63, apresentamos um exemplo dessa estratégia.

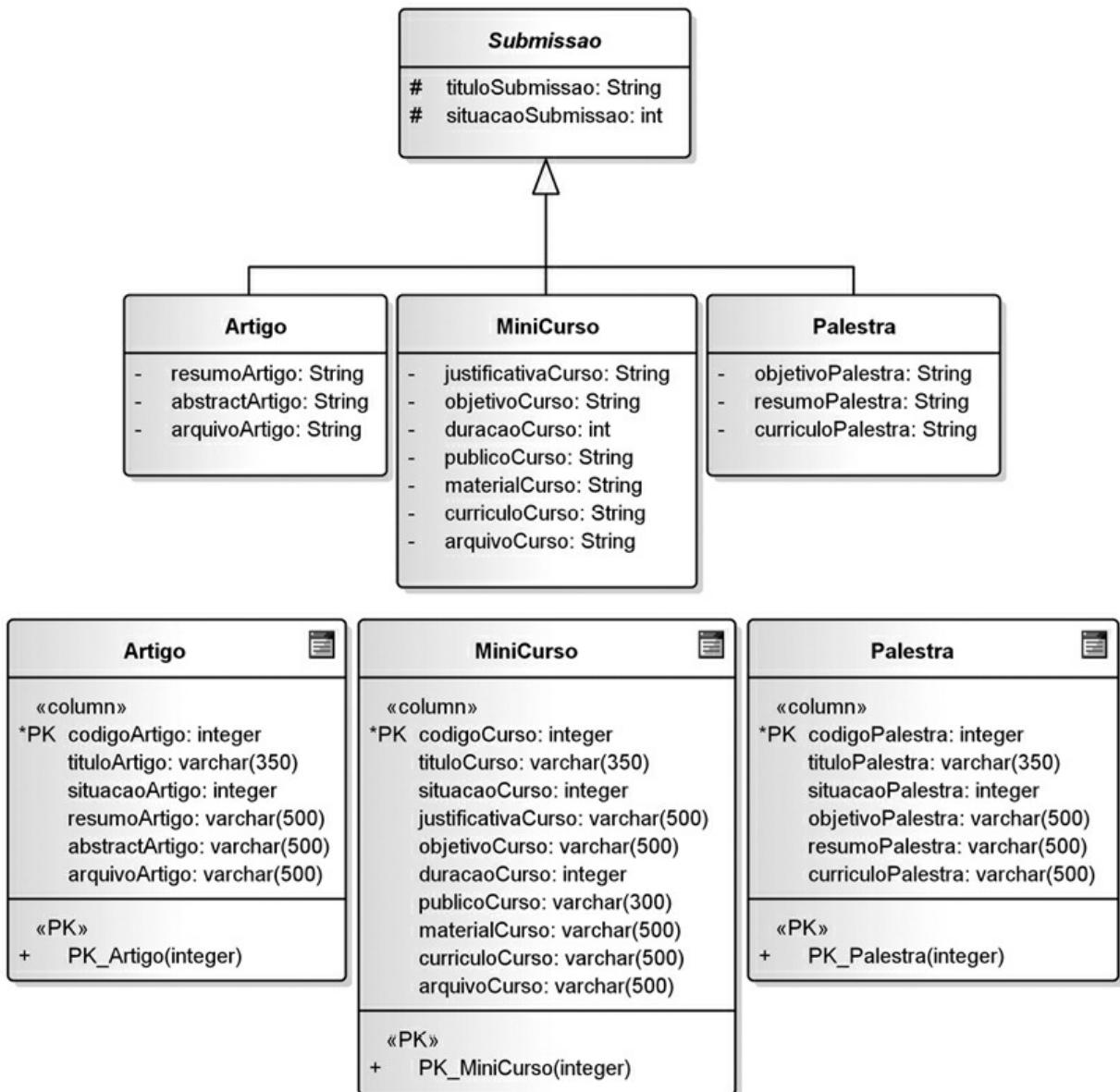


Figura 4.63 – Mapeamento de Herança – Segunda Estratégia.

Aqui, tomamos o mesmo exemplo da figura 4.63 e aplicamos a segunda estratégia de mapeamento. Como o leitor poderá observar, criaram-se três tabelas, uma para armazenar os artigos submetidos, outra para armazenar os minicursos e uma terceira para armazenar as propostas de palestras submetidas. Note que todas as tabelas têm as colunas **titulo** e **situacao**, que representam os atributos da superclasse **Submissao**.

Na terceira estratégia, cria-se uma tabela para cada classe da hierarquia, relacionando-as por meio de chaves estrangeiras. Essa estratégia tenta manter a normalização de dados, de forma que a estrutura final das tabelas

fica bastante parecida com a hierarquia das classes. A figura 4.64 apresenta um exemplo dessa estratégia.

Neste exemplo, criaram-se uma tabela **Submissao** que armazena os dados comuns a todas as submissões e uma tabela para cada tipo de submissão com suas colunas particulares. Observe que existe um relacionamento de 1 para 1 entre essas tabelas e a tabela **Submissao**, onde as tabelas **Artigo**, **Minicurso** e **Palestra** contêm uma chave estrangeira para relacionar-se com a tabela **Submissao**. Essas associações significam que uma submissão pode representar nenhum (0) ou um artigo específico, nenhum ou um determinado minicurso ou nenhuma ou uma palestra específica. Nessa estratégia, não há redundância de dados e o mapeamento é o mais semelhante à hierarquia de classes.

A seguir, mapearemos o modelo conceitual do sistema de controle bancário apresentado anteriormente, conforme demonstra a figura 4.65.

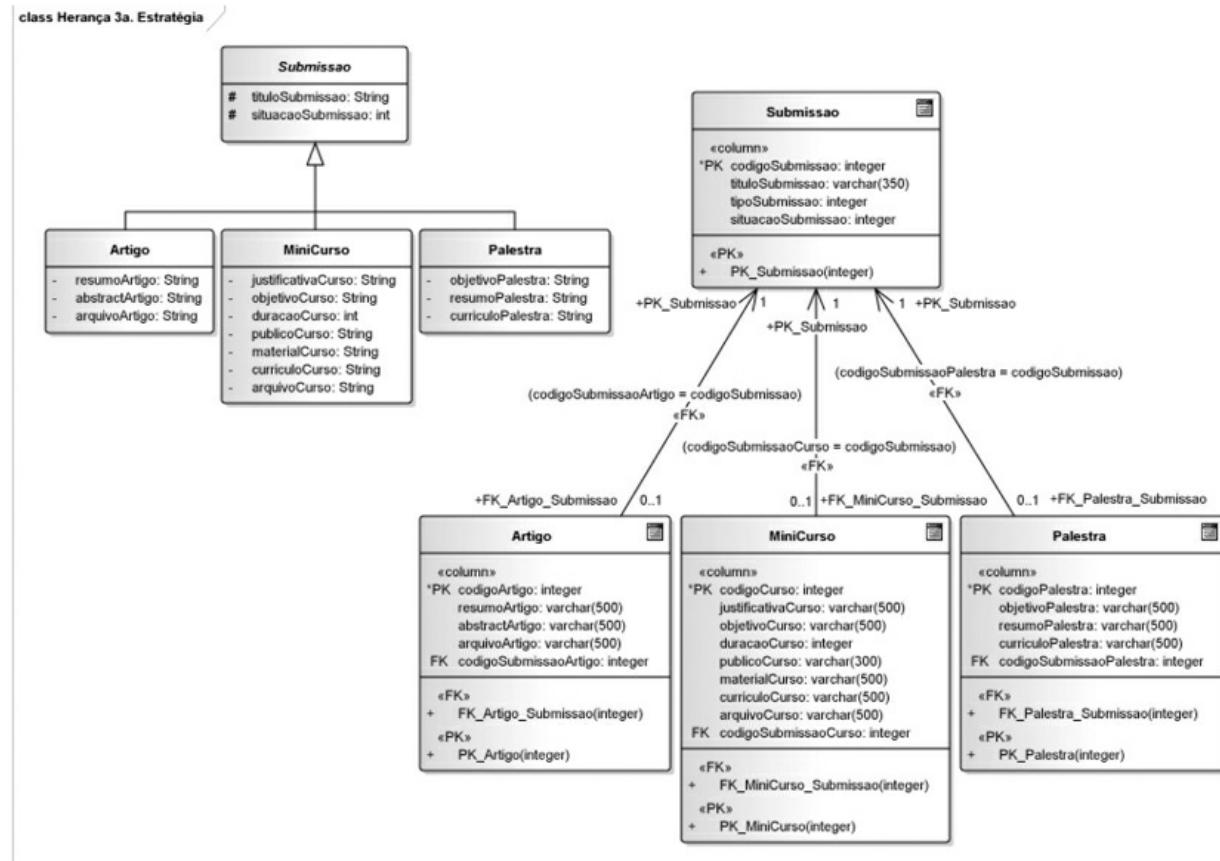


Figura 4.64 – Mapeamento de Herança – Terceira Estratégia.

Para realizar esse mapeamento, em relação à hierarquia de classes,

optamos pela terceira estratégia. Dessa forma, mapeamos uma tabela **Pessoa** que corresponde à classe **Pessoa** do modelo conceitual e duas tabelas **PessoaFisica** e **PessoaJuridica** que correspondem às classes derivadas a partir da classe **Pessoa**. Observe que a tabela **Pessoa** contém colunas correspondentes a todos os atributos da classe **Pessoa**, adicionando, ainda, uma coluna para representar a chave primária da tabela, chamada **codigoPessoa**, e uma coluna para identificar o tipo de pessoa, necessária para determinar se uma linha se refere a uma pessoa física ou jurídica.

Da mesma forma, as tabelas **PessoaFisica** e **PessoaJuridica** contêm colunas correspondentes aos atributos das classes que elas representam, porém não foi necessário criar uma coluna para identificar a chave primária, uma vez que essa função pode ser desempenhada pela coluna que representa o CPF, na tabela **PessoaFisica**, e pela que representa o CNPJ, na tabela **PessoaJuridica**. No entanto, foi necessário acrescentar uma coluna para identificar o código da pessoa que está associada à pessoa física ou jurídica, servindo tal coluna como chave estrangeira.

class Sistema de Controle Bancário - Mapeamento do Modelo Conceitual

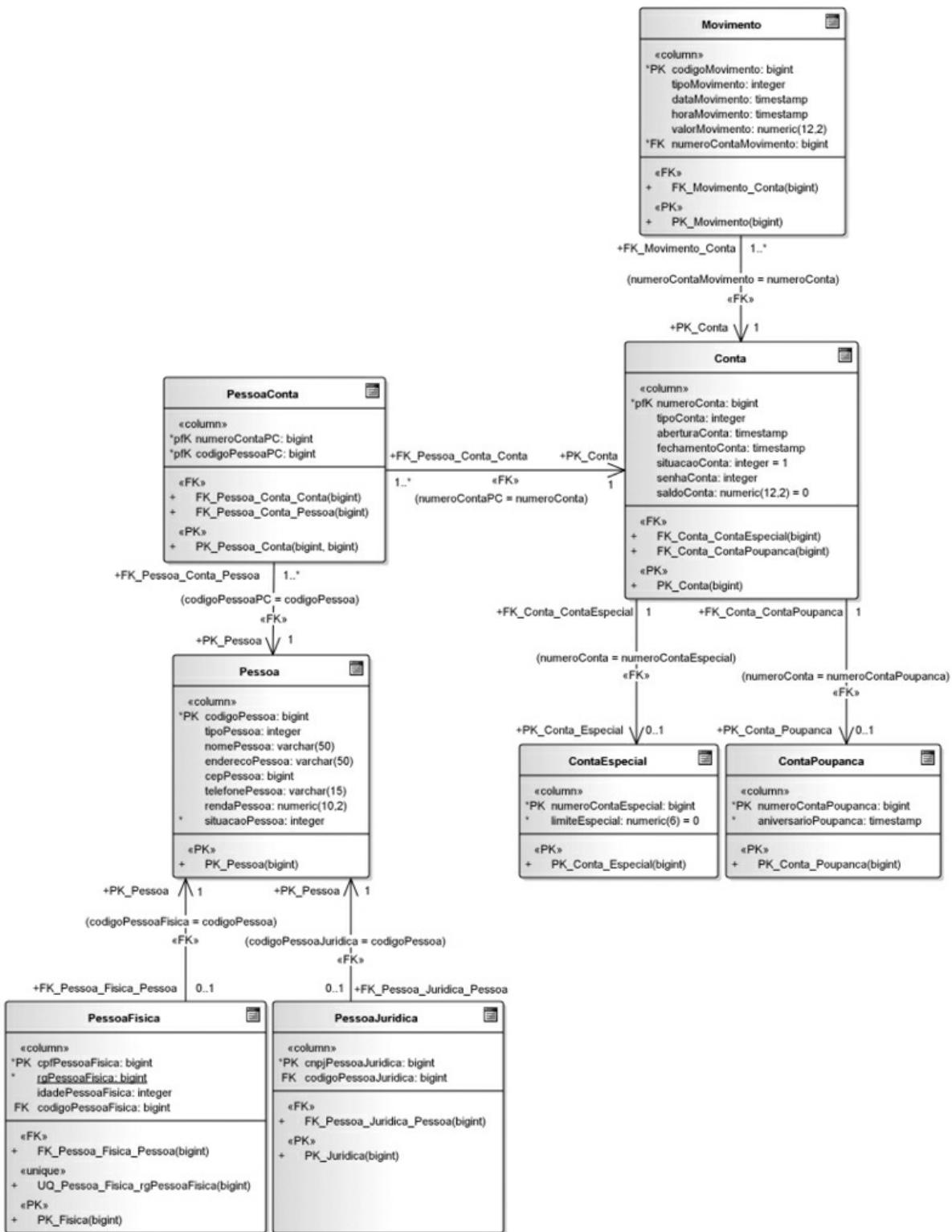


Figura 4.65 – Mapeamento do Modelo Conceitual do Sistema de Controle Bancário.

Em seguida, criamos uma associação entre as tabelas **PessoaFisica** e **Pessoa**, o mesmo ocorrendo entre as tabelas **PessoaJuridica** e **Pessoa**. Essas associações relacionam uma pessoa física a uma pessoa por meio da comparação do valor da chave estrangeira **codigoPessoaFisica** da tabela **PessoaFisica** com a chave primária **codigoPessoa** da tabela **Pessoa**, ocorrendo algo semelhante entre as tabelas **PessoaJuridica** e **Pessoa**, em que comparamos a chave estrangeira **codigoPessoaJuridica**, da tabela **PessoaJuridica**, com a chave primária **codigoPessoa**, da tabela **Pessoa**. Observe que a multiplicidade dessas associações é 0..1 para 1, significando que uma pessoa física ou jurídica está relacionada a uma e somente uma pessoa, e uma pessoa pode estar relacionada a nenhuma ou uma pessoa física, ou a nenhuma ou uma pessoa jurídica. Uma pessoa pode estar associada a nenhuma pessoa física porque pode estar associada a uma pessoa jurídica e vice-versa, sendo as pessoas física e jurídica mutuamente exclusivas.

O mapeamento das classes **ContaComum**, **ContaEspecial** e **ContaPoupanca** foi um pouco diferente pelo fato de a classe **ContaComum** ser uma classe concreta e não abstrata. Dessa forma, mapeamos cada classe em uma tabela correspondente, nomeando a tabela relativa à classe **ContaComum** simplesmente **Conta** e inserindo-lhe uma coluna chamada **tipoConta** para determinar se a linha se refere a uma conta comum, especial ou poupança. Não foi necessário inserir uma coluna para representar a chave primária, uma vez que a coluna **numeroConta** ocupa essa função perfeitamente.

Para manter a coerência com o modelo conceitual, relacionamos as tabelas **ContaEspecial** e **ContaPoupanca** com a tabela **Conta**. O leitor poderá perceber que foi criada uma chave primária correspondente ao número da conta em ambas as tabelas.

Nesses relacionamentos, porém, diferentemente do que ocorre entre a tabela **Pessoa** e as tabelas **PessoaFisica** e **PessoaJuridica**, as chaves estrangeiras estão posicionadas na tabela **Conta**, o que foi necessário porque a pesquisa sempre se dará na tabela **Conta** e, se for percebido que o tipo de conta não se refere a uma conta comum, será pesquisado, por meio do relacionamento, o limite da conta, se esta for especial, ou a data de

aniversário, se for poupança. Note que a coluna `numeroConta`, além de servir como chave primária na tabela `Conta`, serve também como chave estrangeira para pesquisar uma conta especial ou poupança, se isto for necessário.

Observe que a multiplicidade dessas associações é 1 para 0..1, o que significa que uma conta pode estar associada a nenhuma ou somente uma conta especial e que uma conta especial tem de estar associada a somente uma conta, o mesmo ocorrendo com a associação entre as tabelas `Conta` e `ContaPoupança`, uma vez que uma conta pode referir-se a uma conta especial ou a uma conta de poupança ou a nenhuma das duas, quando se constitui em uma conta comum.

Assim, se uma linha da tabela `Conta` corresponder ao tipo conta comum, não haverá ligação nem com a tabela `ContaEspecial` nem com a tabela `ContaPoupança`. Porém, se o tipo for conta especial ou conta de poupança, haverá uma ligação com uma linha da tabela correspondente.

Em seguida, foi feito o mapeamento relativo à associação muitos para muitos entre as classes `Pessoa` e `ContaComum` no modelo conceitual, o que forçou a criação de uma tabela intermediária chamada `PessoaConta`, que tem como chave primária duas colunas que armazenam o número da conta e o código da pessoa. Essas colunas também desempenham a função de chaves estrangeiras, permitindo, assim, relacionar cada linha dessa tabela com a pessoa e conta a qual ela corresponde.

Finalmente, foi mapeada a classe `Movimento` em uma tabela de mesmo nome, contendo colunas correspondentes a todos os seus atributos, adicionando ainda uma coluna para servir de chave primária e outra para servir de chave estrangeira, de maneira que seja possível determinar a qual conta cada movimento corresponde.

## 4.16 Padrão Repository

Quando se trabalha com a camada de domínio, deve-se inserir nas classes de entidade somente os comportamentos relativos à lógica do negócio. A forma como será realizada a persistência dos objetos não faz parte do escopo dessa camada. É quase um consenso que detalhar questões de acesso a dados em classes de entidade é uma péssima decisão de projeto.

Por esse motivo, recomenda-se criar uma nova camada, responsável pela persistência e recuperação dos objetos.

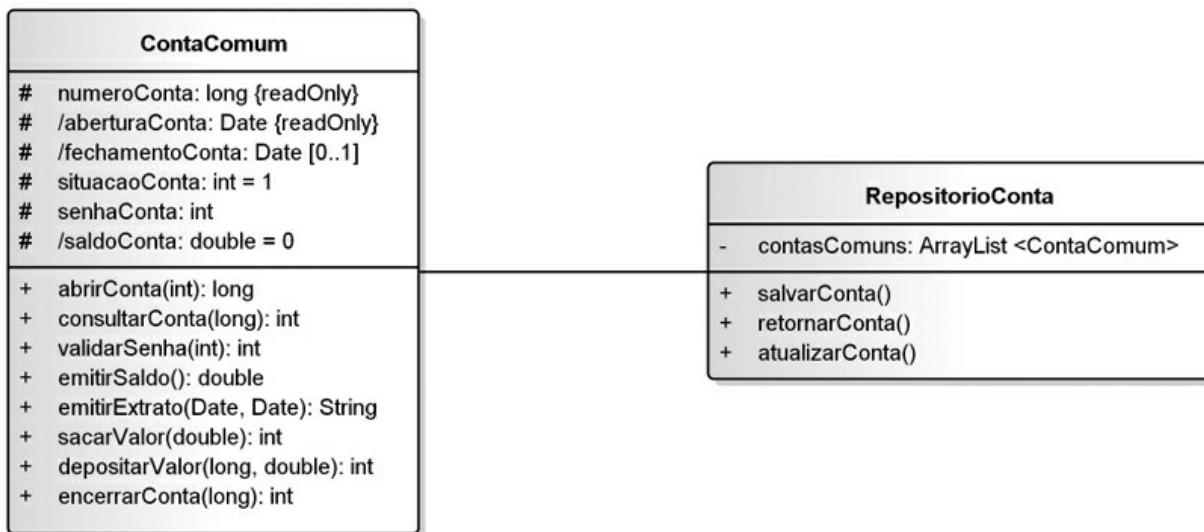
Nesse sentido, o padrão Repository surgiu com o objetivo de abstrair questões relacionadas à manipulação de bancos de dados, como conexões, gravação e recuperação de dados em disco. Assim, o uso desse padrão procura manter a camada de domínio sem saber a forma como seus objetos serão persistidos, isolando a camada de domínio do código necessário para acessar dados em disco e, assim, separando a lógica de acesso a dados da lógica da aplicação.

Para facilitar a comunicação entre essas camadas, o padrão Repository foi concebido para servir de intermediário entre a camada de domínio e a camada de persistência. As classes do tipo Repository representam coleções de objetos em memória, normalmente referentes a uma determinada classe de entidade. Classes desse tipo também contêm algumas operações que podem ser realizadas sobre essas coleções. Basicamente, essas operações caracterizam-se por permitir a adição, remoção e atualização de elementos na coleção, bem como selecionar um ou mais elementos da coleção. Nada impede, todavia, que outros métodos sejam acrescidos a essas classes. As classes Repository pertencem à camada de domínio, embora, por vezes, possam ser representadas em um subpacote dentro dessa camada.

Basicamente, classes de repositório armazenam, excluem, atualizam e consultam objetos na coleção sob sua responsabilidade e, quando necessário, fazem solicitações à camada de persistência para gravar, excluir ou retornar informações. A forma mais simples de projetar classes de repositório é criar uma classe Repository para cada classe de entidade cujos objetos precisarem ser persistidos e definir somente os métodos específicos necessários à manipulação da coleção desses objetos. Outra abordagem é criar uma classe de interface genérica simples e, a partir dela, classes que realizem seus métodos. A figura 4.66 apresenta um exemplo de classe de repositório.

Nesse exemplo, criamos uma classe chamada **RepositoryConta** contendo um atributo do tipo **ArrayList** para representar a coleção de objetos da classe **ContaComum** ao qual essa classe de repositório está associada. Nessa classe também foram definidos três métodos básicos para acrescentar um objeto à coleção, para retornar um ou mais objetos dessa

coleção e atualizar um objeto na coleção. Lembramos que uma conta não pode nunca ser excluída, apenas marcada como encerrada.



*Figura 4.66 – Exemplo de Classe Repository.*

#### 4.17 Padrão DAO (Data Access Object)

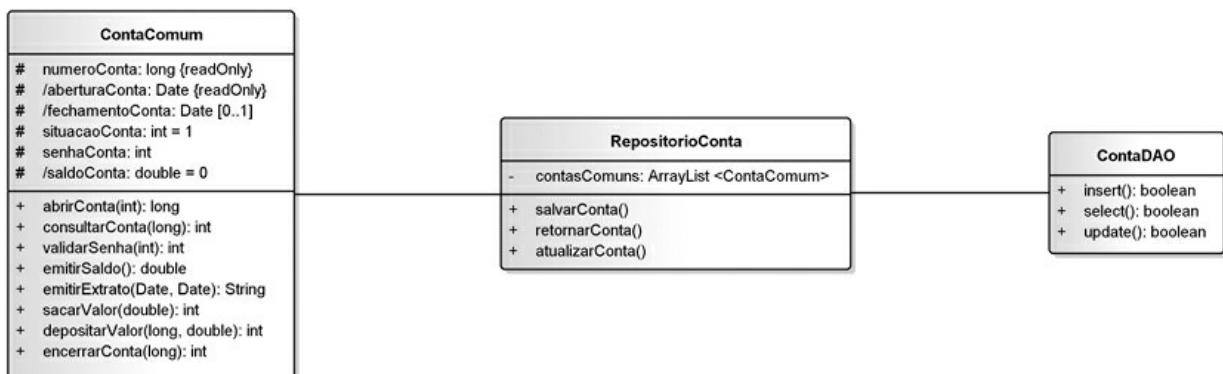
Como já foi dito, o padrão Repository tem o objetivo de auxiliar no processo de persistência de objetos, no entanto classes de repositório pertencem à camada de domínio e apenas gerenciam coleções de objetos de uma determinada classe de entidade. Classes Repository não possuem conhecimento real da infraestrutura de banco de dados utilizada pelo software.

O padrão DAO (Data Access Object ou Objeto de Acesso a Dados) representa classes cujos objetos estão relacionados à infraestrutura da aplicação. Diferentemente das classes de repositório, classes DAO não fazem parte da camada de domínio, e sim da camada de persistência, e encapsulam detalhes referentes à persistência de dados.

Muitas vezes pode ser necessário utilizar ambos os padrões Repository e DAO. Nessa abordagem, o modelo de domínio solicitará e armazenará objetos em um repositório e este, quando necessário, fará solicitações a um DAO, posto que é este quem possui conhecimentos a respeito da estrutura de dados utilizada. Assim, a DAO traduz as chamadas de persistência de um repositório em chamadas adequadas à estrutura de dados utilizada.

Isso permite independência entre as camadas e, caso a infraestrutura seja modificada, as alterações só precisarão ser feitas nas classes DAO. Em geral, as chamadas são genéricas, como insert, update, delete e select. A figura 4.67 apresenta um exemplo simples de classe DAO.

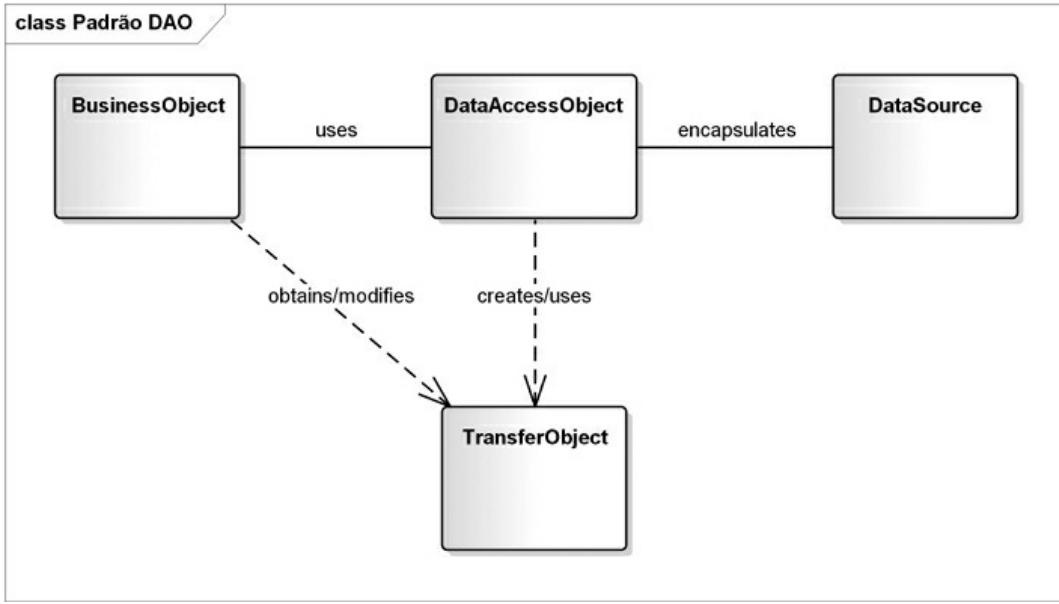
Nesse exemplo, damos continuidade ao exemplo anterior, acrescentando a classe ContaDAO responsável por persistir e recuperar a coleção de objetos mantida pelo repositório. Os métodos dessa classe são basicamente os necessários para inserir, atualizar ou selecionar objetos, porém outros métodos podem ser acrescidos se isto for considerado necessário.



*Figura 4.67 – Exemplo de Classe DAO.*

Esse exemplo não está completamente correto, posto que, como já dissemos, as classes DAO devem estar em uma camada separada, mas decidimos representar as classes de entidade, repositório e DAO na mesma camada para facilitar a explicação, uma vez que ainda não falamos sobre o diagrama de pacotes. O leitor também poderá encontrar exemplos em que é utilizada uma classe de interface DAO e os métodos nela contidos serem realizados por outras classes. Nesse exemplo, no entanto, não achamos necessário proceder dessa maneira.

A figura 4.68 detalha o padrão DAO, especificando os seus componentes:



*Figura 4.68 – Padrão DAO.*

Ao examinarmos essa figura, percebemos que é formada por quatro classes:

- **BusinessObject** – esta classe representa uma classe de entidade que contém a lógica do negócio (representada principalmente por seus métodos). Os objetos dessa classe contêm os dados que devem ser transmitidos para e recuperados da base de dados. A classe BusinessObject utiliza a classe DataAccessObject para persistir e recuperar as informações de seus objetos, conforme demonstra a associação entre as duas classes. Quando se utiliza o padrão Repository com o DAO, a classe Repository passa a servir de intermediária da classe BusinessObject. Observe que a classe BusinessObject não é uma classe real, pois representa um conceito usado pelo padrão. Assim, quando esse padrão for implementado, uma classe de repositório ou uma classe de entidade propriamente dita substituirá a classe BusinessObject.
- **DataAccessObject** – esta é a classe DAO propriamente dita. Como foi explicado nesta seção, ela encapsula o acesso aos dados, que só pode ser feito por meio de seus objetos. Cada instância dessa classe é responsável por um objeto de domínio (uma instância de uma classe de entidade). Quando esse padrão é utilizado, as classes DAO incorporam os mapeamentos entre as classes e tabelas.
- **DataSource** – esta classe representa a origem dos dados, podendo ser

qualquer tipo de repositório físico, como um banco de dados relacional ou um simples arquivo de texto. Como se pode observar pela associação existente entre essa classe e a classe DAO, a última encapsula a fonte dos dados.

- **TransferObject:** esta classe representa objetos intermediários necessários para transferir informações da base de dados ou para a base de dados. Sempre que for necessário recuperar informações de um objeto de entidade ou persistir um objeto na fonte de dados, um objeto dessa classe precisará ser criado. Como se pode perceber ao observar a figura, a classe DAO cria e utiliza esses objetos, enquanto a classe BusinessObject obtém ou modifica informações desses objetos.

O funcionamento desse padrão será ilustrado no capítulo 7, que trata sobre o diagrama de sequência.

## 4.18 Exercícios Propostos

Esta seção dará continuidade à modelagem dos sistemas iniciados no capítulo 3, desta vez enfocando a visão estrutural e estática do diagrama de classes. Demonstraremos aqui as soluções já referentes aos modelos de domínio; as camadas de visão e controle não serão apresentadas, posto que adotamos uma classe de visão e uma classe controladora para cada caso de uso primário.

### 4.18.1 Sistema de Controle de Cinema

Desenvolva o diagrama de classes – modelo de domínio para um sistema de controle de cinema, com base nos seguintes requisitos:

- Um cinema pode ter muitas salas, sendo necessário, portanto, registrar informações a respeito de cada sala, como sua capacidade, ou seja, o número de assentos disponíveis.
- O cinema apresenta muitos filmes. Um filme tem informações como título e duração. Assim, sempre que um filme for apresentado, deve-se registrá-lo também.
- Um filme tem um único gênero, mas um gênero pode se referir a muitos filmes.
- Um filme pode ter muitos atores atuando nele e um ator pode atuar em

muitos filmes. Em cada filme, um ator interpretará um ou mais papéis diferentes. Por uma questão de propaganda, é útil anunciar os principais atores do filme e que papéis eles interpretam.

- Um mesmo filme pode ser apresentado em diferentes salas e horários. Cada apresentação em uma determinada sala e horário é chamada Sessão. Um filme apresentado em uma sessão tem um conjunto máximo de ingressos, determinado pela capacidade da sala.
- Os clientes do cinema podem comprar ou não ingressos para assistir a uma sessão. O funcionário deve intermediar a compra do ingresso. Um ingresso deve conter informações como o tipo de ingresso (meia-entrada ou ingresso inteiro). Além disso, um cliente só pode comprar ingressos para sessões ainda não encerradas.

#### **4.18.2 Sistema de Controle de Clube Social**

Desenvolva o modelo de domínio para um sistema de controle de clube social de acordo com os seguintes requisitos:

- O clube tem muitos sócios e precisa manter informações referentes a eles, como o número de seu cartão de sócio, nome, endereço, telefone e e-mail.
- Um sócio pode ter nenhum ou muitos dependentes, mas um dependente está associado a somente um sócio. O clube precisa manter informações sobre os dependentes de cada sócio, como o número de seu cartão, nome, parentesco e e-mail.
- Um sócio deve pertencer a uma única categoria. No entanto, pode haver muitos sócios pertencentes a uma determinada categoria.
- Um sócio deve pagar mensalidades para poder frequentar o clube. Assim, enquanto permanecer sócio do clube, um sócio poderá pagar muitas mensalidades, mas uma mensalidade pertence a somente um sócio. Eventualmente, um sócio pode não estar adimplente. Nesse caso, serão cobrados juros sobre o valor da mensalidade relativos ao atraso do pagamento. É também possível que um sócio nunca tenha pago suas mensalidades. As informações pertinentes a cada mensalidade são a data de pagamento, o valor, a data em que foi efetivamente paga, os possíveis juros aplicados, o valor efetivamente pago e se está quitada ou não.

### **4.18.3 Sistema de Locação de Veículos**

Desenvolva o modelo de domínio para um sistema de locação de veículos, levando em consideração os seguintes requisitos:

- A empresa tem muitos automóveis. Cada automóvel tem atributos como número da placa, cor, ano, tipo de combustível, número de portas, quilometragem, Renavam, chassi, valor de locação etc.
- Cada carro tem um modelo e uma marca, mas um modelo pode relacionar-se a muitos carros e uma marca pode referir-se a muitos modelos, embora cada modelo só tenha uma marca específica.
- Um carro pode ser alugado por muitos clientes, em momentos diferentes, e um cliente pode alugar muitos carros. É preciso saber quais carros estão locados ou não. Sempre que um carro for locado, será preciso armazenar a data e a hora de sua locação e, quando for devolvido, a data e a hora de devolução.

### **4.18.4 Sistema para Controle de Leilão Via Internet**

Desenvolva o modelo de domínio para um sistema de leilão via internet, de acordo com os seguintes requisitos:

- Cada leilão deve conter informações como data de início, hora de início, data de encerramento e hora de encerramento.
- Em cada leilão, existem diversos itens a serem leiloados. Cada item está associado a um único leilão. Se não for leiloado naquele momento, deverá ser cadastrado como item de outro leilão novamente. Cada item tem um lance mínimo.
- Um item pode receber muitos lances, mas pode não receber nenhum. Nesse último caso, não será arrematado.
- Existem diversos participantes em cada leilão interessados em adquirir os itens ofertados. Os participantes devem se registrar via internet, antes de o leilão iniciar.
- Um participante pode realizar quantos lances quiser, mas não é obrigado a realizar lance algum.

### **4.18.5 Sistema de Controle de Hotelaria**

Desenvolva o modelo de domínio para um sistema de controle de hotelaria de acordo com os seguintes fatos:

- O hotel aluga quartos de diversas categorias (simples, duplo, casal, luxo etc.). O valor dos quartos varia de acordo com a categoria.
- Os quartos do hotel podem ser reservados previamente antes de os hóspedes virem ocupá-los. Para isso, é necessário informar os dados do cliente que os está reservando, a data da reserva e a provável data em que um quarto será desocupado.
- Cada hóspede precisa ser identificado no momento em que ocupa um quarto, mesmo que este seja pago por outro cliente. Caso seu cadastro ainda não exista ou seus dados tenham mudado, será necessário cadastrá-lo.
- Um hóspede pode alugar muitos quartos, em um mesmo momento ou em momentos diferentes, e um quarto pode ser alugado por muitos hóspedes, em momentos diferentes, naturalmente.
- Dependendo da categoria do quarto, terá uma determinada quantidade de itens, tanto no quarto propriamente dito como no frigobar.
- Um hóspede pode consumir itens do frigobar. Cada item tem valores e quantidades diferentes. É preciso registrar o consumo do hóspede para posterior cobrança.
- Um hóspede pode solicitar serviços do hotel, como passar roupas ou lavanderia. Da mesma forma que o consumo, cada serviço solicitado precisa ser registrado.
- Cada quarto ocupado gera diárias sempre ao meio-dia. Uma diária deve ser paga exclusivamente por um determinado hóspede (ou pelo cliente que fez a reserva), mas um hóspede pode pagar muitas diárias.
- É necessário saber qual funcionário foi responsável pela locação e/ou encerramento de cada locação de um quarto.

#### **4.18.6 Sistema de Controle de Imobiliária**

Desenvolva o modelo de domínio para um sistema de controle de imobiliária de acordo com os seguintes requisitos:

- A imobiliária intermedeia a venda ou aluguel de muitos imóveis. Sendo

assim, precisa manter informações sobre todos os imóveis com que já trabalhou, estando eles em oferta ou já vendidos ou locados. Os imóveis podem ser de diversos tipos, como casas, apartamentos, salas comerciais, chácaras ou terrenos.

- A imobiliária negocia com muitos donos de imóveis que desejam vendê-los ou alugá-los. A imobiliária considera dono a pessoa física ou jurídica possuidora de um ou mais imóveis. Um dono deve possuir (ou ter possuído em algum momento) ao menos um imóvel, mas pode possuir muitos.
- Um imóvel pode ser vendido ou alugado diversas vezes (em tempos diferentes obviamente). É necessário registrar cada venda ou locação intermediada pela imobiliária. Essa informação deve incluir os dados do dono e os do comprador/locador, bem como o corretor que intermediou a operação. No caso de venda, deve-se incluir o valor pago pelo imóvel, as taxas e impostos pagos, bem como o percentual recebido pela imobiliária. No caso de locação, deve-se registrar o valor mensal do aluguel, o tempo de vigência do aluguel, os dados dos fiadores ou o valor da caução.
- No caso de locações, a imobiliária se compromete em repassar o valor dos aluguéis ao dono do imóvel, após descontar seu percentual. Assim, sempre que um locador pagar o aluguel (com possível juro em caso de atraso), a imobiliária repassará ao dono o valor pago descontado de sua comissão. Dessa forma, todos os pagamentos de aluguel devem ser registrados.
- A empresa possui muitos corretores trabalhando para ela. É responsabilidade dos corretores atender os clientes da imobiliária e intermediar a venda ou aluguel de imóveis. É necessário, portanto, registrar todos os corretores que trabalham ou já trabalharam para a imobiliária.
- A imobiliária considera clientes as pessoas físicas ou jurídicas que compram ou alugam imóveis oferecidos pela empresa, bem como os donos desses imóveis. Sendo assim, é preciso manter registros dos dados de todos os clientes da imobiliária, atuais ou anteriores.

## 4.19 Solução dos Exercícios

Nesta seção, apresentaremos as soluções dos problemas propostos. Aqui, cumpre chamar a atenção para a declaração de alguns métodos das soluções apresentadas. A rigor é necessário haver um get e um set para cada método de uma classe, mas é inviável e não recomendado representar esses métodos nesses diagramas pelo grande espaço que ocupariam e por não acrescentarem comportamentos importantes ao modelo. Por esse motivo, muitas vezes declaramos métodos “genéricos” para retornar várias informações de uma classe e definimos seu retorno como uma String.

### 4.19.1 Sistema de Controle de Cinema

A figura 4.69 apresenta a solução desse exercício. Pode-se perceber que o diagrama apresentado se refere ao modelo de domínio.

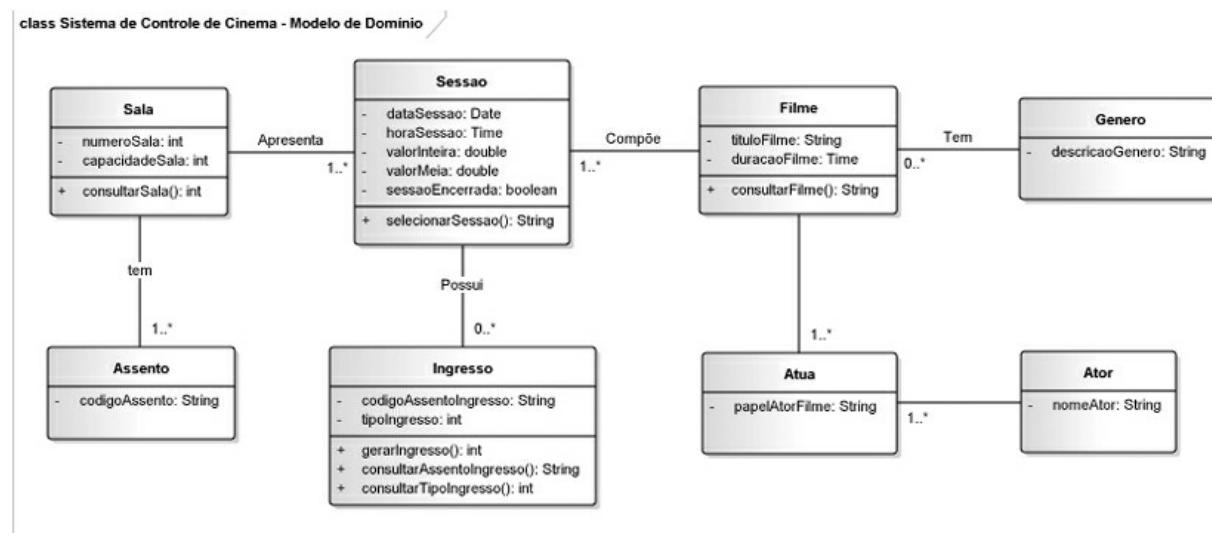


Figura 4.69 – Diagrama de Classes – Modelo de Domínio – para o Sistema de Controle de Cinema – Solução do Exercício.

A seguir, explicaremos as classes de entidade que compõem o diagrama. Nesse diagrama, identificamos somente os métodos considerados mais importantes.

- **Genero** – Essa classe armazena os gêneros de filmes apresentados pelo cinema. Seu único atributo é a descrição do gênero do tipo **String**.
- **Filme** – Essa classe contém o título e a duração de cada filme apresentado no cinema. Seu único método serve para consultar os filmes

cadastrados. Observe que um filme está associado a um único gênero, mas um gênero pode estar associado a muitos filmes.

- **Ator** – Essa classe representa os atores que interpretam papéis nos filmes exibidos no cinema. Seu único atributo é o nome do ator.
- **Atua** – Esta é uma classe intermediária entre as classes **Ator** e **Filme** e representa os papéis que um ator interpreta em cada filme de que participa. Seu único atributo consiste no papel que um ator interpreta. Observe que um ator pode atuar em, no mínimo, um e, no máximo, muitos filmes e que um filme pode ter muitos atores atuando nele, mas um filme deve ter pelo menos um ator.
- **Sala** – Essa classe armazena as informações sobre as salas pertencentes ao cinema. Seus atributos são número da sala e capacidade da sala, ambos do tipo inteiro. Seu único método permite consultar uma determinada sala e retornar sua capacidade.
- **Assento** – Essa classe serve apenas para armazenar os códigos de assento de uma determinada sala.
- **Sessão** – Essa classe representa as sessões de filmes apresentadas pelo cinema. Seus atributos são data da sessão, hora da sessão, valor do ingresso inteiro, valor da meia-entrada e um atributo para determinar se a sessão já foi encerrada ou não. Observe que uma sessão é apresentada em uma única sala, mas em uma sala podem ter sido apresentadas muitas sessões. Note também que em uma sessão é exibido somente um filme, mas um filme pode ser exibido em muitas sessões. O único método definido nessa classe permite selecionar as sessões ainda em aberto, retornando uma **String** com os dados da sessão em questão.
- **Ingresso** – Finalmente, essa classe representa os ingressos vendidos em cada sessão de cinema. Essa classe armazena o código do assento a que se refere o ingresso e o tipo de ingresso, se é um inteiro ou meia-entrada. A classe possui ainda os seguintes métodos:

Método	Descrição
<code>gerarIngresso</code>	Gera um novo ingresso, retornando verdadeiro (1), se foi possível gerar o ingresso, ou falso (0), caso contrário. Observe que uma sessão pode gerar muitos ingressos, contudo pode não gerar ingresso algum, conforme demonstra a multiplicidade.

**consultarAssentoIngresso** Retorna uma String contendo código do assento de um determinado ingresso.

**consultarTipoIngresso** Retorna o tipo de ingresso de um ingresso específico.

## 4.19.2 Sistema de Controle de Clube Social

A solução desse exercício é apresentada na figura 4.70. Como no exercício anterior, esse diagrama se refere ao modelo de domínio, tendo sido somente identificados os métodos considerados pertinentes.

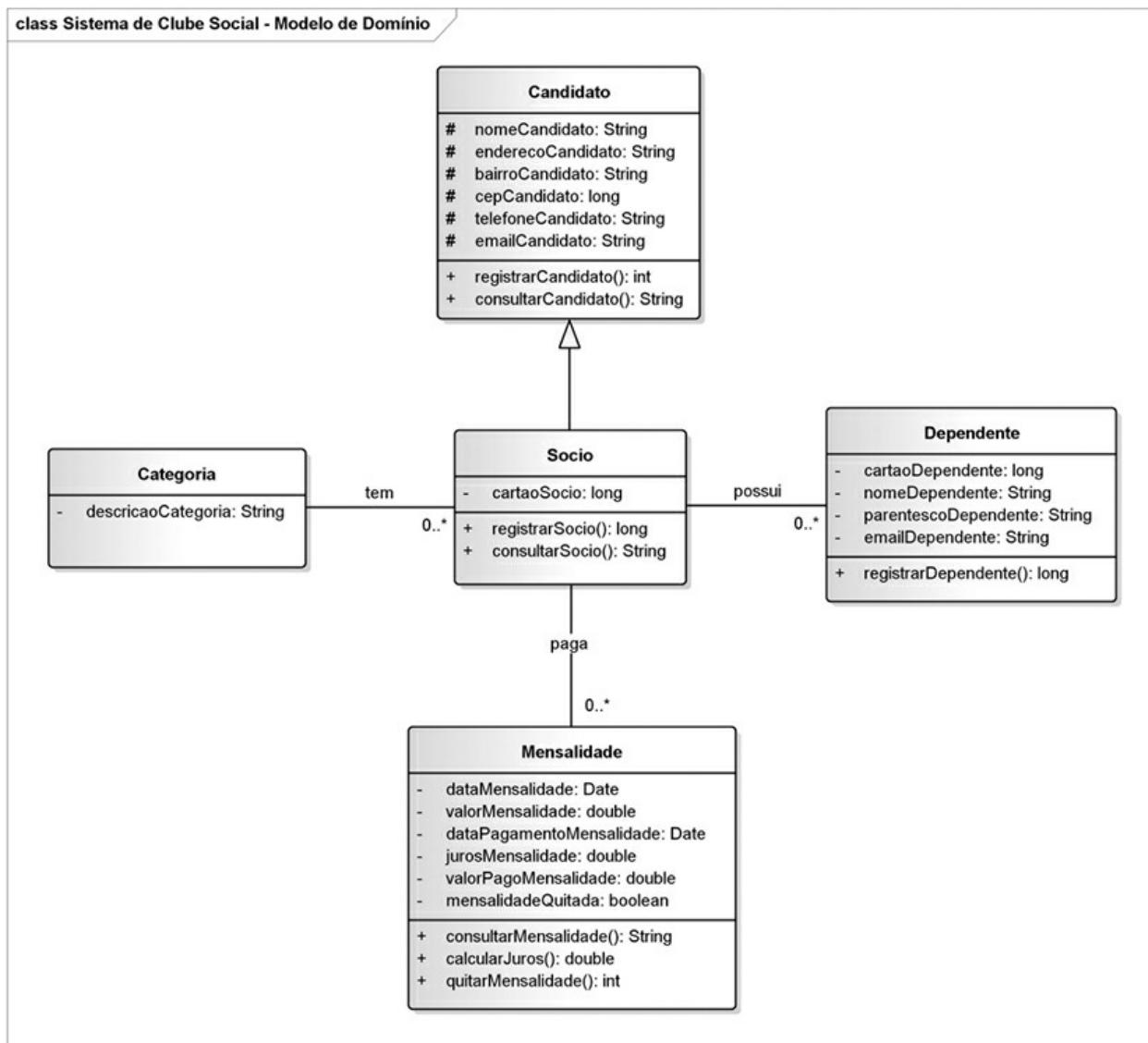


Figura 4.70 – Modelo de domínio para o Sistema de Controle de Clube Social – Solução do Exercício.

A seguir, explicaremos as classes de entidade que compõem este modelo.

- **Candidato** – Essa classe armazena os dados das pessoas que se candidatam a sócios do clube. Os atributos dessa classe são autoexplicativos. Observe que a visibilidade dos atributos é protegida, posto que essa classe é especializada pela classe Socio. A classe tem dois métodos, um para registrar um candidato e outro para consultar um candidato.
- **Socio** – Essa classe armazena as informações referentes aos sócios do clube. Os atributos dessa classe são todos herdados da classe **Candidato**, exceto o atributo referente ao número do cartão do sócio que só existirá se um candidato vier a ser aprovado. A classe tem dois métodos, um para registrar um sócio e outro para consultar um sócio específico.

O método `registrarSocio` retorna um `long` que representa o número do cartão do sócio e o método `consultarSocio` retorna uma `String` contendo os dados de um determinado sócio. Observe que um sócio pertence a uma categoria, mas uma categoria pode estar associada a muitos sócios.

- **Dependente** – Essa classe armazena as informações referentes aos possíveis dependentes de um sócio. Os atributos da classe são autoexplicativos. O único método contido pela classe permite gerar uma nova instância dela. Pode-se perceber que um dependente está relacionado a um único sócio, mas um sócio pode não ter nenhum dependente ou pode ter vários.
- **Categoria** – Essa classe representa as possíveis categorias de sócios estabelecidas pelo clube. Seu único atributo é a descrição da categoria. Pode haver muitos sócios associados a uma determinada categoria, por outro lado pode não haver nenhum sócio associado a uma categoria específica.
- **Mensalidade** – A classe em questão representa as mensalidades que devem ser pagas por cada sócio. Seus atributos são data da mensalidade e data em que a mensalidade foi efetivamente paga, do tipo `Date`, valor da mensalidade, juros da mensalidade, valor efetivamente pago do tipo `double` e um atributo denominado `mensalidadeQuitada` do tipo `boolean`, que determina se a mensalidade foi quitada ou não. Já os métodos da classe são:

Método	Descrição
<b>consultarMensalidade</b>	É disparado para consultar cada mensalidade ainda não paga de um determinado sócio. Retorna uma <b>String</b> com os dados da mensalidade.
<b>calcularJuros</b>	Calcula os juros de uma mensalidade, no caso de esta estar atrasada. Retorna um <b>double</b> contendo o valor atual, após a aplicação de juros, da mensalidade.
<b>quitarMensalidade</b>	Permite a quitação de uma mensalidade, retornando verdadeiro, se a operação foi concluída com sucesso, ou falso, se ocorreu algum problema quando se tentou quitar a mensalidade.

### 4.19.3 Sistema de Locação de Veículos

A solução desse exercício é apresentada na figura 4.71. Da mesma forma que no exercício anterior, esse diagrama já se refere ao modelo de domínio e igualmente só foram identificados os métodos considerados pertinentes.

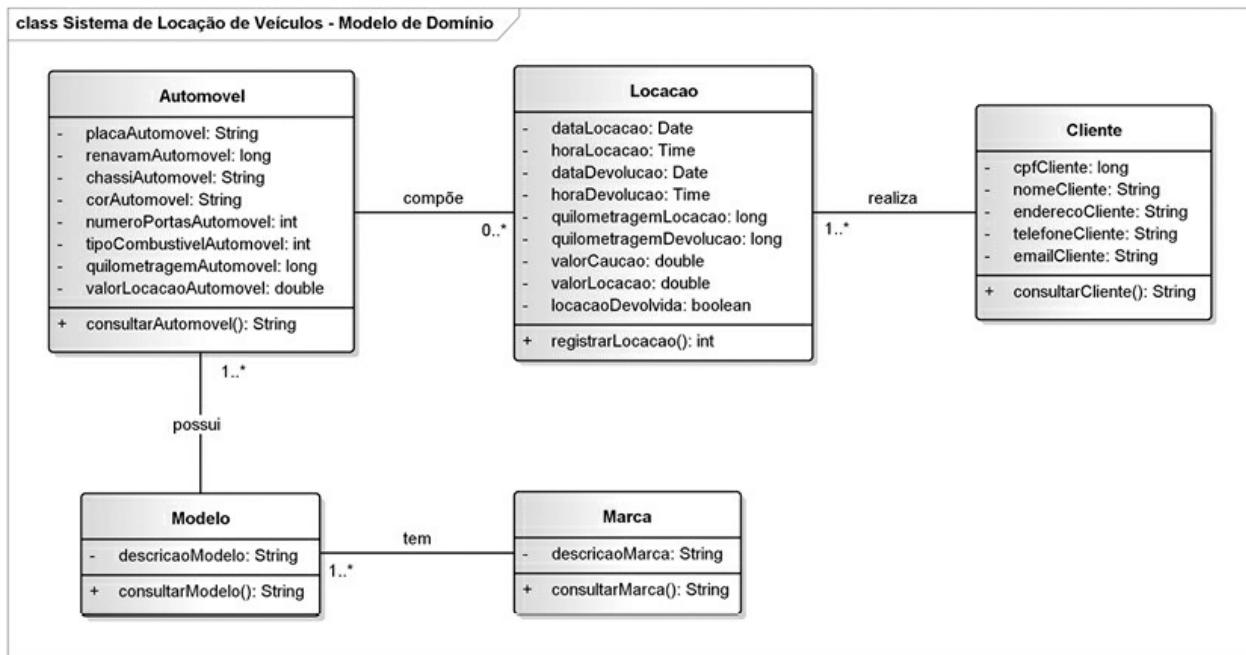


Figura 4.71 – Modelo de domínio para o Sistema de Locação de Veículos – Solução do Exercício.

As classes de entidade identificadas neste diagrama foram as seguintes:

- **Marca** – Essa classe representa as marcas dos carros disponíveis na locadora. Tem como atributo a descrição da marca, do tipo **String**, além do método **consultarMarca**, usado para consultar uma marca específica. Como se pode perceber, o método retorna uma **String**.

- **Modelo** – Já essa classe armazena os modelos das marcas disponíveis na locadora. À semelhança da classe **Marca**, essa classe tem como atributo a descrição do modelo e um método que permite consultar um determinado modelo. Observe que um modelo tem somente uma marca, mas uma marca pode ter muitos modelos.
- **Cliente** – Essa classe, por sua vez, armazena as informações relativas aos clientes que locam veículos. Seus atributos são autoexplicativos. A classe tem ainda um método para consultar um determinado cliente e retornar seus dados.
- **Automóvel** – Essa classe contém as informações dos automóveis locados pela empresa, indo desde a placa do veículo até o valor de locação. Existe ainda um método que permite consultar um determinado automóvel, retornando os dados deste. Um automóvel está relacionado a um único modelo, mas um modelo pode estar associado a muitos automóveis.
- **Locação** – Essa classe apresenta as informações relativas às locações de automóveis já realizadas. A classe tem os atributos data de locação e data de devolução, do tipo **Date**; hora de locação e hora de devolução, do tipo **Time**; quilometragem no momento da locação e quilometragem no momento da devolução, do tipo **long**; tipo de combustível, do tipo **int**; valor de caução e valor de locação, do tipo **double**; um atributo chamado **locacaoDevolvida**, do tipo **boolean**, cuja função é determinar se a locação foi devolvida ou não. A classe tem ainda um método para registrar uma locação, que retorna verdadeiro, se foi executado com sucesso, e falso, caso contrário. Ao examinarmos as associações dessa classe, concluímos que um cliente pode realizar muitas locações (no mínimo, uma), mas uma locação refere-se a um único cliente e a um veículo em particular e um mesmo veículo pode estar relacionado a muitas locações ou não ter sido locado nunca.

#### **4.19.4 Sistema para Controle de Leilão Via Internet**

A solução desse exercício é apresentada na figura 4.72. Novamente, como no exercício 4.71, esse diagrama se refere ao modelo de domínio e novamente só foram identificados os métodos considerados pertinentes.

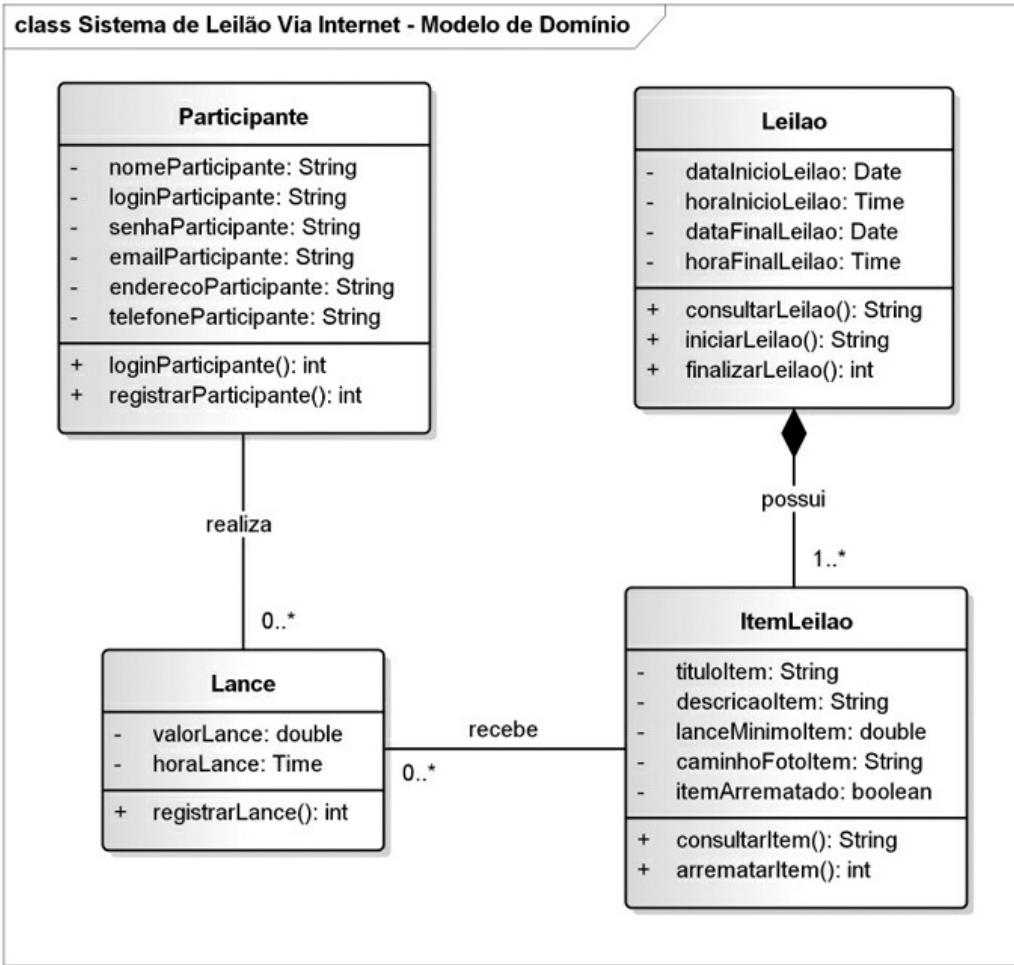


Figura 4.72 – Modelo de Domínio para o Sistema de Controle de Leilão Via Internet.

A seguir, explicaremos as classes de entidade contidas neste diagrama.

- **Participante** – Essa classe refere-se às informações das pessoas que se cadastraram para participar do leilão e, possivelmente, oferecer lances para os itens leiloados. Os atributos da classe são autoexplicativos. A classe contém ainda os métodos para permitir a um participante se logar, denominado `loginParticipante`, e para permitir o autorregistro de um novo participante, chamado `registrarParticipante`. Ambos os métodos retornam verdadeiro, se o método atingiu seu objetivo, ou falso, caso contrário.
- **Leilao** – Aqui identificamos nesta classe os leilões programados ou realizados pela instituição. Seus atributos são a data de início e fim do leilão, do tipo `Date`, bem como a hora de início e fim, do tipo `Time`. Essa

classe tem ainda os seguintes métodos:

Método	Descrição
<b>consultarLeilao</b>	É disparado para consultar os leilões registrados, retornando uma <b>String</b> com os dados referentes a cada leilão.
<b>iniciarLeilao</b>	Tem por função iniciar um leilão. Por depender do retorno de outro método de outra classe, retorna uma <b>String</b> .
<b>finalizarLeilao</b>	Serve para marcar um leilão específico como finalizado. Retorna verdadeiro, se consegue realizar seu intento, ou falso, caso contrário.

- **ItemLeilao** – Essa classe contém as informações dos itens que serão ofertados em cada leilão. Seus atributos são título, descrição e caminho da foto do item, do tipo **String**, lance mínimo, do tipo **double**, e um atributo identificado como “**itemArrematado**”, do tipo **boolean**, cujo objetivo é determinar se o item foi arrematado (verdadeiro) ou não (falso).

Observe que a classe **Leilao** tem uma associação de composição com a classe **ItemLeilao**, o que significa que a informação de um objeto **Leilao** deve ser complementada por um ou mais objetos **ItemLeilao** e que um item de leilão está associado exclusivamente a um leilão específico. Essa relação todo-parte é particularmente clara ao examinarmos os métodos **iniciarLeilao** e **consultarItem**, principalmente se consultarmos o diagrama de sequência referente ao processo **Realizar Leilão**, no final do capítulo sobre o diagrama de sequência.

A classe **ItemLeilao** contém os seguintes métodos:

Método	Descrição
<b>consultarItem</b>	É disparado para consultar um item específico, estando associado ao método <b>consultarItem</b> <b>iniciaLeilao</b> da classe <b>Leilao</b> . Retorna uma <b>String</b> contendo os dados do item.
<b>arrematarItem</b>	Tem o objetivo de definir um determinado item como arrematado, retornando verdadeiro, se for bem-sucedido, ou falso, caso contrário.

- **Lance** – Essa classe é responsável por armazenar os lances realizados sobre cada item ofertado. Seus atributos são valor do lance, do tipo **double**, e o momento em que o lance foi ofertado, do tipo **Time**. Seu único método, **registrarLance**, permite registrar um novo lance. Observe que um lance refere-se a um único item, mas um item pode receber nenhum ou muitos lances. Além disso, um lance é ofertado por

um participante específico, mas um participante pode realizar nenhum ou muitos lances.

#### 4.19.5 Sistema de Controle de Hotelaria

Na figura 4.73, podemos examinar a solução desse exercício. Como nas outras soluções, esse diagrama já se refere ao modelo de domínio, embora um tanto incompleto, uma vez que nesse diagrama só foram declarados os métodos considerados pertinentes ao processo de **Quitar Diárias**.

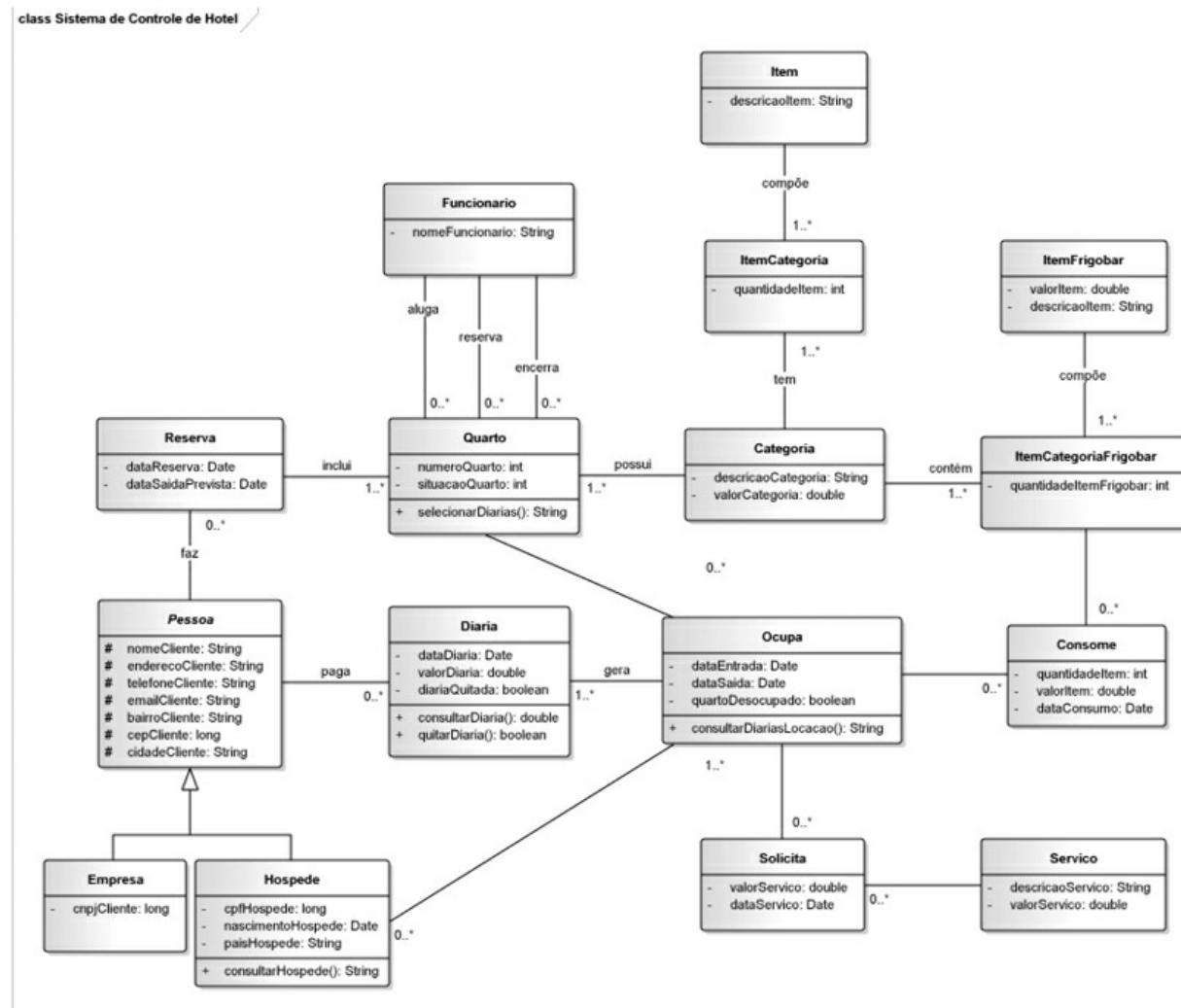


Figura 4.73 – Diagrama de Classes para o Sistema de Controle de Hotelaria – Solução do Exercício.

A seguir, explicaremos as classes de entidade contidas nesse diagrama.

- **Funcionario** – Essa classe armazena as informações dos funcionários

que trabalham ou trabalharam no hotel. Seu único atributo é o nome do funcionário.

- **Quarto** – Aqui, identificamos os quartos alugados pelo hotel. Os atributos dessa classe são o número do quarto e a sua situação, que determina se o quarto está livre (0), ocupado (1) ou reservado (2). Os dois atributos são do tipo **int**. O método **selecionarDiarias** dessa classe é utilizado para selecionar as diárias de um determinado quarto e retorna uma **String** com essas informações. É importante notar que a classe **Funcionario** tem três associações com essa classe, uma vez que um funcionário pode reservar, alugar ou encerrar a estada de um quarto para um hóspede, sendo necessário saber quais funcionários realizaram essas operações. Um quarto pode ser alugado, encerrado ou liberado por um funcionário específico, mas um mesmo funcionário pode realizar muitas dessas operações diversas vezes.
- **Categoria** – A classe em questão contém as informações sobre as categorias de quartos disponíveis no hotel. O tipo de categoria influí no valor do quarto. Seus atributos são a descrição da categoria do tipo **String** e o valor a ser cobrado pelo aluguel de um quarto pertencente a uma categoria, do tipo **double**. Uma categoria pode pertencer a muitos quartos, mas um quarto pode ter somente uma categoria.
- **Item** – Essa classe se refere aos itens que podem estar presentes em um quarto (isso varia de acordo com a categoria). Seu único atributo é a descrição do quarto do tipo **String**.
- **ItemCategoria** – Esta é uma classe intermediária entre as classes **Categoria** e **Item** e identifica os itens contidos nos quartos de uma determinada categoria. Seu único atributo é a quantidade de um determinado item, do tipo **int**, uma vez que essa quantidade pode variar de categoria para categoria. Um objeto dessa classe está associado a um único objeto da classe **Item** e a um único objeto da classe **Categoria**, mas uma categoria pode ter muitos itens e um item pode estar associado a muitas categorias.
- **ItemFrigobar** – Semelhante à classe **Item**, essa classe refere-se aos itens de frigobar que podem estar contidos em um quarto, o que também varia de acordo com a categoria. Seus atributos referem-se à descrição do

item, do tipo **String**, e ao valor do item, do tipo **double**.

- **ItemCategoriaFrigobar** – Já essa classe é semelhante à classe **ItemCategoria**. Refere-se aos itens de frigobar contidos nos quartos de uma determinada categoria. Seu único atributo armazena a quantidade de um determinado item armazenado nos quartos de uma determinada categoria e seu tipo é **int**.
- **Servico** – Essa classe representa os serviços oferecidos pelo hotel. Seus atributos são a descrição do serviço, do tipo **String**, e o valor do serviço, do tipo **double**.
- **Pessoa** – Essa é uma classe abstrata, contendo os atributos comuns às classes **Cliente** e **Hóspede**, especializadas a partir dela. Essas classes serão explicadas a seguir.
- **Empresa** – Essa classe representa os clientes pessoas jurídicas que fazem reservas para determinados hóspedes. Uma empresa não pode ocupar um quarto, mas pode reservar quartos para empregados ou prestadores de serviço, como palestrantes, por exemplo. O único atributo extra dessa classe, além do herdado da classe **Pessoa**, é o CNPJ. Além dos atributos da classe **Pessoa**, a classe cliente também herda as associações para reservar um quarto e pagar diárias. Como um hóspede pode estar a serviço de uma empresa, pode ocorrer de a empresa em questão, e não ele, ser a responsável pela quitação das diárias.
- **Hospede** – O objetivo dessa classe é armazenar as informações de todos os hóspedes que já alugaram, reservaram e/ou ocuparam quartos no hotel (como foi dito na explanação da classe anterior, um quarto não necessariamente é ocupado por quem o aluga). Seus atributos e relacionamentos são herdados da classe Pessoa, exceto os atributos referentes ao CPF, data de nascimento e nacionalidade do hóspede. A classe tem ainda um método para consultar um determinado hóspede, que retorna uma **String** com seus dados.
- **Diaria** – Essa classe armazena as diárias devidas por um hóspede. Seus atributos são a data a que a diária se refere, do tipo **Date**, o valor da diária, do tipo **double**, e o atributo **diariaQuitada**, do tipo **boolean**, que determina se foi quitada (verdadeiro) ou não (falso). A classe tem um método para consultar uma diária, retornando um valor **double**,

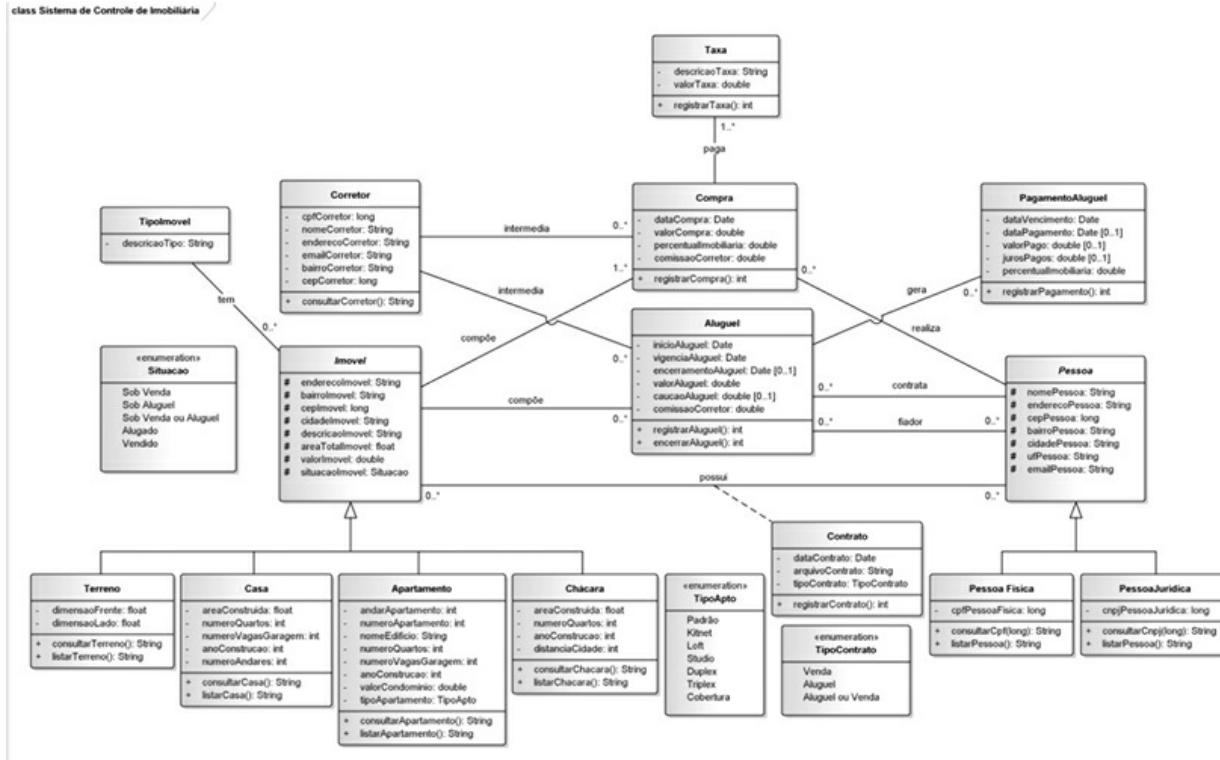
denominado **consultarDiaria**, e um método para quitar uma diária, chamado **quitarDiaria**, que retorna um valor booleano determinando se a operação pôde ser concluída ou não.

- **Ocupa** – Esta classe armazena as informações referentes ao tempo em que um determinado quarto é ocupado por um determinado hóspede. A classe contém os atributos referentes à data em que um hóspede passou a ocupar o quarto e à data em que o hóspede possivelmente desocupou o quarto, ambos do tipo **Date**, além do atributo **quartoDesocupado**, que determina se o quarto encontra-se livre (verdadeiro) ou não (falso), do tipo **boolean**. A classe contém um método denominado **consultarDiariasLocacao**, que retorna uma **String** e é utilizado para retornar os valores das diárias devidas pelo aluguel. Como podemos observar, um hóspede pode ocupar muitos quartos (em épocas diferentes) e um quarto pode ser ocupado por muitos hóspedes.
- **Consome** – Essa classe armazena as informações dos itens de frigobar consumidos por um hóspede na época em que ocupava um determinado quarto. Seus atributos são a quantidade consumida do item, do tipo **int**, seu valor na época, do tipo **double**, e a data em que o item foi consumido, do tipo **Date**. Um hóspede ocupando um quarto pode consumir nenhum ou muitos itens de frigobar, e um item de frigobar pode ser consumido por muitos hóspedes. Contudo, um objeto da classe **Consome** está associado única e exclusivamente a um objeto da classe **Ocupa** e a um objeto da classe **ItemCategoriaFrigobar**.
- **Solicita** – Essa classe identifica os serviços porventura solicitados por um determinado hóspede ocupando um determinado quarto. Seus atributos são o valor do serviço na época em que foi solicitado, do tipo **double**, e a data em que este foi solicitado, do tipo **Date**. Um hóspede em um quarto pode solicitar nenhum ou muitos serviços, e um serviço pode ser solicitado por muitos hóspedes, mas um objeto da classe **Solicita** associa-se exclusivamente a um objeto da classe **Ocupa** e a um objeto da classe **Servico**.

#### 4.19.6 Sistema de Controle de Imobiliária

A figura 4.74 apresenta um exercício sobre o sistema de imobiliária. Como

nas outras soluções, esse diagrama já se refere ao modelo de domínio. Apenas identificamos aqui os métodos considerados mais importantes, principalmente para o processo de venda de um imóvel.



*Figura 4.74 – Diagrama de Classes para o Sistema de Controle de Imobiliária – Solução do Exercício.*

A seguir, explicaremos as classes de entidade contidas nesse diagrama.

- **Imovel** e classes especializadas – Essa é uma classe abstrata que contém os atributos gerais de um imóvel, como endereço, bairro, CEP, cidade, descrição, área total, valor e situação do imóvel. Esse último atributo é do tipo **Situacao**, uma classe de enumeração que contém as possíveis situações porque pode passar um imóvel. Essa classe é especializada em quatro classes concretas que representam os tipos de imóveis com os quais a imobiliária trabalha, que são **Terreno**, **Casa**, **Apartamento** e **Chacara**, cujos atributos acreditamos ser autoexplicativos. A única exceção seria, talvez, o atributo **tipoApartamento**, que pertence ao tipo da classe **TiposApto**, uma classe de enumeração que contém todos os tipos de apartamento que a imobiliária opera. Todas as classes especializadas possuem um método para consultar um objeto específico

e outro para listar os objetos de sua respectiva classe.

- **TipoImovel** – Essa classe armazena os tipos de imóvel com que a imobiliária trabalha. Seu único atributo é a descrição do tipo de imóvel.
- **Corretor** – Essa classe representa os corretores que trabalharam ou já trabalharam para a imobiliária. Seus atributos e métodos são autoexplicativos.
- **Pessoa, Pessoa Física e Pessoa Jurídica** – A classe **Pessoa** é uma classe abstrata que contém os atributos comuns às classes especializadas **PessoaFisica** e **PessoaJurídica**, que, por sua vez, contêm, respectivamente, um atributo para armazenar o CPF e o CNPJ da pessoa. Cada classe especializada contém um método para consultar uma pessoa (por seu CPF ou por seu CNPJ, dependendo da classe) e outro para listar todos os objetos da classe.
- **Contrato** – Essa é uma classe associativa que contém as informações relativas a um contrato estabelecido para a compra ou venda de um imóvel. É produzida pela associação entre a classe **Pessoa** e a classe **Imóvel**, cuja multiplicidade é muitos para muitos. Seus atributos são a data em que o contrato foi redigido, o caminho onde o arquivo do contrato se encontra e o tipo de contrato que se refere a uma classe de enumeração que identifica todos os contratos aceitos pela imobiliária. A classe contém ainda um método para registrar um contrato.
- **Compra** – Essa classe representa as compras de imóveis intermediadas pela imobiliária. Seus atributos são a data em que a compra foi realizada, do tipo **Date**, e o valor da compra, o percentual da imobiliária e a comissão do corretor, do tipo **double**. A classe possui ainda um método para registrar a compra. Observe que essa classe se relaciona com a classe **Corretor**, já que um corretor deve intermediar a compra, com a classe **Imovel**, posto que a compra refere-se a um imóvel, e com a classe **Pessoa**, que representa a pessoa que realiza a compra do imóvel.
- **Taxa** – Essa classe representa as taxas (impostos, taxas de cartório etc.) pagas ao ser realizada uma compra. Seus atributos são descrição e valor da taxa. A classe possui ainda um método para registrar o pagamento de uma taxa.
- **Aluguel** – Essa classe representa os contratos de aluguel de imóvel

intermediados pela imobiliária. Seus atributos são início, vigência e encerramento do aluguel, do tipo **Date**, e valor, caução e comissão do corretor, do tipo **double**. Observe que os atributos encerramento e caução possuem multiplicidade [0..1], o que significa que podem ser deixados vazios, uma vez que a data de encerramento não é determinada, o aluguel possui uma vigência, mas o contrato pode ser rescindido pelo dono do imóvel ou locatário. Além disso, o valor da caução só é exigido se não forem fornecidos fiadores. Os métodos dessa classe permitem registrar um novo contrato de aluguel ou encerrar um contrato já existente. A classe **Aluguel** possui muitas associações e relaciona-se com a classe **Pessoa** por meio de duas associações: a primeira dela refere-se à contratação de um aluguel por uma pessoa e a segunda determina que uma pessoa compromete-se a ser fiador de um aluguel (note que um aluguel pode não ter nenhum fiador, conforme demonstra a multiplicidade). Além disso, a classe **Aluguel** relaciona-se com a classe **Corretor**, pela necessidade de um corretor intermediar o aluguel, com a classe **Imovel**, uma vez que um aluguel refere-se a um imóvel, e com a classe **PagamentoAluguel**, já que é necessário registrar todos os possíveis pagamentos relativos a um determinado aluguel.

- **PagamentoAluguel** – Essa classe representa os pagamentos de um determinado aluguel. Seus atributos referem-se à data do vencimento e à data do pagamento (não são necessariamente os mesmos, podendo haver atrasos), do tipo **Date**, além do valor pago, dos eventuais juros pagos e do percentual da imobiliária, do tipo **double**. Vários desses atributos podem não ser realmente preenchidos, conforme demonstra a sua multiplicidade. A classe possui ainda um método para registrar o pagamento.

## CAPÍTULO 5

# Diagrama de Objetos

O diagrama de objetos tem como objetivo fornecer uma “visão” dos valores armazenados pelos objetos das classes, definidas no diagrama de classes, em um determinado momento do sistema. Assim, embora o diagrama de classes seja estático, podem ser criados diagramas de objetos, onde as possíveis situações pelas quais os objetos das classes passarão podem ser simuladas.

### 5.1 Objeto

Um componente objeto é bastante semelhante a um componente classe, mas os objetos não apresentam métodos, somente atributos, e estes armazenam os valores contidos nos objetos em uma determinada situação. Na verdade, também é possível encontrar diagramas de objetos onde os objetos são representados contendo somente o nome do objeto, sem detalhar os valores de seus atributos.

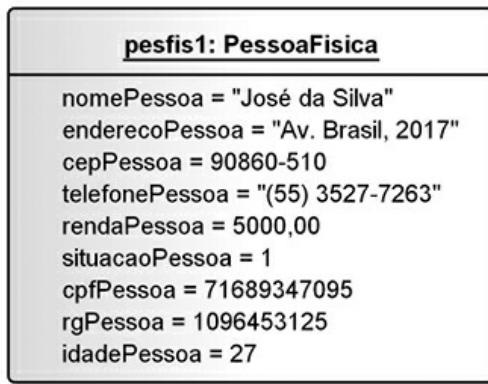
O nome dos objetos está contido, como nas classes, na primeira divisão do retângulo que representa os objetos e pode ser apresentado de três formas:

- O nome do objeto, com todas as letras minúsculas, seguido do símbolo de dois pontos (:) e o nome da classe à qual o objeto pertence, com as letras iniciais maiúsculas. Este é o formato mais completo.
- O nome do objeto omitido, mas mantendo o símbolo de dois-pontos e o nome da classe.
- Somente o nome do objeto, sem dois-pontos.

A figura 5.1 apresenta um exemplo de objeto chamado **pesfis1**, pertencente à classe **PessoaFisica**.

Nesse exemplo, identificamos um objeto individual pertencente à classe **PessoaFisica** do sistema de controle bancário que temos estado a

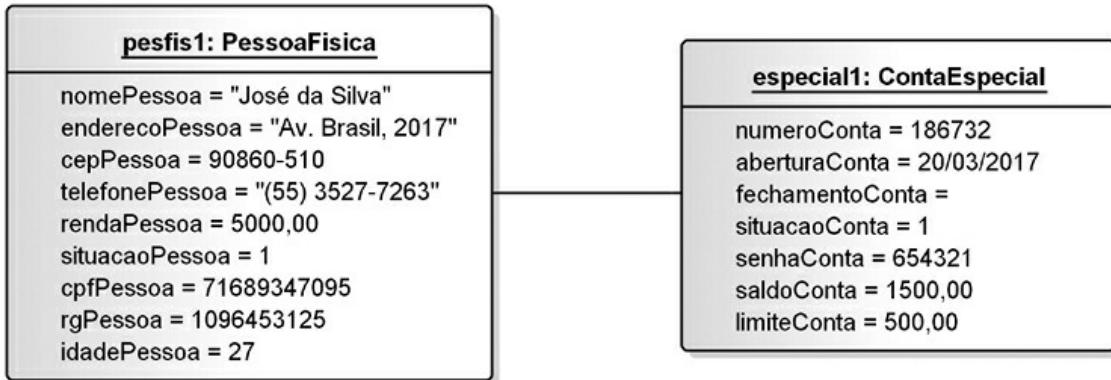
modelar. O leitor poderá perceber que os atributos do objeto em questão foram retirados tanto da classe **Pessoa** como da classe **PessoaFísica**, uma vez que a última classe é derivada da primeira, portanto herda seus atributos. Observe que todos os atributos do objeto possuem valores que foram definidos durante sua instanciação ou ao longo do tempo em que o objeto foi manipulado pelo sistema.



*Figura 5.1 – Exemplo de Objeto.*

## 5.2 Vínculos

Os objetos de um diagrama de objetos apresentam vínculos entre si (links). Tais vínculos nada mais são do que instâncias das associações entre as classes representadas no diagrama de classes, assim como os objetos são instâncias das próprias classes. Um vínculo tem exatamente o mesmo símbolo utilizado pelas associações do diagrama de classes, mas não apresenta multiplicidade porque esta especifica justamente o número de instâncias de uma determinada classe que podem estar envolvidas em uma associação. Assim, um vínculo em um diagrama de objetos liga apenas um único objeto em cada extremidade. A figura 5.2 apresenta um exemplo de vínculo entre objetos.



*Figura 5.2 – Vínculo entre Objetos.*

Nesse exemplo, criamos uma visão por meio do diagrama de objetos referente ao vínculo entre um objeto da classe **PessoaFisica** e um objeto da classe **ContaEspecial** a ele relacionado. Percebemos que o objeto **pesfis1** está ligado ao objeto **esp1**. Observe que os atributos do objeto **ContaEspecial** foram retirados tanto da classe **ContaComum** como da classe **ContaEspecial**, por essa última ser uma especialização da classe **ContaComum**. Observe que a situação da conta está ativa e, por esse motivo, a data de encerramento da conta não foi informada.

### 5.3 Dependência com Estereótipo <<instanceOf>>

É possível representar os objetos instanciados a partir de classes por meio de uma associação de dependência com o estereótipo <<instanceOf>>, como pode ser observado na figura 5.3.

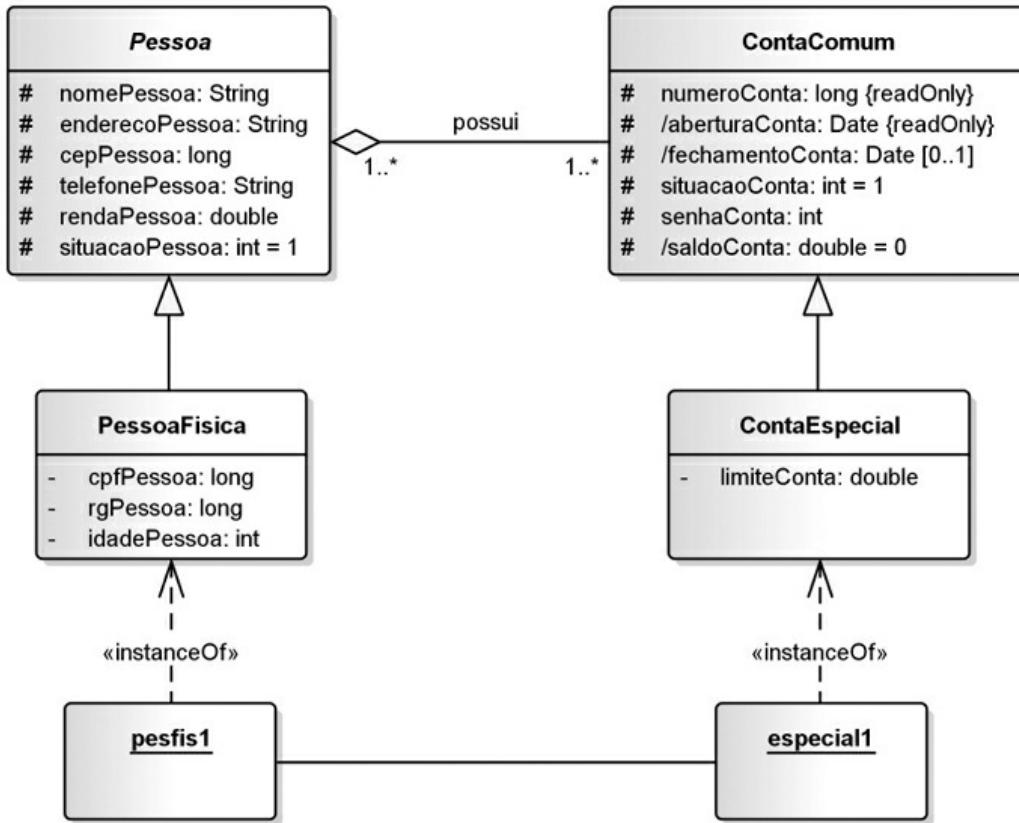


Figura 5.3 – Dependência com Estereótipo <<instanceOf>>.

Nesse exemplo, representamos a instanciação do objeto **pesfis1** a partir da classe **PessoaFisica** e do objeto **especial1** a partir da classe **ContaEspecial**, por meio da associação de dependência apoiada pelo estereótipo <<instanceOf>>. Observe que a associação entre as classes é representada por um vínculo entre os objetos, mas sem multiplicidade, já que, como foi dito, um vínculo une um único objeto a outro.

## 5.4 Exemplo de Diagrama de Objetos

A figura 5.4 apresenta um exemplo de diagrama de objetos, criando uma visão das contas e movimentos de uma pessoa física.

Nesse exemplo, percebe-se que o objeto **pesfis1**, da classe **PessoaFisica**, está vinculado a três objetos: o primeiro da classe **ContaComum**, o segundo da classe **ContaEspecial** e o terceiro da classe **ContaPoupanca**, chamados, respectivamente, de **comum1**, **especial1** e **poupanca1**. Dessa forma, podemos concluir que a pessoa física representada pelo objeto **pesfis1** possui ou possuiu três contas na

instituição bancária. Ao examinarmos o objeto **comum1**, percebemos que essa conta já se encontra encerrada, uma vez que o atributo **situacaoConta** armazena o valor 2, que, por convenção, significa que a conta está inativa, e porque o atributo **fechamentoConta** tem uma data definida. Já os outros dois objetos ainda estão ativos.

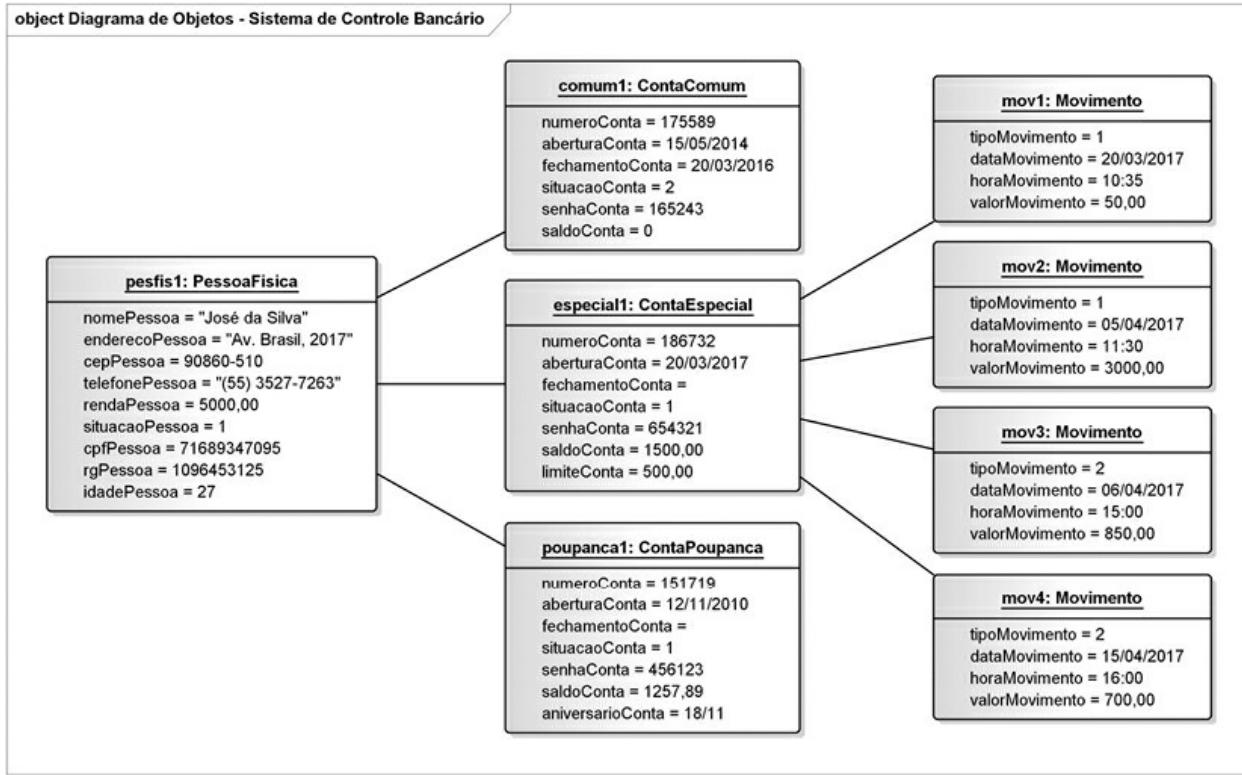


Figura 5.4 – Diagrama de Objetos.

Se continuarmos examinando a figura, perceberemos que o objeto **especial1** está vinculado a quatro objetos da classe **Movimento**, **mov1**, **mov2**, **mov3** e **mov4**. Os dois primeiros objetos referem-se a um depósito de valores, enquanto os dois últimos, a saques.

Como podemos observar, nem todas as classes estão representadas por seus objetos nesse exemplo, o que é, aliás, recomendável. Um diagrama de objetos deve enfocar o menor conjunto possível de classes, porque as classes podem ter um número muito grande de objetos e representar objetos de todas as classes em um diagrama tende a deixá-lo muito extenso e poluído. Assim, o melhor é criar vários diagramas de objetos enfocando pequenas partes do diagrama de classes.

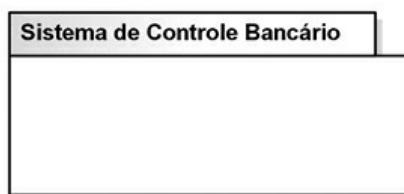
## CAPÍTULO 6

# Diagrama de Pacotes

O diagrama de pacotes descreve como os elementos do modelo estão organizados em divisões lógicas, denominadas pacotes, e demonstra as dependências entre eles. Esse diagrama é muito útil para separar as diversas camadas de um projeto de software, como as de visão, controle, modelo e persistência. Esse diagrama também permite a modelagem de sistemas e/ou subsistemas integrados. Pode ser utilizado para modelar subdivisões da arquitetura de uma linguagem ou representar a arquitetura de um processo de desenvolvimento, entre outras possibilidades.

### 6.1 Pacotes

Pacotes são utilizados para agrupar elementos e fornecer denominações para esses grupos. Um pacote pode representar um sistema, um subsistema, uma biblioteca ou uma etapa de um processo de desenvolvimento, entre outras alternativas. Um pacote pode até conter outros pacotes. A figura 6.1 apresenta um exemplo de pacote.

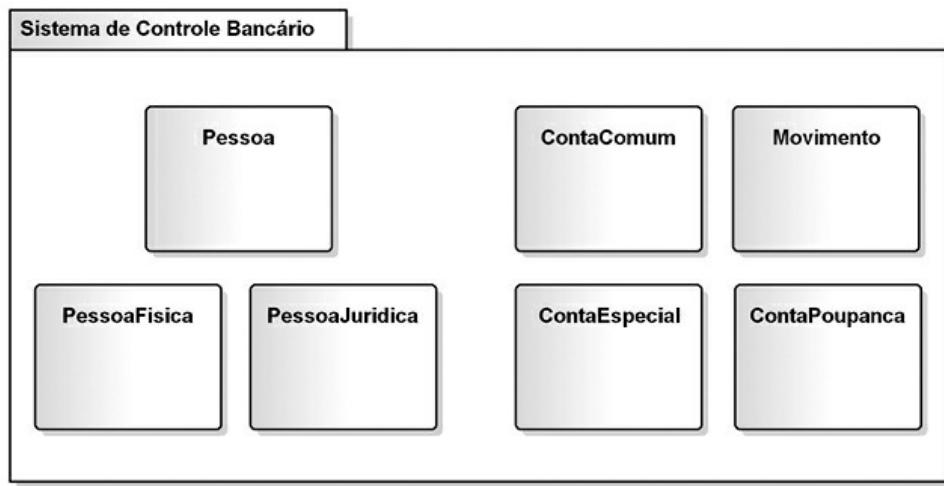


*Figura 6.1 – Exemplo de Pacote.*

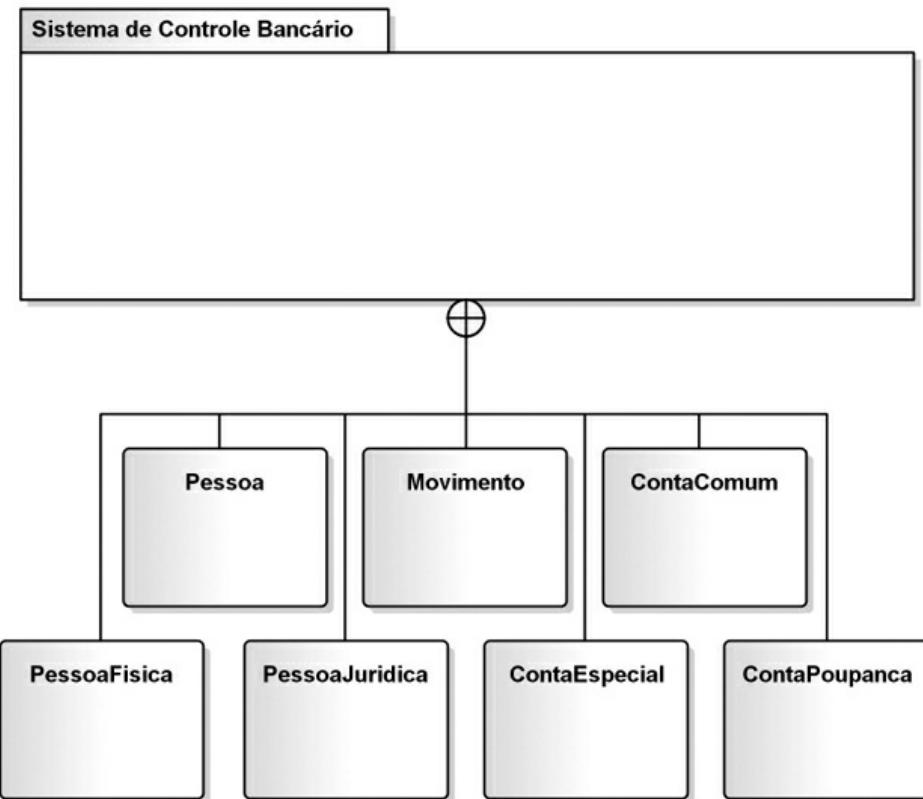
O exemplo apresentado nessa figura contém um pacote intitulado Sistema de Controle Bancário, o que significa que engloba os elementos contidos nesse sistema. Esse exemplo apresenta somente o pacote, sem revelar seu conteúdo. No entanto, é possível encontrar pacotes com seu conteúdo ou parte dele explicitamente declarado, como demonstra a figura 6.2.

Nesse exemplo, identificamos os elementos contidos pelo pacote, em

termos de suas classes. Como o leitor pode notar, não definimos os atributos, métodos nem associações entre as classes. No entanto, isso é perfeitamente possível, podendo um pacote conter um diagrama de outro tipo completo dentro de si. O problema dessa abordagem é o grande espaço ocupado pelo pacote, sendo mais comum encontrar pacotes sem o detalhamento de seu conteúdo. Existe uma notação alternativa para identificar os membros do pacote por meio do conector de aninhamento, representado por um círculo contendo uma cruz, conforme demonstra a figura 6.3.



*Figura 6.2 – Pacote com Detalhe de seus Membros.*



*Figura 6.3 – Pacote com Detalhe de seus Membros – Notação Alternativa com Conector de Aninhamento.*

Também é possível encontrar modelos completos contidos em pacotes, conforme demonstrado a seguir, que demonstra um pacote contendo o modelo de domínio do sistema de controle bancário, incluindo atributos, métodos e associações entre as classes (Figura 6.4).

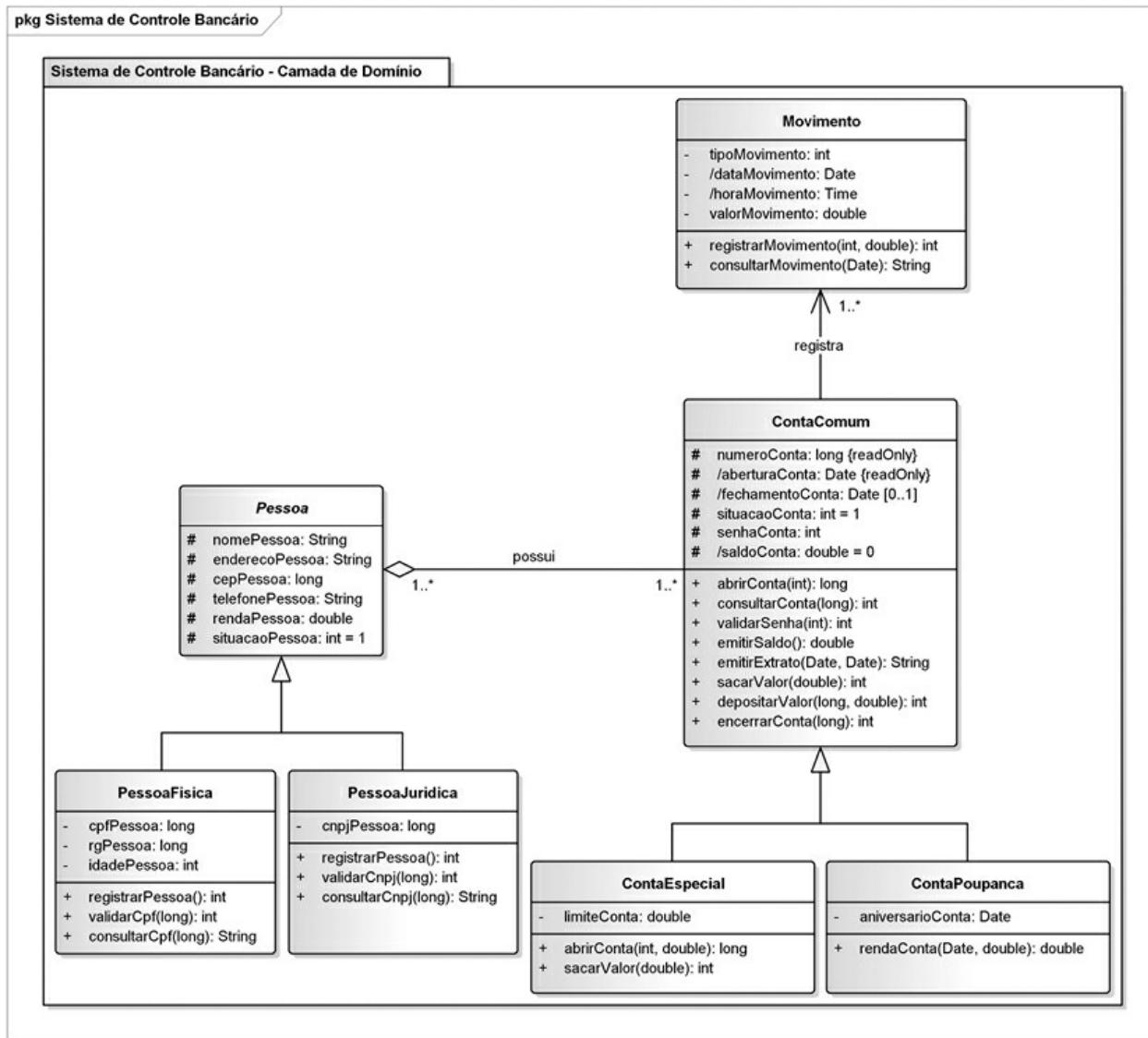


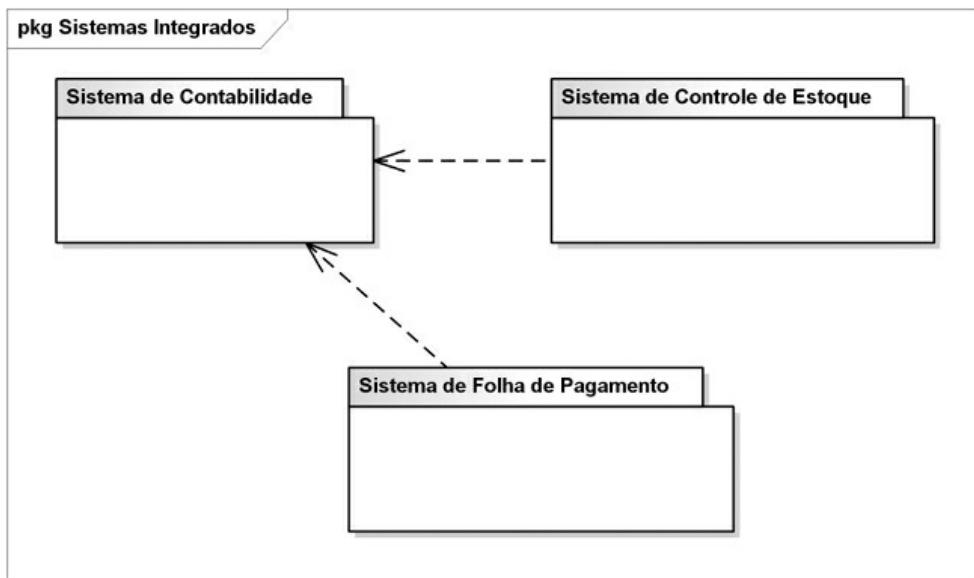
Figura 6.4 – Pacote Contendo um Modelo Completo

Essa alternativa, embora perfeitamente válida, ocupa muito espaço. Assim, em algumas situações, pode ser mais útil apresentar somente o pacote sem detalhar seus membros. Algumas ferramentas CASE podem apresentar diagrama de pacotes, em que cada pacote apresenta uma lista resumida de seus membros, conforme será apresentado nos próximos exemplos.

## 6.2 Dependência

Pacotes muitas vezes contêm dependências entre si. Um relacionamento de dependência informa que o elemento dependente necessita de alguma forma do elemento do qual depende, ou seja, algum elemento interno de

um dos pacotes depende de alguma maneira de elementos do outro pacote. A figura 6.5 apresenta um exemplo de relacionamento de dependência entre pacotes.



*Figura 6.5 – Dependência entre Pacotes.*

Nesse exemplo, existem três sistemas integrados: o sistema de contabilidade, de estoque e de folha de pagamento. Os pacotes estão associados por meio de dependências, indicando que os sistemas de estoque e de folha de pagamento necessitam do sistema de contabilidade para lançar suas operações financeiras. O relacionamento de dependência no diagrama de pacotes pode ter dois estereótipos: <<merge>>, significando que os elementos do pacote que utiliza essa dependência serão unidos aos elementos do outro pacote, e <<import>>, significando que o pacote que utiliza essa dependência está importando alguma característica ou elemento do outro pacote.

### 6.3 Pacotes Contendo Pacotes

É possível que pacotes contenham pacotes. Um exemplo disso é apresentado na figura 6.6.

Nesse exemplo, percebemos que o pacote Repositório foi inserido dentro do pacote que representa a camada de domínio do sistema de controle bancário. Conforme foi explicado no capítulo 4, classes de repositório (repository) são representadas na camada de domínio, no entanto, muitas

vezes, em um pacote interno. Observe que há uma classe de repositório para cada classe concreta da camada de domínio.

## 6.4 Estereótipos Aplicados a Pacotes

É possível aplicar estereótipos aos pacotes, deixando claro que estes representam sistemas, subsistemas, frameworks ou modelos, por exemplo, conforme apresentado na figura 6.7.

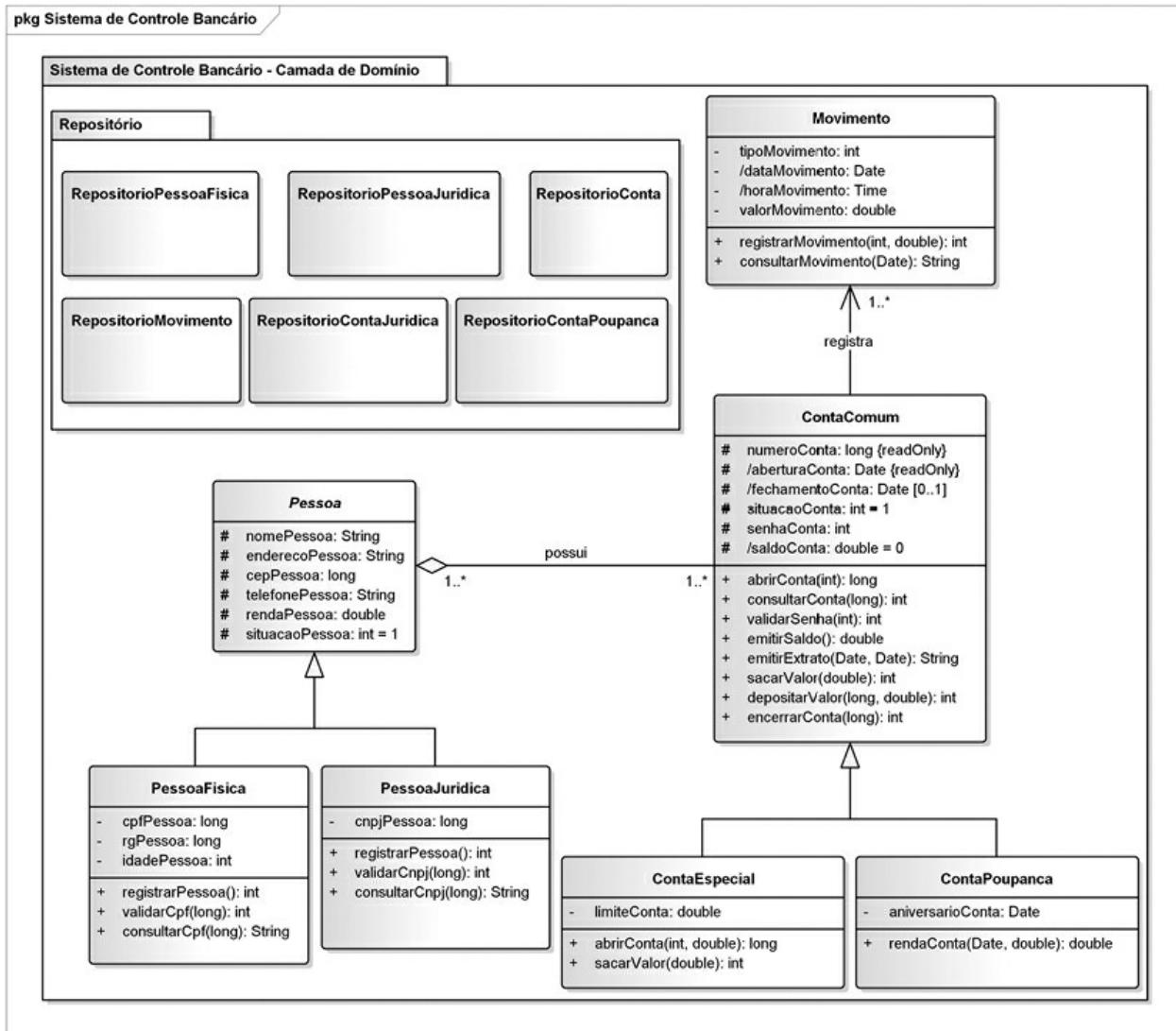
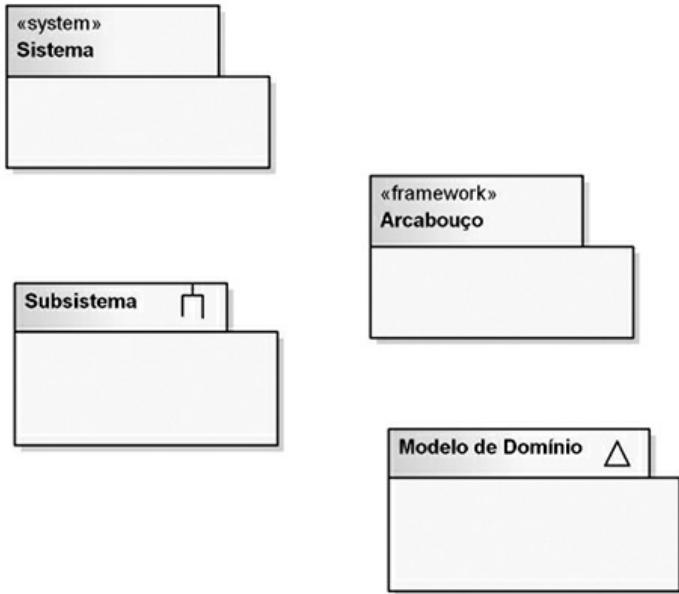


Figura 6.6 – Pacotes Contendo Pacotes.



*Figura 6.7 – Pacotes com Estereótipos.*

Nessa figura, apresentamos pacotes com estereótipos dos quatro tipos enunciados. Observe que enquanto os estereótipos <<system>> e <<framework>> são estereótipos de texto, <<subsystem>> e <<model>> são estereótipos gráficos que modificam um pouco o desenho-padrão do pacote.

## 6.5 Representação de Camadas do Modelo por Meio de Pacotes

Pacotes frequentemente são utilizadas para representar as camadas de uma arquitetura ou as camadas lógicas do modelo geral de um projeto. A figura 6.8 apresenta um exemplo das camadas do sistema de controle bancário por meio de pacotes.

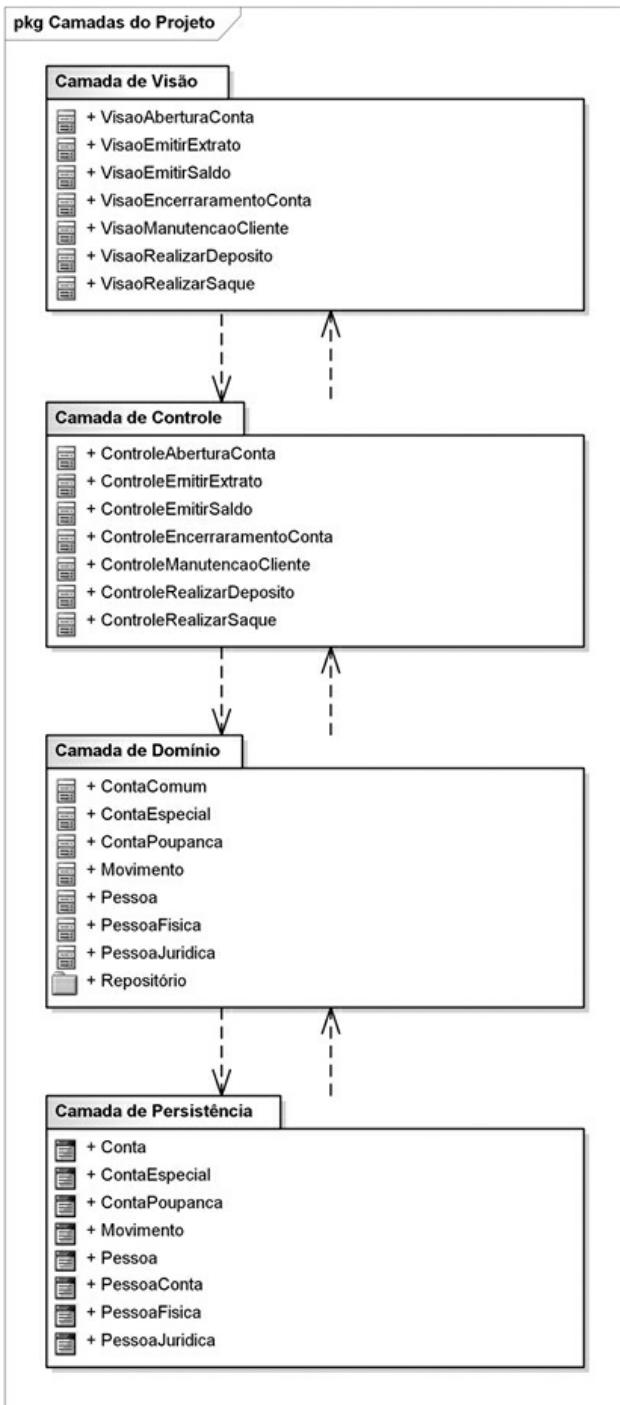


Figura 6.8 – Pacotes Representando Camadas do Modelo.

Nesse exemplo, cada pacote representa uma camada contendo um dos modelos de classes do sistema de controle bancário, ou seja, as camadas de apresentação, controle, domínio e persistência de dados. Assim, o pacote da camada de visão contém as classes de fronteira do sistema, o pacote da camada de controle, as classes controladoras, o pacote da camada de

domínio, as classes de entidade, e o pacote da camada de persistência, o mapeamento das classes de entidade em tabelas (ou classes DAO).

Observe que os pacotes não estão expandidos, ou seja, cada pacote apresenta seu conteúdo em forma de lista, sem detalhar o conteúdo das classes nele contidas. Essa abordagem é muito usada para diminuir o tamanho dos diagramas, posto que representar cada pacote expandido ocuparia um espaço substancial. Observe ainda que o pacote que representa a camada de modelo, além das classes de entidade, contém também um pacote interno que armazena as classes de repositório. Finalmente, é importante destacar que os pacotes possuem dependências entre si, conforme demonstram as associações de dependência empregadas nesse exemplo.

## CAPÍTULO 7

# Diagrama de Sequência

Este é um diagrama comportamental que procura determinar a sequência de eventos que ocorrem em um determinado processo, identificando quais mensagens devem ser disparadas entre os elementos envolvidos e em que ordem. Assim, determinar a ordem em que os eventos ocorrem, as mensagens que são enviadas, os métodos que são chamados e como os objetos interagem dentro de um determinado processo é o objetivo principal desse diagrama.

O diagrama de sequência baseia-se no diagrama de casos de uso, havendo normalmente um diagrama de sequência para cada caso de uso declarado, já que um caso de uso, em geral, refere-se a um processo disparado por um ator. Assim, um diagrama de sequência também permite documentar um caso de uso específico e muitas ferramentas CASE permitem gerar um diagrama de sequência diretamente a partir de um caso de uso.

Obviamente, o diagrama de sequência depende também do diagrama de classes, uma vez que as classes dos objetos utilizados no diagrama de sequência estão descritas nele. No entanto, o diagrama de sequência é uma excelente forma de validar e complementar o diagrama de classes, pois é ao modelar um diagrama de sequência que se percebe quais métodos são necessários declarar em que classes. Isto é o que se recomenda fazer em alguns processos de desenvolvimento de software, como o Processo Unificado, no qual, como foi dito anteriormente, primeiramente se produz o modelo conceitual, durante a fase de análise, e só mais tarde, durante a fase de projeto, produz-se o modelo de domínio, em que serão detalhados os métodos das classes. A descoberta desses métodos é feita por meio do detalhamento dos processos enunciados no diagrama de casos de uso, por meio de diagramas de interação, como os de sequência.

### 7.1 Atores

Os atores modelados neste diagrama são instâncias dos atores declarados no diagrama de casos de uso, representando entidades externas que interagem com o sistema e solicitam serviços, gerando, assim, eventos que iniciam processos. Esses atores costumam ser apresentados como bonecos magros idênticos aos usados no diagrama de casos de uso, porém contendo uma linha de vida. O conceito de linha de vida será explicado nas seções seguintes. A figura 7.1 ilustra um exemplo de como é representado um ator no diagrama de sequência.



*Figura 7.1 – Exemplo de Ator.*

A figura 7.1 representa um cliente que interage com o sistema ou outro ator envolvido no processo. Os atores não são realmente obrigatórios nesse diagrama, mas são utilizados com muita frequência. Além disso, como a maioria dos diagramas de sequência, senão todos, refletem o aspecto dinâmico de um caso de uso. A utilização dos mesmos atores que interagem com o caso de uso em questão facilita a compreensão do processo.

## 7.2 Lifelines

Uma lifeline é um participante individual em uma interação que existe durante um determinado período de tempo (que pode ser o tempo total da interação). Na maioria das vezes, uma lifeline vai se referir a uma instância de uma classe que participa da interação. Assim, lifelines no diagrama de sequência têm a mesma notação utilizada no diagrama de objetos, diferenciando-se por possuírem uma linha vertical tracejada abaixo do retângulo do objeto, que representa o tempo em que este existe na interação, ou seja, sua linha de vida.

Um objeto não precisa necessariamente existir quando o processo é iniciado, podendo ser criado ao longo dele. Dessa forma, os objetos criados durante uma interação não são representados no topo do diagrama, mas só

a partir do momento em que forem criados e obviamente só terão uma linha de vida a partir desse mesmo momento. Quando um objeto é destruído, a sua linha de vida é interrompida com um “X”, significando que o objeto não existe mais no processo. Como na prática, em geral, uma lifeline é um objeto, utilizaremos o termo objeto com frequência ao longo do capítulo. A figura 7.2 apresenta um exemplo de lifeline no diagrama de sequência.

Como é possível perceber ao observarmos a figura, existe uma lifeline chamada **pesfis1**, a qual é uma instância da classe **PessoaFísica**. A linha tracejada vertical que surge a partir do objeto representa o tempo em que este existe na interação.

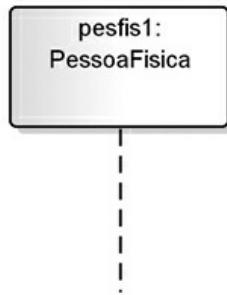


Figura 7.2 – Exemplo de Lifeline.

### 7.3 Mensagens ou Estímulos

As mensagens são utilizadas para demonstrar a ocorrência de eventos, que normalmente forçam a chamada de um método em algum dos objetos envolvidos no processo. Pode ocorrer, no entanto, de uma mensagem representar a comunicação entre dois atores, nesse caso, não disparando métodos. Em geral, um diagrama de sequência é iniciado por um evento externo, causado por algum ator, o que acarreta o disparo de um método em um dos objetos.

As mensagens podem ser disparadas entre:

- um ator e outro ator;
- um ator e uma lifeline (objeto), onde um ator produz um evento que dispara um método em uma lifeline;
- uma lifeline e outra lifeline, o que constitui a ocorrência mais comum de

mensagens, em que uma lifeline transmite uma mensagem para outra, em geral solicitando a execução de um método. Uma lifeline pode até enviar uma mensagem para si mesma, o que é conhecido como autochamada;

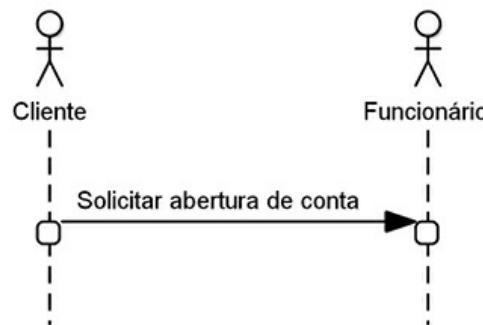
- uma lifeline é um ator, o que normalmente ocorre quando uma lifeline envia uma mensagem de retorno em resposta à chamada de um método solicitado, contendo seus resultados.

As mensagens são representadas por linhas entre dois participantes da interação, contendo setas indicando qual participante enviou a mensagem e qual a recebeu. As mensagens são apresentadas normalmente na posição horizontal entre as linhas de vida dos participantes e sua ordem sequencial é demonstrada de cima para baixo.

Os textos contidos nas mensagens primeiramente identificam qual evento ocorreu e forçou o envio da mensagem e qual método foi chamado. As duas informações são separadas por um símbolo de dois-pontos (:). Podem ocorrer eventos que não disparam métodos. Nesse caso, a mensagem descreve apenas o evento que ocorreu, sem o símbolo de dois-pontos e nenhum texto após estes. Também pode acontecer de somente o método chamado ser descrito, sem detalhar qual evento o causou.

### 7.3.1 Mensagens entre Atores

Esse tipo de mensagem é opcional e sua função basicamente é a de demonstrar interação entre os atores externos que participam de um processo. Isso pode ser útil em algumas interações para tornar sua compreensão mais clara e completa. A figura 7.3 apresenta um exemplo de mensagem disparada entre atores, representando uma conversação entre estes e, portanto, não gera o disparo de nenhum método.

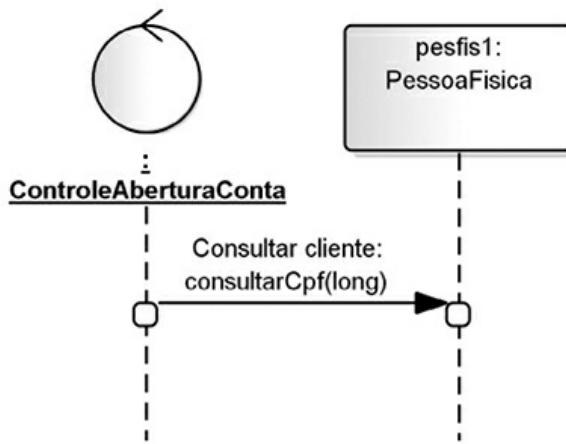


*Figura 7.3 – Mensagem simples entre atores.*

Neste exemplo, percebe-se que a mensagem descreve simplesmente o evento em si, que representa a solicitação de um cliente para abertura de uma conta a um funcionário do banco.

### 7.3.2 Mensagens entre Lifelines

Mensagens entre lifelines são a ocorrência mais comum de troca de mensagens em um diagrama de sequência. Em geral, essas mensagens acarretam a execução de um método. A figura 7.4 mostra um exemplo de uma mensagem enviada por uma lifeline, que dispara um método para consultar um cliente por seu CPF (**consultarCpf**) em outra lifeline.



*Figura 7.4 – Mensagem com Disparo de Método entre Lifelines.*

Nessa figura, podemos perceber que existem duas lifelines representando uma instância da classe controladora **ControleAberturaConta** e uma instância da classe de entidade **ContaComum**. Pode-se perceber que a primeira é uma instância de uma classe controladora pelo símbolo associado ao estereótipo **<<control>>** aplicado à lifeline, todavia, nesse exemplo, optamos por não atribuir o estereótipo **<<entity>>** à outra lifeline apenas para mantermos a notação-padrão desse tipo de objeto. Nada impediria, porém, que utilizássemos o estereótipo de entidade aqui. Observe que somente a instância da classe **ContaComum** recebeu um nome (**pesfis1**). A instância da classe controladora não possui um nome específico, onde somente o nome da classe a qual ela pertence é descrita após os dois-pontos. Na verdade, o nome da lifeline é opcional e, na

maioria das vezes, omitido, sendo somente o nome da classe obrigatório.

Ao examinarmos a mensagem apresentada na figura, podemos perceber que esta descreve o evento que causou seu disparo e, após os dois-pontos, o método que foi disparado por ele. Somente a informação do método disparado é obrigatória e o texto que antecede a descrição do método é meramente ilustrativo. Logicamente, tais métodos podem conter parâmetros e retornar valores. No entanto, deve-se evitar colocar muitos detalhes nas chamadas dos métodos para impedir que o diagrama de sequência torne-se muito extenso. Nessa situação, contudo, foi considerado válido inserir uma justificativa para o disparo da mensagem (a consulta de um cliente) e o detalhamento do tipo de parâmetro que o método para consultar o CPF do cliente (**consultarCpf**) recebe (**long**).

Há, ainda, outra informação importante nessa figura que deve ser explicada. O leitor deve ter percebido que quando uma mensagem é disparada ou recebida, a linha de vida dos elementos envolvidos se torna mais grossa. Isto é chamado Foco de Controle ou Ativação e determina o momento em que um elemento da interação está participando ativamente do processo, em geral, disparando ou recebendo uma mensagem e executando algum método.

### 7.3.3 Mensagens de Retorno

Esse tipo de mensagem identifica a resposta a uma mensagem disparada por uma lifeline (objeto) ou um ator. Uma mensagem de retorno pode retornar informações específicas do método chamado ou apenas um valor indicando se o método foi executado com sucesso ou não. As mensagens de retorno são representadas por uma linha tracejada contendo uma seta fina que aponta para o elemento que recebe o resultado do método chamado. A figura 7.5 apresenta um exemplo de mensagem de retorno.

Aqui, a mensagem de retorno é emitida pela lifeline **pesfis1** para a lifeline da classe **ControleAberturaConta**, em resposta a esta ter disparado o método **consultarCpf** na primeira lifeline. Essa mensagem retorna os dados do cliente consultado, ou seja, o resultado da execução do método, do tipo **String**. A rigor, deveria ser disparado um método para retornar cada atributo da pessoa consultada, no entanto isso tornaria o diagrama extenso demais e, além de ser repetitivo, não acrescentaria informações

realmente pertinentes à interação. Assim, preferimos representar um método genérico de consulta ao cliente.

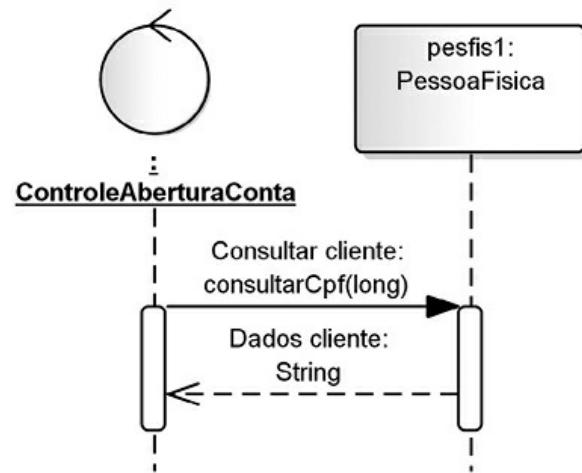
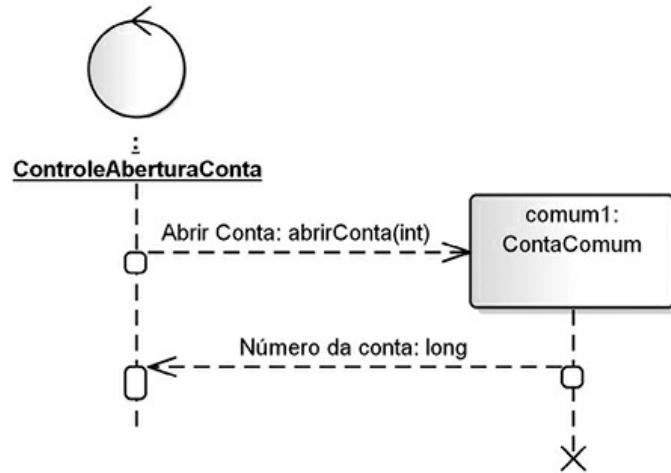


Figura 7.5 – Mensagem de Retorno.

Observe que o retorno da mensagem contém um texto explicativo seguido de dois-pontos (:). Na verdade, esse texto é apenas ilustrativo, não sendo realmente obrigatório. Alguns autores detalham somente o texto nas mensagens de retorno, outros, apenas os valores retornados, enquanto outros informam ambos. Muitos autores recomendam que somente as mensagens de retorno consideradas realmente importantes sejam modeladas para evitar deixar o diagrama muito extenso e poluído.

### 7.3.4 Mensagens Construtoras

Uma lifeline pode existir desde o início do processo ou ser criada durante o decorrer da execução deste. Quando uma mensagem é dirigida a uma lifeline (objeto) já existente, a seta da mensagem atinge a linha de vida da lifeline, engrossando-a, identificando que o foco de controle está sobre o objeto em questão. No entanto, quando a mensagem cria uma nova lifeline, a seta atinge o retângulo que representa a lifeline, indicando que a mensagem representa um método construtor e que a lifeline passa a existir somente a partir daquele momento. A figura 7.6 apresenta um exemplo de mensagem que provoca a criação de uma nova lifeline.



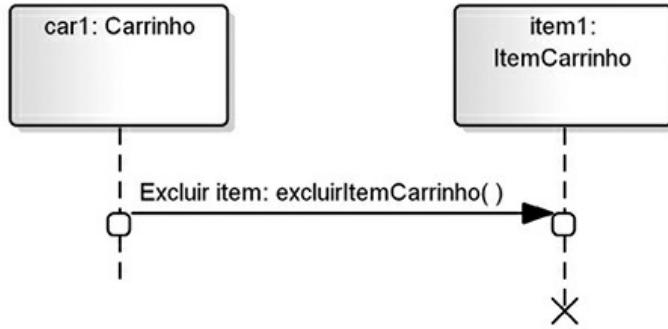
*Figura 7.6 – Exemplo de Mensagem Construtora.*

Ao estudarmos a figura, verificamos que ela representa duas lifelines. A lifeline da classe **ControleAberturaConta** dispara o método **abrirConta** na lifeline **comum1** da classe **ContaComum**, instanciando esse objeto a partir desse momento.

Observe que a lifeline que representa a instância da classe de controle do processo de abertura de conta esteve ativa desde o início do processo, enquanto o objeto (lifeline) **comum1** foi instanciado ao longo do processo por uma mensagem enviada pela lifeline da controladora de abertura de conta. Observe ainda que a lifeline **comum1** não está na mesma altura da lifeline controladora. Isto ocorre para demonstrar o momento em que a lifeline passa a existir no processo. O leitor notará ainda que a figura apresenta também uma mensagem de retorno contendo o número da conta gerada pelo método de abertura de conta.

### 7.3.5 Mensagens Destruitoras

Uma mensagem pode também representar um método destrutor, ou seja, um método que elimina uma lifeline (objeto) não mais necessária à interação. Nesse caso, a mensagem atinge a linha de vida de um objeto e a interrompe com um “X”. A figura 7.7 apresenta um exemplo de chamada de método destrutor.

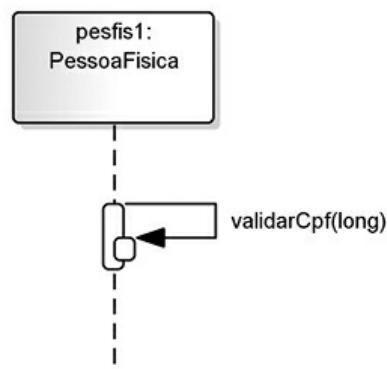


*Figura 7.7 – Mensagem que dispara um método Destruitor.*

Nesse exemplo, existe uma lifeline **car1** pertencente a uma classe **Carrinho**, que representa um carrinho de compras, como os encontrados nas livrarias digitais da internet, e pode conter muitos itens, representados pelas lifelines da classe **ItemCarrinho**. Se em algum momento o cliente resolver cancelar a compra de algum dos itens do carrinho, a lifeline da classe **Carrinho** deverá disparar um método destrutor em uma lifeline da classe **ItemCarrinho**, aqui representado pelo método **excluirItemCarrinho**.

### 7.3.6 Autochamadas ou Autodelegações

Autochamadas são mensagens que uma lifeline envia para si mesma. Nesse tipo de situação, uma mensagem parte da linha de vida do objeto e atinge a linha de vida do próprio objeto. A figura 7.8 demonstra um exemplo de autochamada.



*Figura 7.8 – Autochamada.*

Nesta ilustração, a lifeline (objeto) **pesfis1** dispara em si mesma a

chamada ao método de validação de CPF, `validarCpf`, que recebe como parâmetro um valor do tipo `long` que corresponde ao número do CPF a ser validado.

### 7.3.7 Mensagens Assíncronas

Os exemplos de mensagens apresentadas anteriormente referiam-se a mensagens síncronas, que são mais comumente utilizadas nos diagramas de sequência. Basicamente, lifelines que enviam uma mensagem síncrona devem esperar pelo retorno da mensagem para continuar com o processamento, mas isso não ocorre com mensagens assíncronas. A lifeline pode executar outras ações enquanto espera o retorno da mensagem. Além disso, o elemento que recebe uma mensagem assíncrona não necessariamente precisa atendê-la imediatamente. As mensagens assíncronas diferenciam-se das síncronas por suas setas não serem preenchidas. A figura 7.9 apresenta um exemplo de mensagem assíncrona.

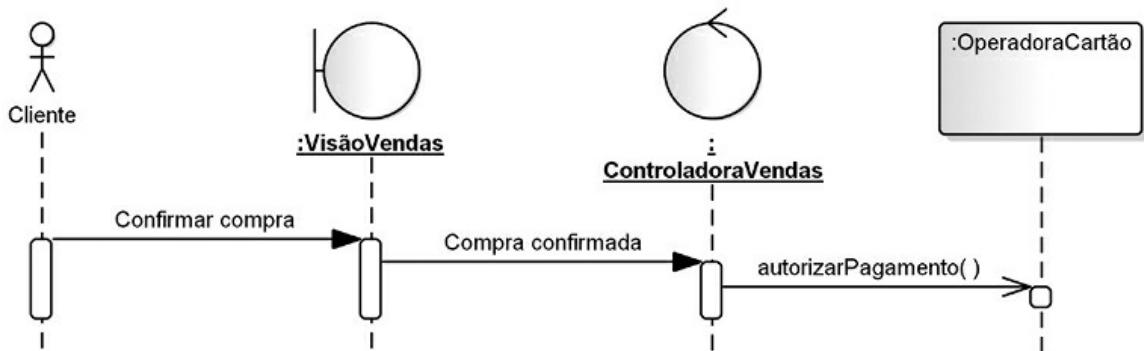


Figura 7.9 – Mensagem Assíncrona.

Este exemplo ilustra uma situação em que um cliente confirma uma compra após ter informado o número de seu cartão de crédito em um sistema de vendas pela internet. Ao receber a confirmação, a interface (uma lifeline do tipo `boundary`) repassa o evento para uma lifeline de controle que, em resposta, envia uma mensagem síncrona para a operadora do cartão para verificar se esta autoriza a compra. Podemos perceber que essa última é uma mensagem síncrona, já que a seta da mensagem é aberta e não preenchida.

### 7.3.8 Restrição de Duração

Às vezes, pode ser necessário estabelecer detalhes de tempo para uma mensagem, como o tempo máximo de espera até que uma mensagem seja disparada. Quando se quer demonstrar o tempo que uma mensagem leva em consideração antes de ser disparada, deve-se usar Restrições de Duração, e a mensagem, em vez de ser representada na horizontal, como é o padrão, é apresentada na diagonal, como demonstra o exemplo da figura 7.10.

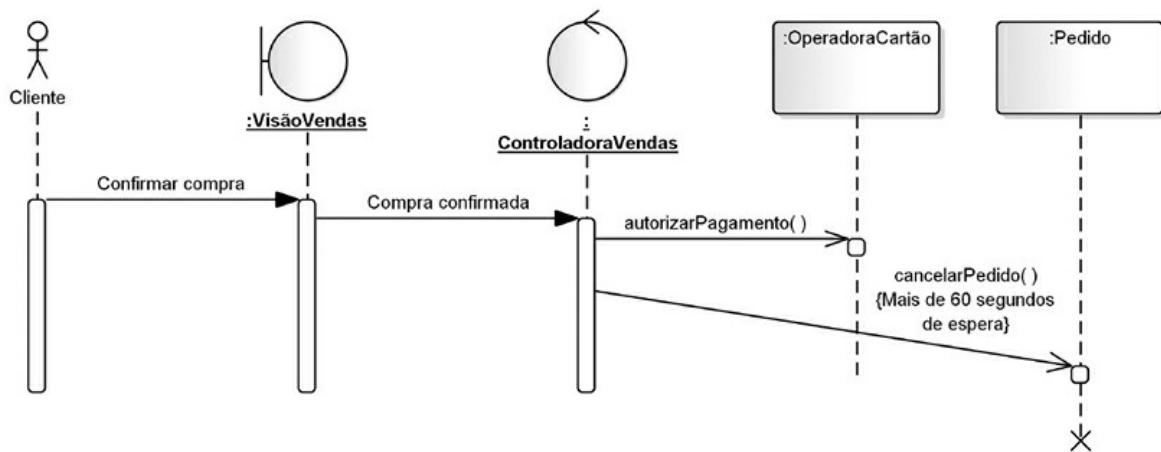
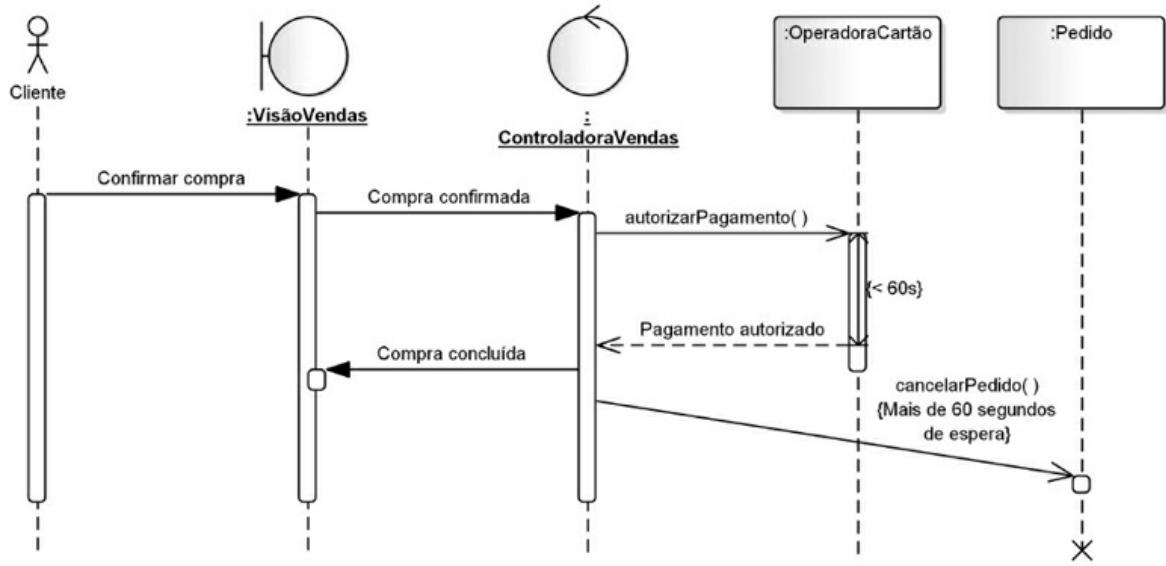


Figura 7.10 – Restrição de Duração.

Aqui damos continuidade ao exemplo anterior, em que, se após 60 segundos de espera o controlador não obtiver resposta da operadora do cartão, o pedido de compra será cancelado e o cliente informado que seu pagamento não foi confirmado pela operadora. Observe que a mensagem que dispara o método que cancelará o pedido tem uma restrição de duração que determina que se devem esperar até 60 segundos antes de disparar o método **cancelarPedido()**, que é um método destrutor, conforme demonstra o “X” ao final da lifeline da classe **Pedido**.

A figura 7.11 apresenta o mesmo exemplo com mais detalhes de restrição de duração, em que definimos que se a autorização de pagamento for recebida dentro de 60 segundos, a compra será autorizada. Observe que o foco de controle da lifeline da classe OperadoraCartão apresenta uma seta dupla e, ao lado, uma restrição contendo o texto “{< 60s}” que estabelece o tempo máximo de espera para autorizar a compra. Isto é chamado observação de duração.



*Figura 7.11 – Detalhes de restrição de duração.*

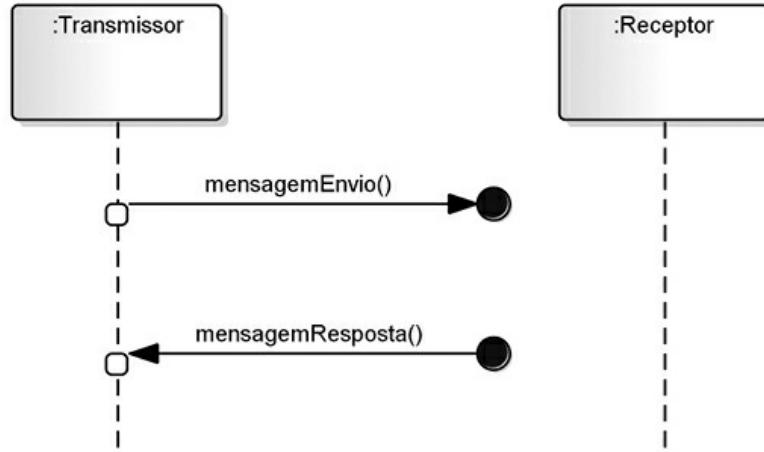
### 7.3.9 Mensagens Perdidas e Mensagens Encontradas

Uma mensagem perdida representa uma mensagem que foi enviada e sua confirmação de recebimento não foi recebida, podendo significar que a mensagem não chegou ao seu destino, ou uma mensagem enviada a um destino não representado no diagrama. Já uma mensagem encontrada representa o recebimento de uma mensagem enviada por um elemento desconhecido ou um elemento não representado no diagrama, ou o recebimento de uma mensagem que fora dada como perdida, pois seu tempo de espera por resposta poderia ter sido encerrado.

Tanto as mensagens perdidas como as mensagens encontradas são representadas por um círculo preenchido. Quando se trata de uma mensagem perdida, o círculo é atingido pela mensagem; já quando se trata de uma mensagem encontrada, esta parte do círculo. Uma aplicação para o uso de mensagens perdidas e mensagens encontradas pode ser a representação de troca de mensagens entre objetos localizados em máquinas (hosts) diferentes, possivelmente distantes entre si, onde a comunicação é realizada por meio de algum tipo de protocolo de rede. A figura 7.12 apresenta um exemplo de mensagem perdida e mensagem encontrada.

Aqui, apresentamos um exemplo de mensagem perdida e outro de mensagem encontrada, em que uma mensagem foi enviada pelo objeto

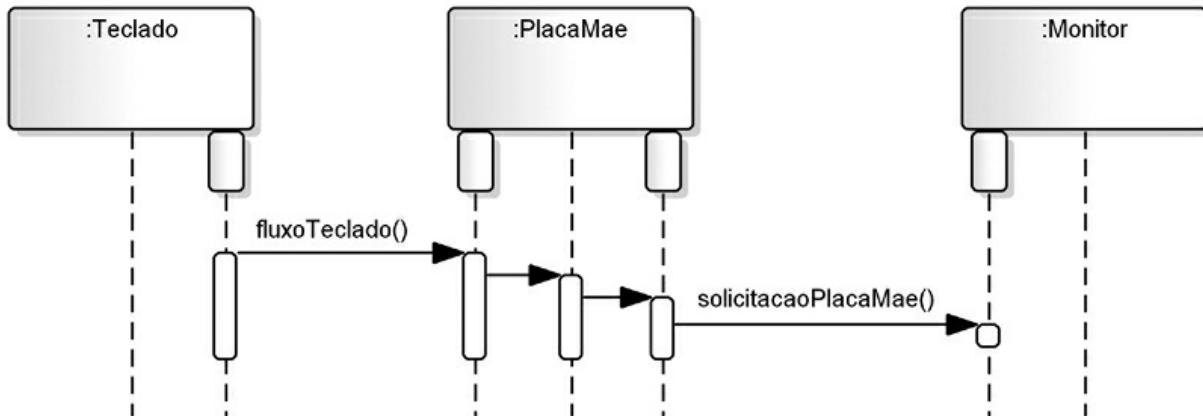
transmissor e, aparentemente, não foi recebida pelo objeto receptor. Após um tempo maior que o tempo máximo de espera, a mensagem de resposta foi recebida, ou seja, “encontrada”.



*Figura 7.12 – Exemplo de Mensagem Perdida e Mensagem Encontrada.*

## 7.4 Portas

O conceito de portas foi explicado no diagrama de classes. É possível representar um objeto (lifeline) no diagrama de sequência contendo instâncias das portas declaradas na classe a que ele pertence. Dessa forma, o objeto poderá ter mais de uma linha de vida, o que permite a representação de mensagens externas e internas no objeto. A figura 7.13 apresenta um exemplo de instâncias de portas em objetos e o envio e recebimento de mensagens por elas.

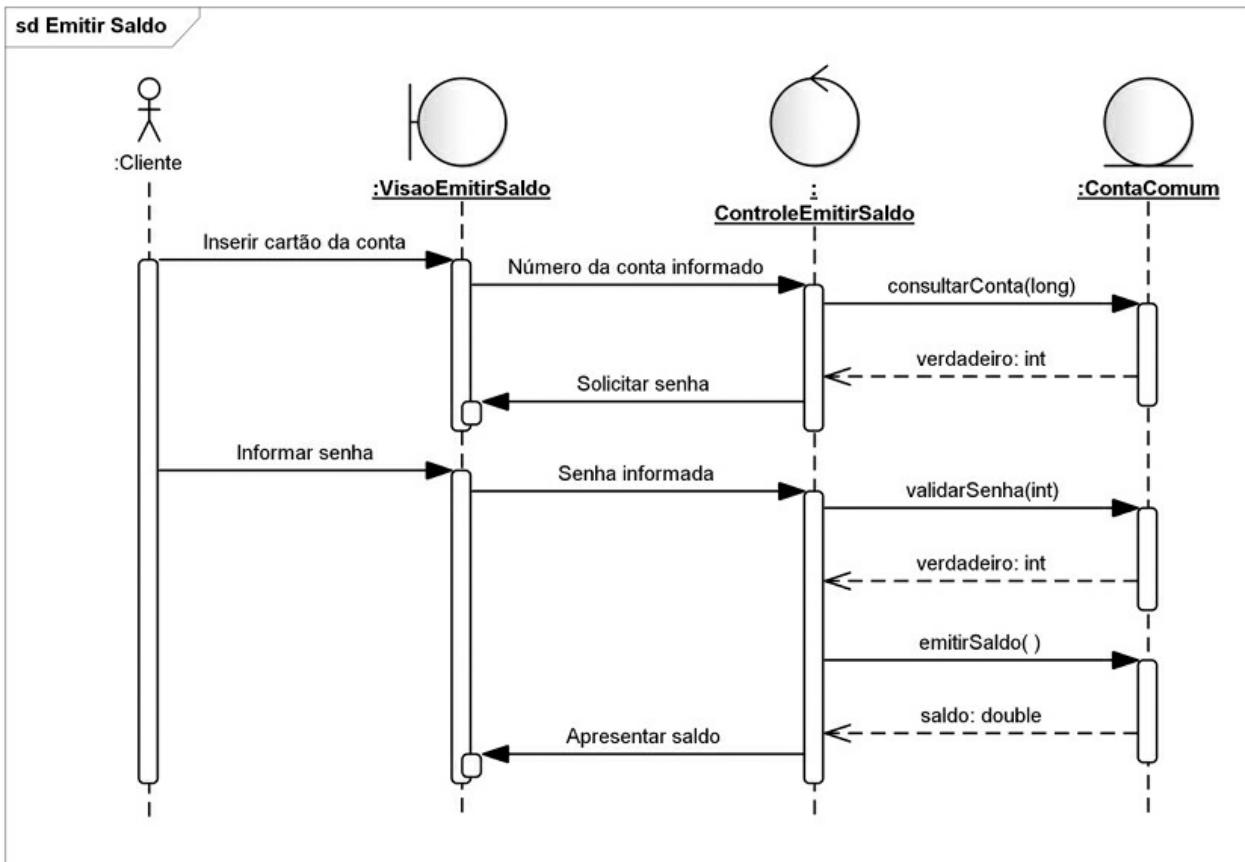


*Figura 7.13 – Portas.*

Nesse exemplo, instanciamos objetos das classes **Teclado**, **PlacaMae** e **Monitor**. Se o leitor voltar ao capítulo 4, perceberá que essas classes se comunicam por meio de interfaces e portas. Aqui, representamos o envio de fluxo de informações do teclado para a placa-mãe e, depois de estas terem sido processadas, a solicitação da placa-mãe ao monitor para que determinadas informações sejam apresentadas. Observe que as portas são representadas por retângulos abaixo do objeto, tendo suas próprias linhas de vida.

## 7.5 Fragmentos de Interação

Os fragmentos de interação são noções abstratas de unidades de interação geral. Um fragmento de interação é uma parte de uma interação, no entanto cada fragmento de interação é considerado uma interação independente. Um fragmento de interação é representado como um retângulo que envolve toda a interação, além de conter uma aba no canto superior esquerdo, contendo um operador que determina a qual tipo de diagrama de interação ele se refere. O operador **sd**, por exemplo, indica que o fragmento é um diagrama de sequência ou de comunicação. O texto seguinte ao operador contém a descrição da interação que está sendo modelada, normalmente contendo apenas o nome da interação. Na prática, mesmo um diagrama de sequência completo pode ser considerado um fragmento de interação no momento em que possa ser referenciado em outro diagrama, como será apresentado ao longo deste capítulo. A figura 7.14 apresenta um exemplo de fragmento de interação.



*Figura 7.14 – Exemplo de Fragmento de Interação – Processo de Emissão de Saldo.*

Essa figura representa o processo de emissão de saldo do sistema de controle bancário que estivemos modelando ao longo dos capítulos anteriores. Observe que esse exemplo contém quatro lifelines: a primeira representa uma instância do cliente que inicia o processo de emissão de saldo (por meio de um caixa eletrônico ou de uma página web); a segunda representa uma instância da classe **VisaoEmitirSaldo** que o eleitor deve lembrar ser uma classe **boundary** (ou seja, uma classe responsável pela interface com o usuário); a terceira representa uma instância da classe **ControleEmitirSaldo** que, no capítulo 4 sobre o diagrama de classes, foi definida como uma classe de controle, conforme demonstra seu estereótipo; a última, uma instância da classe **ContaComum**, que é uma classe de entidade.

Nesse processo, o cliente deve informar o número de sua conta, inserindo o cartão da conta na interface do caixa eletrônico ou digitando o número em uma página web. Esse número é repassado da interface para a

controladora do sistema, que disparará o método `consultarConta` na lifeline da classe `ContaComum` e, caso o número da conta se refira a uma conta válida, o controlador solicitará à interface que apresente uma mensagem pedindo a senha da referida conta ao cliente.

O cliente, então, deverá digitar sua senha, que será repassada da interface à controladora, que, por sua vez, solicitará o disparo do método `validarSenha` e, se essa for a senha correta, o disparo do método `emitirSaldo` para retornar o valor do saldo da conta que o cliente informou. O controlador pedirá, então, que a interface apresente o saldo ao cliente.

Observe que o diagrama demonstra o retorno “verdadeiro” para os primeiros dois métodos, destacando que se trata de um valor inteiro. Obviamente, um valor inteiro não pode conter o texto “verdadeiro”, e sim somente números, de modo que optamos por inserir a palavra “verdadeiro” para indicar o sucesso da operação, mas o retorno correto seria um número, sendo 1 para determinar que o cliente foi encontrado ou a senha informada é válida e 0 para o contrário, por exemplo. Já o retorno do último método é um `double` que conterá o saldo atual da conta.

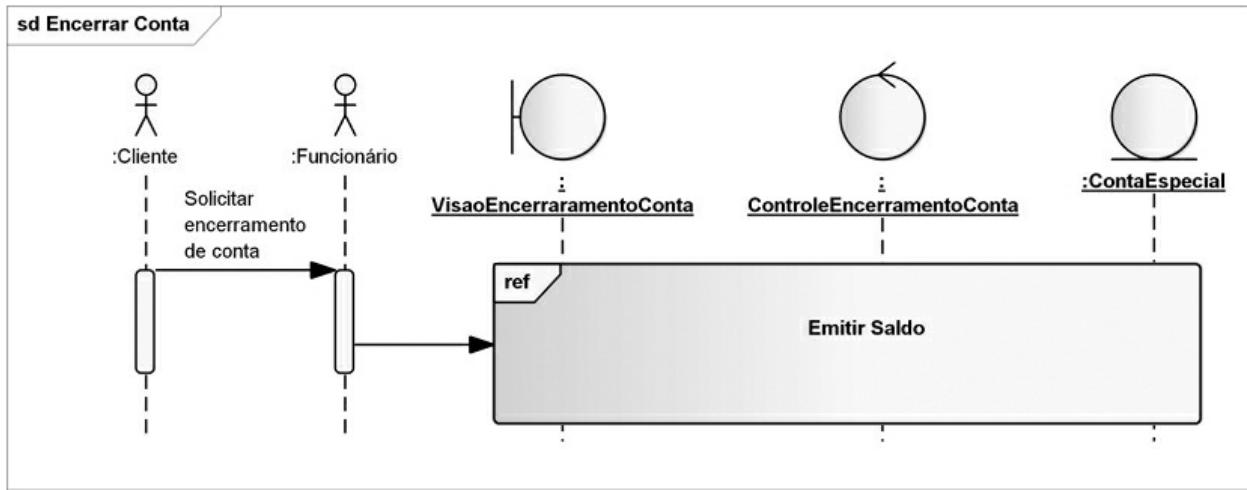
Observe, ainda, que aplicamos o estereótipo `<<entity>>` à lifeline da classe `ContaComum`, em vez de utilizarmos o desenho-padrão explicado anteriormente. Agimos dessa forma para tornar claro que a lifeline é uma classe de entidade e para demonstrar essa alternativa de representação.

Uma última observação ainda se faz necessária. Nos exemplos de diagramas de sequência apresentados neste livro, representamos explicitamente somente a chamada de métodos nas lifelines de classes de entidade. O leitor notará que as mensagens entre as lifelines de fronteira e de controle apenas descrevem os eventos ocorridos. Como foi destacado no capítulo 4, sobre o diagrama de classes, classes de fronteira e de controle podem igualmente conter métodos, todavia, neste livro, optamos por representar somente os métodos contidos pelas classes de entidade. Sendo assim, no diagramas de sequência, apenas modelamos os métodos contidos nessas classes.

## 7.6 Usos de Interação (Ocorrências de Interação antes da UML 2.1.1)

Uma das principais vantagens do uso de fragmentos de interação

caracteriza-se pela possibilidade de se poder referenciá-los por meio do operador **Ref**, que é a abreviatura de **Referred** (referido) e significa que se deve procurar por um diagrama cujo nome é o mesmo do nome apresentado após o operador **Ref**, ou seja, o fragmento faz referência a outro diagrama, não detalhado no diagrama em questão e que deve ser inserido neste. A isto chama-se uso de interação (esse recurso passou a chamar-se uso de interação a partir da UML 2.1.1; antes era chamado ocorrência de interação) e permite que se montem diagramas mais complexos que fazem referência a outros diagramas como se fossem subrotinas, detalhadas em separado, diminuindo, assim, o tamanho do diagrama e facilitando sua leitura e compreensão. A figura 7.15 apresenta um exemplo de uso de interação em um fragmento de interação.



*Figura 7.15 – Exemplo de Ocorrência de Interação.*

Nesse exemplo, enfocamos o início do processo de encerramento de conta do sistema de controle bancário, em que o cliente solicita ao funcionário o encerramento de uma conta. Para encerrar uma conta, é necessário primeiro verificar seu saldo, para determinar se é preciso sacar ou depositar algum valor.

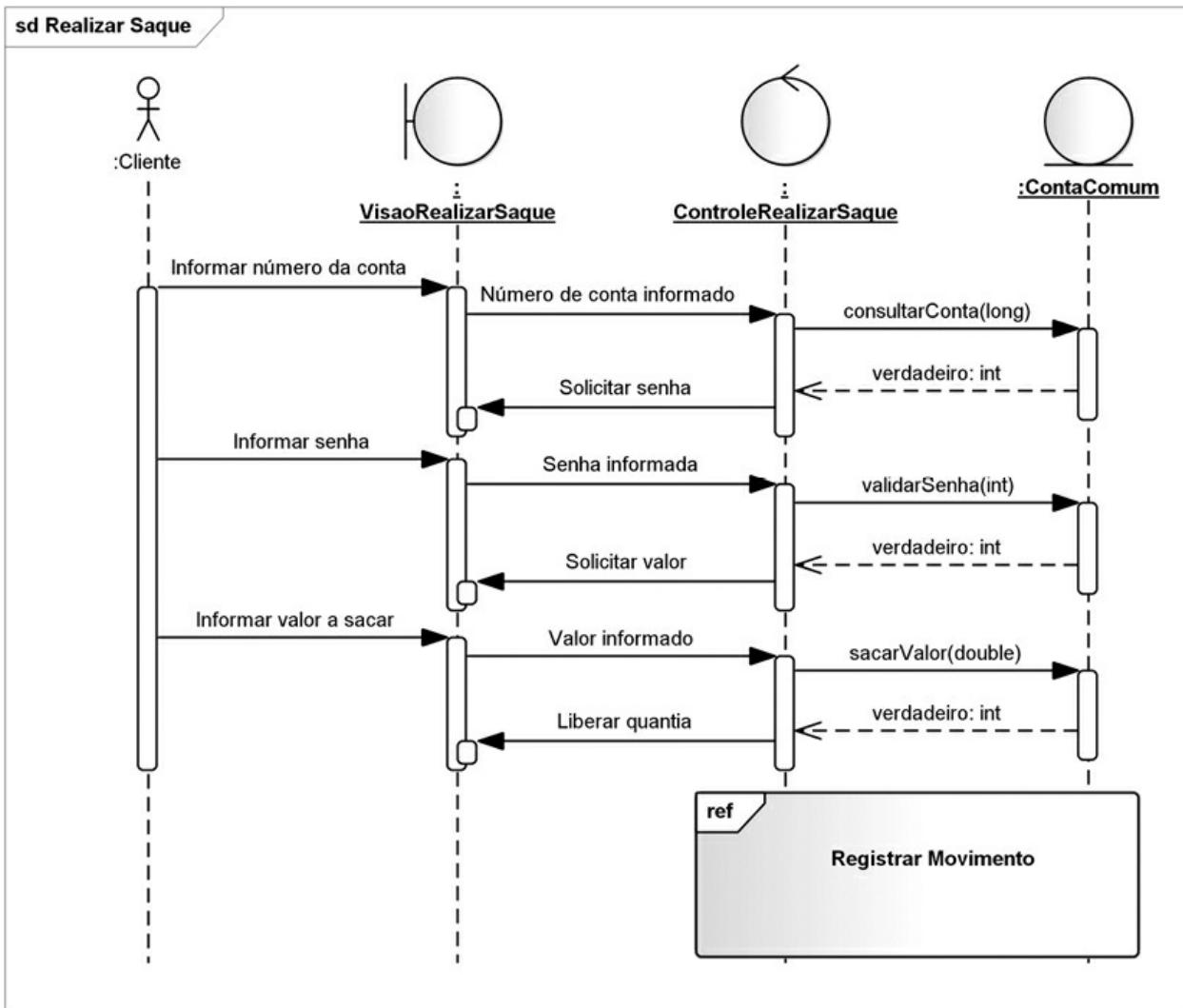
Como o processo de emissão de saldo já se encontra modelado em um diagrama à parte, é contraproducente ter que modelar esses passos novamente, e se houver alguma mudança no processo, este terá que ser alterado nos dois diagramas. Para evitar isso, faz-se uma referência a esse diagrama por meio de um uso de interação. Observe que o uso de interação é colocado sobre as linhas de vida dos objetos envolvidos no processo, e o

ator (poderia ser também um objeto) simplesmente solicita sua execução por meio de uma mensagem. Essa mensagem poderia conter texto, mas o próprio uso de interação já é autoexplicativo.

Assim, no processo de encerramento de conta, primeiramente o funcionário chamará o processo de emissão de saldo, que está detalhado em outro diagrama, e após a execução deste, o processo de encerramento de conta continuará normalmente.

É possível encontrar usos de interação simplesmente sobrepostos às linhas de vida dos objetos que fazem parte do processo, sem nem ao menos chamá-las por meio de uma mensagem, como se as instruções contidas nos usos de interação fossem adicionadas automaticamente ao diagrama, como pode ser visto na figura 7.16, que se refere ao processo de realizar saque.

Dessa vez, apresentamos o processo de realizar saque do sistema de controle bancário para ilustrar esse novo exemplo. Ao estudarmos o diagrama, percebemos que, da mesma forma que para visualizar o saldo de uma conta, é necessário primeiro informar o número da conta e, se esta estiver correta, informar a senha dela para realizar um saque. Se a senha estiver correta, o sistema solicitará o valor que o cliente deseja sacar. O cliente, então, informará o valor desejado na interface, que o repassará para o controlador e este, por sua vez, disparará o método **sacarValor** em um objeto da classe **ContaComum**. Esse método, caso haja saldo suficiente, diminuirá o valor solicitado do saldo da conta e autorizará o sistema a liberar o dinheiro pedido.



*Figura 7.16 – Exemplo de Uso de Interação – Processo de Realizar Saque.*

Conforme foi definido no diagrama de casos de uso, ao realizar um saque, é necessário registrar esse movimento e, como os passos do processo de registrar movimento estão detalhados em outro diagrama de sequência, apenas colocamos seu uso de interação sobre as linhas de vida dos objetos que participarão do processo, sem ao menos enviar uma mensagem a esse uso de interação.

Os usos de interação podem se constituir em uma simples chamada a outro fragmento de interação ou passar parâmetros para este receber o retorno da chamada deste ou ambos. Muitas vezes, as associações de inclusão e extensão do diagrama de casos de uso denotam a necessidade da existência de usos de interação nos diagramas de sequência, já que um diagrama de sequência é uma forma de documentar um caso de uso e, se este tem uma

associação de inclusão ou extensão com outro caso de uso, é muitas vezes preciso referenciá-la no diagrama de sequência, por meio de usos de interação.

## 7.7 Portões (Gates)

Um portão é uma interface entre fragmentos, um ponto de conexão para relacionar uma mensagem fora de um uso de interação com uma mensagem dentro do uso de interação. Portões podem ser representados de duas formas. A mais comum representa o portão simplesmente pelo encontro da seta da mensagem no retângulo do uso de interação, como ocorre na figura 7.15. Quando, porém, se deseja explicitamente representar uma mensagem transmitida por algum elemento externo ou o envio de uma mensagem para fora de um fragmento, o portão é representado por um pequeno quadrado, podendo este ser atingido por uma mensagem ou uma mensagem ser enviada dele.

## 7.8 Fragmentos Combinados e Operadores de Interação

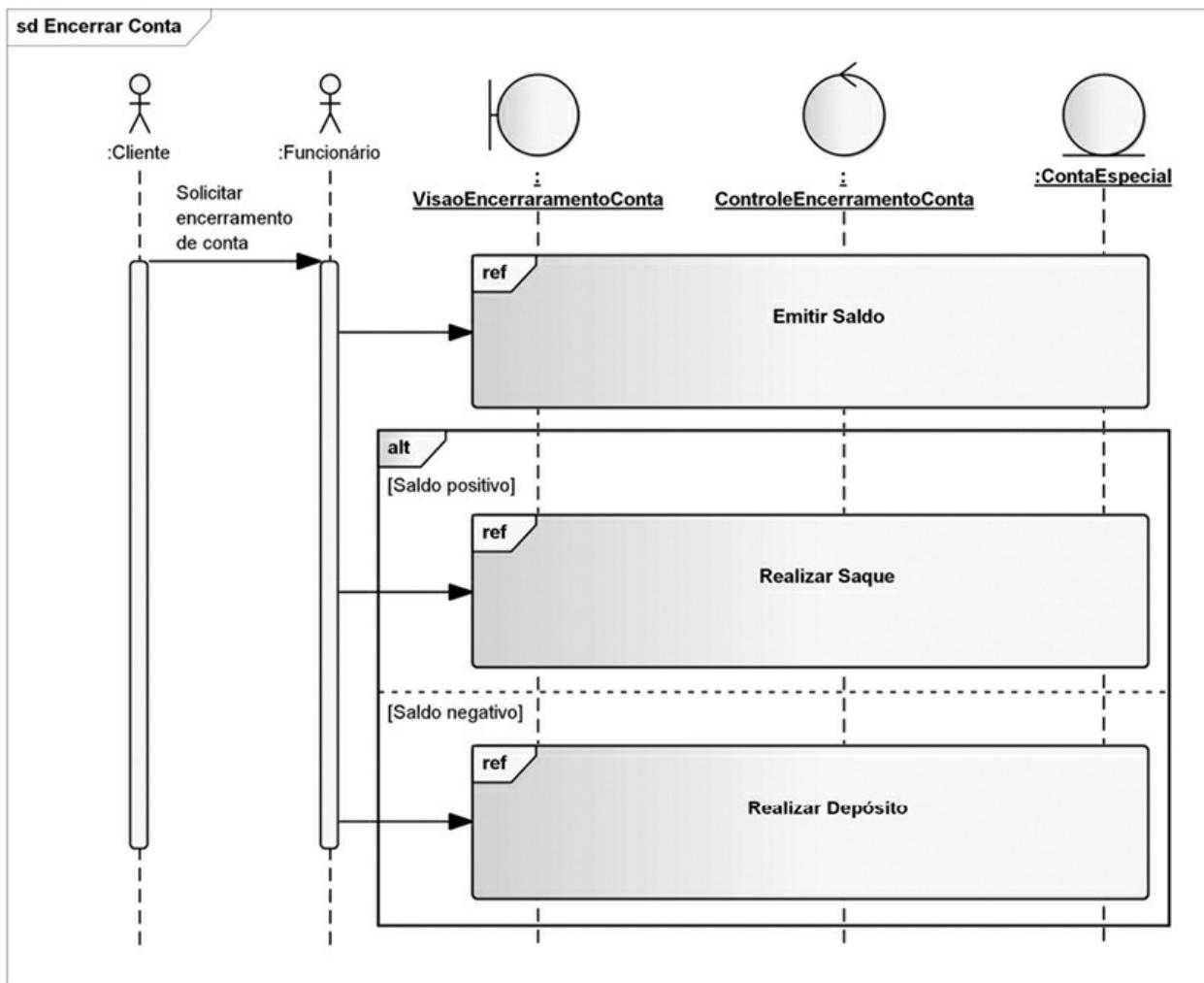
Nas versões anteriores à versão 2.0 da UML, os diagramas de sequência tinham dificuldade em trabalhar questões como testes se-senão, laços ou processamentos paralelos. Essas questões foram abordadas a partir da versão 2.0 por meio do uso de fragmentos combinados, que permitem uma modelagem semi-independente da parte do diagrama onde se deve enfocar problemas como os enunciados.

Os fragmentos combinados são representados por um retângulo que determina a área de abrangência do fragmento no diagrama, além de conterem ainda uma subdivisão na extremidade superior esquerda para identificar a descrição do fragmento combinado e seu operador de interação, que define o tipo de fragmento que está sendo modelado. Alguns dos operadores de interação mais comuns são listados e exemplificados a seguir:

- **Alt** – Abreviatura de Alternatives (Alternativas). Este operador de interação define que o fragmento combinado representa uma escolha entre dois ou mais comportamentos. Esse tipo de fragmento combinado costuma utilizar condições de guarda (texto entre colchetes que estabelece uma regra ou condição), também conhecidas como restrições

de interação, para definir o teste a ser considerado na escolha de um dos comportamentos. A figura 7.17 apresenta um exemplo de fragmento combinado com o operador **alt**.

Aqui, damos continuidade ao processo de encerramento de conta, em que, depois de verificar o saldo da conta, deverá ser feita uma escolha entre duas operações. Se o saldo da conta for positivo, o processo executará um saque e entregará ao cliente o valor depositado. Já se o saldo estiver negativo, o cliente deverá depositar o valor necessário para cobrir o saldo negativo da conta antes de encerrá-la. Observe que em cada uma das alternativas foi feita uma referência a um uso de interação: a primeira relativa ao processo de realizar saque e a segunda, ao processo de realizar depósito. Uma vez que esses processos já se encontram modelados em outros diagramas, é mais prático somente os referenciar.



*Figura 7.17 – Exemplo de Fragmento Combinado com Operador Alt.*

Observe que fragmentos combinados que utilizam o operador de interação **alt** têm ao menos uma divisão, representada por uma linha tracejada, separando as ações executadas nas alternativas. Cada uma dessas divisões é chamada separador de operando de interação e o conteúdo representado em cada divisão é conhecido como operando de interação, ou seja, uma área de atuação de um fragmento combinado. Um fragmento combinado contém ao menos um operando de interação, e, em alguns casos, deve ter ao menos dois, como nesse exemplo, e em outros não poderá ter mais de um, quando não existem cenários alternativos nem paralelos. Nesse último caso, o operando de interação representa todo o conteúdo do fragmento combinado.

- **Opt** – Abreviatura de Option (Opção). Esse operador de interação determina que o fragmento combinado representa uma escolha de comportamento em que esse comportamento será ou não executado, não havendo uma escolha entre mais de um comportamento possível. A figura 7.18 apresenta um exemplo de fragmento combinado utilizando o operador de interação opt.

A figura 7.18 dá continuidade ao processo de encerramento de conta, em que, após ser realizado um saque ou depósito, pode ser necessário dar manutenção no cadastro do cliente, tornando-o inativo, caso a conta a ser encerrada seja a única por ele possuída. Por esse motivo, utilizamos um fragmento combinado com operador **Opt**, significando que os passos nele contidos serão ou não executados dependendo de sua condição, determinada pela restrição de interação “Se for a única conta”.

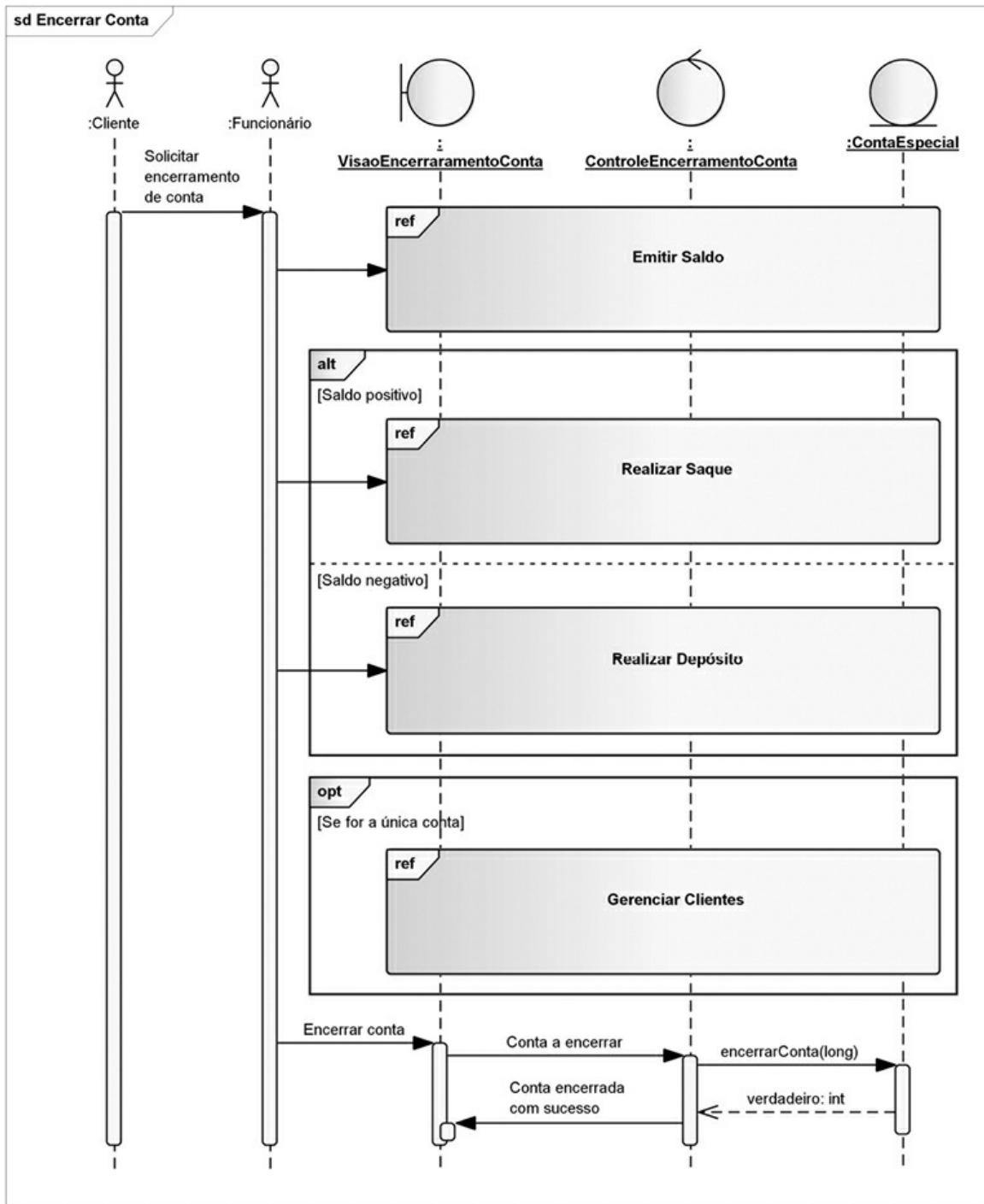


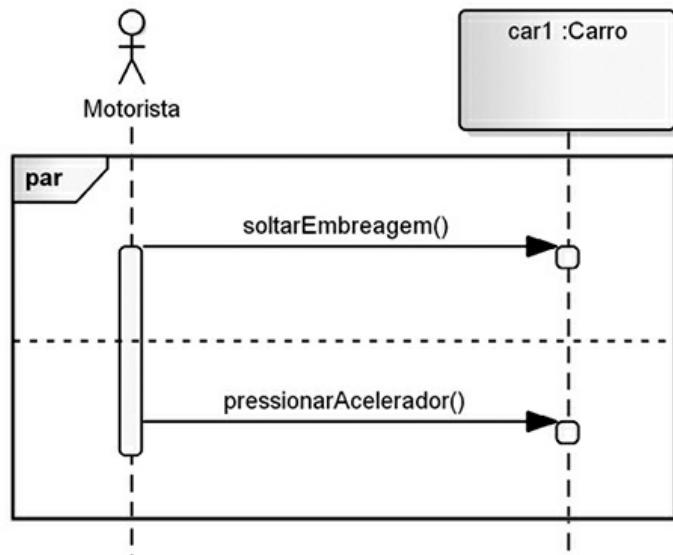
Figura 7.18 – Exemplo de Fragmento Combinado com Operador Opt.

O leitor notará que o processo **Gerenciar Clientes** é um processo referenciado, ou seja, um uso de interação, e, se formos examinar o diagrama de casos de uso do sistema de controle bancário, perceberemos que o caso de uso **Encerrar Conta** tem uma associação de extensão com o processo **Gerenciar Clientes**. No caso de associações de extensão, é

preciso fazer um teste para determinar se o caso de uso será ou não estendido, aqui refletido na restrição de interação. Quando se tratar de associações de inclusão, não é preciso haver um fragmento combinado do tipo **opt**, bastando inserir um uso de interação sobre a linha de vida dos objetos, como acontece com o uso de interação **Emitir Saldo**.

Para concluir a descrição do processo de encerramento de conta, depois de verificar se é necessário alterar o cadastro do cliente, o funcionário solicitará o encerramento da conta em questão. Essa solicitação será repassada ao controlador, que disparará o método **encerrarConta** e, se este for bem-sucedido, ordenará à interface que apresente uma mensagem de sucesso.

- **Par** – Abreviatura de Parallel (Paralelo). Esse operador de interação determina que o fragmento combinado representa uma execução paralela de dois ou mais comportamentos. A figura 7.19 apresenta um exemplo de fragmento combinado utilizando o operador de interação **par**.

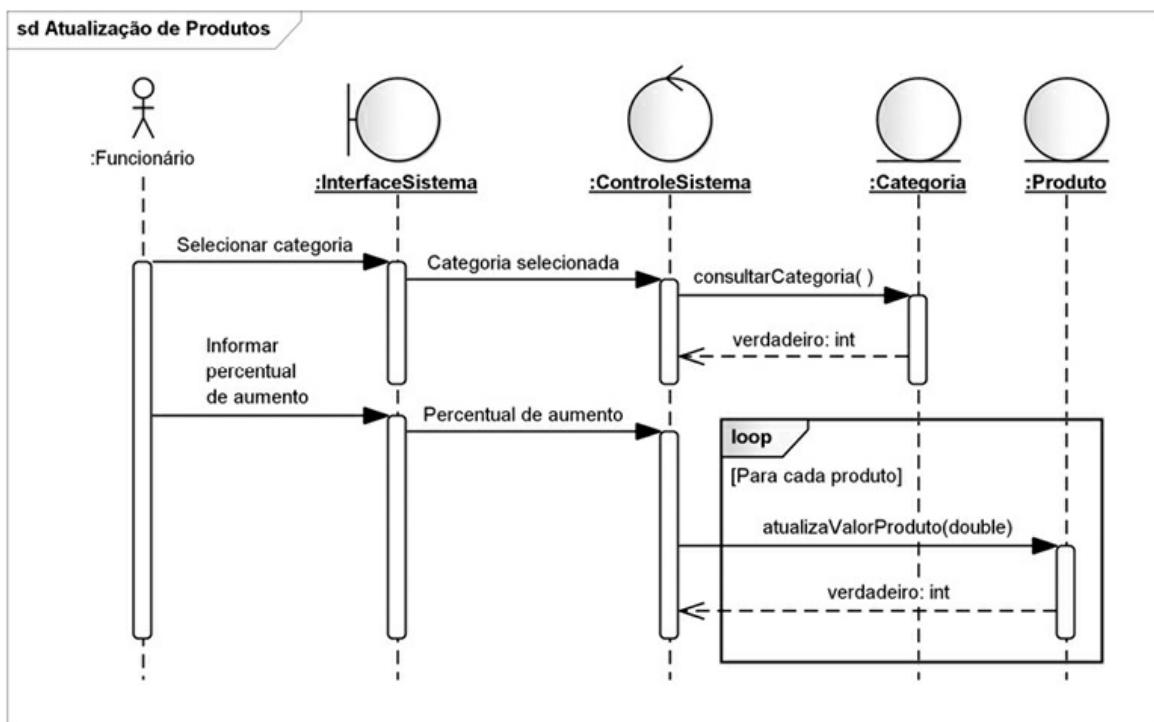


*Figura 7.19 – Exemplo de Fragmento Combinado com Operador Par.*

Identificamos aqui uma situação em que o ator **Motorista** deve realizar duas operações simultâneas sobre a lifeline **car1** da classe **Carro** para poder dirigí-lo: soltar a embreagem e pressionar o acelerador. Note que uma linha tracejada divide os operandos de interação, representando cada operação paralela.

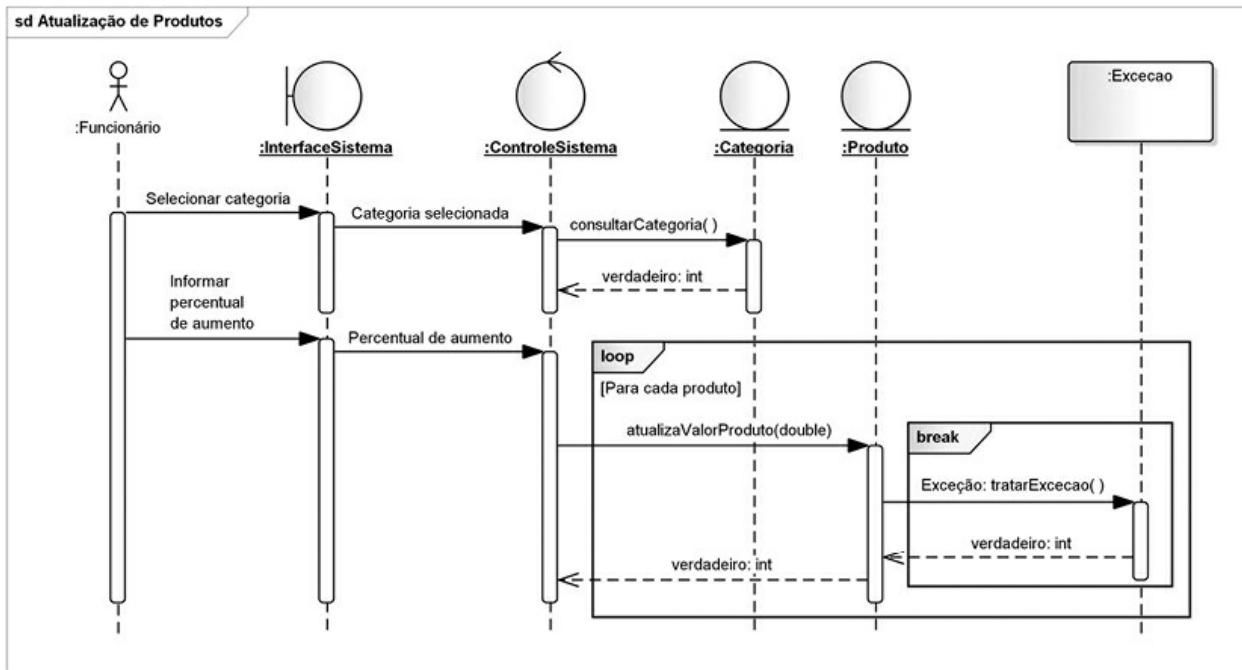
- **Loop** – Abreviatura de Looping (Laço). Esse operador de interação determina que o fragmento combinado representa um laço que poderá ser repetido diversas vezes. A figura 7.20 demonstra um exemplo de fragmento combinado utilizando o operador de interação **loop**.

Nesse exemplo, identificamos um processo utilizado para aumentar os produtos de uma categoria. Ao observarmos a figura, percebemos que o funcionário seleciona uma categoria, informa o percentual de aumento e manda aumentar o valor de todos os produtos pertencentes à categoria selecionada. Podemos perceber ainda que há um laço por meio do qual o valor dos produtos da categoria é atualizado, em que o método para atualizar o valor (**atualizaValorProduto**) é aplicado a cada produto pertencente à categoria selecionada, conforme demonstra a restrição de interação “[Para cada produto]”.



*Figura 7.20 – Exemplo de Fragmento Combinado com Operador Loop.*

- **Break** (Quebra) – Esse operador de interação indica uma “quebra” na execução normal do processo. É usado principalmente para modelar o tratamento de exceções. A figura 7.21 apresenta um exemplo de fragmento combinado utilizando o operador de interação **break**.



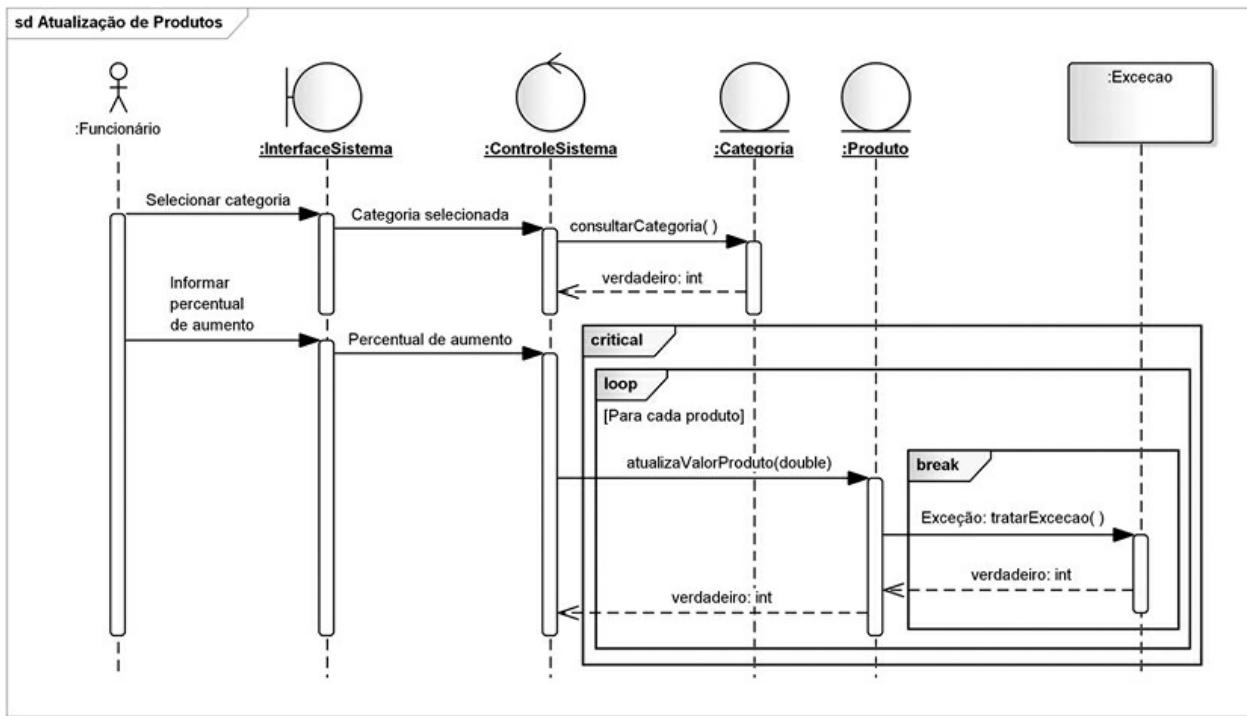
*Figura 7.21 – Exemplo de Fragmento Combinado com Operador Break.*

Aqui, demos continuidade ao processo do exemplo anterior, em que identificamos uma situação em que pode ocorrer uma exceção, em razão de algum registro estar corrompido ou algum campo ter valores inválidos. Assim, criamos um método `tratarExcecao`, pertencente à classe `Excecao`, para tratar possíveis exceções que venham ocorrer durante a atualização. Observe que como uma exceção interrompe o desenrolar normal do laço, além de não ocorrer normalmente, ela está contida em um fragmento combinado do tipo `break`. Esse exemplo ilustra também a possibilidade de um fragmento combinado poder conter outro fragmento combinado.

Observe ainda que a lifeline da classe de exceção não recebeu o estereótipo `<<entity>>` como as lifelines das classes `Categoria` e `Produto`. Isto ocorre porque a classe da lifeline em questão não é uma classe de entidade, ou seja, não pertence ao domínio do problema, trata-se apenas de uma classe auxiliar utilizada pela interação. A classe `Excecao` é também uma classe transitória cujos objetos não precisam ser persistidos.

- **Critical Region** (Região Crítica) – esse operador de interação identifica uma operação atômica que não pode ser interrompida por outro processo até ser totalmente concluída. A figura 7.22 apresenta um

exemplo de fragmento combinado utilizando esse operador.



*Figura 7.22 – Exemplo de Fragmento Combinado com Operador Critical Region.*

Nessa figura, melhoramos o exemplo utilizado para ilustrar os dois últimos operadores de interação, envolvendo o laço de atualização de produtos com um fragmento combinado do tipo **critical**, indicando que a atualização dos produtos não deverá ser interrompida até que o processo seja totalmente concluído.

Existem ainda alguns operadores de interação menos utilizados, apresentados a seguir:

- **Neg** – Abreviatura de Negative (Negativo) – esse operador de interação representa eventos considerados inválidos, que não devem ocorrer. Todos os eventos não contidos em um fragmento combinado do tipo **neg** (quando existir um) são considerados positivos.
- **Assertion** (Afirmação) – esse operador de interação é o oposto do anterior, representando eventos considerados válidos. Todos os eventos não contidos em um fragmento combinado do tipo **assertion** são automaticamente considerados negativos.
- **Ignore** (Ignorar) – o operador de interação **Ignore** determina que as

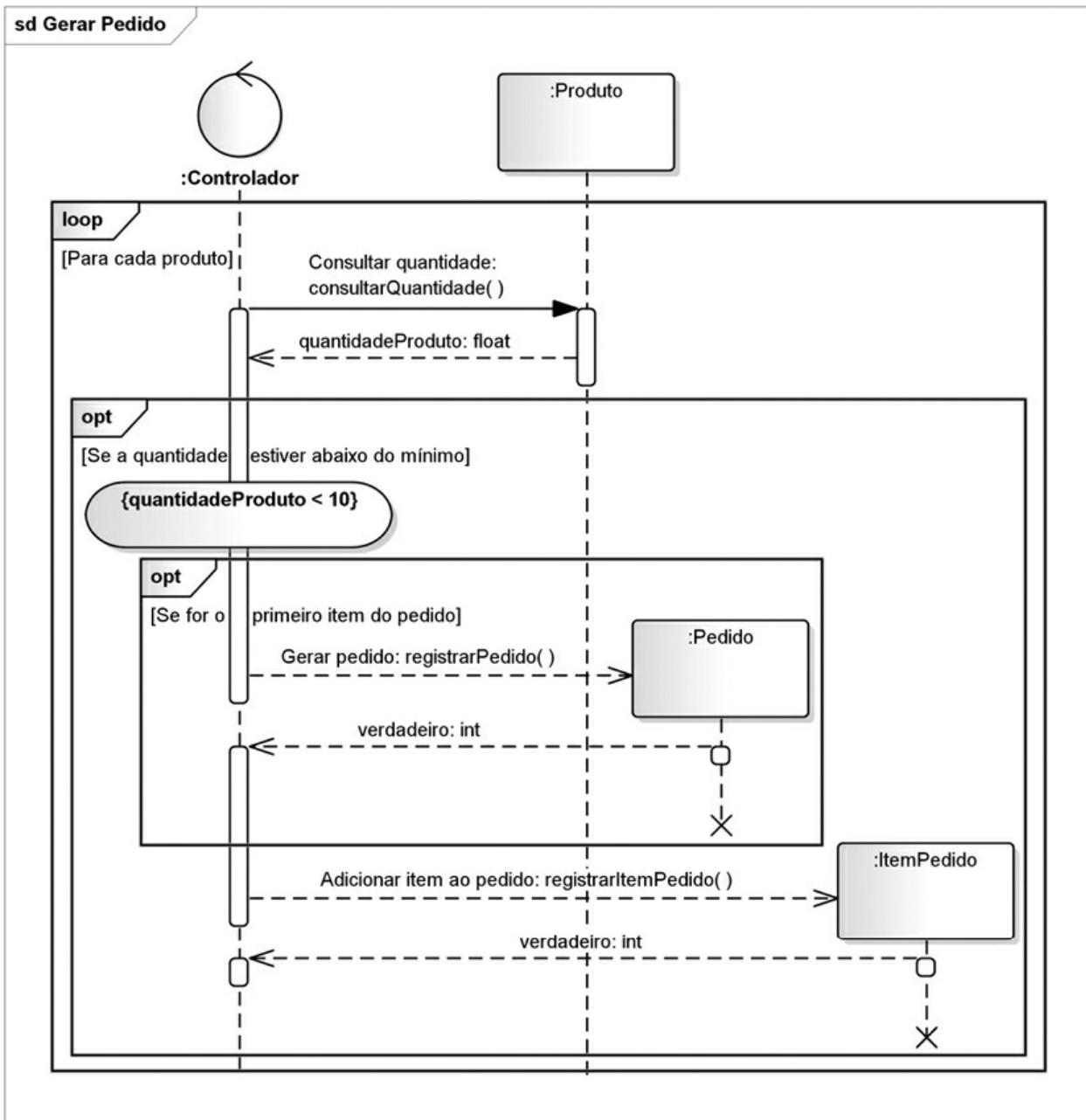
mensagens contidas no fragmento devem ser ignoradas. Essas mensagens podem ser consideradas insignificantes e são intuitivamente ignoradas se aparecerem em uma execução correspondente. Alternativamente, pode-se entender **ignore** como significando que as mensagens ignoradas podem aparecer em qualquer lugar nos eventos.

- **Consider** (Considerar) – esse operador de interação é o oposto do anterior e determina que as mensagens devem obrigatoriamente ser consideradas e que todas as outras não contidas no fragmento devem ser automaticamente desconsideradas. Tanto o operador de interação **ignore** como **consider** são frequentemente utilizados com os operadores **neg** e **assertion**, de maneira que um fragmento pode conter o outro.
- **Seq** – Abreviatura de Weak Sequencing (Sequência Fraca) – esse operador de interação identifica uma situação na qual as ocorrências de evento devem atender a essas propriedades:
  - As ordenações das ocorrências de evento dentro de cada um dos operandos são mantidas no resultado.
  - Ocorrências de evento em linhas de vida diferentes de operandos diversos podem vir em qualquer ordem.
  - Ocorrências de evento na mesma linha de vida de operandos diferentes são ordenadas de tal forma que uma ocorrência de evento do primeiro operando venha antes do segundo operando.
- **Strict** – Abreviatura de Strict Sequencing (Sequência Estrita) – o operador de interação **strict** apresenta um refinamento do operador **weak sequencing** e garante que todas as mensagens no fragmento combinado sejam ordenadas do início ao fim.

## 7.9 Invariante de Estado (StateInvariant)

Um invariante de estado define uma restrição a ser aplicada em tempo de execução aos participantes da interação. Pode ser usado para especificar diferentes tipos de restrições, como valores de atributos ou variáveis, estados internos ou externos etc. Um invariante de estado é um fragmento de interação e deve ser colocado sobre uma linha de vida. A restrição de invariante de estado estabelece uma condição para que o comportamento pretendido de um ou mais elementos do diagrama possa ser executado.

Essa condição deve ser avaliada durante o tempo de execução. Um invariante de estado pode ser representado por um círculo posicionado sobre a linha de vida do elemento ou por um texto entre chaves. A figura 7.23 apresenta um exemplo de invariante de estado.



*Figura 7.23 – Exemplo de Invariante de Estado.*

Nesse exemplo, demonstramos uma situação em que é gerado um pedido automático para todos os produtos cuja quantidade estiver abaixo do mínimo. Há quatro lifelines representando as instâncias das classes

**Controlador, Produto, Pedido e ItemPedido.** Há um fragmento combinado do tipo loop que representa um laço em que a controladora dispara uma mensagem para consultar a quantidade de cada produto. Ao receber a quantidade do produto, a controladora verifica se essa quantidade está abaixo do mínimo, o que é representado por um fragmento combinado do tipo opt e por uma invariante de estado contendo uma restrição que apresenta o teste para verificar se a quantidade do produto é menor que dez. Em caso positivo, faz-se um segundo teste, representado por um segundo fragmento combinado do tipo opt que determina se o produto com quantidade abaixo do mínimo é o primeiro item do pedido. Nesse caso, é gerada uma nova lifeline da classe **Pedido**. Independentemente disso, sempre que a quantidade de um produto estiver abaixo do mínimo, será gerada uma nova lifeline da classe **ItemPedido**, representando um novo item do pedido.

## 7.10 Exemplos de Diagramas de Sequência para o Sistema de Controle Bancário

Nesta seção, exemplificaremos os principais diagramas de sequência referentes ao sistema de controle bancário. Os processos de **Emitir Saldo**, **Encerrar Conta** e **Realizar Saque** foram previamente explicados. Para ter uma compreensão completa desses diagramas, é útil acompanhá-los com o diagrama de casos de uso e o diagrama de classes desse mesmo sistema, já explicados anteriormente.

### 7.10.1 Processo de Abertura de Conta Comum – Modelo Preliminar

O diagrama de sequência pode ser aplicado ainda durante a fase de análise de requisitos, para se ter uma ideia geral de como será a interação de um processo. Contudo, durante a fase de análise, os métodos não costumam estar definidos ainda, posto que já fazem parte da solução e, portanto, só devem ser identificados na fase de projeto. Dessa forma, o diagrama de sequência deve ser tratado como uma caixa preta, sem especificar como o processo se desenrolará internamente, identificando-se somente os atores envolvidos no processo, a interface do sistema e, eventualmente, a classe controladora, conforme pode ser visto na figura 7.24.

Aqui, enfocamos o processo de abertura de conta comum, mas somente

são apresentadas a interação entre os usuários, a interface e a controladora, não havendo nenhuma lifeline de classe de entidade declarada e tampouco qualquer chamada de método. Esse diagrama apresenta apenas o pedido de execução de tarefas ao sistema e o retorno dos resultados destas, não havendo detalhamento de como os serviços solicitados pelos atores serão realizados pelo sistema. Assim, quando o funcionário solicitar a consulta de um cliente, só será demonstrado que a controladora pedirá à interface para apresentar os dados do cliente consultado, sem definir como tais dados foram recuperados. O detalhamento do processo deve ser inserido somente na fase de projeto. Na seção 7.10.2, apresentaremos esse mesmo diagrama, já identificando as lifelines de classes de entidade envolvidas no processo e os métodos disparados entre elas.

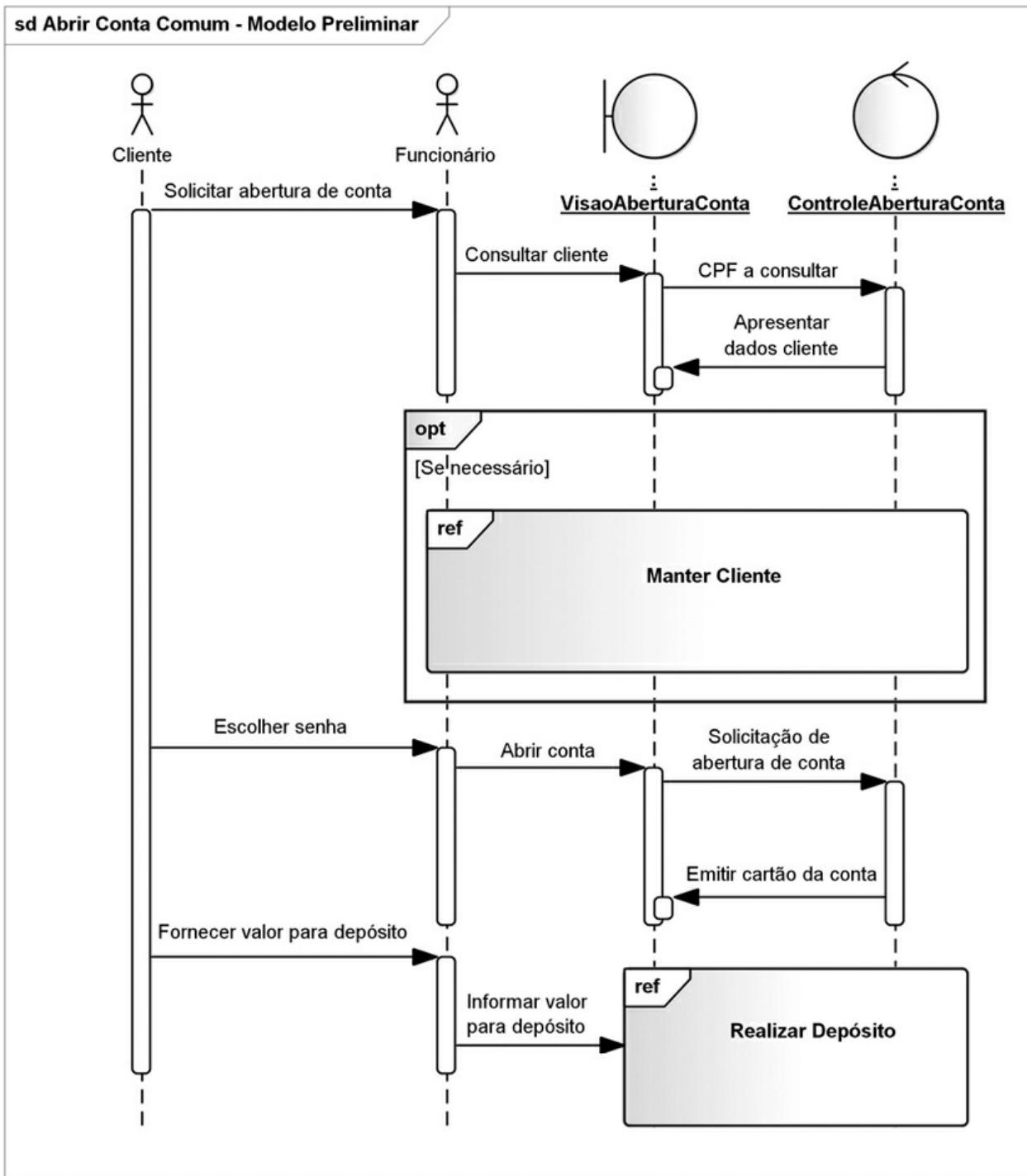


Figura 7.24 – Processo de Abertura de Conta Comum – Modelo Preliminar.

### 7.10.2 Processo de Abertura de Conta Comum – Modelo Detalhado

Neste exemplo, damos continuidade ao processo representado pelo caso de uso **Abrir Conta Comum**, iniciado na seção anterior. Desta vez,

apresentamos o processo completo. Esse processo envolve os atores **Cliente** e **Funcionário**, uma instância (lifeline) da classe de fronteira **VisaoAberturaConta**, uma instância da classe de controle **ControleAberturaConta** e instâncias das classes de entidade **PessoaFisica** e **ContaComum**. A figura 7.25 apresenta o diagrama de sequência referente a esse processo. Não aplicamos os estereótipos de entidade nas lifelines **pesfin1** e **comum1** para manter semelhança com os exemplos iniciais deste capítulo.

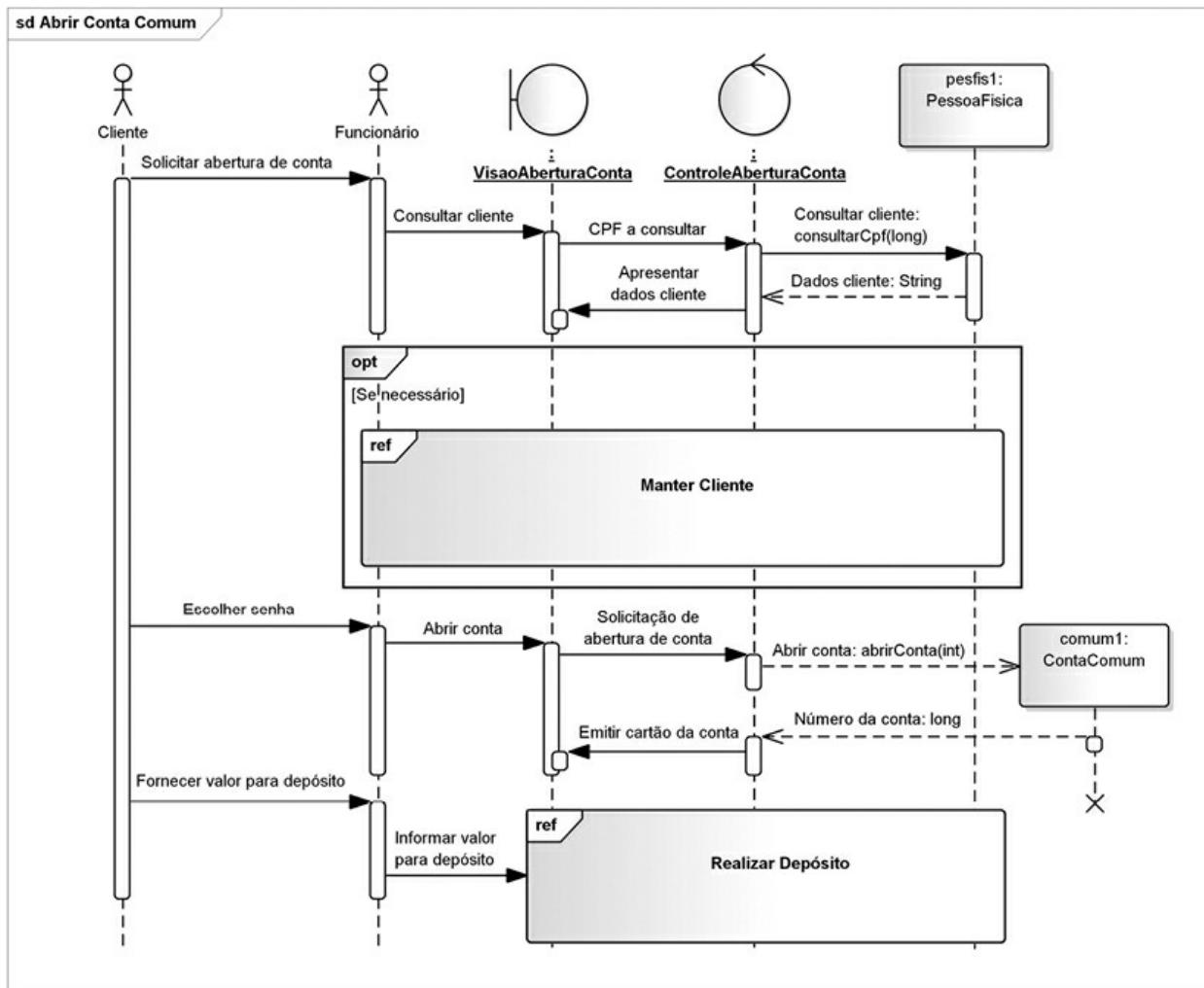


Figura 7.25 – Processo de Abertura de Conta Comum.

Como podemos verificar, primeiramente o cliente que deseja ter uma conta no banco apresenta um pedido de abertura de conta a um funcionário, o qual consultará o cadastro de clientes, o que acarretará o disparo do método **consultarCpf** em um objeto da classe **PessoaFísica**,

passando como parâmetro o CPF do cliente, para determinar se o solicitante já se encontra cadastrado. Se o cliente já estiver registrado, a consulta retornará as informações do cliente, caso contrário, retornará um valor significando que o cliente ainda não possui cadastro na instituição.

Em seguida, conforme demonstra o fragmento combinado do tipo **opt**, o cadastro do cliente poderá ser atualizado, caso necessário, podendo gerar uma nova instância da classe **Cliente**, se o solicitante não tiver cadastro, ou simplesmente atualizar os dados dele, se for necessária alguma modificação. Como o processo de manutenção de cadastro de clientes já se encontra modelado em outro diagrama, este foi referenciado por meio de um uso de interação, o que está de acordo com o diagrama de casos de uso desse sistema, uma vez que existe uma associação de extensão entre o caso de uso **Abrir Conta Comum** e o caso de uso **Manter Cliente**.

Depois disso, o cliente escolherá uma senha para a nova conta a ser aberta. A seguir, o funcionário solicitará o serviço de abertura de conta por meio da interface do sistema, que repassará o pedido ao controlador, o qual, por sua vez, disparará o método **abrirConta**, que gerará um novo objeto da classe **ContaComum** e retornará o número da nova conta criada. Se a operação for realizada com sucesso, o controlador ordenará, então, a emissão do cartão da conta.

Finalmente, o cliente deve fornecer um valor inicial para depósito e o funcionário, por sua vez, solicitará a execução do serviço de **Realizar Depósito**, concluindo o processo. Uma vez que o processo de **Realizar Depósito** é um caso de uso independente, apenas referenciamos esse processo por meio de um uso de interação sobreposta sobre as linhas de vida dos objetos do diagrama e representamos a solicitação de sua execução por meio do disparo de uma mensagem pelo funcionário.

Se observarmos o diagrama de caso de uso desse sistema, perceberemos haver uma associação de inclusão entre os casos de uso **Abrir Conta Comum** e **Realizar Depósito**. Por esse motivo, o processo está só referenciado e não é necessário nenhum teste, pois uma associação de inclusão indica uma obrigatoriedade. Por isso, só é preciso colocar o uso de interação sobre as linhas de vida dos objetos.

O leitor notará que esse diagrama modela, em geral, o “caminho feliz” da interação, considerando que as chamadas dos métodos para consultar o

cliente e abrir a conta foram bem-sucedidas. Situações de exceção costumam só ser modeladas quando são importantes para a interação, evitando deixar o diagrama muito extenso.

### 7.10.3 Processo de Realizar Depósito

Neste processo, os componentes são os mesmos do diagrama anterior, exceto pelo objeto da classe **PessoaFísica**, desnecessário aqui, porque um cliente não precisa se identificar para depositar um valor. A figura 7.26 apresenta o diagrama de sequência referente ao processo de **Realizar Depósito**.

Nesse diagrama, o cliente informa ao funcionário (poderia ser um caixa eletrônico também e, nesse caso, não haveria esse segundo ator) o número da conta que deverá receber o valor a ser depositado. O funcionário em resposta verificará se essa conta existe, solicitando sua consulta à interface do sistema, a qual repassará o número da conta ao controlador, que, por sua vez, disparará o método **consultarConta**. Se a conta informada for encontrada, o funcionário solicitará o valor a ser depositado e o cliente o fornecerá. O funcionário, então, informará à interface do sistema a quantidade do valor a depositar. Esse valor será repassado ao controlador, que disparará o método **depositarValor**. Se esse método for realizado com sucesso, o controlador pedirá à interface que apresente uma mensagem informando que a operação foi concluída.

O sistema deverá, ainda, registrar o movimento realizado sobre a conta em questão. Para isso, ele faz referência ao processo de **Registrar Movimento**, que, como foi definido no diagrama de casos de uso, tem relação de inclusão com o caso de uso **Realizar Depósito**. Assim, é necessário apenas posicioná-lo sobre as linhas de vida dos objetos. Como esse é um processo interno do sistema, não é necessário o disparo de nenhuma mensagem pelos atores para que ele seja executado.

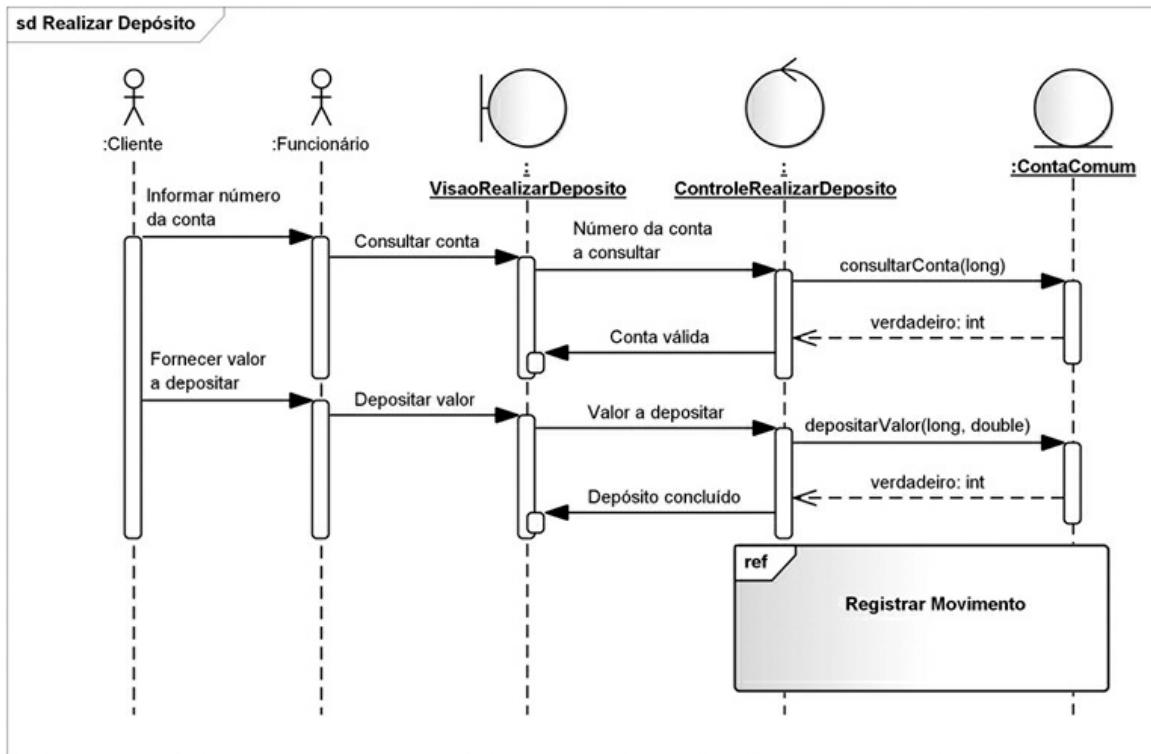
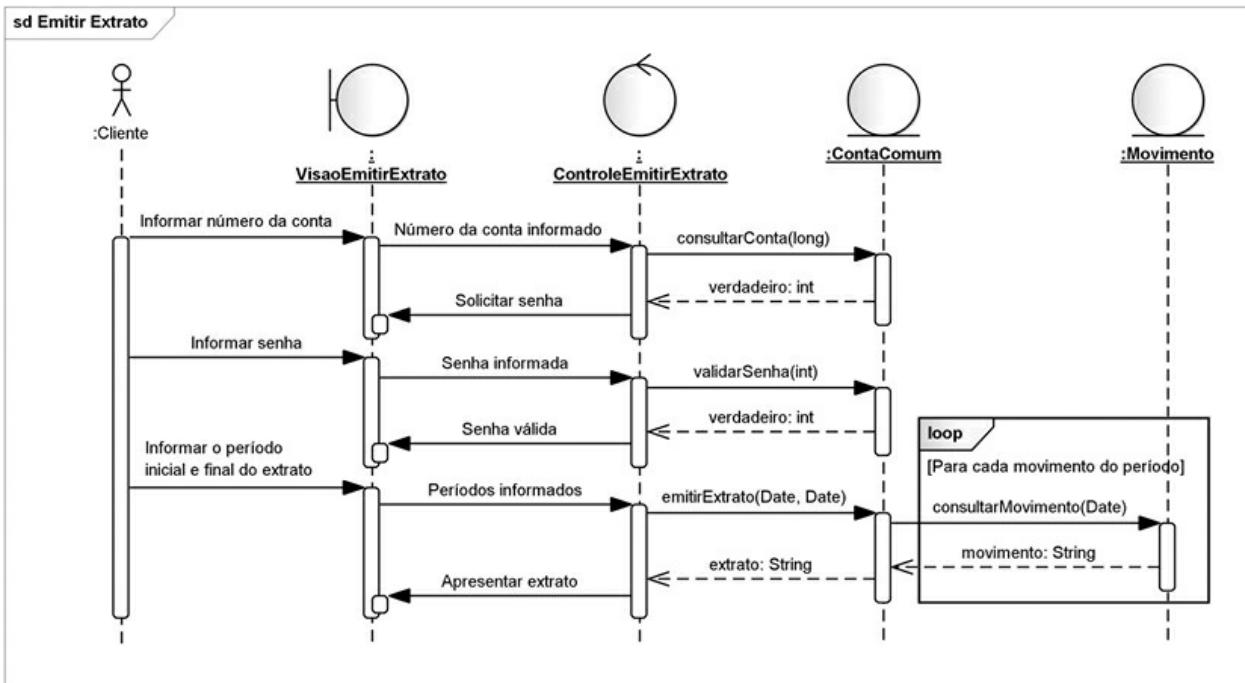


Figura 7.26 – Processo de Depósito.

#### 7.10.4 Processo de Emissão de Extrato

Este exemplo envolve o ator **Cliente** e lifelines das classes **VisaoEmitirExtrato**, **ControleEmitirExtrato**, **ContaComum** e **Movimento**. Não é necessária a interação com o funcionário, uma vez que a emissão do extrato pode ser feita diretamente por um caixa eletrônico ou pela internet. A figura 7.27 apresenta o diagrama de sequência referente ao processo de emissão de extrato.



*Figura 7.27 – Processo de Emissão de Extrato.*

Nesse processo, o cliente fornece o número da sua conta à interface, a qual repassa o número informado ao controlador, que ordenará o disparo do método **consultarConta** em um objeto da classe **ContaComum** para determinar se existe uma conta-corrente comum com um número de conta igual ao informado. Em caso positivo, o método retornará um valor, indicando que a conta é válida para o objeto de controle.

No caso de a conta ser válida, a lifeline da classe controladora informará à lifeline da classe de fronteira para solicitar a senha da conta ao cliente. Por sua vez, o cliente digitará sua senha e a interface a enviará para a lifeline da classe controladora, que, por sua vez, chamará o método de validação de senha, **validarSenha** no objeto da classe **ContaComum**, passando como parâmetro a senha informada. O método de validação retornará um valor indicando se a senha é válida ou não.

Caso a senha esteja correta, o cliente selecionará a opção de extrato e fornecerá os períodos que determinarão o intervalo desejado do relatório. Ao receber essas informações, a interface as retransmitirá à controladora, que, então, disparará o método **emitirExtrato** na lifeline da classe **ContaComum** e esta, por sua vez, disparará o método **consultarMovimento** em cada lifeline da classe **Movimento** que esteja associada à conta informada e esteja dentro do período solicitado, conforme demonstra o

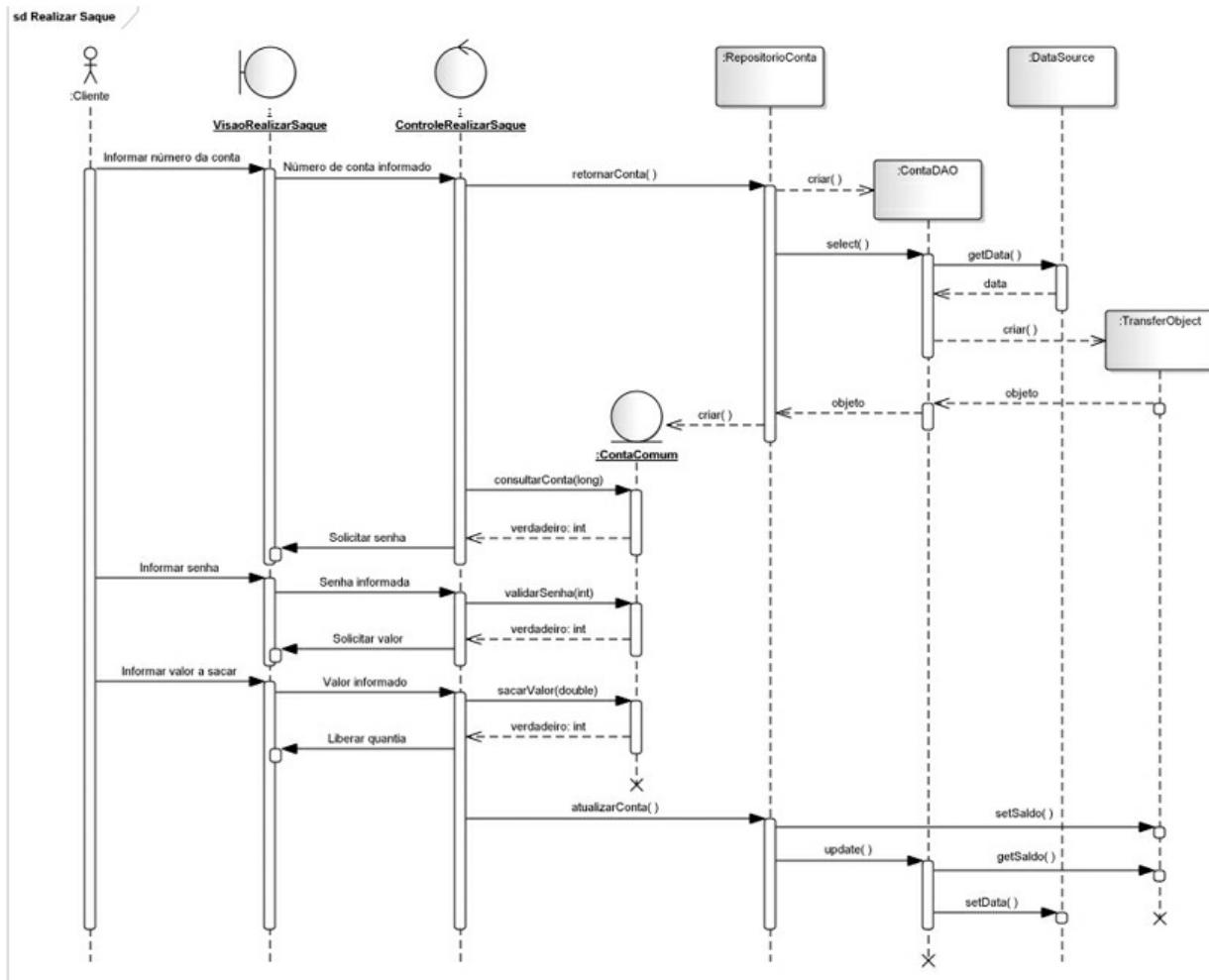
fragmento combinado do tipo loop e a condição de guarda a ele associada, retornando uma **String** com os dados de cada movimento.

Os valores contidos nas instâncias que satisfizerem à condição imposta pelo cliente serão, então, retornados à interface com a instrução fornecida pelo controlador de serem impressas e entregues ao cliente.

## 7.11 Padrões Repository e DAO

Estes padrões foram explicados no capítulo 4, sobre o diagrama de classes. Nesta seção, demonstraremos na figura 7.28 como os aplicar em uma interação. Neste exemplo, estendemos o diagrama sobre o processo de realizar saque para acomodar as lifelines das classes necessárias a esses padrões. Assim, essa interação passa a incluir a recuperação e a persistência de objetos em uma base de dados.

Se o leitor comparar essa figura com a interação de realizar saque original, perceberá diversas mudanças. A primeira é que ao ser notificada do evento de que um número de conta foi informado, a controladora pedirá que a lifeline da classe **RepositorioConta**, que representa a classe de repositório associada à conta comum, recupere a conta consultada. Isso faz com que essa lifeline crie uma lifeline da classe **ContaDao**, que é responsável pela persistência de objetos da classe **ContaComum**. Após essa lifeline ser criada, o repositório solicitará a recuperação do objeto pesquisado e, em resposta, a DAO buscará essa informação em uma fonte de dados. Ao receber essa informação, a DAO criará uma lifeline da classe **TransferObject** (literalmente objeto de transferência) e retornará o objeto criado para o repositório. A lifeline **TransferObject** será utilizada principalmente quando for necessário atualizar atributos nesse objeto, mas também para recuperar informações desse objeto, quando a DAO precisar persistir a lifeline de entidade, por exemplo.



*Figura 7.28 – Processo de Saque com os Padrões Repository e DAO.*

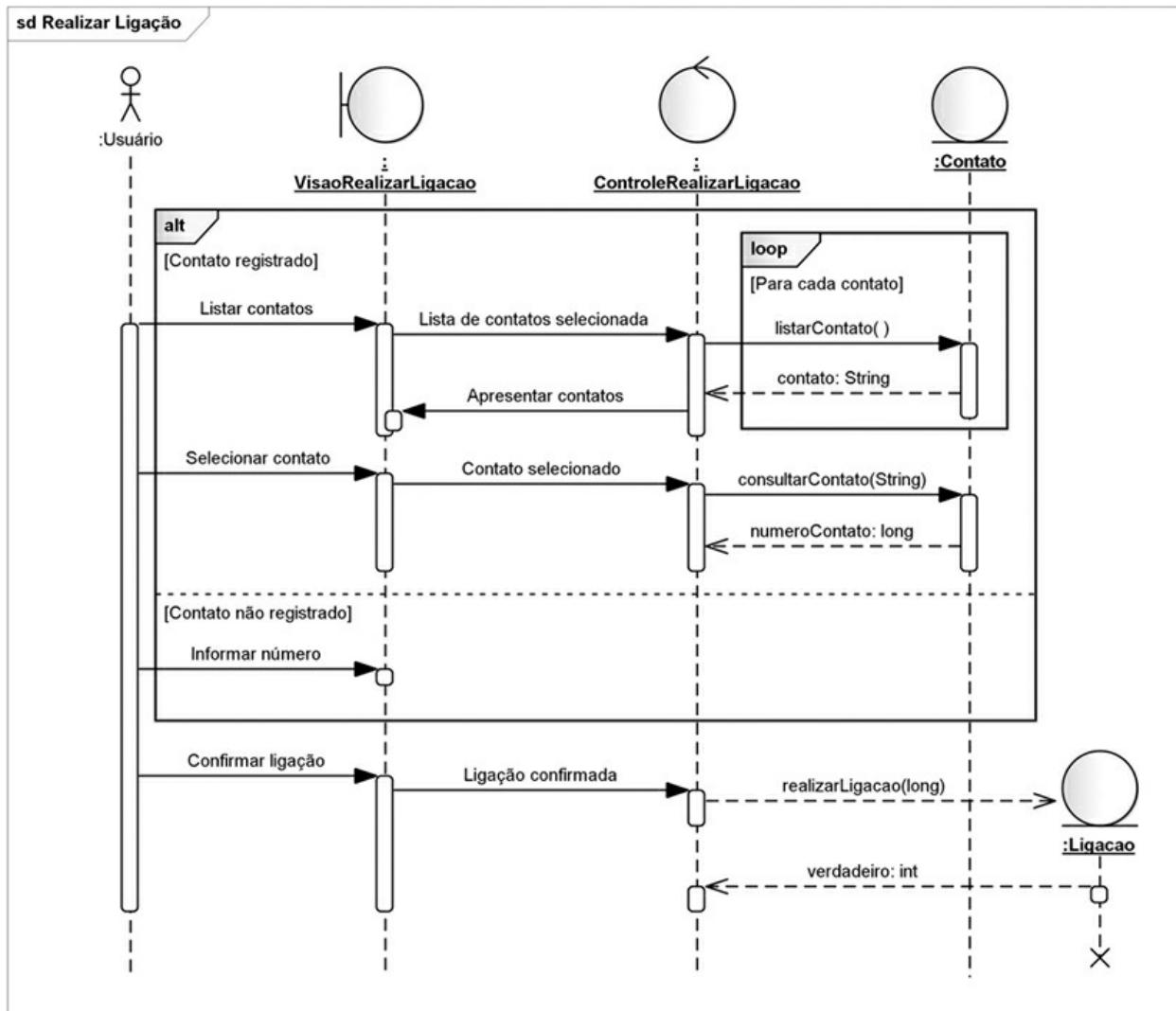
Com base no objeto retornado, o repositório cria uma lifeline da classe de entidade **ContaComum**. A partir desse ponto, o processo fica idêntico ao apresentado no processo de realizar saque original até o momento em que a quantia solicitada é liberada. Depois disso, a controladora solicita ao repositório que atualize a conta e esta, por sua vez, atualizará o atributo saldo na lifeline **TransferObject**. Em seguida, o repositório pedirá que a DAO registre a mudança no saldo da conta e a DAO registrará a atualização na base de dados.

É importante destacar que nesse exemplo estamos utilizando lifelines **Repository** e **DAO** combinadas. Se fôssemos utilizar somente a DAO, a classe de entidade assumiria o papel de **BusinessObject**, porém, nessa ilustração, esse papel é representado pela classe repositório. Além disso, somente o atributo saldo é atualizado na lifeline **TransferObject**, mas

poderiam ser diversos atributos, dependendo do processo, e a atualização na base de dados só se daria no final do processo.

## 7.12 Exemplo de Diagrama de Sequência – Processo de Realizar Ligação para o Sistema de Telefone Celular

A figura 7.29 apresenta o diagrama de sequência para o processo de realizar ligação do sistema de telefone celular que vem sendo modelado nos capítulos anteriores.



*Figura 7.29 – Processo de Realizar Ligação.*

Ao examinarmos essa figura, percebemos que há uma lifeline representando uma instância do ator Usuário que interage com o software, além das lifelines correspondentes à interface e à controladora desse

processo, bem como lifelines referentes às classes de entidade **Contato** e **Ligaçāo**.

Também se pode perceber que a modelagem da interação se inicia por meio de um fragmento combinado do tipo **alt**, contendo duas alternativas: a primeira enfoca o cenário em que o usuário possui o número do contato para quem ele deseja ligar armazenado no aparelho, enquanto a segunda enfoca o cenário no qual o usuário não possui armazenado o número para o qual ele quer ligar.

No primeiro cenário, o usuário solicita a consulta de seus contatos. Esse evento é notificado à controladora pela interface e esta, em resposta, dispara o método **listarContato** na lifeline da classe **Contato** para retornar cada contato armazenado. Como isso pode se repetir muitas vezes, inserimos um fragmento combinado do tipo **loop** para representar esse laço. De posse das informações dos contatos, a controladora ordena à interface que apresente os contatos recuperados.

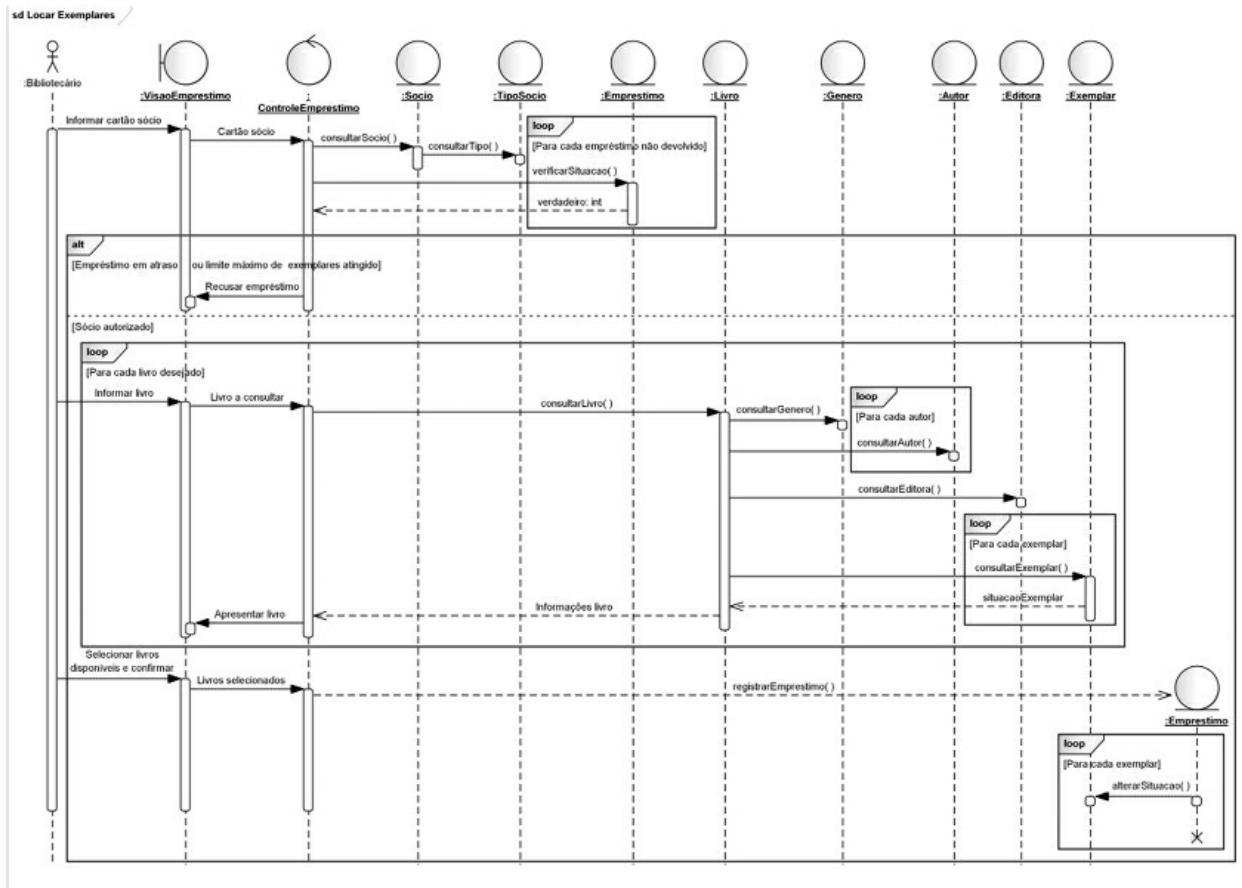
O usuário, então, selecionará o contato desejado. Quando isso ocorrer, a interface notificará a controladora, que, por sua vez, consultará o número do contato por meio do método **consultarContato**, que retornará o número do contato na forma de um valor **long**.

No segundo cenário, em que o usuário não possui o número armazenado, ele simplesmente o informará à interface.

Após a escolha entre as duas alternativas, o usuário confirma a ligação. A lifeline da interface repassa esse acontecimento para a controladora e esta dispara o método **realizarLigacao**, gerando um novo objeto da classe **Ligacao** que armazenará o número da ligação, bem como a data e a hora em que foi realizada.

## 7.13 Exemplo de Diagrama de Sequência – Processo de Locação de Exemplares para o Sistema de Biblioteca

A seguir, daremos continuidade à modelagem do sistema de biblioteca, apresentando o diagrama de sequência equivalente ao processo de locação de exemplares (Figura 7.30).



*Figura 7.30 – Processo de Locação de Exemplares.*

Nesta interação existem muitas lifelines, como o leitor pode observar, indo da lifeline do ator Bibliotecário, passando pela interface e controladora e incluindo diversas lifelines de entidade.

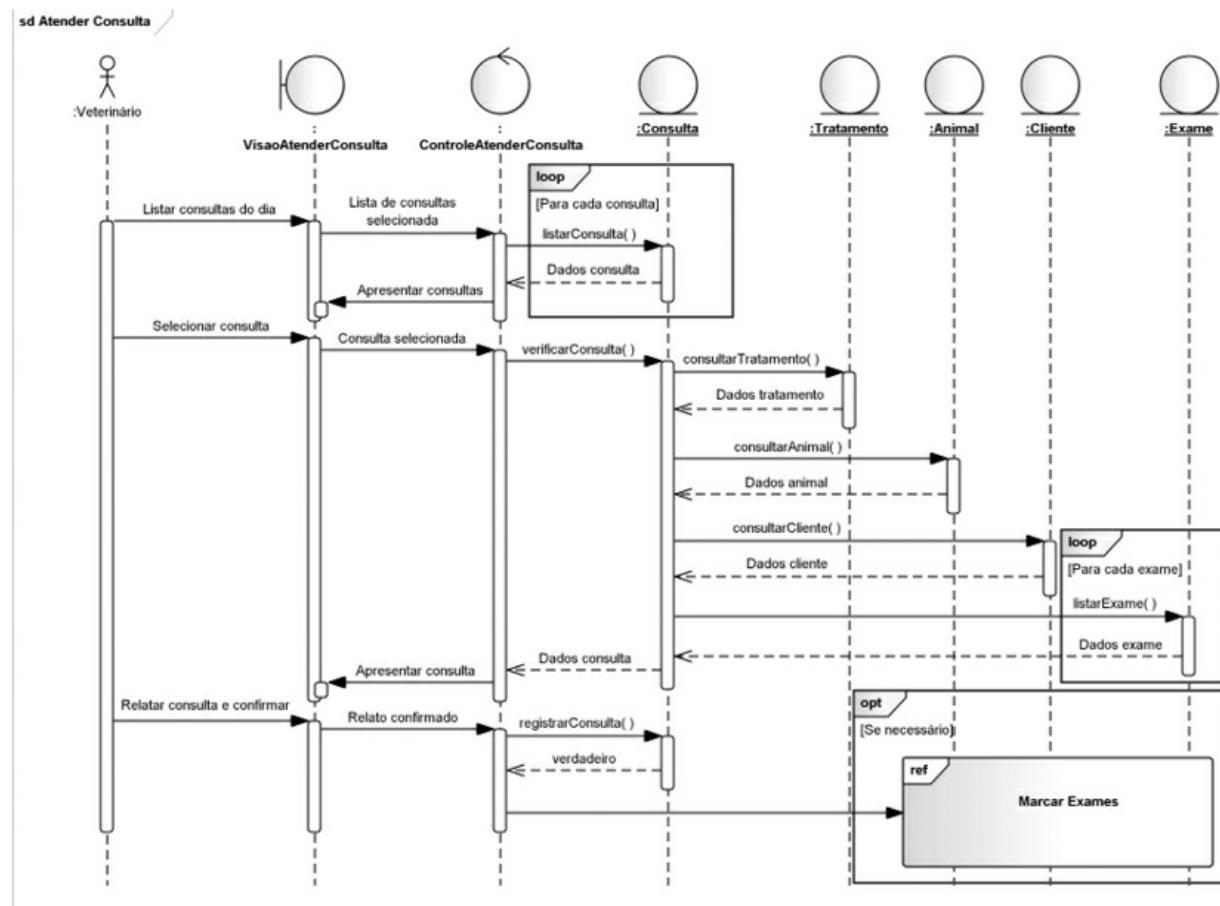
O processo se inicia com o bibliotecário informando o cartão do sócio, o que acarreta o disparo de vários métodos com o objetivo de consultar o sócio e seu tipo e verificar a situação de empréstimos anteriores. Caso haja algum empréstimo em atraso ou o número máximo de exemplares que o sócio pode locar tiver sido atingido, o sócio será informado de que não pode mais locar livros, o que será demonstrado por um fragmento combinado do tipo **alt**.

Caso o sócio esteja autorizado a solicitar empréstimos, poderá consultar os livros que deseja. Para cada livro desejado, o sistema disparará métodos para consultar o livro, seu gênero, seus autores, sua editora e a situação de cada exemplar do livro escolhido. Essas informações serão apresentadas pela interface a pedido da controladora.

Caso haja exemplares dos livros pesquisados disponíveis, será registrado um novo empréstimo. Observe que uma nova lifeline da classe **Emprestimo** é gerada, por meio do método **registrarEmprestimo**, e a situação de cada exemplar incluído no empréstimo será alterada para indicar que os exemplares estão locados por meio do método **alterarSituacao**.

## 7.14 Exemplo de Diagrama de Sequência – Processo de Atendimento de Consulta para o Sistema de Clínica Veterinária

A seguir, apresentamos o processo de atendimento de consulta do sistema de veterinária, por meio de um diagrama de sequência (Figura 7.31).



*Figura 7.31 – Processo de Atendimento de Consulta.*

Essa interação se inicia com o veterinário solicitando a listagem das consultas agendadas para o dia. Essa listagem é recuperada por meio do método **listarConsulta**.

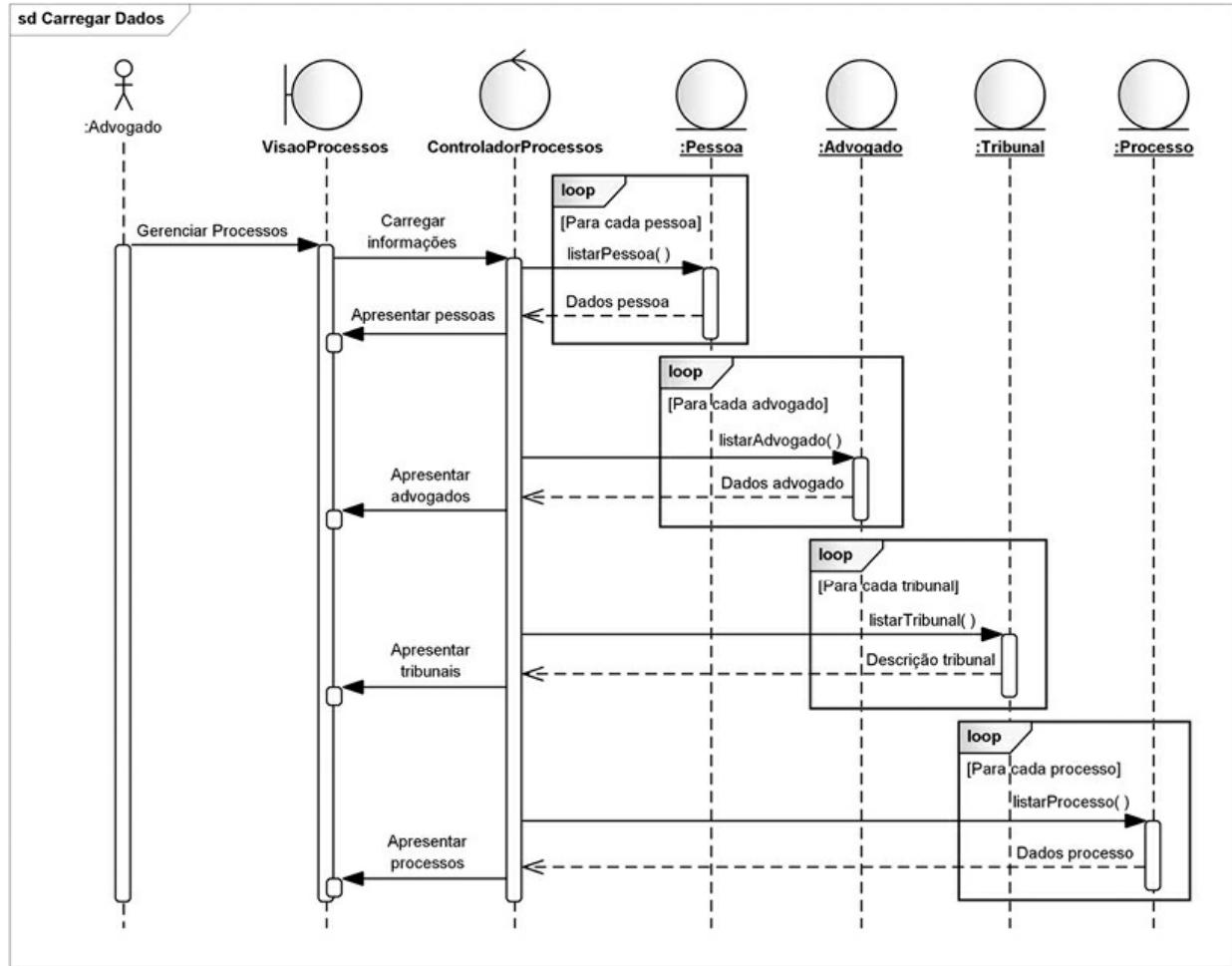
Com base na listagem, o veterinário seleciona a consulta desejada. Isso

acarreta o disparo de diversos métodos para consultar o tratamento ao qual a consulta se refere, o animal que está sendo tratado, o cliente dono do animal e os possíveis exames associados à consulta.

Após atender o animal, o veterinário relata o que foi realizado e confirma tal informação. Tal procedimento faz o método `registrarConsulta` atualizar a consulta. Se forem pedidos exames, então será feita uma referência ao processo de `Marcar Exames` para registrar cada exame solicitado.

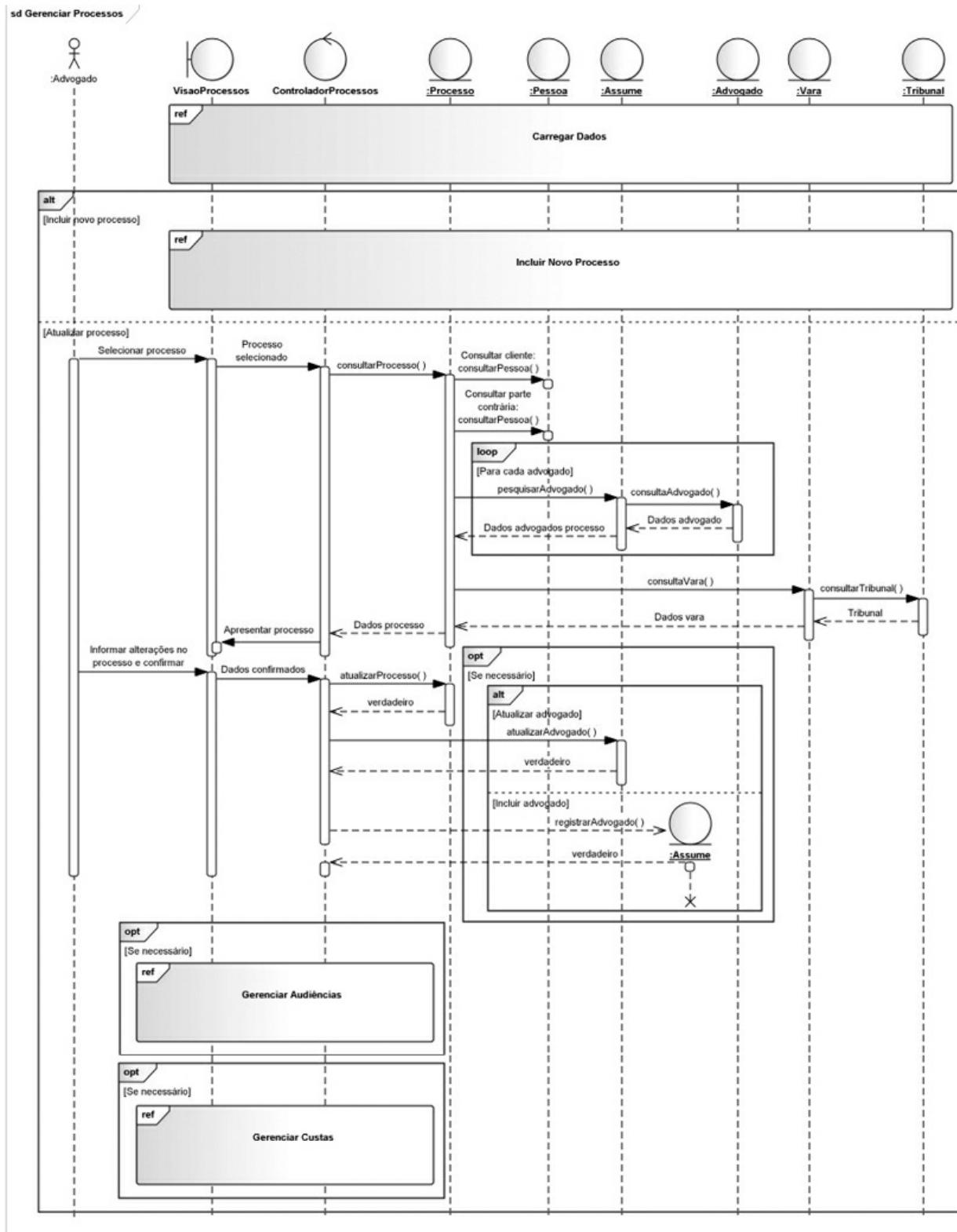
## **7.15 Exemplo de Diagrama de Sequência – Funcionalidade para Gerenciamento de Processos do Sistema de Controle de Advocacia**

A seguir, apresentamos os diagramas de sequência para o processo de gerenciamento de processos do sistema de controle de advocacia. Como esse processo é extenso, foi dividido em três fragmentos de interação. O diagrama de sequência referente ao processo principal faz referência a outros dois fragmentos. Apresentaremos, primeiramente, o subprocesso para carregar os dados necessários à interação (Figura 7.32).



*Figura 7.32 – Carregar Dados.*

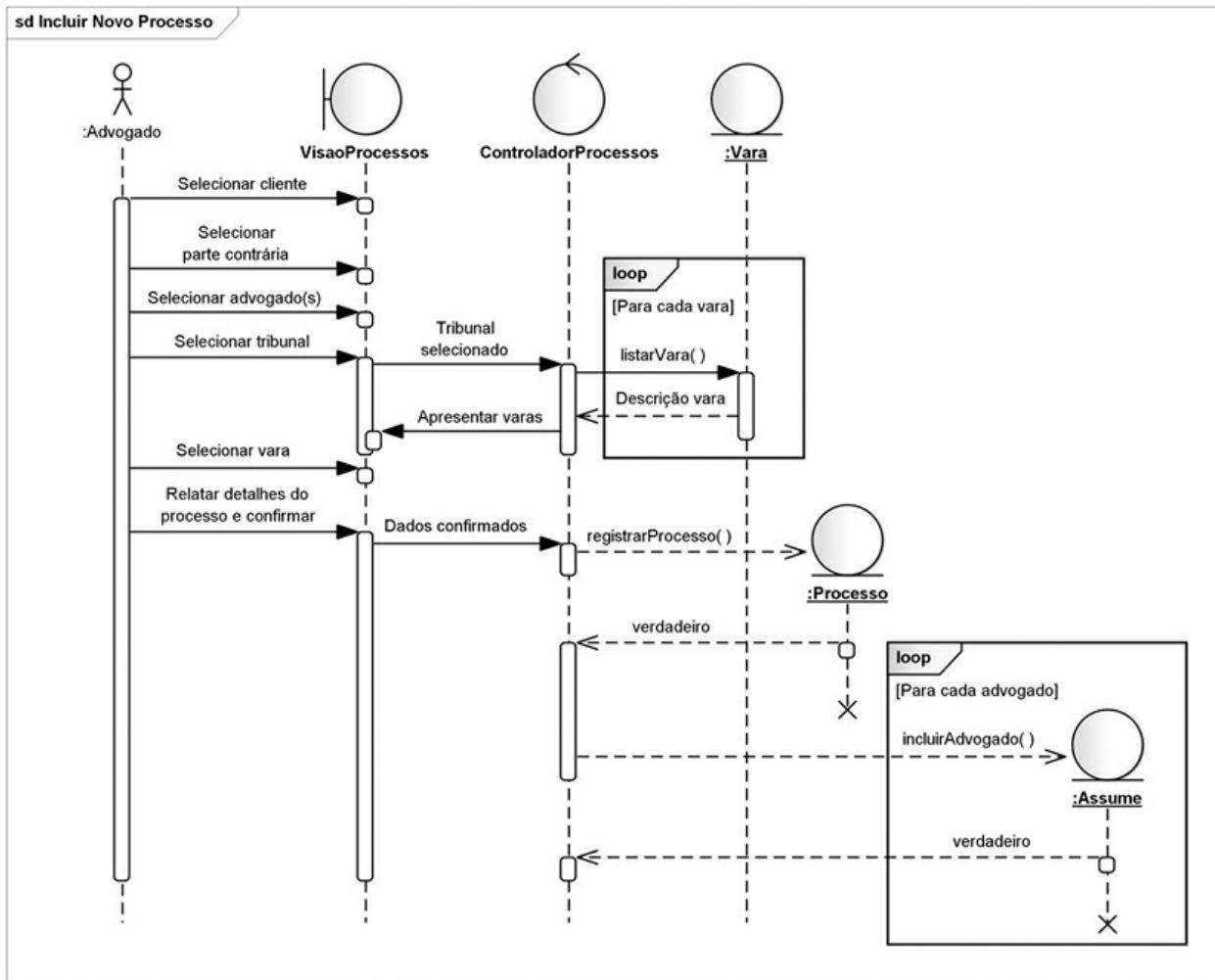
Esse diagrama foi produzido apenas para diminuir o tamanho do diagrama principal de gerenciamento de processos. Sua função, como seu próprio nome indica, é carregar os dados utilizados no gerenciamento do processo. Dessa forma, nesse fragmento de interação, quando o advogado solicita a funcionalidade de gerenciamento de processos, são disparados métodos para listar todas as pessoas, advogados ativos, tribunais e processos em andamento mantidos pelo sistema. A figura 7.33 apresenta o fragmento de interação principal referente ao gerenciamento de processos.



*Figura 7.33 – Processo de Gerenciar Processos.*

Como o leitor pode perceber, esse fragmento se inicia com uma referência

ao fragmento de interação “Carregar Dados”, que já foi explicado, demonstrando que essa interação é executada antes de mais nada. Depois, há um fragmento combinado do tipo **alt**, que identifica os dois cenários principais dessa interação. O primeiro cenário representa a situação em que se deseja incluir um novo processo. Esse cenário faz referência ao fragmento de interação “Incluir Novo Processo”, que será explicado na figura 7.34.



*Figura 7.34 – Processo de Inclusão de um Novo Processo.*

Já o segundo cenário demonstra a situação em que se deseja alterar um processo já existente. Nesse cenário, o advogado seleciona o processo que deseja alterar e, ao ser notificada do evento, a controladora em resposta dispara métodos para retornar as informações do processo, além das informações do cliente e da parte contrária, dos advogados que já assumiram o processo, bem como da vara e do tribunal onde o processo

tramita.

Após receber essas informações, o advogado faz as alterações necessárias e confirma isso. Esse evento faz a controladora atualizar os dados do processo, por meio do método **atualizarProcesso** e, se necessário, alterar a situação de um dos advogados envolvidos no processo (que pode deixar de atuar no processo, por exemplo) ou incluir um novo advogado no processo. Observe que essa possibilidade está modelada por meio de um fragmento combinado do tipo **opt**, contendo, por sua vez, um fragmento combinado do tipo **alt**, que identifica as duas possibilidades. Note, ainda, que há duas lifelines da classe **Assume** envolvidas por esse fragmento. Essa classe identifica todos os advogados que já assumiram o processo em questão. Assim, quando se altera a situação de um determinado advogado no processo, é disparado o método **atualizarAdvogado** em uma lifeline da classe **Assume** já existente, enquanto quando se deseja incluir um novo advogado no processo, é chamado o método **registrarAdvogado**, que cria uma nova lifeline da classe.

Para finalizar a interação, há dois fragmentos combinados do tipo **opt** representando cenários em que são referenciadas as interações para gerenciar as audiências ou as custas do processo, caso isso seja necessário. A figura 7.34 detalha o fragmento de interação para incluir um novo processo.

Nesse fragmento de interação, o advogado seleciona o cliente, a parte contrária, os advogados e o tribunal onde o processo será julgado. Todas essas informações já se encontram na interface do sistema, tendo sido carregadas durante a execução do fragmento de interação “Carregar Dados” já explicado. Quando a controladora é notificada da escolha do tribunal, no entanto, solicita que sejam consultadas todas as varas do tribunal selecionado, por meio do método **listarVara**. Como essa pesquisa pode se repetir muitas vezes, foi modelada dentro de um fragmento combinado do tipo **loop**.

Depois, o advogado seleciona a vara em que o processo tramitará, descreve os detalhes do processo e confirma tais informações. Isso faz com que a controladora dispare o método **registrarProcesso** para gerar uma nova lifeline da classe **Processo**. Em seguida, para cada advogado atribuído ao processo, a controladora disparará o método

`incluirAdvogado` para criar uma nova lifeline da classe `Assume`, que contém as informações referentes aos advogados associados a cada processo. Essa parte da interação está contida em um fragmento combinado do tipo `loop`, uma vez que se trata de um laço.

## 7.16 Exercícios Propostos

Aqui, será dada continuidade aos exercícios propostos anteriormente, retomando a modelagem dos sistemas iniciados nos capítulos anteriores. Abordaremos em cada exercício apenas um dos processos considerados mais importante em cada sistema.

### 7.16.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

Desenvolva o diagrama de sequência para o processo de venda de ingressos de cinema, de acordo com a documentação do caso de uso referente a esse processo e das seguintes declarações:

- Quando a opção de venda de ingressos for selecionada pelo funcionário, o sistema deverá carregar todas as sessões ainda não encerradas, detalhando o horário, o filme apresentado e o número da sala.
- Quando o funcionário escolher a sessão a que o cliente deseja assistir, o sistema apresentará os assentos disponíveis.
- O funcionário deverá, então, informar os assentos desejados pelo cliente, bem como se os ingressos são inteiros ou meias-entradas. O sistema em resposta gerará os ingressos de acordo com as escolhas do cliente.

### 7.16.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade

Desenvolva o diagrama de sequência referente ao processo de pagamento de mensalidade do sistema de controle de clube social, de acordo com a documentação do caso de uso desse processo e das seguintes especificações:

- O sócio deve informar o número de seu cartão ao atendente, que consultará no sistema a mensalidade do mês a ser pago e, se existirem, as possíveis mensalidades em atraso, já com o valor acrescido de juros até o

dia da consulta.

- O sócio, então, poderá escolher quais mensalidades pagar, caso haja mais de uma. O clube exige que sejam pagas primeiro as mensalidades com mais atraso.
- Ao ser realizado o pagamento, o atendente quitará as mensalidades em questão.

### **7.16.3 Sistema de Locação de Veículos – Processo de Locação de Veículo**

Desenvolva o Diagrama de Sequência para o processo de locação de veículo do sistema de controle de aluguel de automóveis, levando em consideração as determinações a seguir, bem como a documentação do caso de uso associado a esse processo.

- Primeiramente, o funcionário deve selecionar o cliente que está locando o automóvel em uma lista. Para isso, ao selecionar a opção locação, o sistema deve carregar todos os clientes da empresa.
- Em seguida, o funcionário deve informar o veículo que o cliente deseja locar, selecionando o automóvel em uma lista, que também foi carregada pelo sistema quando o processo foi iniciado.
- Ao selecionar o veículo desejado, o sistema apresentará detalhes do automóvel, como ano, cor e quilometragem, além do modelo e marca do veículo.
- Finalmente, caso o cliente queira realmente locar o veículo selecionado, ele informará o período em que o locará e para qual finalidade. Isso gerará uma fatura de locação, que o cliente deverá pagar para concluir a locação.

### **7.16.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão**

Desenvolva o diagrama de sequência para o processo de **Realizar Leilão** do sistema de leilão via internet, de acordo com a documentação do caso de uso em que se baseia essa interação e das seguintes informações:

- Ao selecionar a opção realizar leilão, o leiloeiro faz o sistema selecionar todos os leilões não encerrados, apresentando-os ao leiloeiro.

- O leiloeiro deve, então, escolher o leilão que deseja realizar, o que faz o sistema carregar todos os itens a serem leiloados.
- Enquanto houver itens a ser leiloados, o leiloeiro deverá buscar o próximo item a ser leiloadado e apresentá-lo na página do leilão, destacando a foto do item, uma breve descrição do produto e o lance mínimo para arrematá-lo.
- Em seguida, o leiloeiro deve aguardar um determinado tempo. Os lances serão ofertados pelos participantes.
- Cada lance que suplante o anterior deve ser anunciado, destacando o participante que o fez. Sempre que um lance suplantar o anterior, deverá ser registrado.
- Quando os lances se esgotarem, depois de um tempo de espera após a oferta do último lance, o vencedor será anunciado e o item, marcado como arrematado.
- Ao concluir-se os itens, o leiloeiro deve finalizar o leilão.

### **7.16.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias**

Desenvolva o diagrama de sequência para o processo de pagamento das diárias para o sistema de hotelaria, considerando a documentação do caso de uso dessa interação e, ainda, as seguintes declarações:

- O hóspede se dirige ao funcionário e informa os quartos que deseja pagar.
- O funcionário, por meio do sistema, deve, então, buscar todas as diárias ainda não pagas relativas ao quarto e apresentá-las ao hóspede.
- Ao realizar o pagamento, as diárias serão quitadas, podendo o hóspede permanecer no hotel ou encerrar sua estada.
- Caso o hóspede tenha solicitado algum serviço ou consumido algum item de frigobar, deverá pagá-los igualmente.

### **7.16.6 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel**

Desenvolva o diagrama de sequência referente ao processo de venda de um imóvel para o sistema de controle de imobiliária, considerando a

documentação do caso de uso desse processo e sabendo que:

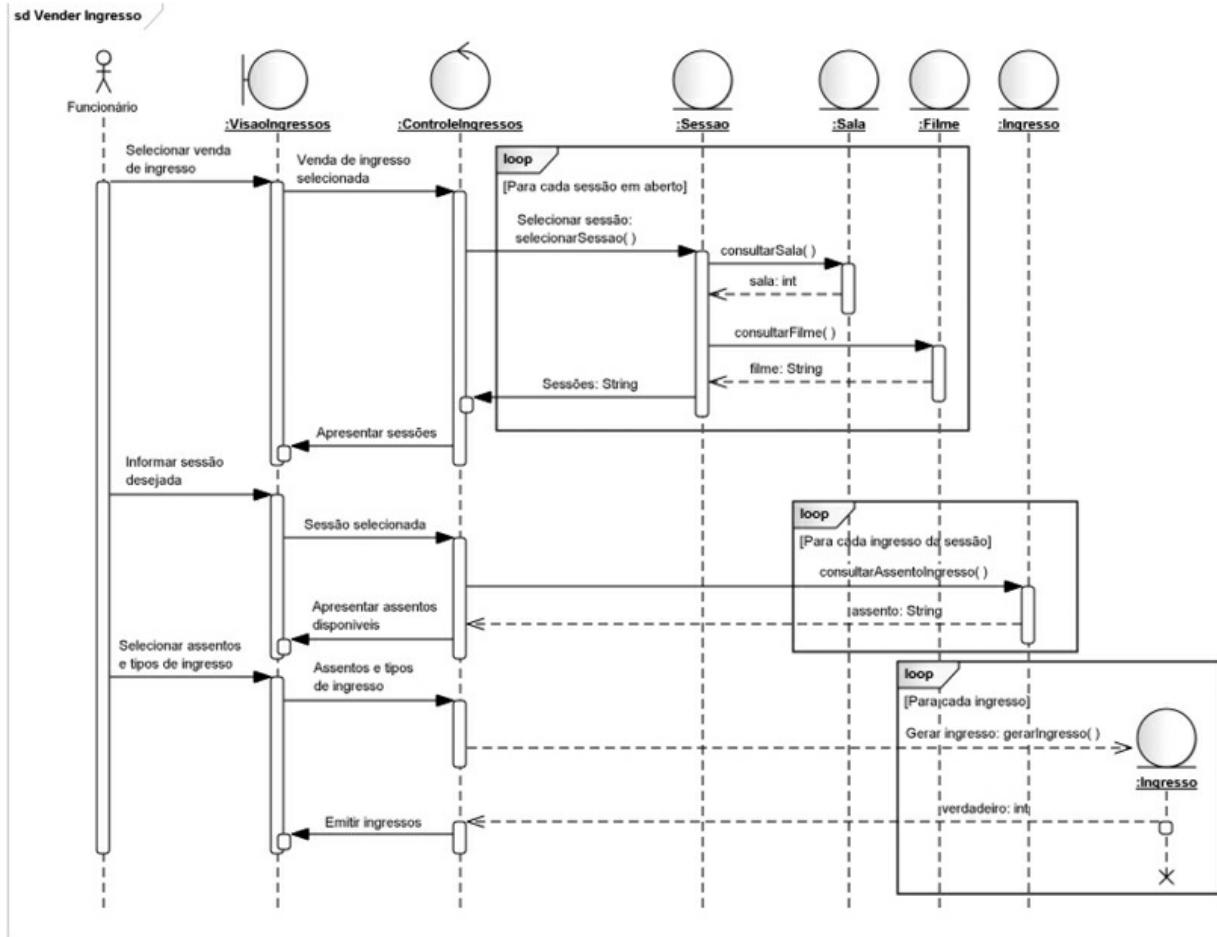
- o corretor deve selecionar o tipo de imóvel e consultar os imóveis do tipo selecionado que estão à venda;
- com os imóveis à venda, o sistema deve listar os clientes da imobiliária;
- o corretor deve, então, selecionar o imóvel que será vendido e seu comprador, informar as taxas da transação (impostos, tarifas de cartório etc.), o valor pago pelo imóvel e confirmar a venda;
- o sistema em resposta calculará as comissões da imobiliária e do corretor, registrará a venda e todas as taxas pagas durante a transação.

## 7.17 Solução dos Exercícios

A seguir, apresentaremos as soluções relativas a cada exercício proposto na seção anterior. Sugerimos que o leitor acompanhe a explicação de cada exercício, consultando o diagrama de casos de uso e o diagrama de classes de cada um dos sistemas para ter uma compreensão mais completa da solução.

### 7.17.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

Podemos visualizar o detalhamento desse processo na figura 7.35. Nesse processo, participam o ator Funcionário e lifelines da classe de fronteira e de controle, além de lifelines das classes de entidade **Sessão**, **Sala**, **Filme** e **Ingresso**. O processo será explanado a seguir.



*Figura 7.35 – Processo de Venda de Ingresso.*

Nesse processo, o funcionário seleciona a opção de venda de ingresso na interface do sistema. Esse evento é repassado à controladora, que iniciará um laço, representado pelo fragmento combinado do tipo **loop**, no qual, para cada sessão ainda não encerrada, será disparado o método **selecionarSessao**. Esse método consulta a sala associada à sessão por meio do método **consultarSala** e, em seguida, o filme nela apresentado, por meio do método **consultarFilme**. Essas informações serão, então, retornadas à controladora, que ordenará sua apresentação na interface.

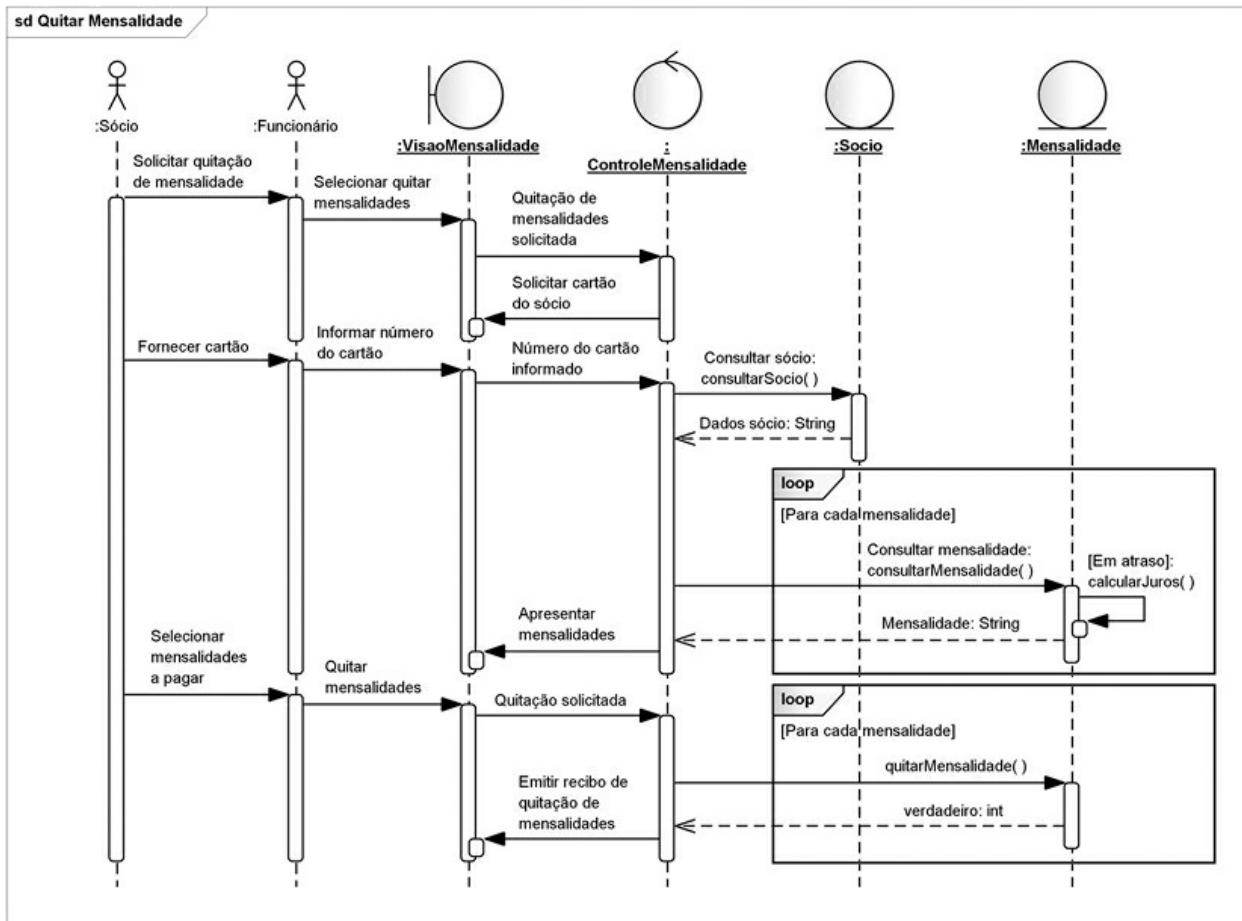
Com base nessas informações, o funcionário informará a sessão desejada pelo cliente e confirmará. Isso fará com que a interface informe à controladora a sessão escolhida. Em resposta a isso, a controladora pesquisará todos os ingressos já vendidos para a sessão selecionada para determinar quais assentos ainda estão disponíveis. Observe que é disparado o método **consultarAssentoIngresso** para retornar o atributo

contendo o assento de cada ingresso. Ao receber essas informações, a controladora pede que a interface apresente os assentos disponíveis.

Depois, o funcionário informará os assentos desejados pelo cliente, bem como o tipo de cada ingresso (se inteiro ou meia-entrada). Ao ser notificada desse evento, a controladora em resposta solicitará a geração de uma nova lifeline da classe **Ingresso** para cada ingresso desejado pelo cliente, por meio da chamada do método **gerarIngresso**. Observe que a geração de ingressos foi representada dentro de um laço representado por um fragmento combinado do tipo **loop**. Se o método for concluído com sucesso, a controladora pedirá que a interface emita os ingressos solicitados que deverão ser entregues ao cliente pelo funcionário.

## 7.17.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade

Esse processo está detalhado na figura 7.36. Aqui, participam os atores Sócio e Funcionário, com esse último representando os funcionários do clube. No processo, interagem também as lifelines das classes de fronteira e de controle, além de lifelines das classes de entidade **Socio** e **Mensalidade**. O processo será detalhado a seguir.



*Figura 7.36 – Processo de Pagamento de Mensalidade.*

Nesse processo, o sócio procura um funcionário e lhe informa que deseja quitar mensalidades. O funcionário, então, solicita esse serviço à interface, que repassa o pedido à controladora e esta, em resposta, pede que a interface apresente um formulário no qual se deve inserir o número do cartão do sócio.

O sócio, então, fornecerá o número de seu cartão, que será repassado ao controlador. Em posse desse número, a controladora chamará o método **consultarSocio** para consultar o sócio em questão. Esse método, caso o sócio seja encontrado, retornará os dados dele. Em seguida, a controladora consultará as mensalidades do sócio por meio do disparo do método **consultarMensalidade** em cada lifeline associada ao sócio consultado, como demonstra o fragmento combinado do tipo **loop**. Dentro desse laço existe ainda um teste, determinado por uma condição de guarda (texto entre colchetes), que estabelece que se a mensalidade está em atraso, deve-se disparar o método **calcularJuros**. Observe que esta é uma

autochamada. O conteúdo de cada mensalidade será, então, retornado à controladora, que as enviará para serem apresentadas na interface.

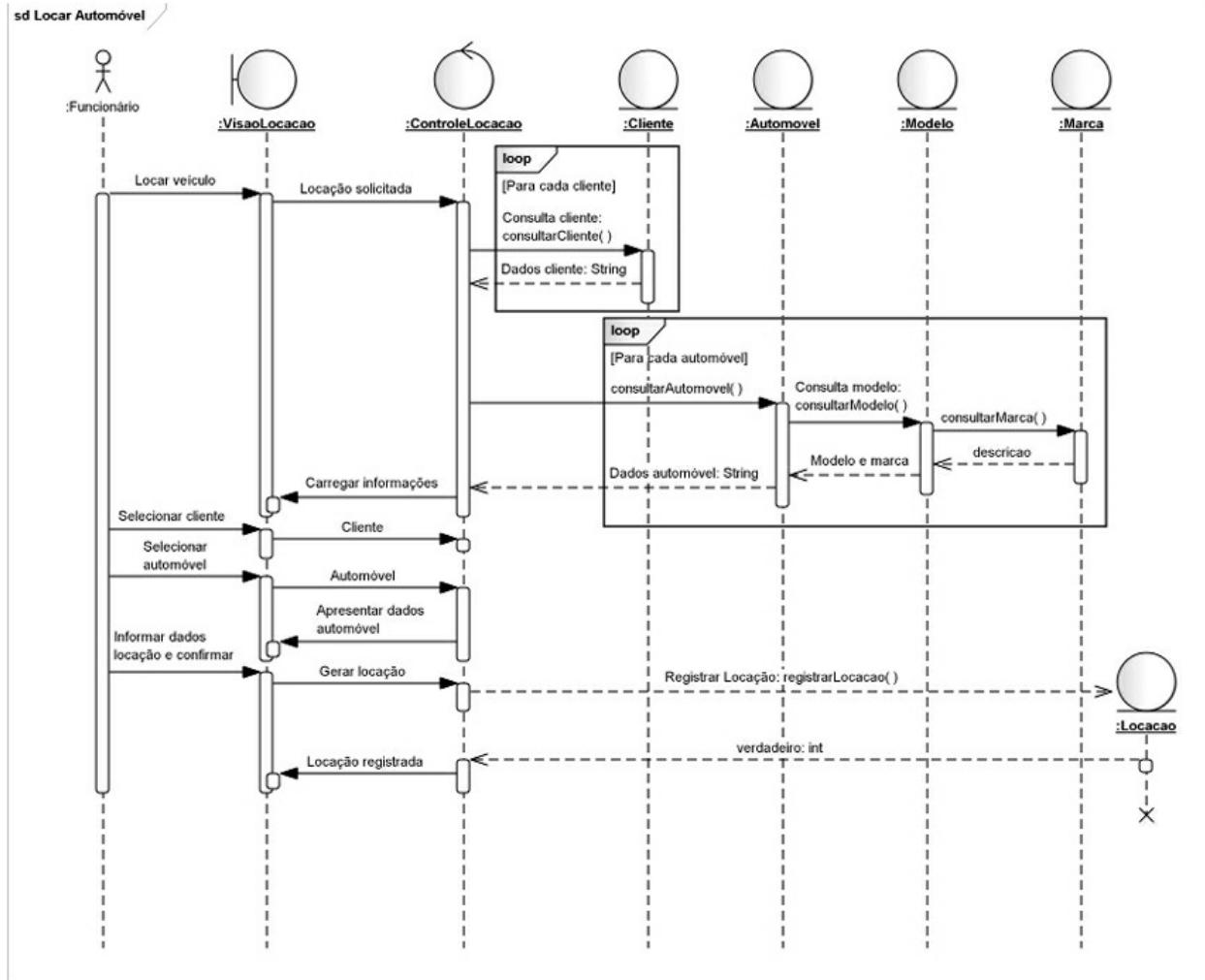
Com base nas informações apresentadas, o sócio escolherá quais mensalidades deseja pagar e o funcionário mandará quitá-las através da interface, que as repassará à controladora. Ela, então, disparará o método **quitarMensalidade** para cada uma das mensalidades selecionadas, conforme demonstra o fragmento combinado do tipo loop, definindo-as como quitadas. Depois de receber o retorno de que as mensalidades foram quitadas, a controladora mandará a interface emitir o recibo de quitação.

### 7.17.3 Sistema de Locação de Veículos – Processo de Locação de Veículo

Podemos visualizar a solução para esse exercício na figura 7.37. Deste processo, participa somente o ator **Funcionário** (poderíamos ter colocado também o ator **Cliente**, mas o diagrama ficaria largo demais desnecessariamente). Nesse processo interagem também as lifelines da classe de fronteira e de controle associadas a essa interação, além de lifelines das classes de entidade **Cliente**, **Automovel**, **Modelo**, **Marca** e **Locacao**. A seguir, explicaremos a solução.

O processo se inicia quando o funcionário seleciona a opção de locar veículo na interface. Esse evento é repassado pela interface à controladora, que disparará o método **consultarCliente** para consultar cada cliente registrado no sistema, conforme demonstra o fragmento combinado do tipo **loop**. Após a execução desse laço, a controladora terá recebido os dados de todos os clientes registrados.

Em seguida, a controladora inicia outro laço para selecionar todos os automóveis disponíveis para locação, disparando o método **consultarAutomovel** para cada automóvel em condições de locação, conforme demonstra o fragmento combinado do tipo **loop**. Esse método consulta o modelo do automóvel consultado, por meio do método **consultarModelo**, e este, por sua vez, consulta a marca desse modelo por meio do método **consultarMarca**. Esses dados são, então, retornados à controladora, que os manda apresentar na interface com o formulário de locação de veículos.



*Figura 7.37 – Processo de Locação de Veículo.*

O funcionário seleciona o cliente que deseja locar o veículo e, em seguida, o veículo desejado. Isso é feito diretamente na interface do software. A seguir, o funcionário deve informar os dados da locação, como o período de locação, para qual finalidade e para onde o cliente pretende ir. A confirmação da locação na interface fará com que esta avise a controladora da ocorrência desse evento, o que fará com que a controladora dispare o método `registrarLocacao` para registrar a nova locação, instanciando uma nova lifeline da classe `Locacao` e, caso o método seja executado com sucesso, a controladora mandará a interface informar que a locação foi realizada e o veículo está liberado.

#### 7.17.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão

Na figura 7.38, podemos visualizar o processo de **Realizar Leilão**. O processo envolve os atores **Leiloeiro** e **Participante** – esse último representa todos os possíveis participantes do leilão. Além disso, o processo envolve lifelines da classe de fronteira e de controle associadas a essa interação, bem como lifelines das classes de entidade **Leilao**, **ItemLeilao** e **Lance**. A seguir, detalharemos os passos desse processo.

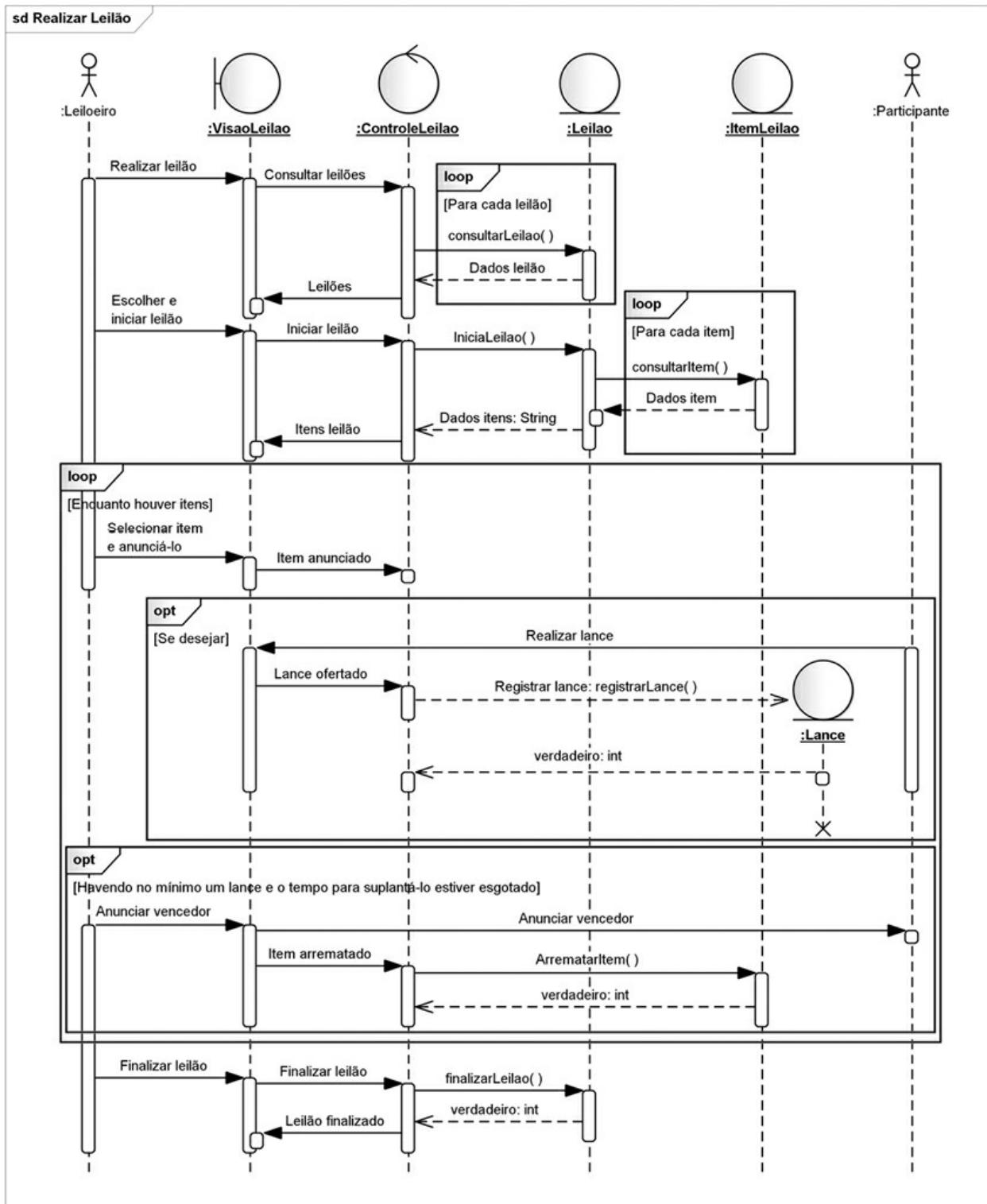


Figura 7.38 – Processo de Realizar Leilão.

O leiloeiro inicia o processo selecionando a opção **Realizar Leilão** na interface do sistema, o que faz com que esta avise a controladora de que é necessário apresentar todos os leilões ainda não encerrados para o leiloeiro.

A controladora, então, dispara o método **consultarLeilao** em todos os objetos da classe **Leilao** que ainda não foram encerrados, para retornar seus dados. Observe que como esse método deve ser disparado em muitos objetos, esse laço foi representado por meio de um fragmento combinado do tipo **loop**.

Ao receber essa listagem, o leiloeiro deve escolher um leilão e iniciá-lo. Essa ação faz a interface avisar o controlador desse evento, causando o disparo do método **iniciaLeilao** em uma lifeline da classe **Leilao**. Esse método precisa selecionar todos os itens da classe **ItemLeilao** associados ao objeto escolhido da classe **Leilao**. Assim, ele dispara o método **consultarItem** em cada objeto da classe **ItemLeilao** associado ao objeto **Leilao** selecionado, como demonstra o fragmento combinado do tipo **loop**. Os itens do leilão iniciado são, então, apresentados pela interface ao leiloeiro.

A partir desse momento, o processo entra em um laço que será somente concluído quando não houver mais itens a leiloar, como podemos perceber ao observarmos o fragmento combinado do tipo **loop** que envolve todos os componentes do processo e contém a condição de guarda “Enquanto houver itens”.

Assim, enquanto existirem itens, o leiloeiro selecionará um deles de acordo com a ordem e irá anunciá-lo por meio da interface. Como se trata de um leilão via internet, todos os participantes serão notificados de qual item está sendo anunciado e poderão, se assim o desejarem, oferecer lances.

O fragmento combinado do tipo **opt**, logo a seguir, modela a situação em que o participante deseja oferecer um lance para um item. O lance oferecido deve ser repassado pela interface à controladora, que disparará o método **registrarLance** para registrá-lo, criando uma nova lifeline da classe **Lance**.

Já o segundo fragmento combinado do tipo **opt** modela a situação em que, tendo ocorrido ao menos um lance e o tempo para suplantá-lo estiver esgotado, o leiloeiro deverá anunciar o participante vencedor, através da interface, que se encarregará de comunicar ao participante que este venceu, bem como anunciar isto na página. Ao mesmo tempo, ela notificará isso à controladora e esta se encarregará de definir o item como arrematado, por

meio do método `arrematarItem`.

Finalmente, quando todos os itens tiverem sido ofertados, o leiloeiro solicitará o encerramento do leilão à interface, que repassará o pedido à controladora, causando o disparo do método `finalizarLeilao`, que definirá o leilão como encerrado.

### 7.17.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias

A figura 7.39 apresenta a solução desse problema. Além do ator funcionário, o diagrama conta com instâncias da classe de fronteira e de controle relativas a essa interação, bem como lifelines das classes de entidade **Quarto**, **Ocupa**, **Hospede** e **Diaria**. A seguir, detalharemos os passos desse processo.

Nesse processo, o funcionário informa o número do quarto e solicita o serviço de quitação de diárias à interface do sistema, a qual repassa o número do quarto para a controladora que, em resposta, dispara o método `selecionarDiarias` em um objeto da classe **Quarto** para selecionar as diárias devidas pelo hóspede.

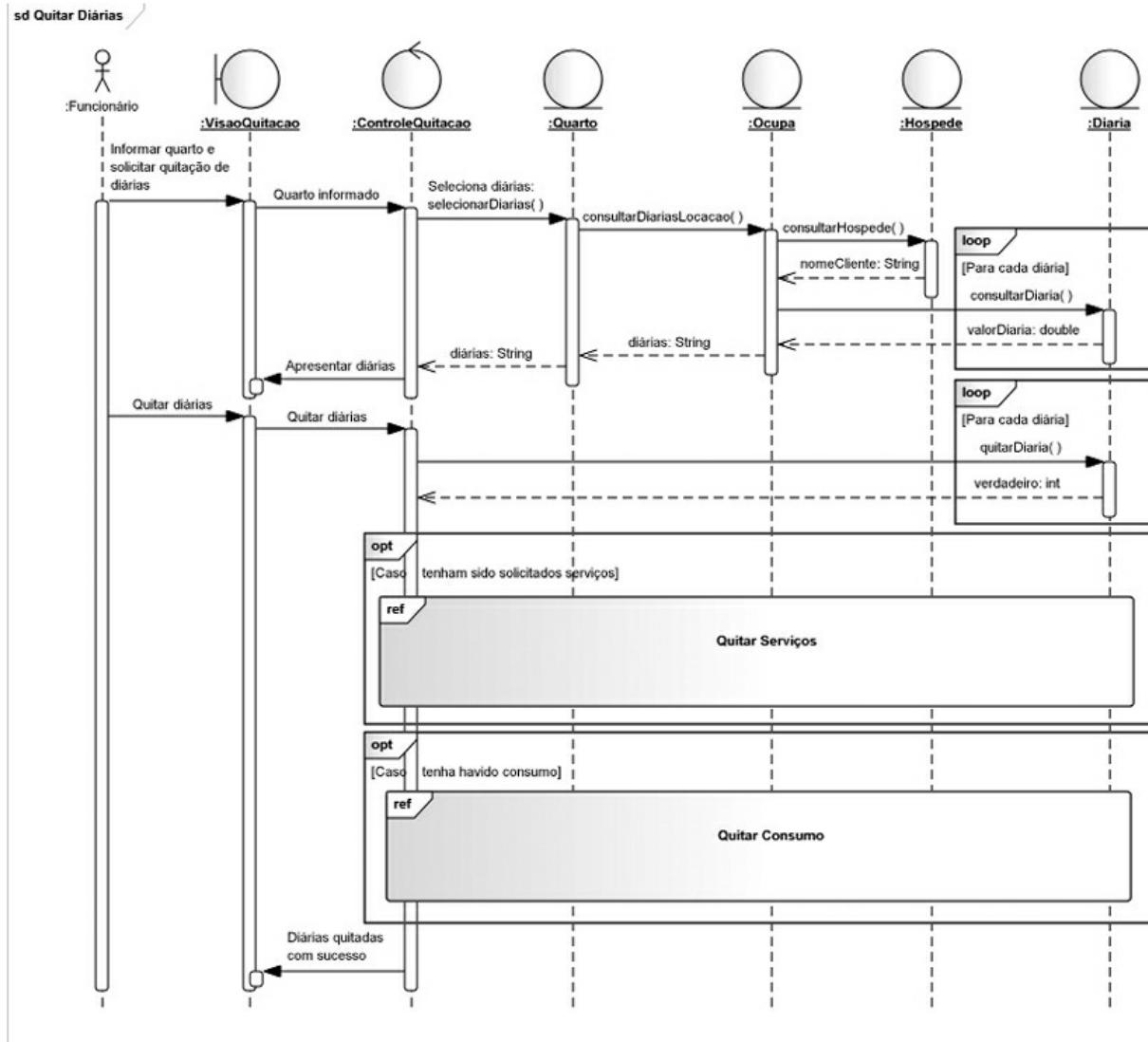


Figura 7.39 – Processo de Pagamento de Diárias.

O método **selecionarDiarias** deve retornar o nome do hóspede que alugou o quarto (o que não significa que seja o mesmo que o está ocupando) com as diárias devidas. Para isso, é necessário buscar informações contidas nos objetos **Ocupa**, **Hospede** e **Diaria**. Assim, esse método primeiramente dispara o método **consultarDiariasLocacao** em um objeto da classe **Ocupa**, para que este, por sua vez, consulte o hóspede ao qual está associado, por meio do método **consultarHospede** e, depois de obter seu retorno na forma de uma **String** contendo o nome do hóspede, dispare o método **consultarDiaria** em cada objeto da classe **Diaria** a ele associado, como demonstra o fragmento combinado do tipo **loop**, retornando o valor da diária. De posse desses valores, o método

`consultarDiariasLocacao` os retornará na forma de uma `String` para o método `selecionarDiarias` que o chamou, que, por sua vez, os retornará à controladora.

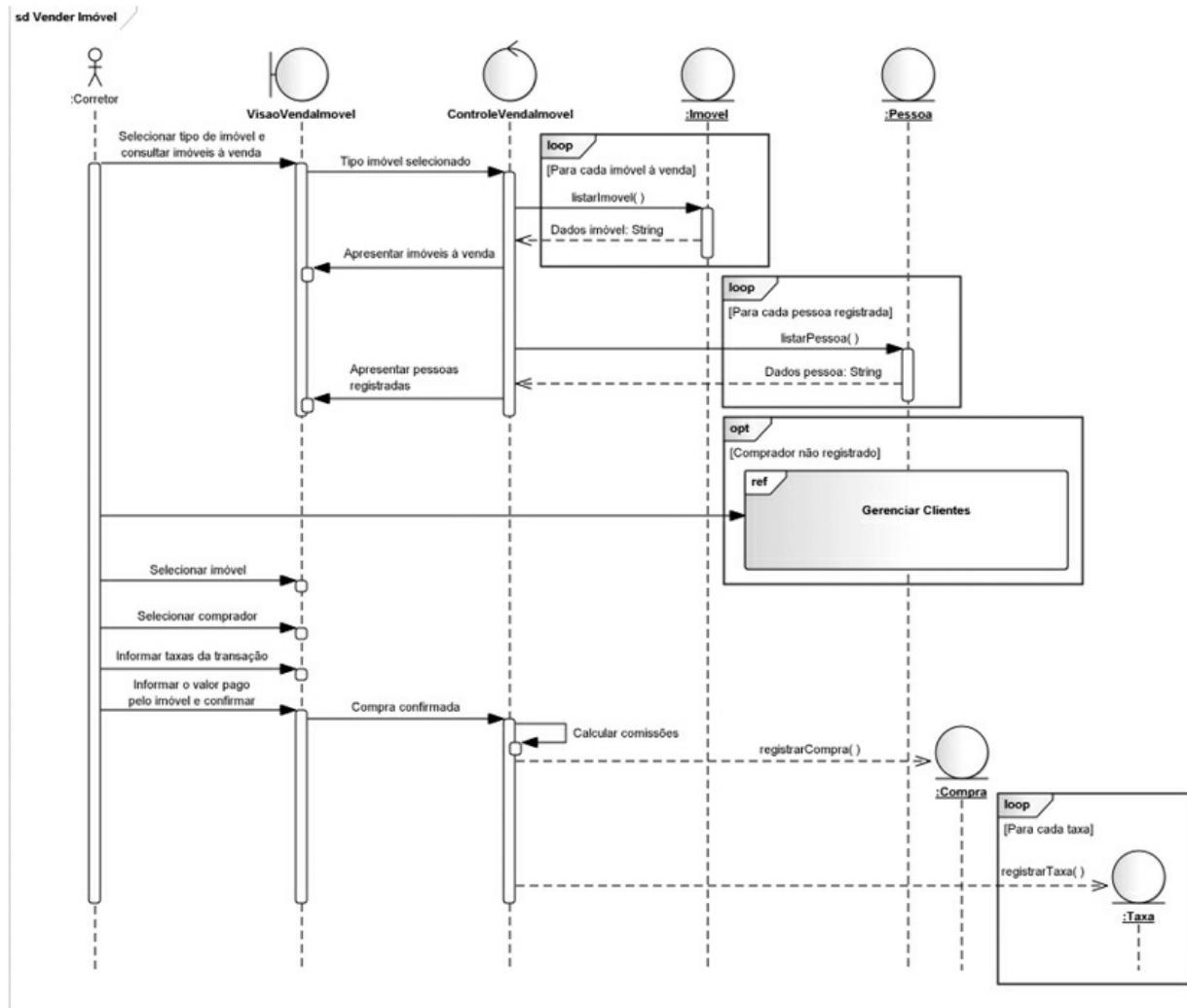
Ao obter esse relatório, o funcionário selecionará a opção de quitar diárias na interface, que repassará esse evento à controladora e esta, em resposta, disparará o método `quitarDiaria` em cada objeto associado ao quarto alugado, como demonstra o fragmento combinado do tipo `loop`. Esse método retornará verdadeiro para cada execução bem-sucedida.

Caso tenham sido solicitados serviços ou tenha havido consumo do frigobar, o hóspede deverá quitá-los também. Como os processos de **Quitar Serviços** e **Quitar Consumo** estão modelados em diagramas separados, foram apenas referenciados por meio de um uso de interação. Observe que a execução desses usos de interação depende de uma condição estabelecida por um fragmento combinado do tipo `opt` para cada uma delas, o que significa que poderão ser executadas ou não.

Finalmente, depois de quitar as diárias e, eventualmente, os possíveis serviços solicitados e/ou consumos ocorridos, o sistema apresentará a mensagem de que as diárias foram quitadas com sucesso.

### 7.17.6 Sistema de Controle de Imobiliária

A figura 7.40 ilustra a interação referente ao processo de venda de um imóvel. Essa interação modela lifelines do ator corretor, das classes de fronteira e de controle associadas a esse processo e lifelines das classes de entidade `Imovel`, `Pessoa`, `Compra` e `Taxa` (descritas no capítulo 4, sobre o diagrama de classes). A seguir, detalharemos os passos desse processo.



*Figura 7.40 – Processo de Venda de Imóvel.*

Essa interação se inicia com o corretor selecionando o tipo de imóvel que será vendido e consultando todos os imóveis desse tipo que estão à venda. A controladora em resposta a esse evento consulta todos os imóveis que satisfaçam a condição da pesquisa, bem como todos os clientes da imobiliária, e solicita à lifeline da classe de fronteira que apresente essas informações.

Caso o cliente não esteja cadastrado, então será feita uma referência ao processo de Gerenciar Clientes para registrá-lo. Como este é um cenário opcional, foi modelado dentro de um fragmento combinado do tipo **opt**.

Depois, o corretor seleciona o imóvel a ser vendido e o cliente que vai comprá-lo, bem como as taxas pagas e o valor total da venda do imóvel. A confirmação dessas informações faz com que a lifeline de fronteira informe

esse evento à controladora que, em resposta, calculará as comissões devidas e gerará uma nova lifeline da classe **Compra**, além de uma lifeline da classe **Taxa** para cada tarifa ou imposto pago durante a transação.

## CAPÍTULO 8

# Diagrama de Comunicação

O diagrama de comunicação era conhecido como diagrama de colaboração até a versão 1.5 da UML, tendo seu nome modificado para diagrama de comunicação a partir da versão 2.0. O diagrama está amplamente associado ao diagrama de sequência: na verdade, um complementa o outro. As informações mostradas no diagrama de comunicação são, com frequência, praticamente as mesmas apresentadas no diagrama de sequência, porém com um enfoque diferente, visto que esse diagrama não se preocupa com a temporalidade do processo, concentrando-se em como os elementos do diagrama estão vinculados e quais mensagens trocam entre si durante um processo.

Por ser muito semelhante ao diagrama de sequência, o diagrama de comunicação utiliza muitos de seus componentes, como atores e objetos, incluindo seus estereótipos de fronteira e controle. No entanto, os objetos no diagrama de comunicação não têm linhas de vida. Além disso, esse diagrama não suporta ocorrências de interação ou fragmentos combinados, como o diagrama de sequência, por isso é utilizado para a modelagem de processos mais simples ou com menos detalhamento.

Da mesma forma que no diagrama de sequência, um diagrama de comunicação enfoca um processo, normalmente baseado em um caso de uso. As semelhanças entre ambos são tão grandes que existem até mesmo ferramentas CASE capazes de gerar um dos diagramas a partir do outro. Nas seções seguintes, detalharemos o diagrama de comunicação.

### 8.1 Lifelines

As lifelines do diagrama de comunicação representam o mesmo que no diagrama de sequência, ou seja, participantes individuais de uma interação e, em geral, instâncias de classes que participam do processo. No entanto, diferentemente do diagrama de sequência, e apesar do nome, as lifelines

utilizadas por esse diagrama não têm linha de vida nem foco de controle, tendo a mesma representação utilizada no diagrama de objetos e as mesmas regras de nomenclatura, mas sem haver um detalhamento dos valores de seus atributos. Uma vez que, na prática, uma lifeline representa, em geral, um objeto, eventualmente poderemos usar essa nomenclatura. A figura 8.1 apresenta um exemplo de lifeline.



*Figura 8.1 – Exemplo de Lifeline.*

Como podemos notar, da mesma forma que no diagrama de sequência, o texto do retângulo (lifeline) informa o nome do objeto e o da classe à qual ele pertence, nessa ordem. As duas informações vêm separadas pelo símbolo de dois-pontos (:). Assim, a lifeline representada na figura chama-se **comum1** e é uma instância da classe **ContaComum**.

## 8.2 Vínculos

Um vínculo nada mais é do que uma instância de uma associação definida no diagrama de classes e identifica uma ligação entre duas lifelines envolvidas em um processo. Assim, a existência de um vínculo é caracterizada sempre que dois objetos (lifelines) colaboram entre si dentro de um processo, seja pelo envio, seja pelo recebimento de mensagens, ou ambos. Tal vínculo é representado por uma linha unindo as duas lifelines. A figura 8.2 apresenta um exemplo de vínculo entre duas lifelines, demonstrando que uma lifeline da classe de controle **ControleEmitirSaldo** está vinculada a uma lifeline da classe de entidade **ContaComum**.



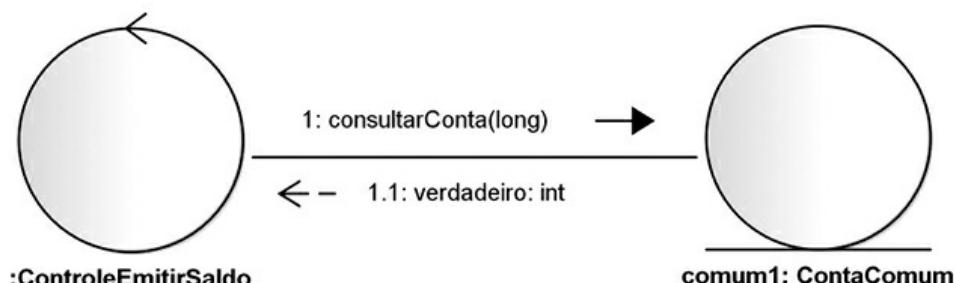
*Figura 8.2 – Exemplo de Vínculo entre Lifelines.*

Observe que os mesmos estereótipos aplicados no diagrama de sequência podem ser igualmente aplicados às lifelines no diagrama de comunicação, assim a lifeline da controladora recebeu o estereótipo **control** e a lifeline da classe **ContaComum** recebeu o estereótipo **entity** que, por serem gráficos, mudaram o desenho-padrão dos componentes. Por esse motivo, a lifeline da classe **ContaComum** apresenta um desenho diferente do apresentado na figura anterior.

### 8.3 Mensagens

As mensagens identificadas nesse diagrama são as mesmas definidas no de sequência e, em geral, representam chamadas de métodos. No entanto, não há preocupação com a temporalidade, ou seja, a ordem em que elas são chamadas não é relevante, o que importa é que são disparadas entre os elementos envolvidos no processo. A única noção temporal passada por esse diagrama é a numeração das mensagens, indicando a ordem em que ocorrem. Uma mensagem é representada por uma seta indicativa da direção para onde a mensagem foi enviada, ou seja, a ponta da seta indica a lifeline (objeto) que receberá a mensagem, enquanto a lifeline na direção oposta será a responsável por seu envio.

É necessário, primeiro, existir um vínculo entre as lifelines para que as mensagens possam ser inseridas. Um único vínculo pode suportar muitas mensagens (podendo estas ser mensagens de retorno), porém não é possível existir mais de um vínculo entre as mesmas lifelines. A figura 8.3 mostra um exemplo de mensagens trocadas entre objetos vinculados.



*Figura 8.3 – Exemplo de mensagens.*

Esse exemplo demonstra que a lifeline da classe controladora

`ControleEmitirSaldo` dispara uma mensagem para que seja executado o método `consultarConta`, passando como parâmetro o número da conta a ser consultada na lifeline `comum1` da classe `ContaComum` e esta, em resposta, envia uma mensagem de retorno informando que a conta foi encontrada.

## 8.4 Atores

Os atores apresentados nesse diagrama são exatamente os mesmos utilizados no diagrama de sequência, ou seja, são instâncias dos atores representados no diagrama de casos de uso. Assim, esses atores representam as entidades externas que interagem com o sistema de alguma forma. Os atores desse diagrama não têm linha de vida nem foco de controle como no diagrama de sequência, sendo representados exatamente da mesma forma que no diagrama de casos de uso, exceto por possuírem o símbolo de dois-pontos (:) na frente da descrição de seu nome, pois são instâncias de um ator definido em um modelo de casos de uso. A figura 8.4 apresenta um exemplo de um ator.

Aqui, demonstramos que existe um vínculo entre uma instância do ator `Cliente` e uma lifeline da classe `VisaoEmitirSaldo`. Por meio desse vínculo, o ator envia uma mensagem à lifeline informando que o cartão de uma conta de banco foi inserido. Essa mensagem não representa um disparo de método, apenas a ocorrência de um evento sobre a interface causado pelo ator.



Figura 8.4 – Um Ator Interagindo com uma Lifeline.

## 8.5 Autochamada

Uma lifeline pode disparar uma mensagem em si própria, o que é conhecido como autochamada, em que a mensagem parte da lifeline e

retorna à própria lifeline. A figura 8.5 apresenta um exemplo de autochamada em uma lifeline.

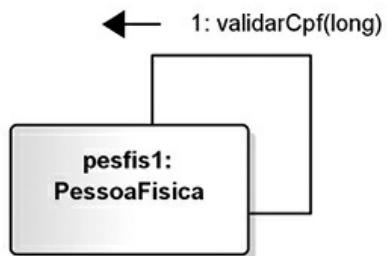


Figura 8.5 – Autochamada.

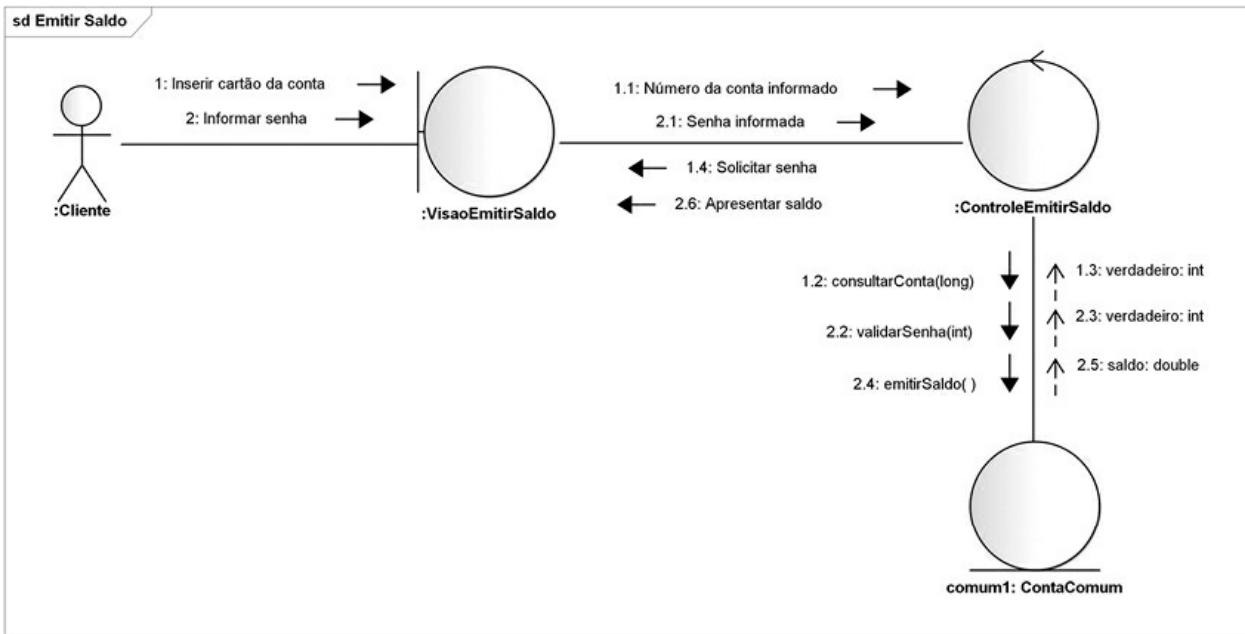
Nessa figura, a lifeline **pesfis1** da classe **PessoaFísica** envia uma mensagem para si mesma, solicitando o disparo do método para validar um CPF.

## 8.6 Exemplo de diagrama de comunicação – Processo de Emissão de Saldo

Nesta seção, apresentaremos um exemplo de diagrama de comunicação, referindo-se este ao processo de emissão de saldo, representado pelo caso de uso **Emitir Saldo** e já modelado no diagrama de sequência. A figura 8.6 apresenta o diagrama referente a esse processo.

Nessa figura interagem o ator **Cliente** e lifelines das classes **VisaoEmitirSaldo**, **ControleEmitirSaldo** e **ContaComum**. O ator tem um vínculo com a interface, esta, com a lifeline da classe controladora, e essa última, com a lifeline da classe **ContaComum**.

Existem aqui dois grupos de mensagens, conforme podemos observar pela numeração delas. O primeiro grupo inicia-se com o ator fornecendo um número de conta à interface. Esta, por sua vez, repassará esse número à controladora que, por meio do vínculo com a lifeline da classe **ContaComum**, disparará por meio de uma mensagem o método **consultarConta**, que recebe como parâmetro o número da conta. Em resposta, a lifeline da classe **ContaComum** retornará verdadeiro através de uma mensagem de retorno, se a conta existir, ou falso, caso contrário.



*Figura 8.6 – Processo de Emissão de Saldo.*

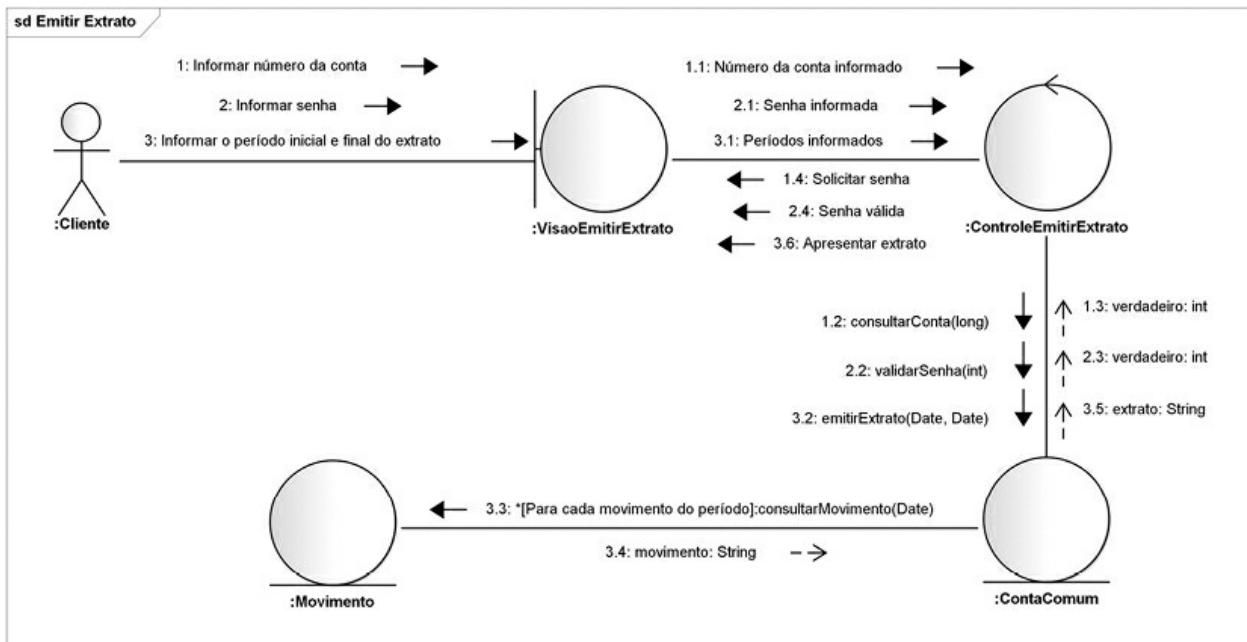
O segundo grupo também é iniciado pelo ator, que, dessa vez, fornece a senha da conta por meio do vínculo com a interface. A interface repassa essa senha para a controladora e esta dispara o método **validarSenha** na lifeline da classe **ContaComum**, passando como parâmetro a senha recebida. Em resposta, a lifeline da classe **ContaComum** retornará verdadeiro, se a senha for válida, ou falso, caso contrário. Em seguida, o controlador solicitará a execução do método **emitirSaldo** na lifeline da classe **ContaComum** e esta em resposta, enviará uma mensagem de retorno contendo o valor armazenado na conta consultada. O controlador toma esse valor e solicita sua apresentação na interface, encerrando o processo.

## 8.7 Condições de Guarda e Iterações

Condições de Guarda são textos entre colchetes que estabelecem condições ou validações para que uma mensagem possa ser enviada. Já iterações representam uma situação em que a mensagem pode ser enviada várias vezes, correspondendo muitas vezes a um laço. As iterações são representadas por um asterisco (\*) na frente da mensagem e, em geral, vêm apoiadas por condições de guarda. Uma vez que o diagrama de comunicação não suporta fragmentos combinados, muitas vezes é necessário lançar mão desse artifício para representar situações opcionais ou laços. A figura 8.7 apresenta um exemplo de mensagem com iteração e

condição de guarda enfocando o processo de emissão de extrato.

O processo aqui representado refere-se ao caso de uso Emitir Extrato, igualmente já descrito no diagrama de sequência. Esse processo envolve o ator **Cliente** e lifelines das classes **VisaoEmitirExtrato**, **ControleEmitirExtrato**, **ContaComum** e **Movimento**.



*Figura 8.7 – Condição e Iteração no Diagrama de Comunicação – Processo de Emissão de Extrato.*

Esse diagrama representa três grupos de mensagens. O primeiro grupo inicia-se com o cliente informando o número da conta na interface. A interface repassa o número da conta para a controladora e esta chama o método **consultarConta**, passando como parâmetro o número da conta em uma lifeline da classe **ContaComum**. Em resposta, esse objeto enviará uma mensagem de retorno à controladora, determinando se a conta foi encontrada (verdadeiro) ou não (falso). Supondo que a conta tenha sido encontrada, a controladora enviará uma mensagem à interface, solicitando que esta peça ao cliente a senha da conta.

Quando o cliente fornece a senha à interface, inicia-se o segundo grupo de mensagens. A interface repassa a senha para a controladora e esta solicita a validação da senha, por meio do disparo do método **validarSenha**, na lifeline da classe **ContaComum**, passando como parâmetro a senha. O objeto da classe **ContaComum** retornará uma mensagem

contendo verdadeiro se a senha estiver correta. Nesse caso, a controladora pedirá que a interface solicite os períodos do extrato.

Ao informar os períodos na interface, o cliente inicia o terceiro e último grupo de mensagens. Os períodos são transmitidos à controladora, que disparará o método `emitirExtrato`, passando como parâmetro as datas iniciais e finais desejadas pelo cliente em uma lifeline da classe `ContaComum`. O objeto da classe `ContaComum` disparará, por sua vez, o método `consultarMovimento`, passando como parâmetro as mesmas datas, em cada lifeline da classe `Movimento` que estiver associada à lifeline da classe `ContaComum` e dentro do período solicitado. Observe que essa última mensagem se caracteriza por ser uma iteração, como demonstra o asterisco (\*) no início da mensagem, ou seja, poderá ser disparada muitas vezes. Além disso, a mensagem é acompanhada pela condição de guarda com o texto “Para cada movimento do período”, que estabelece a condição para que a iteração seja executada. Cada movimento selecionado será retornado ao objeto da classe `ContaComum`, que, por sua vez, os repassará à controladora e esta os mandará apresentar na interface.

## 8.8 Exercícios Propostos

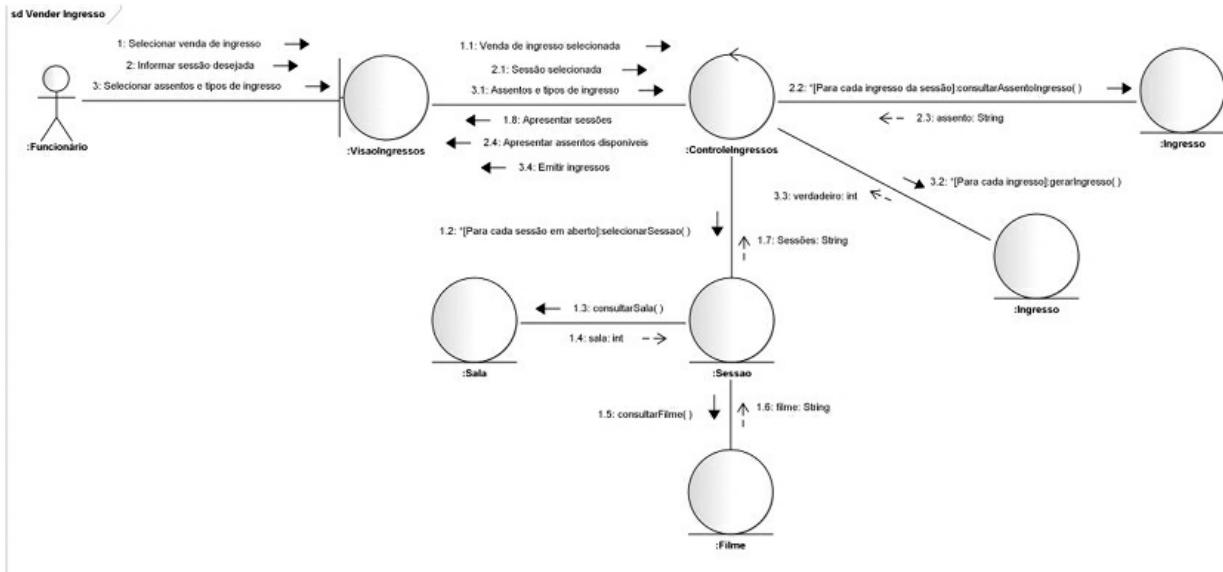
Os exercícios aqui propostos são exatamente os mesmos descritos no final do capítulo 7, que trata do diagrama de sequência, uma vez que o objetivo dos dois diagramas é o mesmo. Assim, não achamos necessário inserir a descrição dos exercícios novamente. Passaremos, assim, diretamente para a solução dos exercícios, excetuando-se o processo de quitar diárias do sistema de controle de hotel, por este utilizar ocorrências de interação não suportadas pelo diagrama de comunicação, utilizado para modelar processos mais simples. Alternativamente, se o leitor desejar, será possível construir um diagrama de comunicação abrangendo os três processos de quitar diária, serviço e consumo, ou criar três diagramas separados, um para cada processo.

## 8.9 Solução dos Exercícios

Na solução dos exercícios, inseriremos somente as figuras com a solução destes, uma vez que a descrição das soluções dos processos se encontra no capítulo 7 sobre o diagrama de sequência. É importante destacar, no

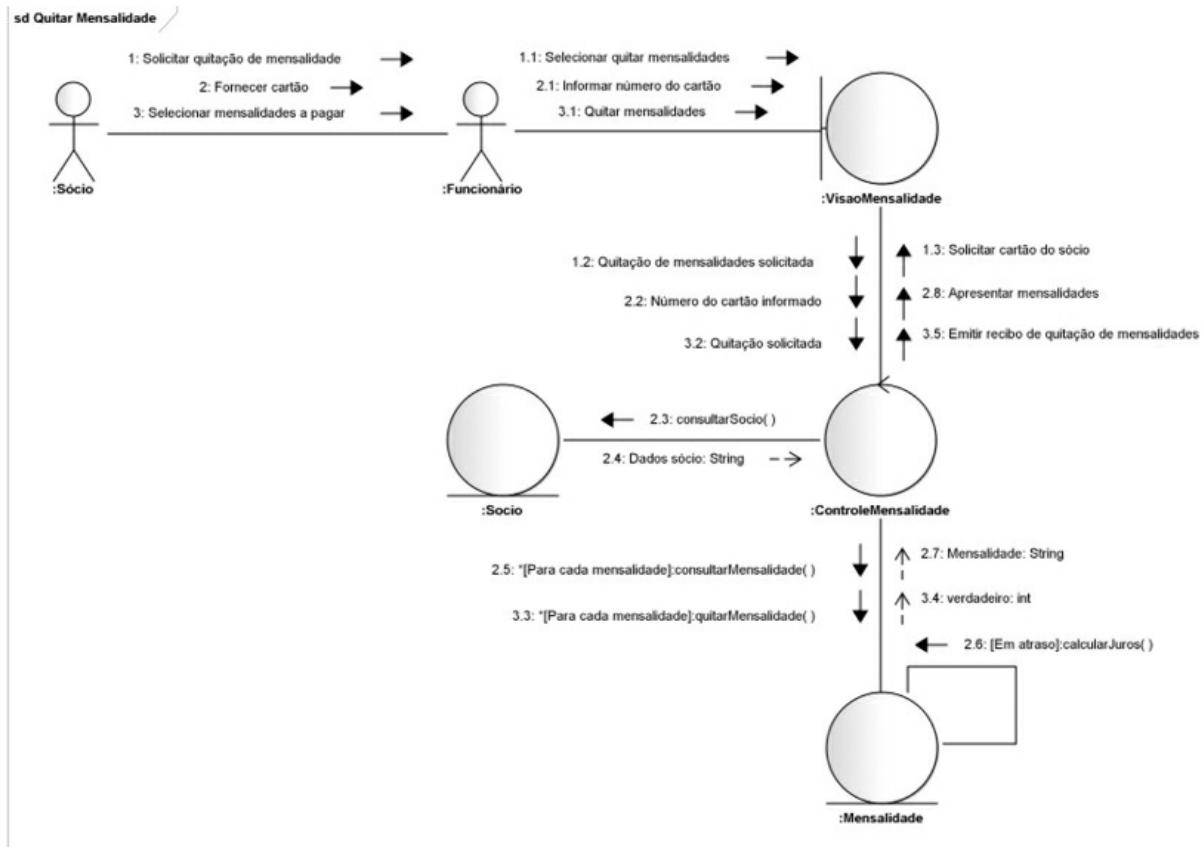
entanto, que fragmentos combinados do tipo opt costumam ser representados por condições de guarda nesse diagrama e fragmentos combinados do tipo loop são representados pelo símbolo de iteração (\*).

### 8.9.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos



*Figura 8.8 – Processo de Venda de Ingressos.*

### 8.9.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade



*Figura 8.9 – Processo de Pagamento de Mensalidade.*

### 8.9.3 Sistema de Locação de Veículos – Processo de Locação de Veículo

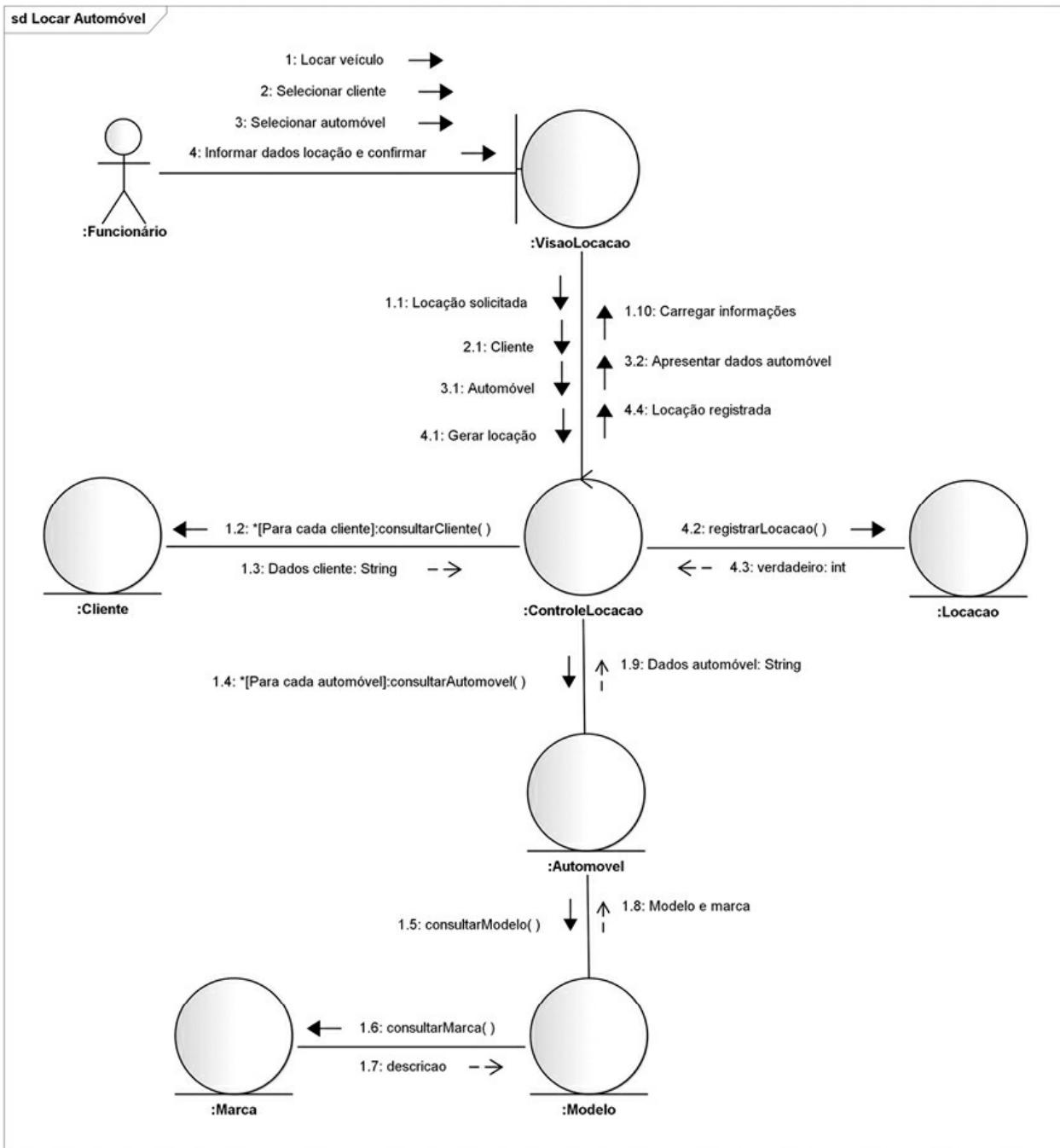


Figura 8.10 – Processo de Locação de Veículo.

#### 8.9.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão

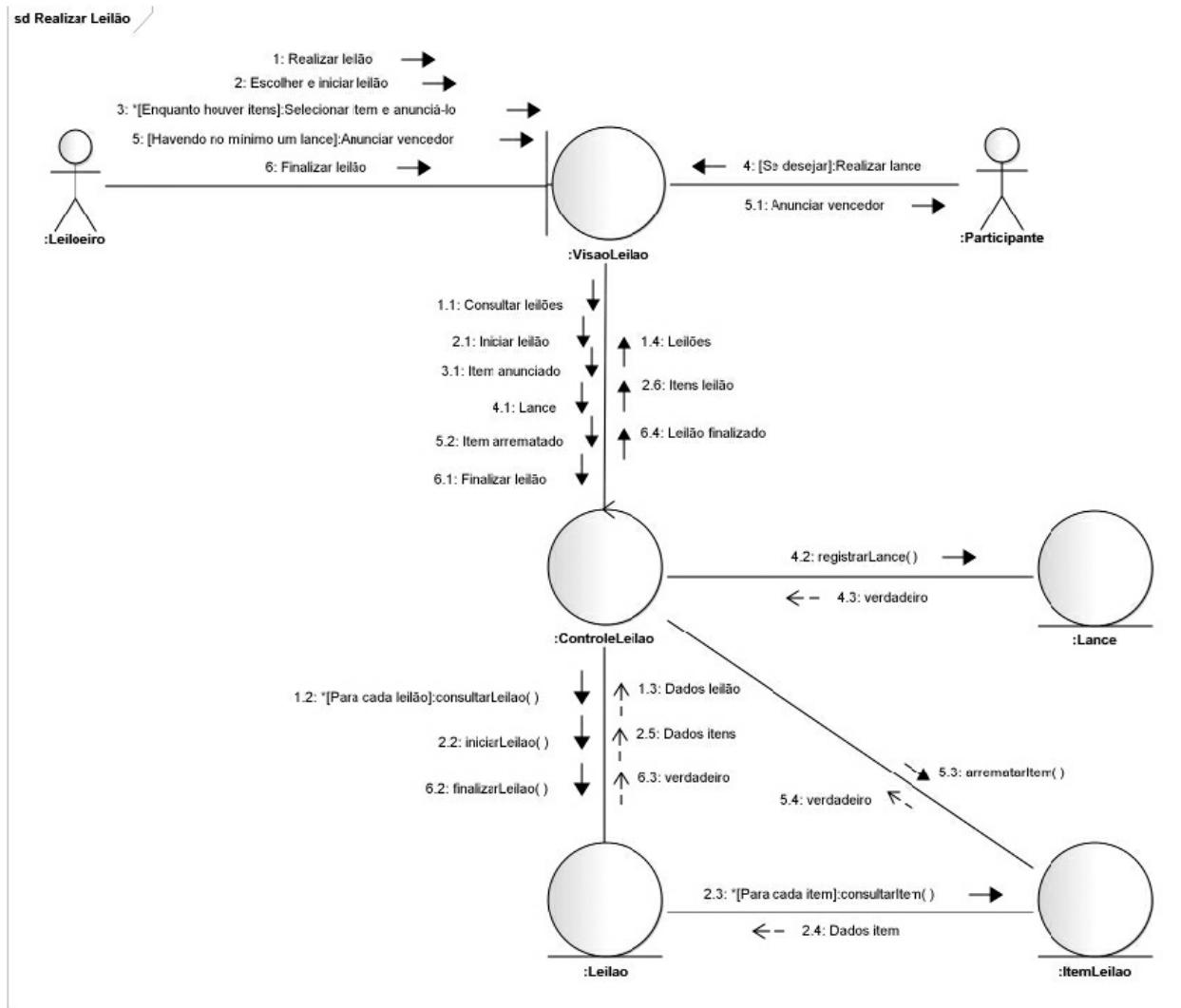


Figura 8.11 – Processo de Realizar Leilão.

## 8.9.5 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel

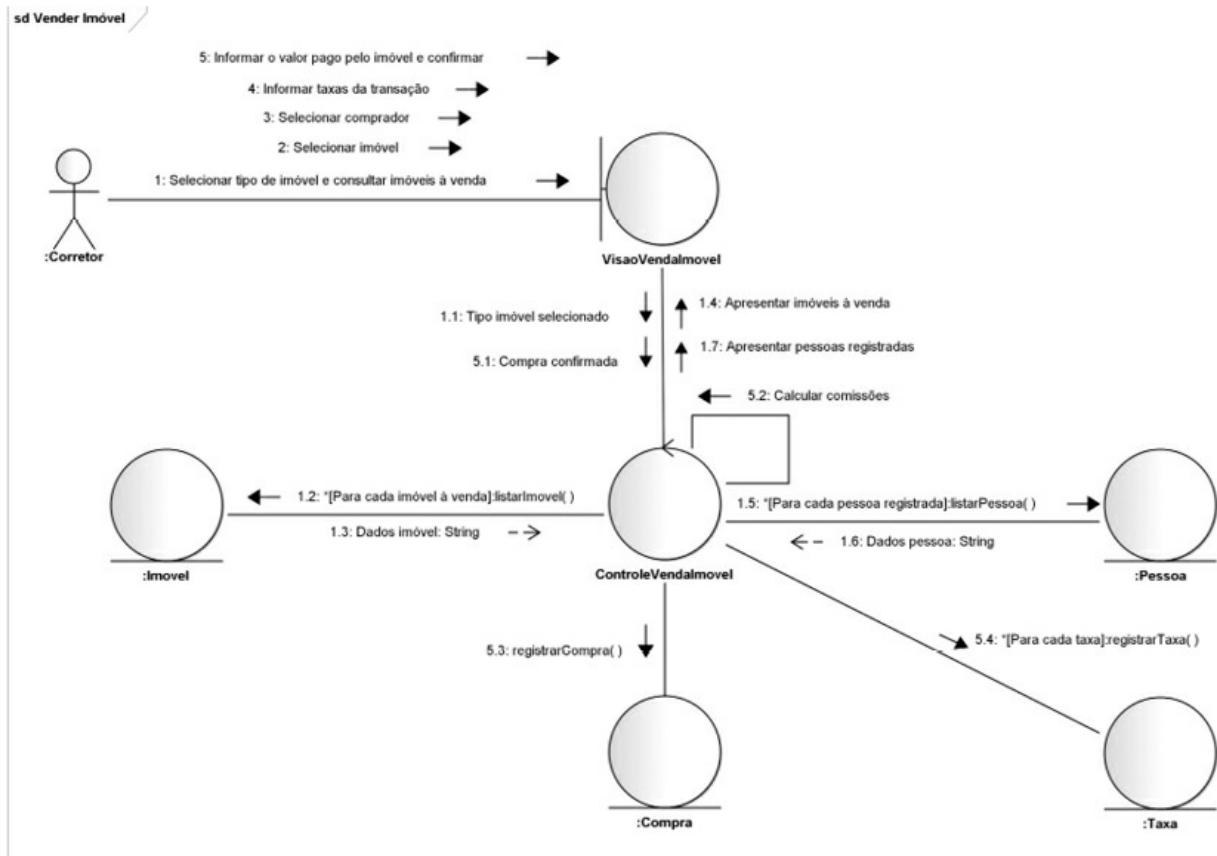


Figura 8.12 – Processo de Venda de Imóvel.

## CAPÍTULO 9

# Diagrama de Máquina de Estados

Esse diagrama era chamado de diagrama de gráfico de estados ou simplesmente diagrama de estados nas versões anteriores da UML. A partir da versão 2.0 da linguagem, seu nome foi modificado para diagrama de máquina de estados.

O diagrama de máquina de estados demonstra o comportamento de um elemento por meio de um conjunto finito de transições de estado. Além de ser utilizado para expressar o comportamento de uma parte do sistema, quando é chamado de máquina de estado comportamental, também pode ser usado para expressar o protocolo de uso de parte de um sistema, quando identifica uma máquina de estados de protocolo. Neste capítulo nos concentraremos em máquinas de estado comportamentais.

Uma máquina de estados comportamental pode ser usada para especificar o comportamento de vários elementos do modelo. O elemento modelado muitas vezes é uma instância de uma classe. No entanto, pode-se usar esse diagrama para modelar o comportamento de um caso de uso, por exemplo.

## 9.1 Estado

Um estado representa a situação em que um elemento (muitas vezes, um objeto) se encontra em determinado momento durante o período em que participa de um processo. Um objeto pode passar por diversos estados dentro de um mesmo processo. Um estado pode demonstrar:

- a espera pela ocorrência de um evento;
- a reação a um estímulo;
- a execução de alguma atividade;
- a satisfação de alguma condição.

### 9.1.1 Estado Simples

Um estado simples não tem subestados, ou seja, não pode ser subdividido em estados internos. É o tipo mais comum de estado. Por esse motivo, será referido simplesmente como estado ao longo do capítulo. A figura 9.1 apresenta um exemplo de estado simples.



Figura 9.1 – Exemplo de Estado Simples.

O estado ilustrado nesse exemplo é um dos estados pelos quais passa um objeto da classe **ContaComum**. Aqui, o objeto se encontra no estado de ContaAtiva, ou seja, pode-se depositar valores nessa conta, bem como sacar valores, emitir saldos e extratos, além de outras operações.

## 9.2 Transições

Uma transição representa um evento que causa uma mudança no estado de um objeto, gerando um novo estado. Uma transição é representada por uma linha ligando dois estados, contendo uma seta em uma de suas extremidades, apontando para o novo estado gerado. A figura 9.2 apresenta um exemplo de transição entre estados.



Figura 9.2 – Exemplo de Transição entre Estados.

Nesse exemplo é apresentada uma transição que representa o evento de encerramento da conta. Aqui, existem dois estados: no primeiro, o objeto da classe **ContaComum** se encontra no estado ativo, enquanto, no segundo, o objeto se encontra no estado inativo. O evento que causou a mudança do estado do objeto é aqui representado por uma seta que parte do primeiro estado e atinge o segundo.

Uma transição pode ou não conter uma descrição (recomenda-se que tenha, para facilitar sua compreensão). A descrição de um evento pode

tanto conter uma ordem para realizar alguma tarefa como ser simplesmente uma informação avisando a ocorrência do evento. Além da descrição, uma transição pode também conter condições de guarda e parâmetros.

### 9.3 Estado Inicial

O estado inicial tem como função somente determinar o início da modelagem dos estados de um elemento. É representado por um círculo preenchido, a partir do qual é gerada uma transição que determina o início do processo. Da mesma forma que nas transições entre os objetos, uma transição de um estado inicial pode conter ou não uma descrição, o que, às vezes, é útil para identificar o evento que iniciou o processo.

### 9.4 Estado Final

O estado final tem a função apenas de indicar o final dos estados modelados. É representado por um círculo não preenchido envolvendo um segundo círculo preenchido. A figura 9.3 apresenta um exemplo de estados inicial e final.

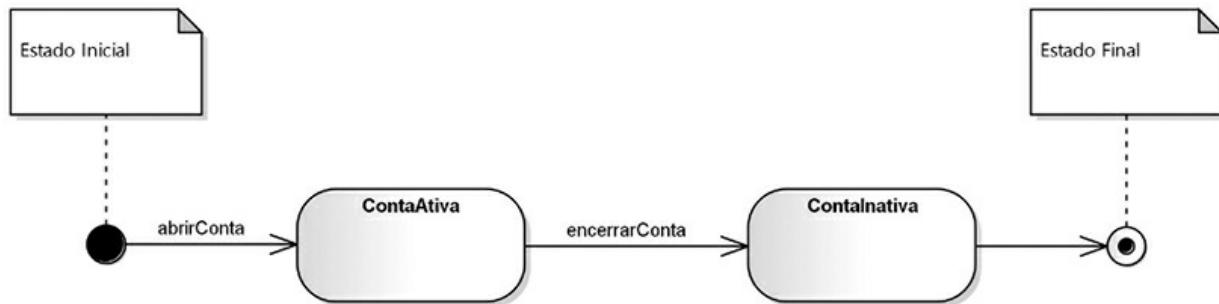


Figura 9.3 – Exemplo de Estados Inicial e Final – Processo de Emissão de Saldo.

Nesse exemplo, usamos o componente nota para identificar os estados iniciais e finais desse diagrama. O diagrama em questão representa os estados pelos quais passa um objeto da classe **ContaComum**, que se resumem basicamente aos estados de contas ativa e inativa. O primeiro estado ocorre quando a conta é aberta, já o segundo ocorre quando a conta é encerrada.

Como o leitor pode perceber, os estados desse objeto são poucos. Isso é

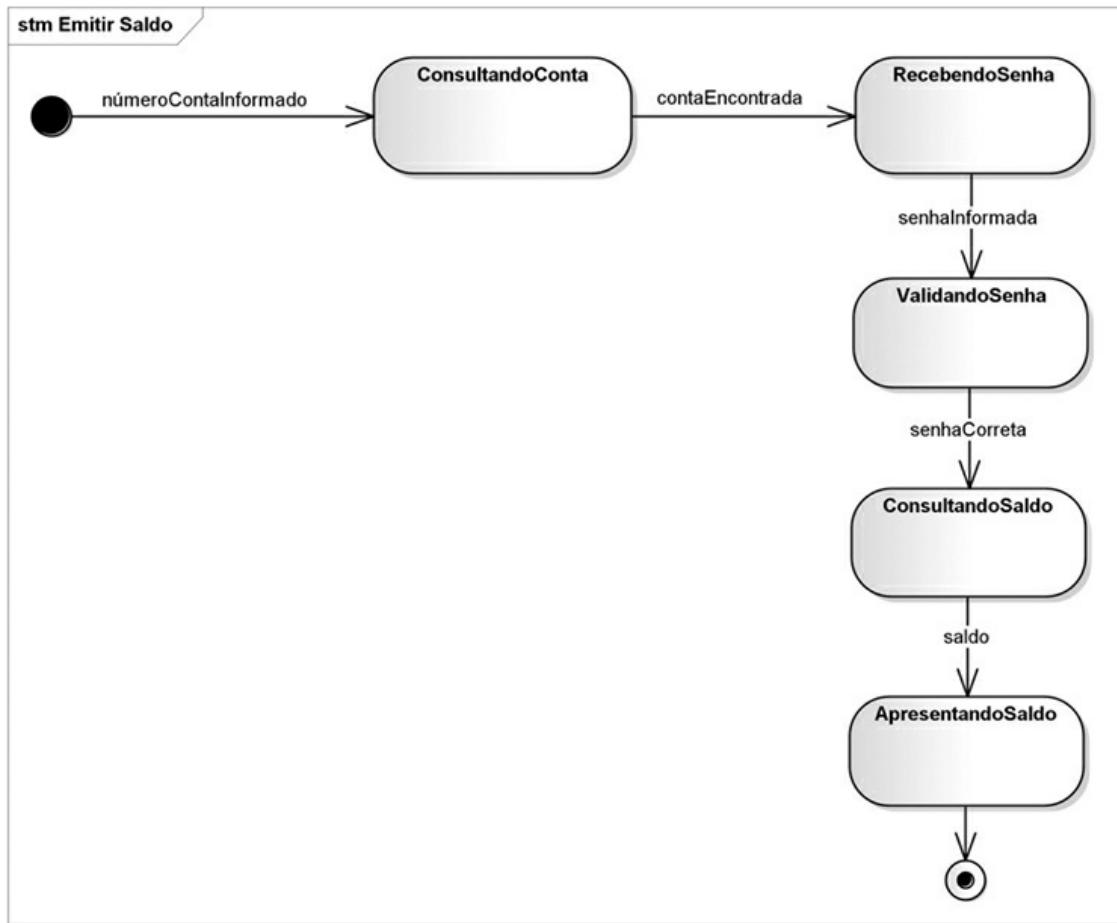
comum em sistemas de cunho comercial, em que os objetos não costumam possuir muitos estados. Em situações assim, é mais prático utilizar o diagrama de máquina de estados para modelar os estados de um processo representado por um caso de uso – que, nessa situação, passa a ser tratado como um objeto. Dessa forma, no restante deste capítulo, utilizaremos essa prática com frequência e muitos dos exemplos que apresentaremos serão relativos aos estados de um caso de uso.

## 9.5 Exemplo de Diagrama de Máquina de Estados – Processo de Emissão de Saldo

Aqui, tomaremos o caso de uso referente ao processo de emissão de saldo e representaremos os estados desse processo. Como já dissemos, em sistemas comerciais, essa prática é mais comum, posto que os objetos individuais costumam possuir poucos estados e estes muitas vezes são bastante simples (Figura 9.4).

Nesse exemplo é gerada uma transição a partir do estado inicial que dá início ao processo. Essa transição representa o evento em que um número de conta é informado, o que produz o estado **ConsultandoConta**.

Depois de a conta ter sido consultada, um novo evento é gerado, caracterizando-se pela informação de que a conta foi encontrada. Isto gera um novo estado, denominado **RecebendoSenha**, que é estático, ou seja, um estado em que não é realizada nenhuma atividade, apenas se aguarda que algo aconteça. Nesse estado específico, aguarda-se que a senha da conta seja informada.



*Figura 9.4 – Processo de Emissão de Saldo.*

No momento em que a senha é fornecida, é produzido o estado **ValidandoSenha**. Quando a validação for concluída, será solicitada a emissão do saldo, o que cria o estado **ConsultandoSaldo**. No momento em que esse estado é concluído, é gerada uma transição contendo o saldo a ser apresentado. Essa transição cria o estado **ApresentandoSaldo**, o que conclui esse processo, conforme demonstra o estado final.

É importante destacar que quando um estado está executando uma atividade, costuma ser descrito no gerúndio. No entanto, pode acontecer de um estado estar em uma situação estática, à espera de que um evento ocorra e, nessas situações, não deverá ser descrito no gerúndio.

## 9.6 Atividades internas

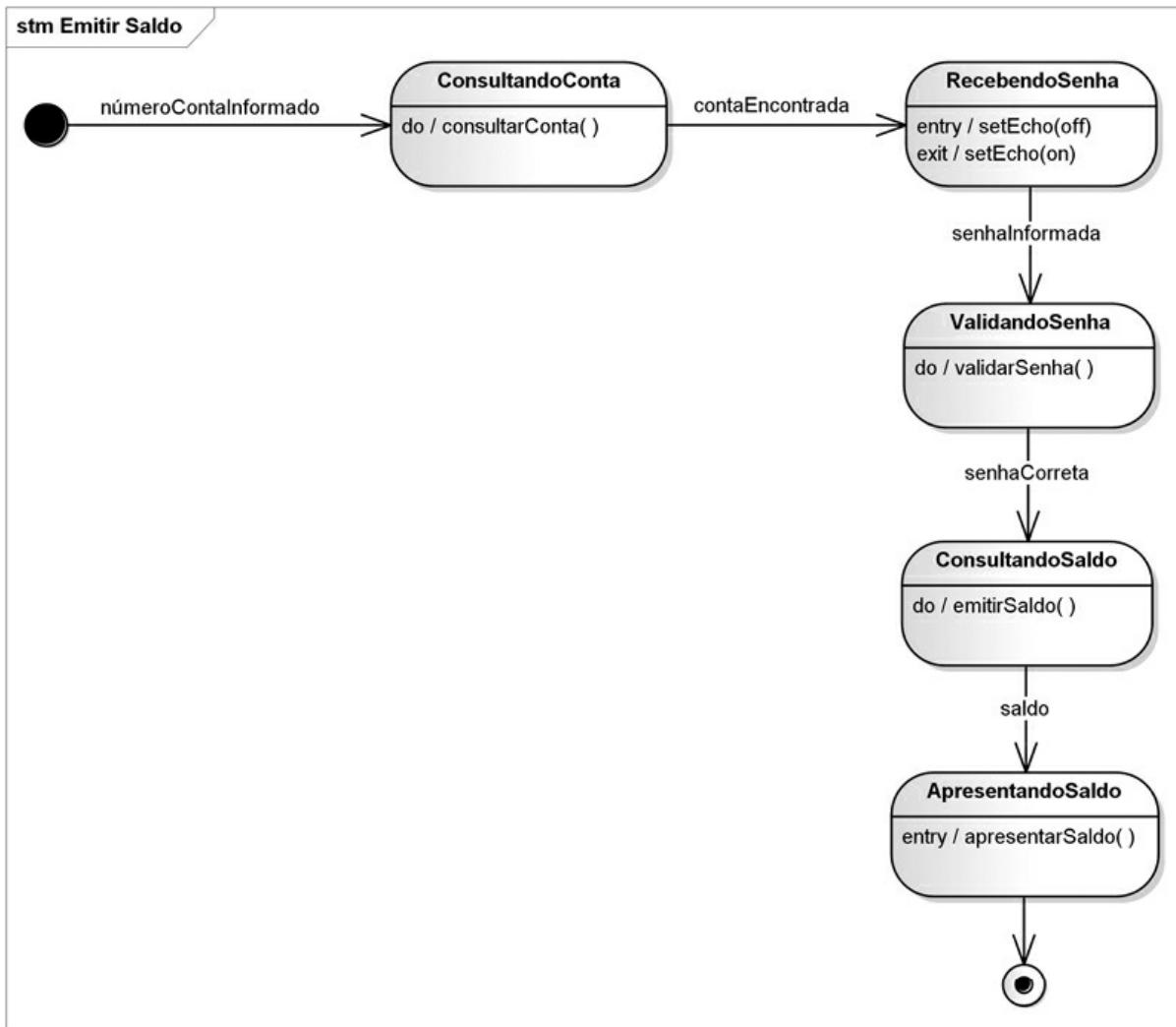
São as atividades que um objeto pode executar quando se encontra em um estado. O detalhamento delas não é obrigatório e nem sempre um estado as possuirá, mas elas fornecem informações extras sobre o que o objeto faz

quando se encontra em um estado, o que pode ser útil para compreender a função de um estado específico. Essas atividades podem ser detalhadas por meio das seguintes cláusulas:

- **Entry** – Identifica uma atividade executada quando o objeto assume (entra em) um estado. Sempre que um estado é assumido, executa o comportamento identificado por essa cláusula antes de qualquer outra ação.
- **Exit** – Identifica uma atividade executada quando o objeto sai de um estado. Sempre que se vai sair de um estado, o comportamento identificado por essa cláusula é executado antes de o estado ser abandonado.
- **Do** – Identifica uma atividade realizada durante o tempo em que o objeto se encontra em um estado. Atividades internas do tipo **Do** também são chamadas de atividades de estado.

As cláusulas **Entry** e **Exit**, também chamadas ações de estado, estão mais associadas às transições do que ao estado propriamente dito, já que são executadas quando o objeto assume um novo estado ou está mudando de estado. Além disso, seu tempo de execução costuma ser inferior as das atividades de estado, podendo representar a simples atribuição de um valor a um atributo ou a geração de uma saída, enquanto as atividades de estado geralmente representam métodos executados pelo objeto.

As atividades internas são representadas em uma segunda divisão do estado, conforme demonstra a figura 9.5, na qual enriquecemos a figura 9.4 adicionando atividades internas aos estados, quando necessário.



*Figura 9.5 – Processo de Emissão de Saldo com Detalhamento de Atividades Internas.*

Ao observar esse exemplo, o leitor notará que quando o objeto se encontra no estado **ConsultandoConta**, é executado o método **consultarConta**, conforme demonstra a cláusula **do**. Já quando o objeto entra no estado **RecebendoSenha**, é definido que qualquer caractere digitado não poderá ser mostrado, como é especificado pela cláusula **entry**, e quando o objeto sair desse estado, os caracteres passarão novamente a ser apresentados, conforme definido pela cláusula **exit**.

Da mesma forma, quando o objeto assumir o estado **ValidandoSenha**, será executado o método **validarSenha**, e quando o objeto estiver no estado **ConsultandoSaldo**, será executado o método **emitirSaldo**. Finalmente, quando o estado **ApresentandoSaldo** for atingido, o saldo

será apresentado, como é demonstrado pela cláusula `entry`.

## 9.7 Transições Internas

Transições internas não produzem modificações no estado de um objeto. A figura 9.6 apresenta um exemplo de transição interna.

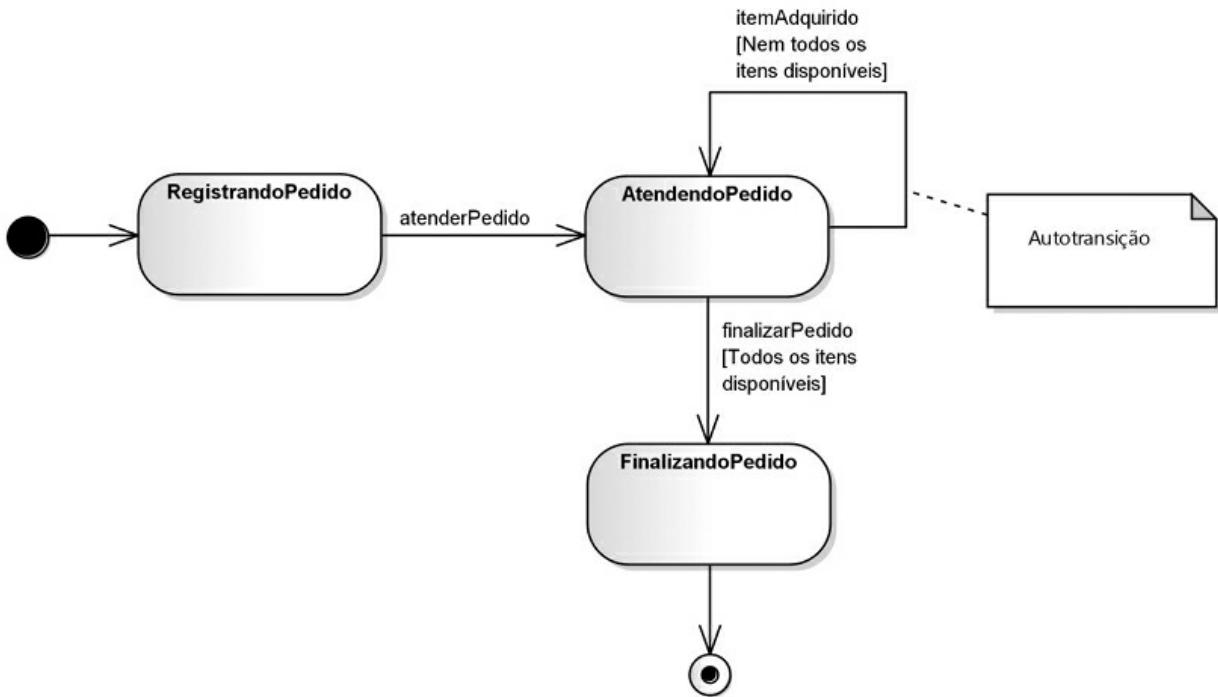


Figura 9.6 – Exemplo de Transição Interna.

Nesse exemplo, enfocamos o estado **RegistrandoPessoa**, cuja função é cadastrar um novo cliente da instituição bancária, representado por uma pessoa física. Nesse estado, existem duas cláusulas: a primeira representa uma atividade de estado, conforme demonstra a cláusula **Do**, que determina que deverá ser executado o método `registrarPessoa` quando o objeto se encontrar nesse estado. A segunda representa uma transição interna que determina que o CPF da pessoa deverá ser validado por meio da execução do método `validarCpf`, que será chamado durante a execução do método `registrarPessoa`, ou seja, antes de concluir o registro da pessoa, é necessário verificar se o CPF informado está correto. Assim, embora o objeto esteja validando o CPF informado pelo cliente, ainda se encontra no estado **RegistrandoPessoa**, não havendo uma mudança no estado do objeto.

## 9.8 Autotransições

Autotransições apresentam pequenas diferenças em relação às transições internas. Essas últimas ocorrem durante um estado do objeto, sem modificá-lo, enquanto as autotransições saem do estado atual do objeto, podendo executar alguma ação quando dessa saída, e retornam ao mesmo estado. Uma autotransição é representada por uma seta de transição que parte do objeto e retorna ao próprio objeto. A figura 9.7 apresenta um exemplo de autotransição.



*Figura 9.7 – Exemplo de Autotransição.*

Nesse exemplo, enfocamos o processo de atendimento de um pedido. Pode acontecer de, ao atender a um pedido, nem todos os itens solicitados estarem disponíveis em estoque, sendo preciso adquiri-los. Assim, sempre que um novo item for adquirido, ocorrerá uma autotransição. No entanto, se algum item ainda estiver indisponível, o objeto voltará ao estado **Atendendo Pedido**. Somente quando todos os itens do pedido estiverem disponíveis, será possível passar ao estado seguinte.

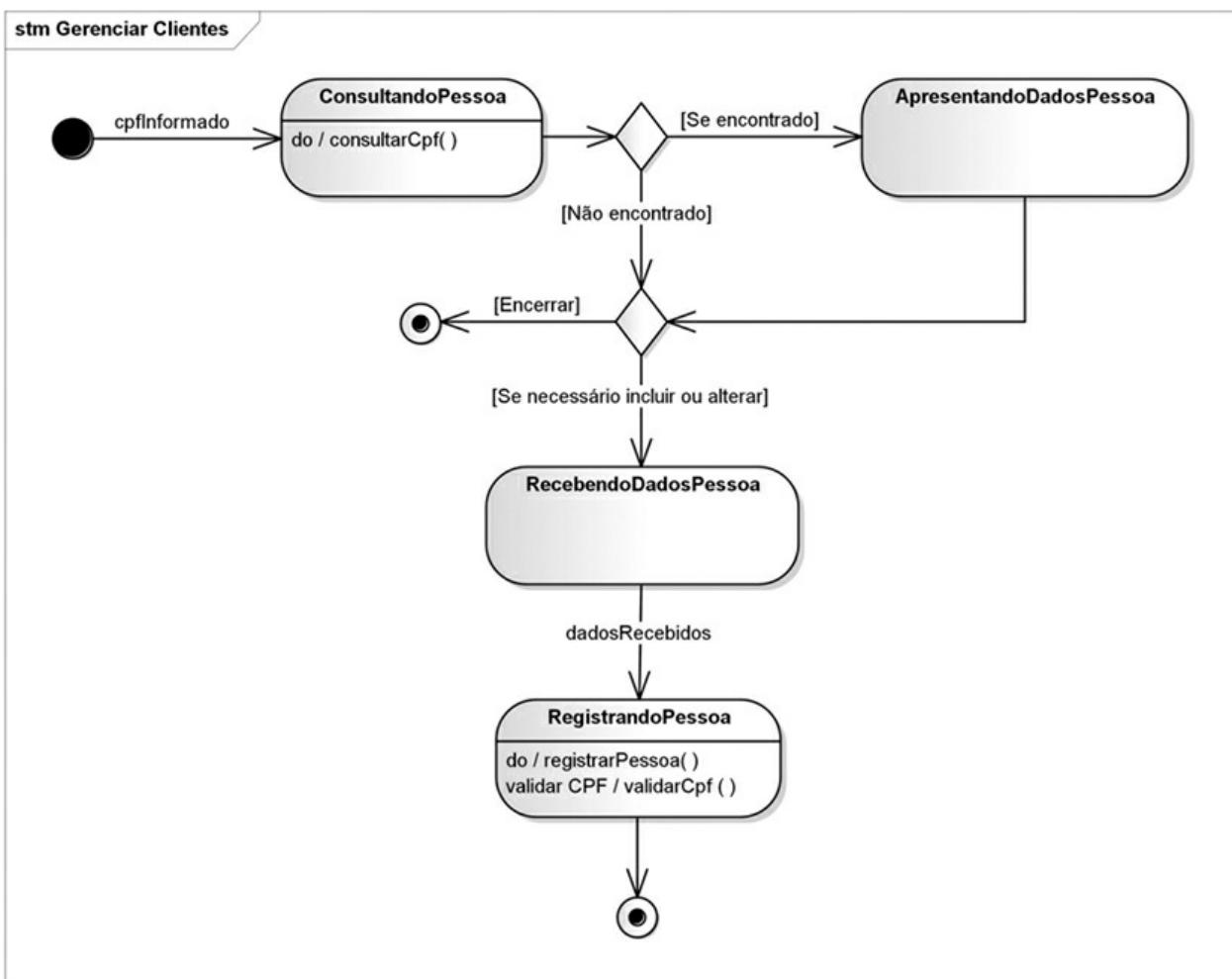
Observe que existem duas condições de guarda associadas às transições que determinam a condição para que se permaneça no estado **Atendendo Pedido** ou se passe para o estado **Finalizando Pedido**. O evento que causa a autotransição é identificado pela aquisição de um novo item. A esse evento existe uma condição de guarda associada, definida como “Nem todos os itens disponíveis”, o que determina que se deve voltar ao mesmo estado quando essa condição for verdadeira. Já a transição que leva ao novo estado **Finalizando Pedido** tem como condição que todos os itens estejam disponíveis.

## 9.9 Pseudoestado de Escolha

Conhecido nas versões anteriores como estado de ponto de escolha

dinâmico, o pseudoestado de escolha representa um ponto na transição de estados de um objeto em que deve ser tomada uma decisão, segundo a qual um determinado estado será ou não gerado, normalmente em detrimento de diversos outros possíveis estados. Dessa forma, um pseudoestado de escolha representa uma decisão, apoiada por condições de guarda, em que se decidirá qual o próximo estado do objeto a ser gerado.

Um pseudoestado de escolha pode ser representado por um losango ou um círculo vazio de onde partem duas ou mais possíveis transições. A figura 9.8 apresenta um exemplo de pseudoestado de escolha enfocando o processo de gerenciamento do cadastro de clientes do sistema de controle bancário.



*Figura 9.8 – Pseudoestado de Escolha – Gerenciar Clientes.*

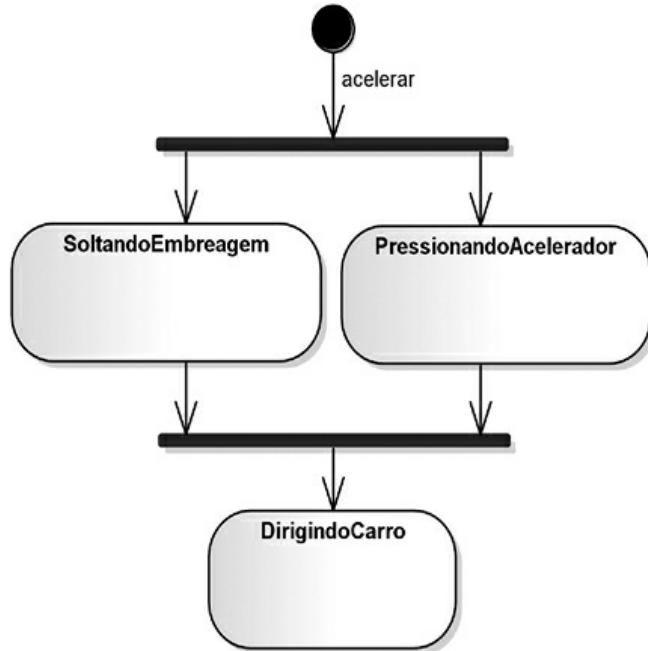
Ao observarmos a máquina de estados apresentada no diagrama, percebemos que o processo se inicia quando é informado o CPF da pessoa,

o que gera o estado **ConsultandoPessoa**, em que é executado o método **consultarCpf**. Quando o estado é concluído, é necessário fazer uma escolha, representada por um pseudoestado de escolha, em que, se for encontrada uma pessoa com o CPF informado, o processo deverá passar ao estado **ApresentandoDadosPessoa** (conforme demonstra a condição de guarda) e, em seguida, a um segundo pseudoestado de escolha.

Se a pessoa não for encontrada, então o estado **ApresentandoDadosPessoa** não será gerado, passando-se diretamente ao segundo pseudoestado de escolha, em que se deve escolher encerrar o processo ou fornecer os dados da pessoa para inserir uma nova pessoa ou alterar os dados da pessoa encontrada, o que produz o estado **RecebendoDadosPessoa** e, no momento em que este for concluído, passe ao estado **RegistrandoPessoa**, no qual serão executados os métodos **registerPessoa** e **validarCpf**, já explicados.

## 9.10 Barra de Bifurcação/união

É utilizada quando da ocorrência de estados paralelos, causados por transições concorrentes. Sua função é determinar o momento em que o processo passou a ser executado em paralelo e em quantos subprocessos se dividiu (evento conhecido como bifurcação) ou determinar o momento em que dois ou mais subprocessos se uniram em um único processo (evento conhecido como união). Pode ser representada tanto por uma barra horizontal quanto vertical. A figura 9.9 apresenta um exemplo de barra de bifurcação/união, em que são modelados dois estados paralelos pelos quais passa um objeto da classe carro no momento em que um ator necessita dirigí-lo.

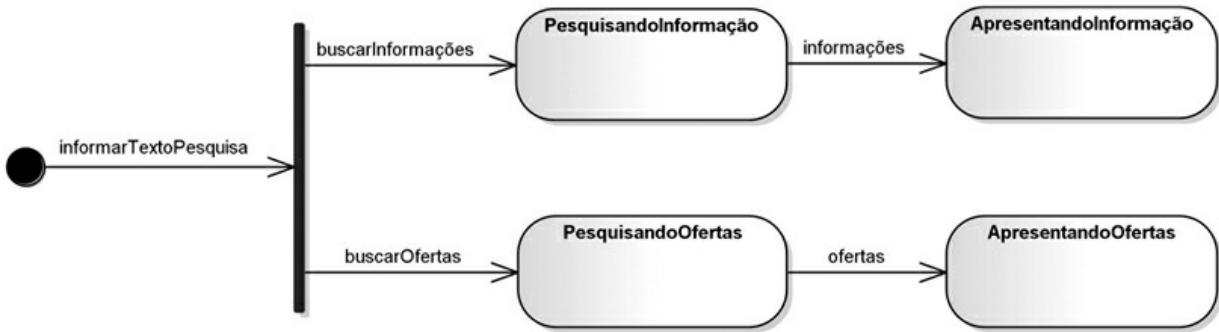


*Figura 9.9 – Barra de Bifurcação/União.*

Esse exemplo representa um trecho de um diagrama que modela os estados pelos quais passa um objeto da classe **Carro** no processo que o usuário leva para destrancá-lo, ligá-lo e dirigí-lo. No trecho ilustrado na figura, enfocamos o momento final do processo, quando o motorista começa a guiar o carro, mas, para isso, ele precisa realizar duas tarefas simultaneamente: soltar a embreagem e pressionar o acelerador. Dessa forma, foi inserida uma barra de bifurcação/união horizontal, indicando que os dois estados ocorrem em paralelo. Em seguida, usa-se uma nova barra de bifurcação/união para unir os dois subprocessos e gerar o estado **DirigindoCarro**.

Outro exemplo para o uso da barra de bifurcação/união pode ser visto na figura 9.10, em que enfocamos um processo que se refere a uma pesquisa de internet, na qual, em paralelo com a busca por informações solicitadas, são pesquisadas também ofertas relacionadas à pesquisa.

Nesse exemplo, no momento em que o evento **informarTextoPesquisa** é disparado, uma barra de bifurcação/união divide os estados gerados pelo evento, havendo um fluxo para buscar e apresentar as informações pertinentes à pesquisa e outro para buscar e apresentar as ofertas relacionadas a esta.



*Figura 9.10 – Uso da barra de bifurcação/união no processo de Solicitar Pesquisa.*

## 9.11 Estados Compostos

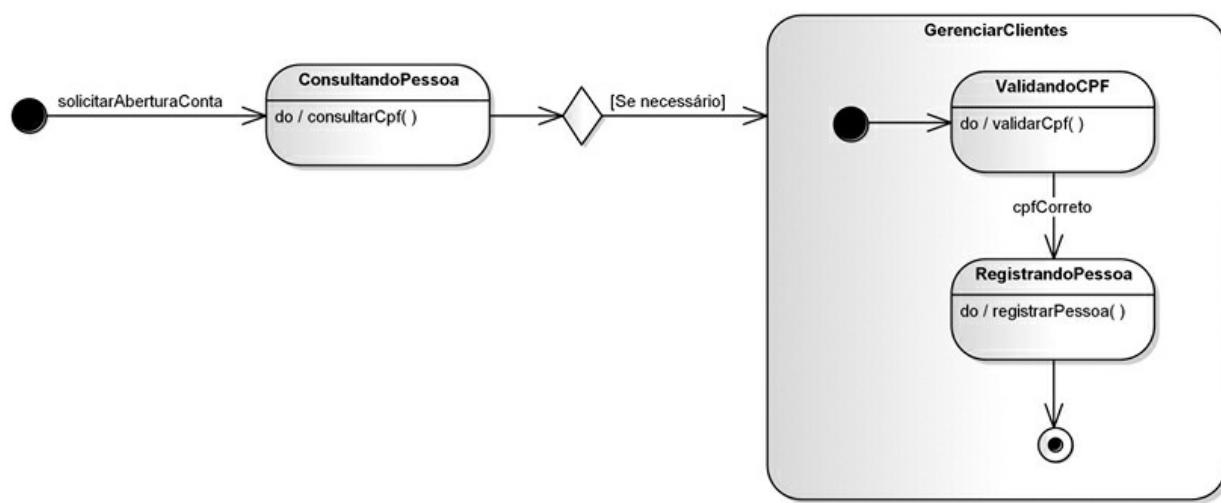
Estados compostos são o segundo tipo de estado suportado pelo diagrama de máquina de estados. O primeiro é o estado simples, explicado anteriormente neste capítulo, e o terceiro é a máquina de subestados, que será explicada mais adiante.

Um estado composto contém internamente dois ou mais estados chamados subestados. Um subestado é chamado direto, quando não é contido por outro estado, ou indireto, caso contrário. Assim, estado composto é um estado que foi “explorado”, de maneira a apresentar detalhadamente todas as etapas pelas quais passa o objeto quando no estado em questão.

Um estado composto pode apresentar somente uma região ou ser decomposto em duas ou mais regiões ortogonais, quando se torna um estado ortogonal (chamado estado concorrente em versões anteriores), em que cada região terá estados e transições. Cada região de um estado composto pode ter um pseudoestado inicial e um estado final. Uma transição para um estado composto representa uma transição para o pseudoestado inicial de cada região. Um estado composto não pode ser reutilizado, diferentemente do que ocorre com as máquinas de subestado.

Um estado composto pode facilitar a compreensão de um determinado estado de maneira bem mais detalhada, em que são discriminados os diversos subestados pelos quais passa o objeto quando no estado composto. A figura 9.11 demonstra um exemplo de estado composto com apenas uma região.

No exemplo dessa figura, iniciamos a modelar o processo de abertura de conta no qual o evento **solicitarAberturaConta** produz uma transição que gera o estado **ConsultandoPessoa**. Depois de esse estado ser concluído, passa-se a um pseudoestado de escolha (somente identificamos uma de suas transições, uma vez que o exemplo está incompleto), no qual, se for necessário (como demonstra a condição de guarda), gera-se uma transição para um estado composto intitulado **GerenciarClientes**, que representa os estados necessários para dar manutenção no cadastro de pessoas, ou seja, para incluir ou alterar o registro de uma pessoa.



*Figura 9.11 – Estado Composto.*

O leitor notará que transformamos o estado **RegistrandoPessoa**, exemplificado anteriormente, em um estado composto, dividindo-o em dois subestados. Nesse caso, tomamos a transição interna para validação de CPF e a transformamos em um subestado, assim como criamos um segundo subestado, denominado **RegistrandoPessoa**. Dessa forma, antes de concluir o cadastro ou a atualização da pessoa, primeiramente se valida seu CPF e, caso este esteja realmente correto, registram-se os dados da pessoa.

O estado inicial apresentado dentro do estado composto é um pseudoestado inicial, já que o diagrama tem um estado inicial. Esse pseudoestado inicial refere-se somente ao início dos subestados do estado composto.

## 9.12 Pseudoestado de História

Um pseudoestado de história representa o registro do último subestado em que um objeto se encontrava, quando, por algum motivo, o processo foi interrompido. Assim, por meio do pseudoestado de história, podemos retornar exatamente ao último subestado em que o objeto encontrava-se quando da interrupção do processo. Um pseudoestado de história é representado por um **H** dentro de um círculo, conforme ilustra a figura 9.12.

Nesse exemplo, enfocamos um processo de aumento de produtos de uma determinada categoria. Primeiramente, selecionamos a categoria do produto e, depois, o percentual de aumento para os produtos da categoria escolhida. O recálculo de valor de todos os produtos da categoria em questão é representado por um estado composto, em que um produto específico da categoria é selecionado, seu valor de venda é recalculado e o produto, atualizado. Observe que um estado de história mantém o subestado atual do processo.

Eventualmente, por um motivo qualquer, pode ser necessário suspender temporariamente o processo, conforme demonstra a transição à esquerda do estado composto, que gera o estado **ProdutoSuspenso**. Tão logo o motivo da interrupção seja sanado, o processo de recálculo poderá ser retomado a partir do momento em que foi suspenso, por meio do estado de história. Observe que é gerada uma transição do estado **ProdutoSuspenso** diretamente para o estado de história, que será responsável por continuar o processo exatamente a partir do subestado em que havia sido interrompido.

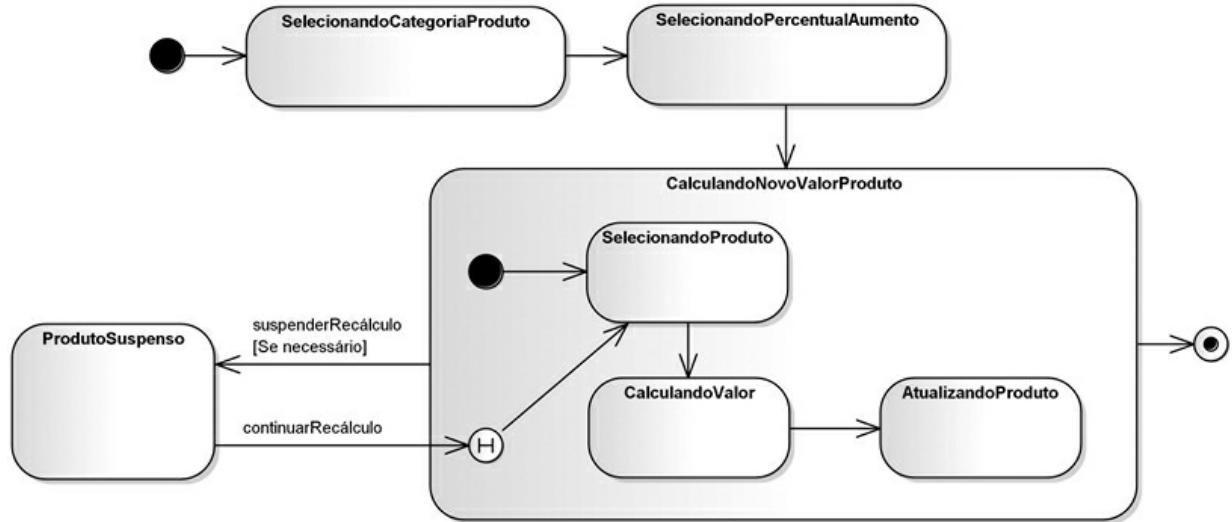


Figura 9.12 – Pseudoestado de História.

Existe também o pseudoestado de história profunda, representado por um **H** seguido de um símbolo de asterisco (**H\***) envolvido por um círculo, utilizado quando da ocorrência de estados compostos dentro de estados compostos. Assim, o estado de história profunda guarda e recupera o último subestado em qualquer dos estados compostos onde possa estar.

## 9.13 Estados Compostos Ortogonais

Um estado ortogonal (chamado concorrente em versões anteriores) é um estado composto que possui mais de uma região, onde cada uma apresenta um conjunto de estados e os estados de cada região são assumidos paralelamente, o que força o processo a se dividir em dois ou mais subprocessos concorrentes. Os processos concorrentes são separados por uma linha tracejada dentro do estado ortogonal, conforme pode ser observado na figura 9.13.

Esta figura apresenta uma segunda alternativa para o processo de **Solicitar Pesquisa**. Nesse exemplo, o processo se inicia com um estado estático que aguarda que o usuário digite um texto a pesquisar. O evento de inserção de um texto para pesquisa causa uma transição para um estado composto ortogonal contendo duas regiões. Na primeira região, como seu título na forma de condição de guarda informa, é feita a busca por informações relevantes à pesquisa, enquanto na segunda região é feita a busca por ofertas relacionadas ao texto pesquisado. Os estados de cada

região ocorrem paralelamente.

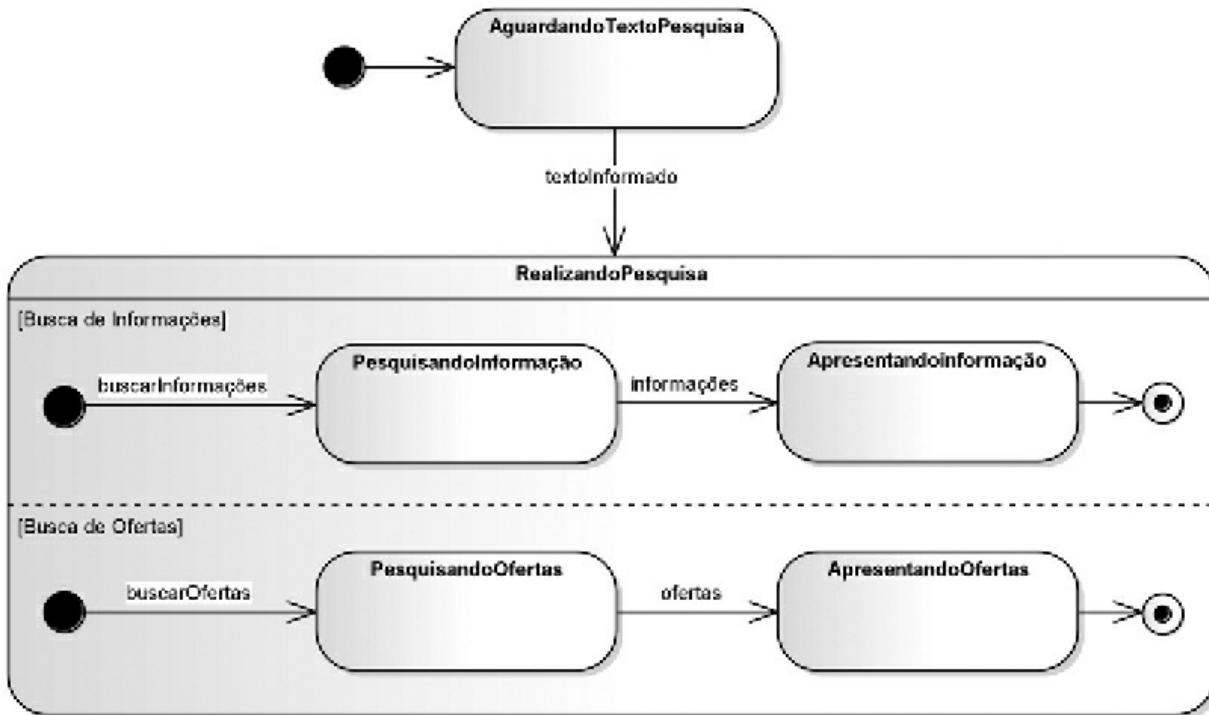
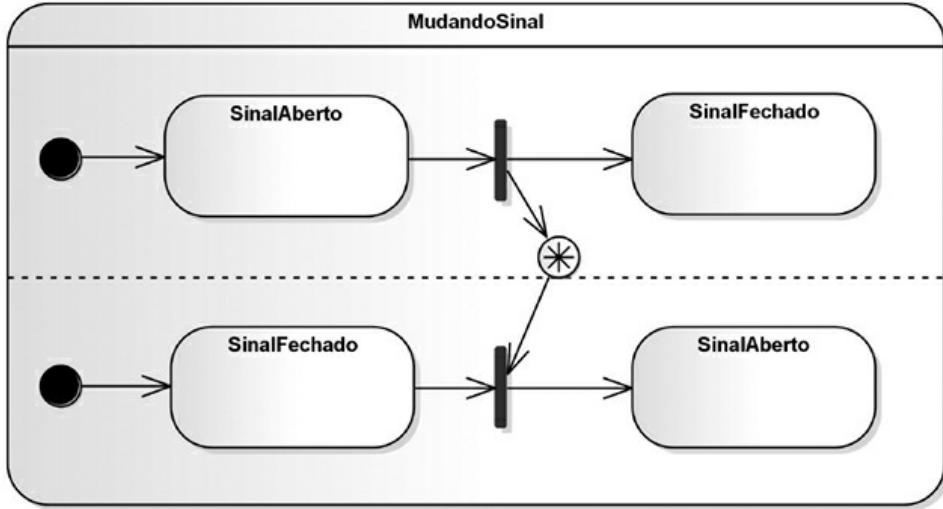


Figura 9.13 – Exemplo de Estado Composto Ortogonal.

## 9.14 Estado de Sincronismo

Em alguns processos pode eventualmente ser necessário que estados de regiões diferentes estejam de alguma forma sincronizados, por vezes sendo necessário que um estado de uma região espere por um estado de outra. Assim, é preciso existir um estado de sincronismo cuja função é permitir que os relógios de duas ou mais regiões estejam sincronizados em um determinado momento do processo. O estado de sincronismo é representado por um símbolo de asterisco (\*) dentro de um círculo, conforme pode ser observado na figura 9.14.

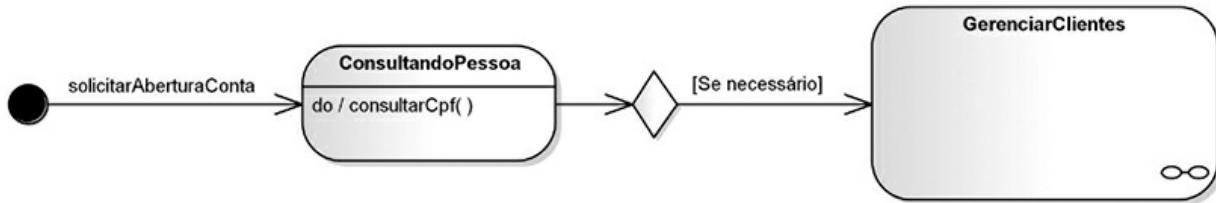


*Figura 9.14 – Estado de Sincronismo.*

Nesse exemplo, há dois sinais de trânsito. No momento em que o sinal de um muda, o outro deve mudar também, automaticamente. Assim, o estado do primeiro sinal de trânsito é representado como **SinalAberto**, e o do segundo, como **SinalFechado**. Quando o primeiro sinal de trânsito recebe a ordem de trocar seu sinal, este gera duas transições, uma para trocar seu próprio estado e outra para avisar o estado de sincronismo para alterar também o estado do segundo sinal. Assim, no momento em que o estado do primeiro sinal mudar para **SinalFechado**, o estado do segundo sinal automaticamente terá de mudar para **SinalAberto**.

## 9.15 Estado de Submáquina

Um estado de submáquina é um mecanismo de decomposição que permite a fatoração de comportamentos comuns e seu reuso. Um estado de submáquina é equivalente a um estado composto. No entanto, seus subestados não são descritos no diagrama, o que indica que terão de ser demonstrados em outro diagrama. Além disso, diferentemente de um estado de submáquina, um estado composto não pode ser reutilizado. Um estado de submáquina é representado por um retângulo com as bordas arredondadas sem divisões internas, contendo no canto inferior direito um símbolo que representa um diagrama de máquina de estados, significando que o estado em questão possui subestados internos. A figura 9.15 apresenta um exemplo de estado de submáquina.

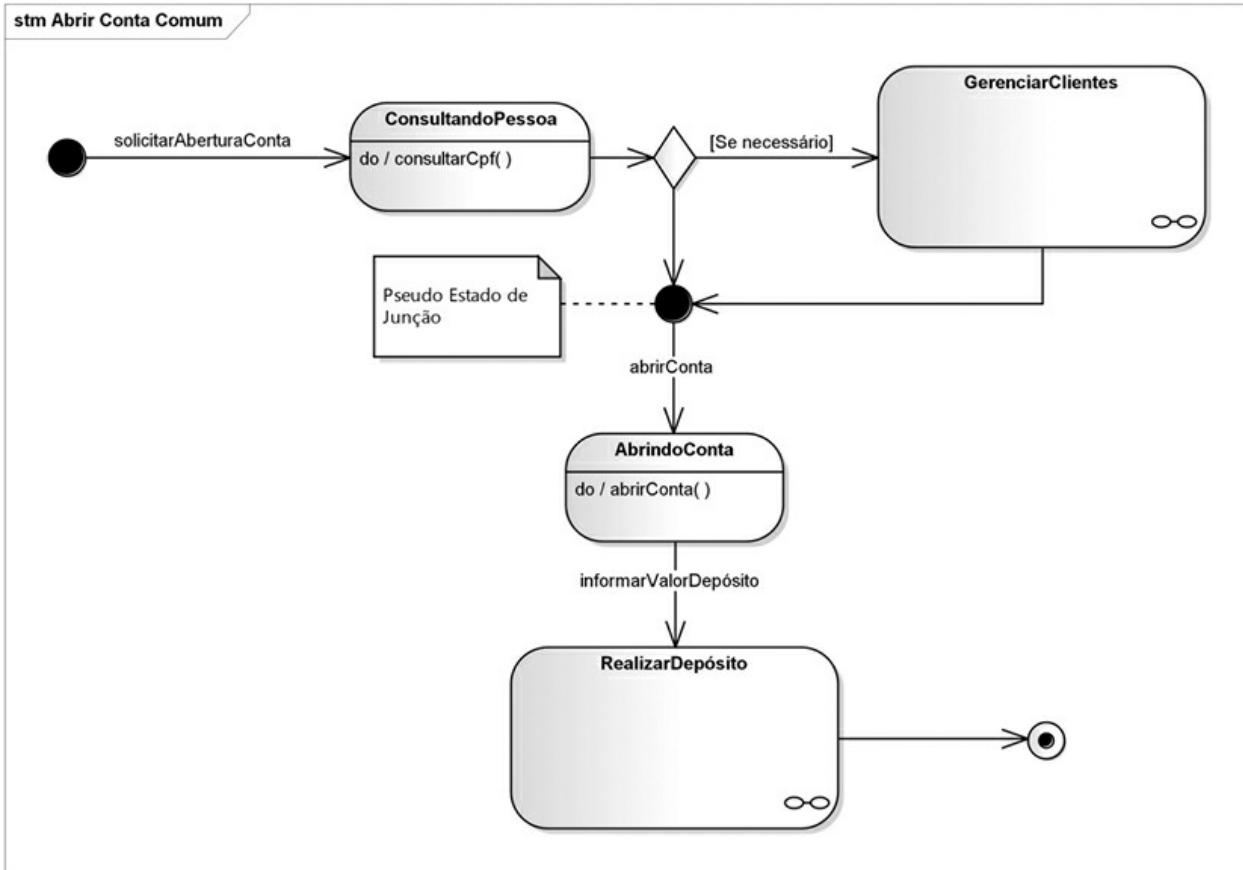


*Figura 9.15 – Estado de Submáquina.*

Aqui, utilizamos novamente o exemplo parcial do processo de **Abertura de Conta** já apresentado e substituímos o estado composto denominado **Gerenciar Clientes** por um estado de submáquina de mesmo nome, em que os subestados não são representados, mas deverão estar detalhados em outro diagrama de máquina de estados, normalmente com o nome do próprio estado de submáquina. Essa abordagem permite produzir diagramas mais enxutos, deixando os diagramas mais fáceis de entender, além de permitir referenciar outros diagramas já modelados.

## 9.16 Pseudoestado de Junção

Esse pseudoestado é utilizado para projetar caminhos transacionais complexos. Pode unir múltiplos fluxos em um único ou dividir um fluxo em diversos, sendo possível utilizar condições de guarda como auxílio. Um pseudoestado de junção é representado por um círculo preenchido. A figura 9.16 apresenta um exemplo de pseudoestado de junção utilizado durante o processo de abertura de conta.



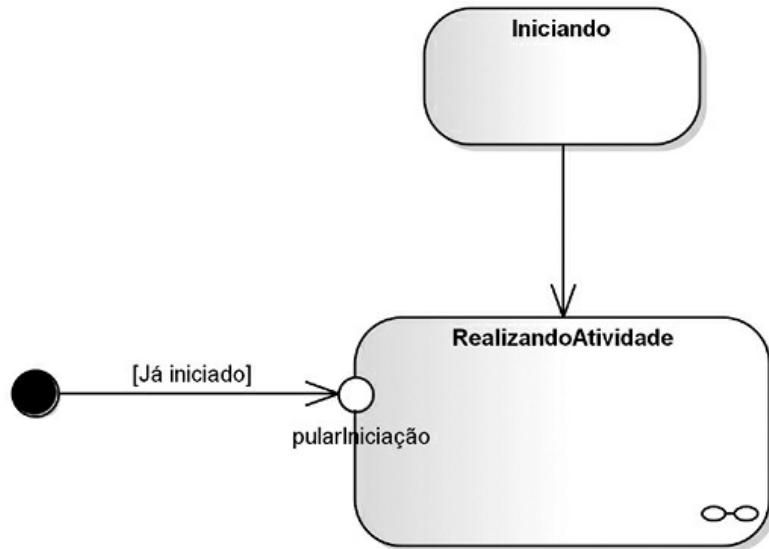
*Figura 9.16 – Pseudoestado de Junção – Processo de Abertura de Conta.*

Aqui, apresentamos o diagrama de máquina de estados equivalente ao processo de abertura de conta, que iniciamos a modelar nos exemplos anteriores. Depois de realizar o teste para determinar se é necessário efetuar manutenção no cadastro de clientes, observe que o fluxo foi dividido em dois, sendo novamente unido por um pseudoestado de junção, identificado aqui por uma nota. Após a união dos fluxos, gera-se uma transição para o estado **AbrindoConta**, em que é executado o método **abrirConta** e, depois de sua conclusão, gera-se uma transição para registrar o depósito inicial. Observe que essa transição aponta para um estado de submáquina em que estão detalhados os estados do processo de **Realizar Depósito**. Após a conclusão desse estado de submáquina, o processo é encerrado.

## 9.17 Pseudoestado de Ponto de Entrada e Pseudoestado de Ponto de Saída

Os pseudoestados de ponto de entrada e de ponto de saída são utilizados

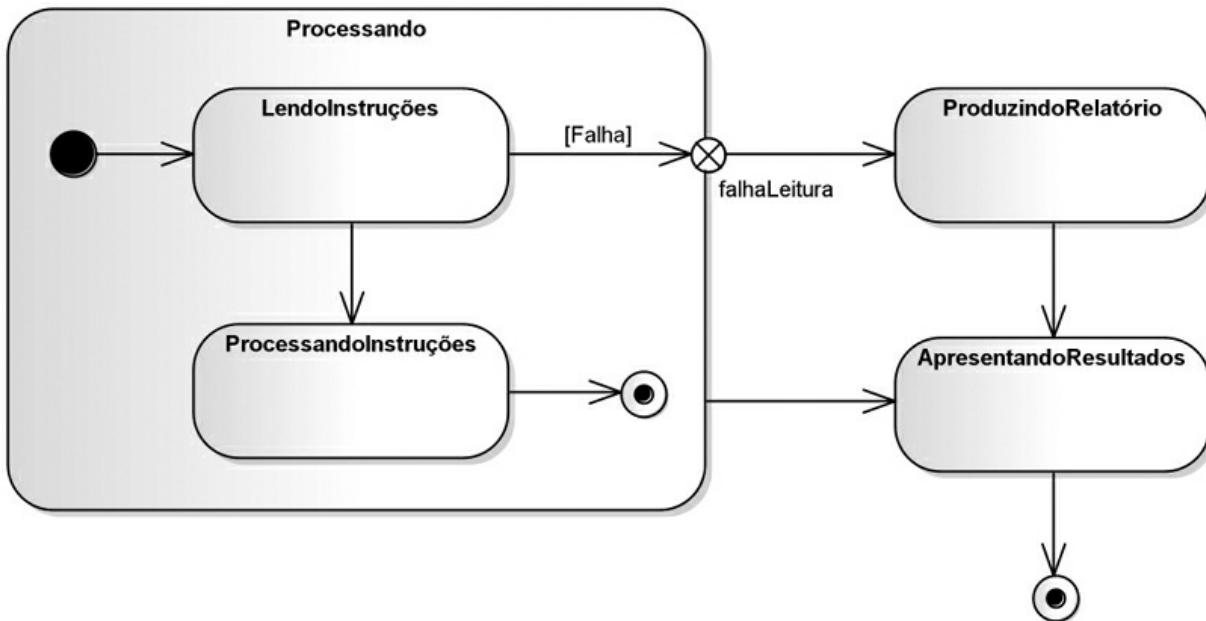
com estados de submáquina ou estados compostos. Demonstram pontos de entrada e saída que serão somente usados em casos excepcionais. O estado de entrada demonstra um caminho alternativo e é representado por um círculo vazio na borda do estado de submáquina. A figura 9.17 apresenta um exemplo de pseudoestado de ponto de entrada.



*Figura 9.17 – Pseudoestado de Ponto de Entrada.*

Nesse exemplo, o processo normal executa, primeiro, o estado denominado **Iniciando** para, depois, executar a atividade representada pelo estado de submáquina. No entanto, pode ser que não seja necessário executar o estado preliminar. Dessa forma, foi inserido no estado de submáquina um pseudoestado de entrada, chamado **pularIniciação**, por meio do qual se pode executar a atividade sem passar pela primeira etapa.

Em relação ao estado de saída, este normalmente demonstra uma exceção que causa o cancelamento do fluxo normal estado. É representado por um círculo com um X. A figura 9.18 apresenta um exemplo de pseudoestado de ponto de saída.



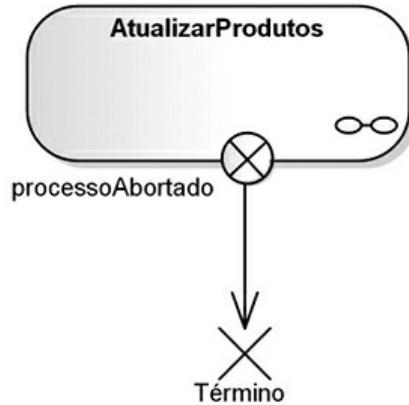
*Figura 9.18 – Pseudoestado de Ponto de Saída.*

Aqui, representamos um estado composto responsável por ler e processar instruções. Após sua conclusão, passa-se ao estado **ApresentandoResultados**. Porém, durante a leitura das instruções, pode ocorrer uma exceção, denotando falha de leitura das instruções. Essa exceção interrompe o processamento das instruções e gera um relatório de erros. Isso é representado por um pseudoestado de ponto de saída chamado de **falhaLeitura**.

Esses estados normalmente representam exceções, motivo pelo qual os exemplos apresentam também transições normais que não necessitam utilizar esses pseudoestados nem as entradas e saídas alternativas por eles representados.

## 9.18 Pseudoestado de Término

Força o término da execução de uma máquina de estados, em razão, por exemplo, da ocorrência de uma exceção. É representado por um “X”, conforme demonstra a figura 9.19.



*Figura 9.19 – Pseudoestado de Término.*

### 9.19 Exemplo de Diagrama de Máquina de Estados – Emitir Extrato

Nesta seção, apresentaremos os estados relativos ao caso de uso **Emitir Extrato** do sistema de controle bancário que vimos modelando ao longo do livro, como pode ser visto na figura 9.20.

O processo se inicia quando a transição contendo o número da conta a consultar é disparada, produzindo o estado **ConsultandoConta**, no qual é executado o método **consultarConta**. O estado seguinte, **RecebendoSenha**, é um estado estático que aguarda que uma senha seja informada. Quando a senha é recebida, é produzido o estado **ValidandoSenha** cujo objetivo é determinar se a senha informada corresponde à senha da conta consultada. Nesse estado é executado o método **validarSenha**.

Na sequência da validação da senha, produz-se o estado **RecebendoPeríodos**, que é um estado estático em que se aguarda o fornecimento dos períodos do extrato. No momento em que os períodos são informados, passa-se a um estado composto, em que se inicia a emissão do extrato propriamente dita. Nesse estado composto, há dois subestados. O primeiro deles representa o início do extrato, em que é executado o método **emitirExtrato**. A partir desse estado, gera-se uma transição para um novo estado no qual serão consultados todos os movimentos do período relativos à conta consultada, chamado **ConsultandoMovimento**, em que é executado o método **consultarMovimento**. O leitor notará que há uma autotransição para esse

estado, enquanto ainda houver movimentos dentro dos períodos informados.

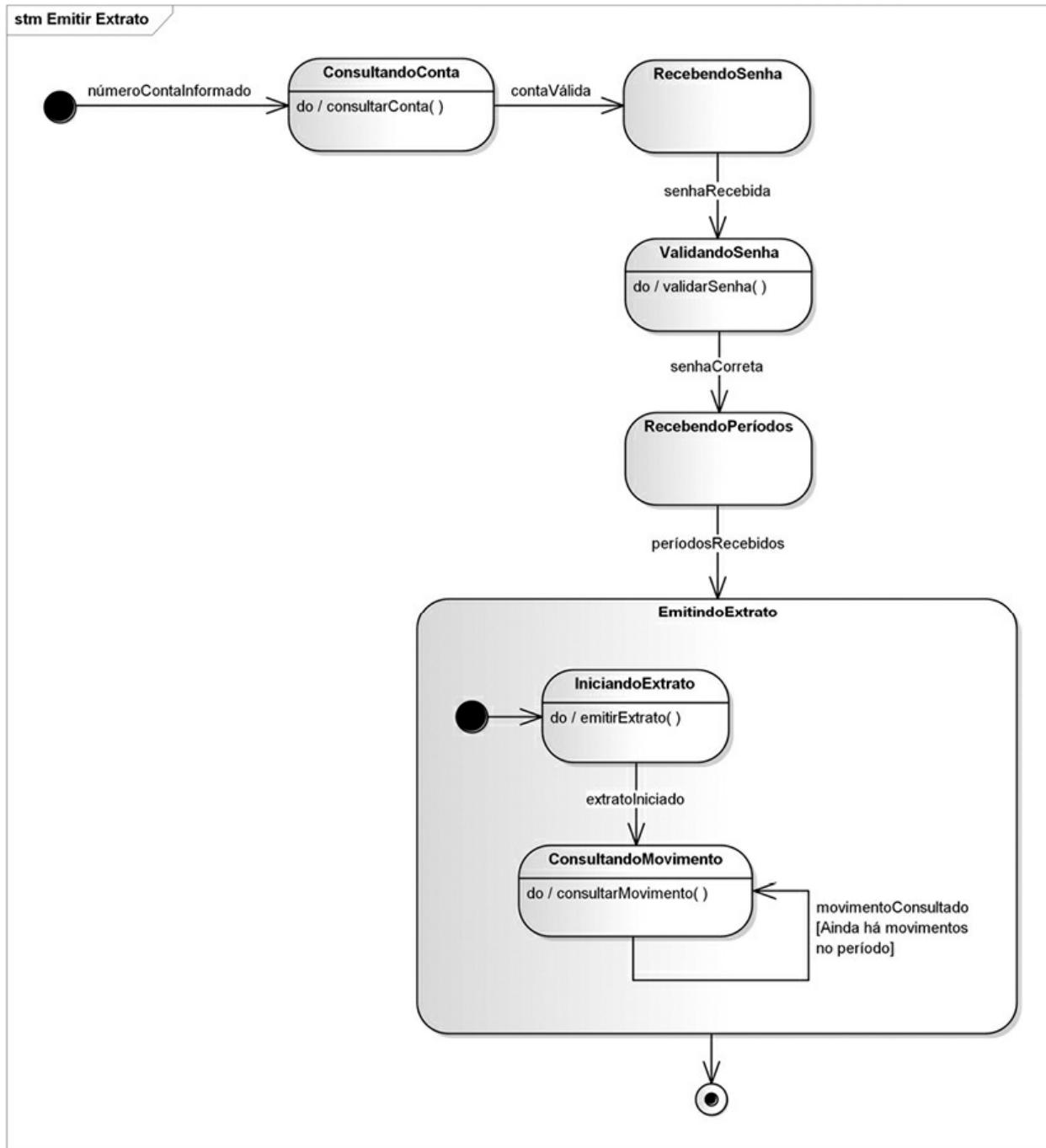
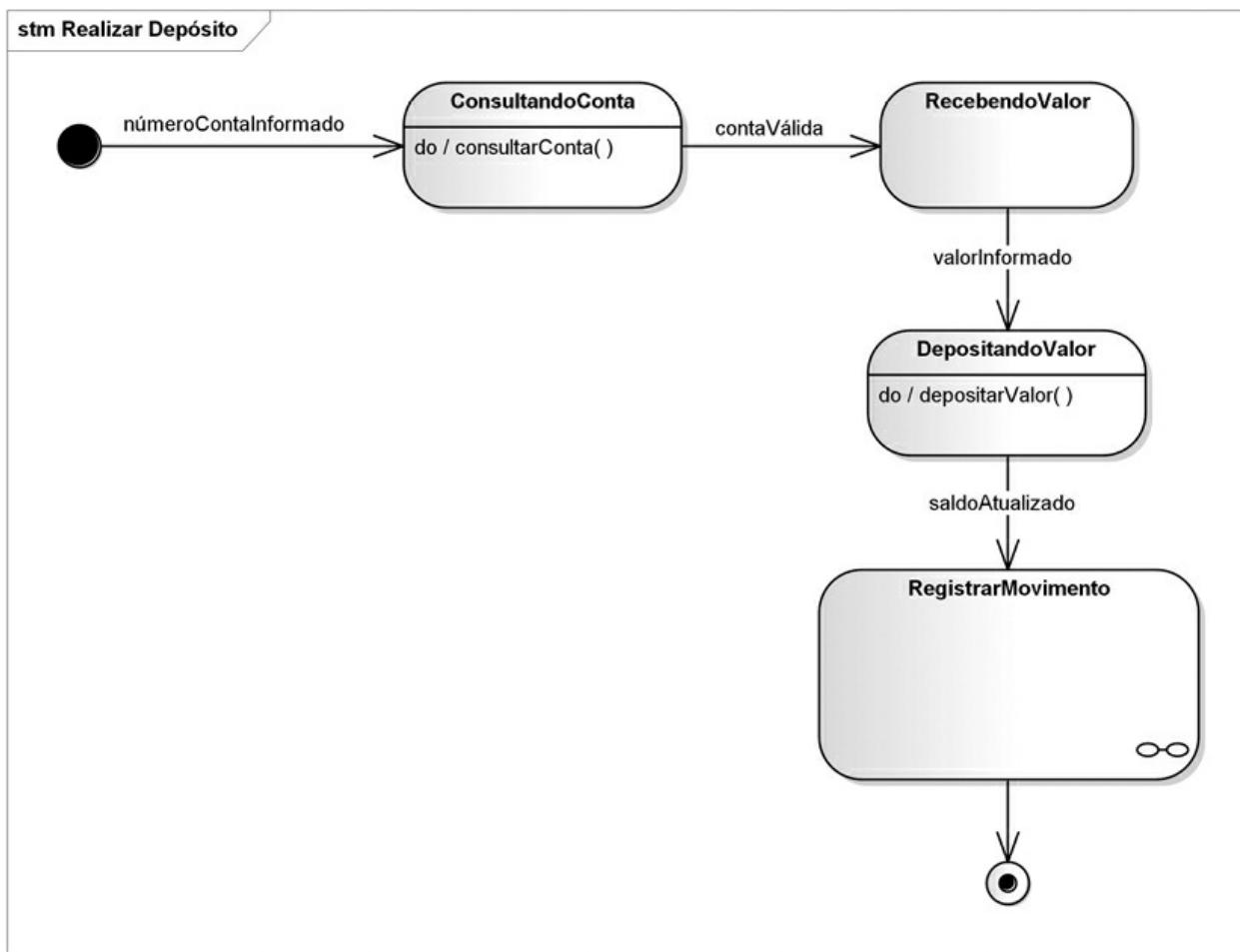


Figura 9.20 – Processo de Emissão de Extrato.

## 9.20 Exemplo de Diagrama de Máquina de Estados – Realizar Depósito

Nesta seção serão apresentados os estados relativos ao caso de uso

**Realizar Depósito** do sistema de controle bancário, por meio da figura 9.21.



*Figura 9.21 – Processo de Realizar Depósito.*

Esse diagrama se inicia quando o cliente fornece o número da conta para a qual deseja depositar um valor. Esse evento produz o estado **ConsultandoConta** em que é disparado o método **consultarConta**. Depois de a conta ter sido consultada, o evento que representa a informação de que a conta é válida gera o estado **RecebendoValor**, um estado estático em que se aguarda que o valor a depositar seja informado.

No momento em que o valor para depósito for fornecido, passa-se ao estado **DepositandoValor**, em que é executado o método **depositarValor**. A conclusão desse estado produz uma transição que leva a um estado de submáquina que representa o processo de **Registrar Movimento**. Depois de o registro do movimento ter sido concluído, essa máquina de estados é concluída.

## 9.21 Exemplo de Diagrama de Máquina de Estados – Realizar Saque

Já nesta seção, demonstraremos os estados relativos ao caso de uso **Realizar Saque**, apresentados pela figura 9.22.

Novamente, o processo se inicia com a informação do número da conta, o que gera o estado **ConsultandoConta**. Após esse estado ter sido concluído, passa-se a um estado estático que aguarda o recebimento da senha. O evento que representa o fornecimento da senha produz o estado **ValidandoSenha**, em que é executado o método **ValidarSenha**. A conclusão desse estado produz outro estado estático em que se espera que o valor a sacar seja informado.

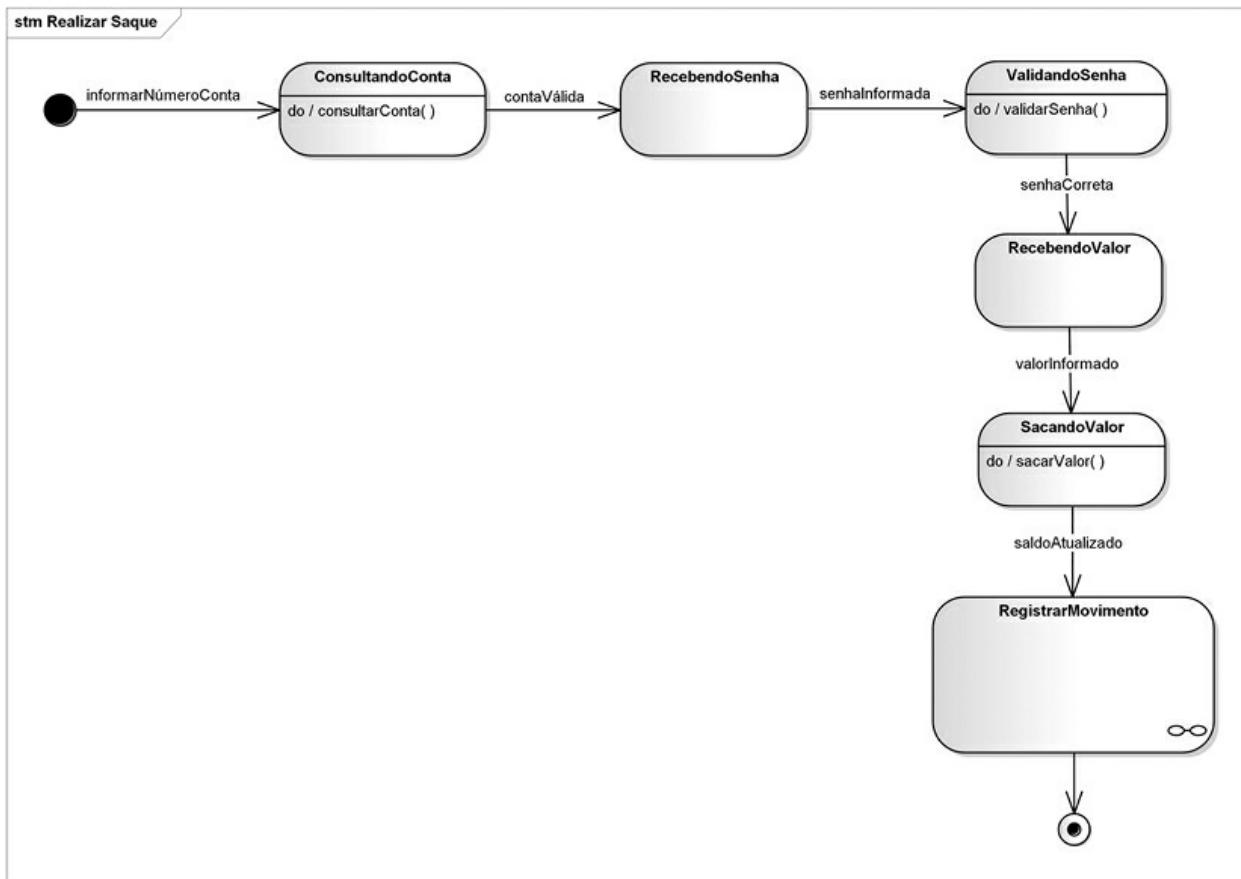


Figura 9.22 – Processo de Realizar Saque.

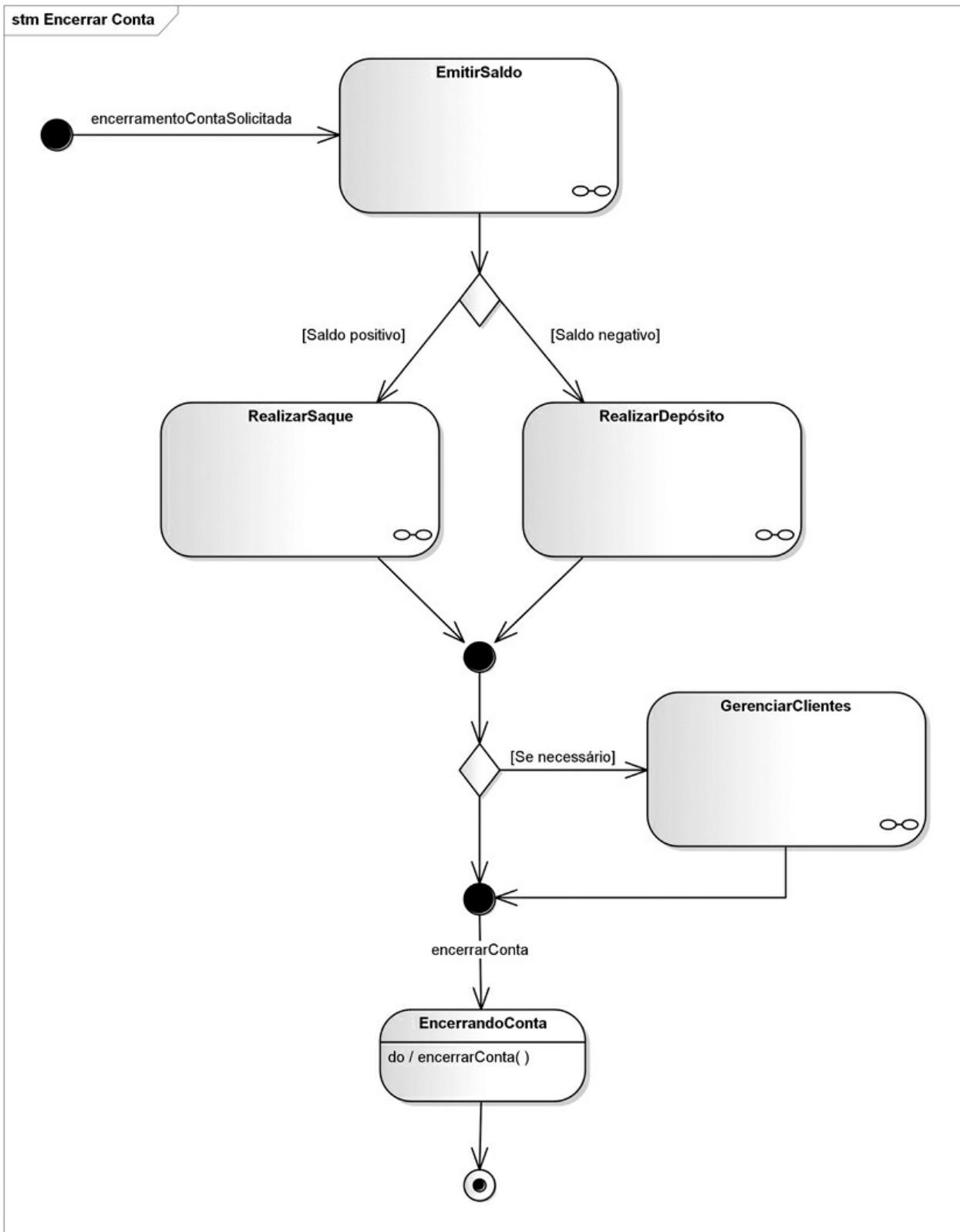
Ao concluir-se esse estado, ou seja, quando é informado o valor desejado para saque, uma transição gera o estado **SacandoValor**, no qual é chamado o método **SacarValor**. A retirada de um valor exige o registro do movimento, por isso, após a conclusão do estado, uma transição atinge um estado de submáquina que representa o processo de **Registrar Movimento**.

e, após seu término, a máquina de estados é concluída.

## 9.22 Exemplo de Diagrama de Máquina de Estados – Encerrar Conta

Finalmente, nesta seção apresentaremos os estados pertencentes ao caso de uso **Encerrar Conta** do sistema de controle bancário, como ilustra a figura 9.23. Esse diagrama é um pouco mais complexo que os anteriores.

Uma vez que diversos processos mais simples haviam sido modelados anteriormente, lançamos mão de estados de submáquina para representá-los e deixar esse diagrama menor. Assim, o processo de encerramento de conta se inicia com o evento de solicitação de encerramento de conta, mas a transição que o representa atinge um estado de submáquina relacionado ao processo de **Emitir Saldo**, uma vez que é necessário, primeiro, saber quanto se tem na conta antes de encerrá-la. A partir desse estado de submáquina, gera-se uma transição para um pseudoestado de escolha, em que é preciso escolher entre realizar um saque ou um depósito, dependendo de se o saldo estiver positivo ou negativo.



*Figura 9.23 – Processo de Encerrar Conta.*

Assim, se o saldo estiver positivo, gera-se uma transição para o estado de

submáquina **Realizar Saque**. Já se o saldo estiver negativo, gera-se uma transição para o estado de submáquina **Realizar Depósito**. Observe que o fluxo dividido pelo pseudoestado de escolha é unido novamente por um pseudoestado de junção e, a partir dele, é produzida uma nova transição para outro pseudoestado de escolha, em que se deve decidir se é necessário manter o cadastro do cliente, caso a conta a encerrar seja a única possuída por ele. Note ainda que novamente o fluxo dividido é unido por um novo pseudoestado de junção e, a partir deste, é produzida uma transição para o estado **Encerrando Conta**, no qual é executado o método `encerrarConta`. A conclusão desse estado também conclui o diagrama.

Poderíamos, no lugar dos estados de submáquina, ter usado estados compostos, pois estes são equivalentes às máquinas de subestados, exceto por detalharem seus subestados e não poderem ser reutilizados. Porém, isso deixaria o diagrama bem mais extenso. Poder-se-ia também detalhar todo esse processo por meio de estados simples, se o projetista assim o preferisse.

## 9.23 Exercícios Propostos

Como tem ocorrido nos capítulos anteriores, continuaremos a modelagem dos sistemas propostos pelos exercícios, enfocando agora o diagrama de máquina de estados.

### 9.23.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

Desenvolva o diagrama de máquina de estados referente ao processo de venda de ingressos para um sistema de controle de cinema, sabendo que:

- Ao selecionar a opção de venda de ingressos, o sistema deverá apresentar todas as sessões ainda não encerradas. Cada sessão deve informar o título do filme e a sala em que será apresentado.
- A partir da listagem apresentada, o funcionário deverá escolher a sessão desejada pelo cliente.
- Em resposta, o sistema deverá apresentar os assentos disponíveis da sessão.
- Com base nessa informação, o funcionário informa os assentos desejados pelo cliente e o tipo de ingresso que ele deseja, ou seja,

ingressos inteiros ou meias-entradas.

- Finalmente, o funcionário deverá gerar os ingressos referente à sessão escolhida.

### **9.23.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade**

Desenvolva o diagrama de máquina de estados referente ao processo de pagamento de mensalidade para um sistema de clube social, levando em consideração os seguintes fatos:

- Primeiramente, deve-se consultar o sócio que deseja pagar mensalidades.
- Após a consulta do sócio, deve-se consultar a(s) mensalidade(s) por ele devida(s).
- Se houver alguma mensalidade em atraso, será necessário calcular os juros referentes ao atraso do pagamento.
- Em seguida, deve-se informar quais mensalidades o sócio deseja quitar.
- Finalmente, deve-se quitar a(s) mensalidade(s) informada(s).

### **9.23.3 Sistema de Locação de Veículos – Processo de Locação de Veículo**

Desenvolva o diagrama de máquina de estados referente ao processo de locação de veículo para um sistema de aluguel de veículos, considerando que:

- Ao selecionar a opção de locação de veículos, o sistema deve carregar todos os clientes registrados.
- Em seguida, o sistema deve apresentar todos os veículos disponíveis. A listagem decorrente deve mostrar a descrição do automóvel, seu modelo e marca.
- A partir dessa listagem, o funcionário deve selecionar o cliente.
- Depois de o cliente ter sido selecionado, deve-se selecionar o automóvel.
- Finalmente, depois de selecionar o veículo, o funcionário poderá mandar gerar a locação do automóvel selecionado.

## **9.23.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão**

Desenvolva o diagrama de máquina de estados referente ao processo de realizar leilão para um sistema de controle de leilão via internet, de acordo com os seguintes requisitos:

- Ao receber a solicitação do serviço de realizar leilões, o sistema deve apresentar todos os leilões ainda não encerrados.
- A partir dessa listagem, o leiloeiro deve selecionar qual leilão deseja iniciar.
- No momento que um leilão for escolhido para ser iniciado, o sistema precisa carregar todos os itens a serem leiloados nele.
- A partir da listagem dos itens a serem leiloados, o leiloeiro deve escolher um item a leiloar e anunciá-lo.
- Se houver algum lance para o item anunciado, o sistema deve anunciá-lo e, em seguida, registrá-lo.
- Existe um tempo máximo de espera para que haja lances. Enquanto esse tempo não for atingido, o item permanecerá sendo anunciado.
- Quando o tempo máximo de espera por um lance for atingido, o processo deverá verificar se houve ofertas para o item, caso em que se deve anunciar o participante que ofereceu o maior lance como vencedor. Caso contrário, deve-se simplesmente encerrar o anúncio do item.
- Depois de ter sido encerrado o anúncio de um item, deve-se verificar se ainda há itens a anunciar, caso em que o processo passa a anunciar o novo item. Caso contrário, o leilão deve ser encerrado.

## **9.23.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias**

Desenvolva o diagrama de máquina de estados referente ao processo de pagamento de diárias para um sistema de controle de hotelaria, de acordo com as seguintes definições:

- No momento em que o hóspede informa o número do quarto para quitar as diárias, o sistema deve consultar o hóspede e todas as diárias

devidas pelo aluguel do quarto, apresentando-as ao funcionário.

- A partir dessa listagem, deve-se quitar as diárias apresentadas.
- Se tiver havido a solicitação de qualquer serviço no período em que o quarto estava ocupado, estes devem ser quitados também.
- Isso também ocorre se houver quaisquer consumos de frigobar, sendo necessário também os quitar.

### **9.23.6 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel**

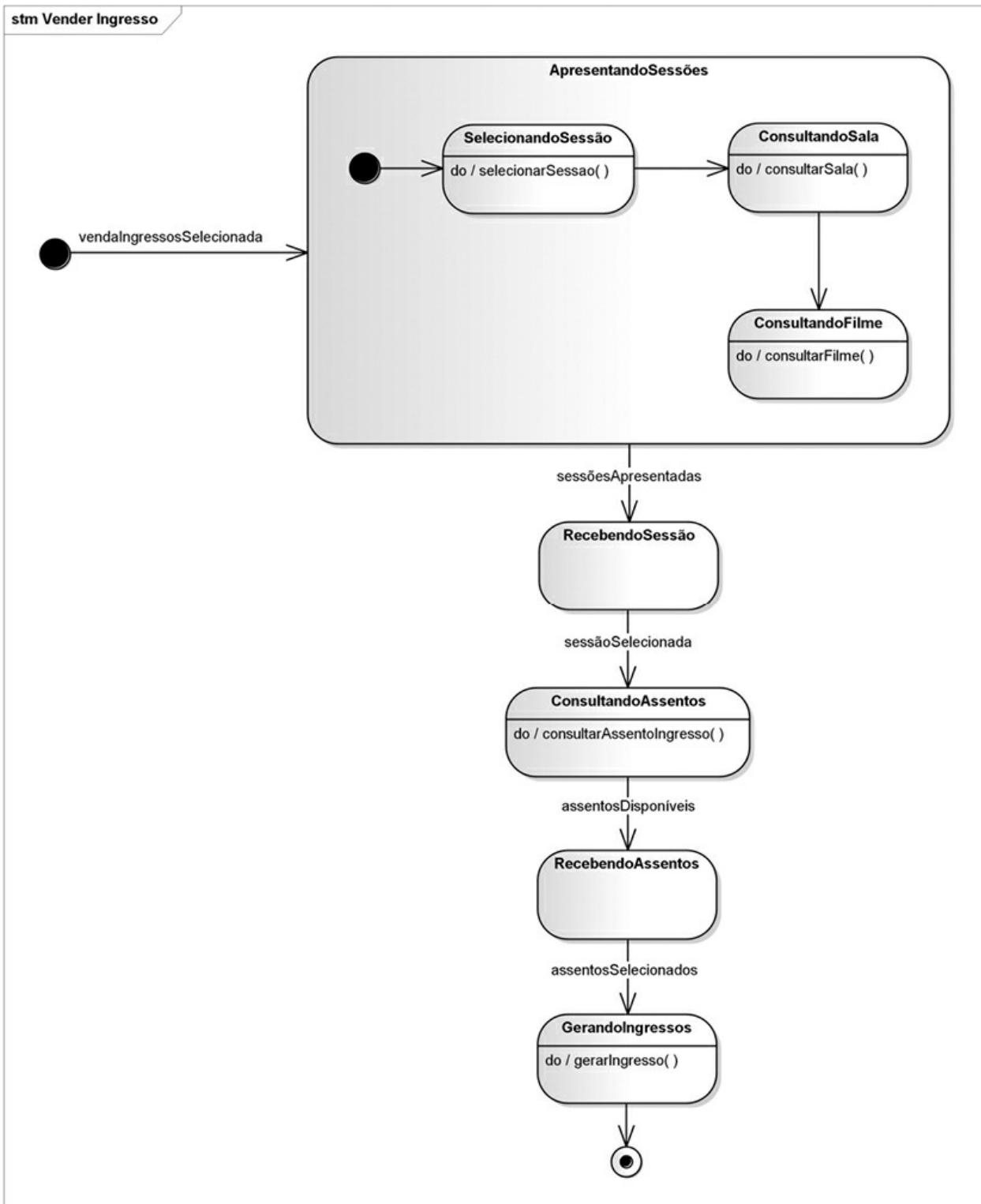
Desenvolva o diagrama de máquina de estados referente ao processo de venda de imóvel para um sistema de controle de imobiliária, sabendo que:

- Primeiramente, o corretor deve selecionar o tipo de imóvel que será vendido.
- Em seguida, o sistema deve listar todas os imóveis do tipo selecionado, bem como todas as pessoas registradas no sistema.
- Caso o cliente não esteja registrado, deve-se, então, registrá-lo.
- A seguir, deve-se selecionar o imóvel vendido, o cliente que o comprou e todas as taxas pagas na transação.
- Finalmente, deve-se calcular as comissões da imobiliária e do corretor, registrar a compra e todas as taxas pagas.

## **9.24 Solução dos Exercícios**

### **9.24.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos**

A figura 9.24 apresenta a solução para esse exercício. A seguir, explicaremos cada um de seus estados.



*Figura 9.24 – Processo de Venda de Ingressos.*

Esse processo se inicia com o funcionário escolhendo a opção de venda de ingressos. Essa escolha gera uma transição para o estado composto

**ApresentandoSessões**, que contém três subestados. No primeiro, é selecionada uma sessão ainda não encerrada, em que é executado o método **selecionarSessao**. O subestado seguinte consulta a sala onde ocorrerá a sessão, chamando o método **consultarSala**, e o terceiro subestado consulta o filme que será apresentado, executando o método **consultarFilme**.

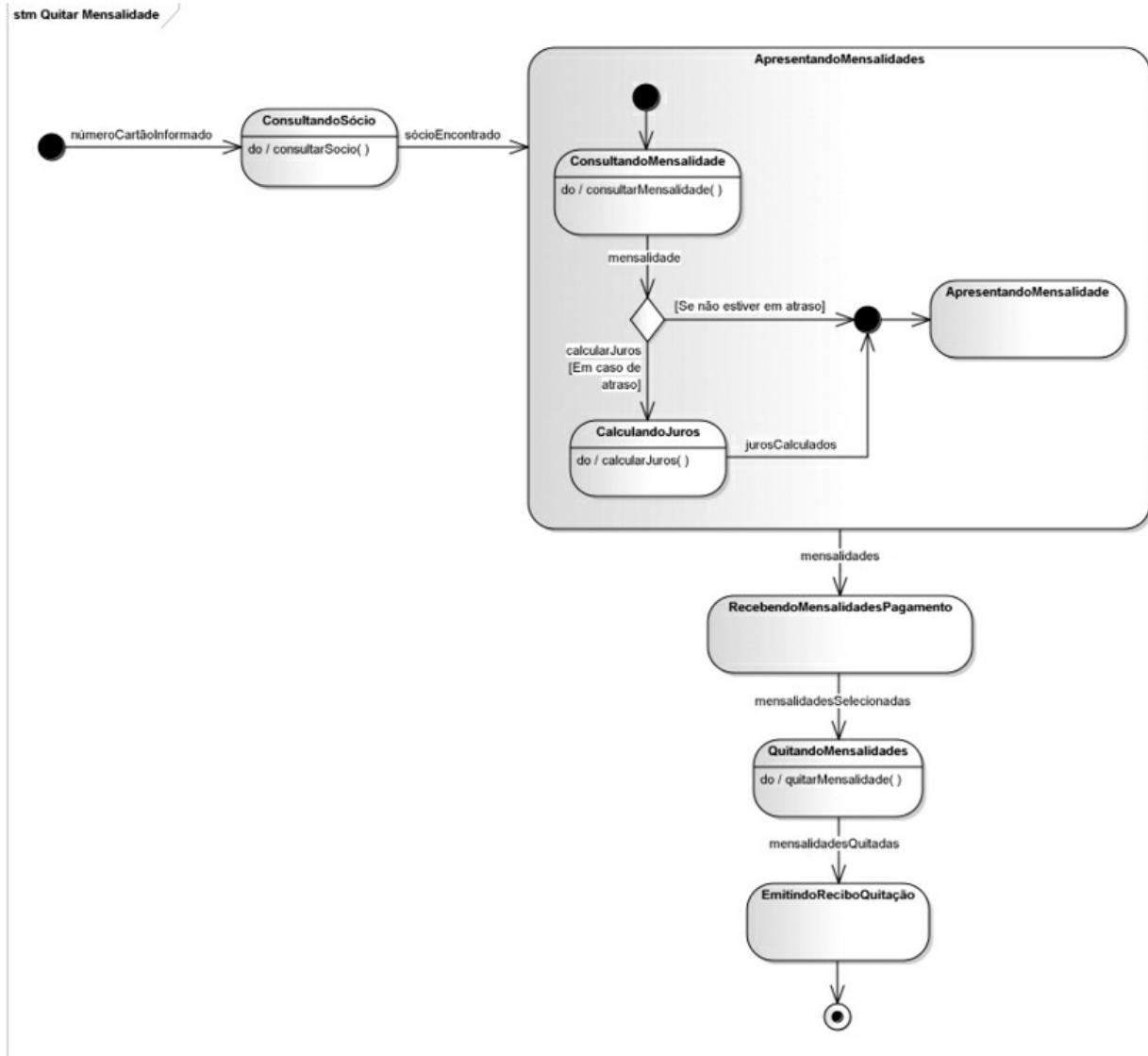
Ao finalizar a consulta de sessões disponíveis é gerada uma transição para um estado de espera (ou estático), em que o funcionário deverá escolher uma sessão. A escolha de uma sessão causa a transição para o estado **ConsultandoAssentos** em que todos os assentos disponíveis são apresentados. Ao concluir esse estado, passa-se ao estado estático **RecebendoAssentos**, em que se aguarda que os assentos desejados sejam escolhidos.

Quando os assentos forem escolhidos, passa-se ao último estado, **GerandoIngressos**, no qual será disparado o método **gerarIngresso** e, após a conclusão desse estado, a máquina de estados será encerrada.

Poder-se-ia argumentar que deveriam ter sido representadas autotransições nos estados descritos no plural, como **ConsultandoAssentos** ou **GerandoIngressos**, porém não consideramos necessário deixar isso explícito, uma vez que são estados simples e bastante autoexplicativos. Todavia, não estaria incorreto representar autotransições nesses estados para destacar que estes poderiam se repetir várias vezes.

## **9.24.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade**

A seguir, por meio da figura 9.25, apresentamos a solução para esse exercício.



*Figura 9.25 – Processo de Pagamento de Mensalidade.*

Esse processo se inicia com o sócio fornecendo o número de seu cartão, o que gera o estado **ConsultandoSócio** e a execução do método **consultarSocio**. Depois dessa consulta, uma transição passa ao estado composto **ApresentandoMensalidades**. Esse estado possui três subestados, **ConsultandoMensalidade**, **CalculandoJuros** e **ApresentandoMensalidade**.

No primeiro subestado é consultada uma mensalidade específica e executado o método **consultarMensalidade**. Após a mensalidade ser consultada, uma nova transição atinge um pseudoestado de escolha em que deverá ser escolhido um entre dois fluxos possíveis. Caso o sócio não tenha mensalidades em atraso, passa-se ao estado

**ApresentandoMensalidade**, que simplesmente apresenta as informações da mensalidade a ser paga. Caso a mensalidade esteja em atraso, então se gera o estado **CalculandoJuros**, em que é disparado o método `calcularJuros` e, após o término desse método, passa-se ao estado **ApresentandoMensalidade** já explicado. Observe que o fluxo dividido é unido por um pseudoestado de junção.

Após as mensalidades terem sido apresentadas, passa-se a um estado estático em que se espera que o sócio escolha quais mensalidades deseja pagar. Essa escolha gera uma nova transição, que produz o estado **QuitandoMensalidades**, em que é executado o método `quitarMensalidade`. O término desse estado se caracteriza por uma transição para o estado **EmitindoReciboQuitação**, após o que a máquina de estados se encerra.

### **9.24.3 Sistema de Locação de Veículos – Processo de Locação de Veículo**

Apresentaremos aqui os estados da solução deste exercício, ilustrado pela figura 9.26.

Esse processo inicia-se com o funcionário solicitando o serviço de locação de veículos, o que causa uma transição para o estado **ConsultandoClientes**, que carrega os clientes registrados por meio da execução do método `consultarClientes`. Poder-se-ia acrescentar uma autotransição a esse estado, posto que pode se repetir várias vezes enquanto houver clientes a consultar.

Após a conclusão desse estado, passa-se ao estado composto **ConsultandoVeículos**, para carregar os veículos disponíveis. Esse estado composto apresenta três subestados. No primeiro, é consultado um automóvel e executado o método `consultarAutomovel`, no segundo é consultado o modelo do automóvel pesquisado no subestado anterior, em que é executado o método `consultarModelo` e, finalmente, no terceiro subestado, é consultada a marca do automóvel e executado o método `consultandoMarca`.

Ao ser finalizado, esse estado composto gera uma transição contendo os veículos disponíveis e passa-se ao estado **SelecionandoCliente**, em que o processo aguarda que o funcionário escolha o cliente interessado na locação. A escolha de um cliente gera uma transição para o estado

**SelecionandoVeículo**, em que o processo aguarda que seja selecionado um dos automóveis disponíveis. Por sua vez, a seleção de um automóvel gera uma transição para o estado **GerandoLocação**, em que o veículo é locado por meio da execução do método `registrarLocacao`.

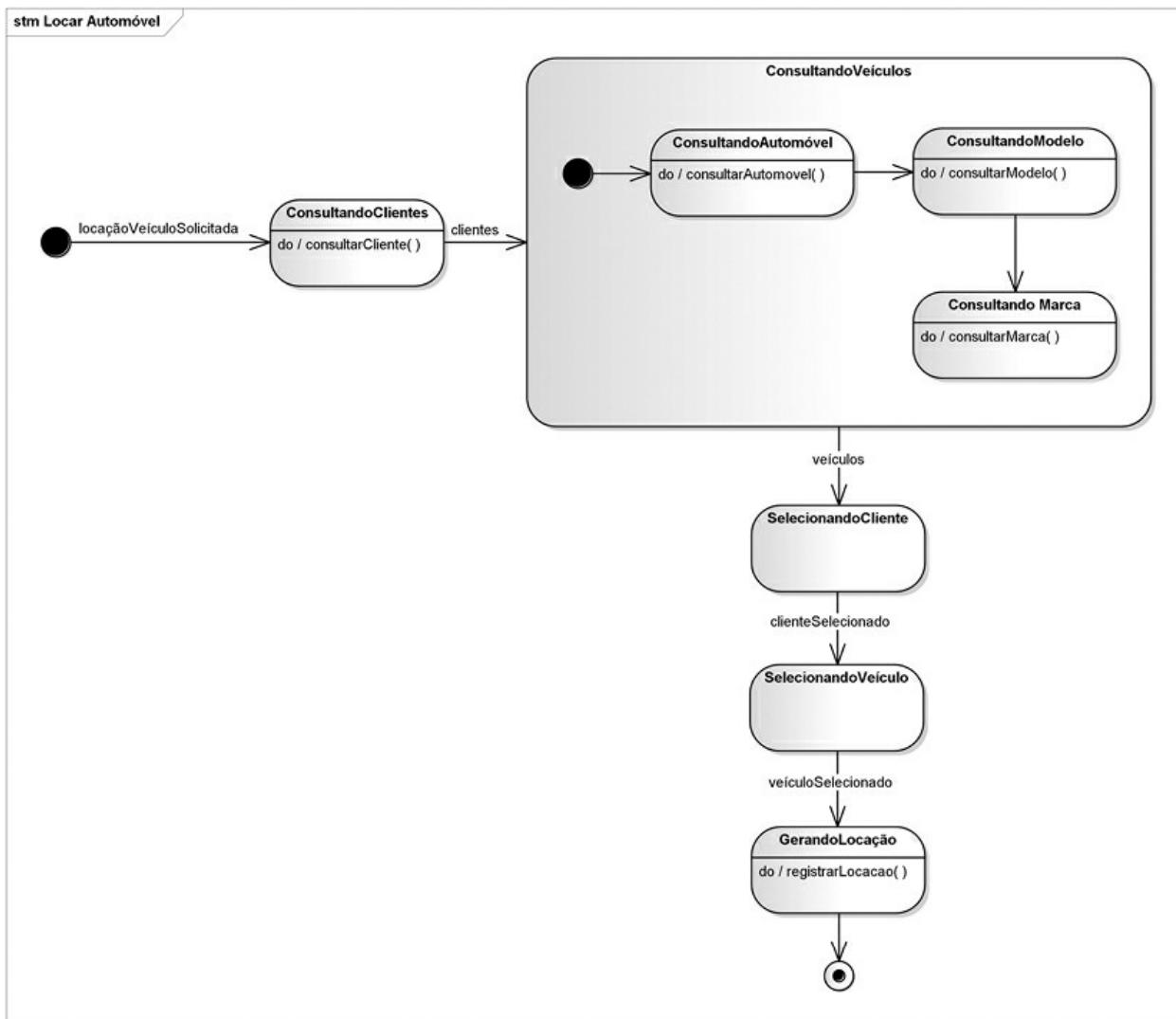


Figura 9.26 – Processo de Locação de Veículo.

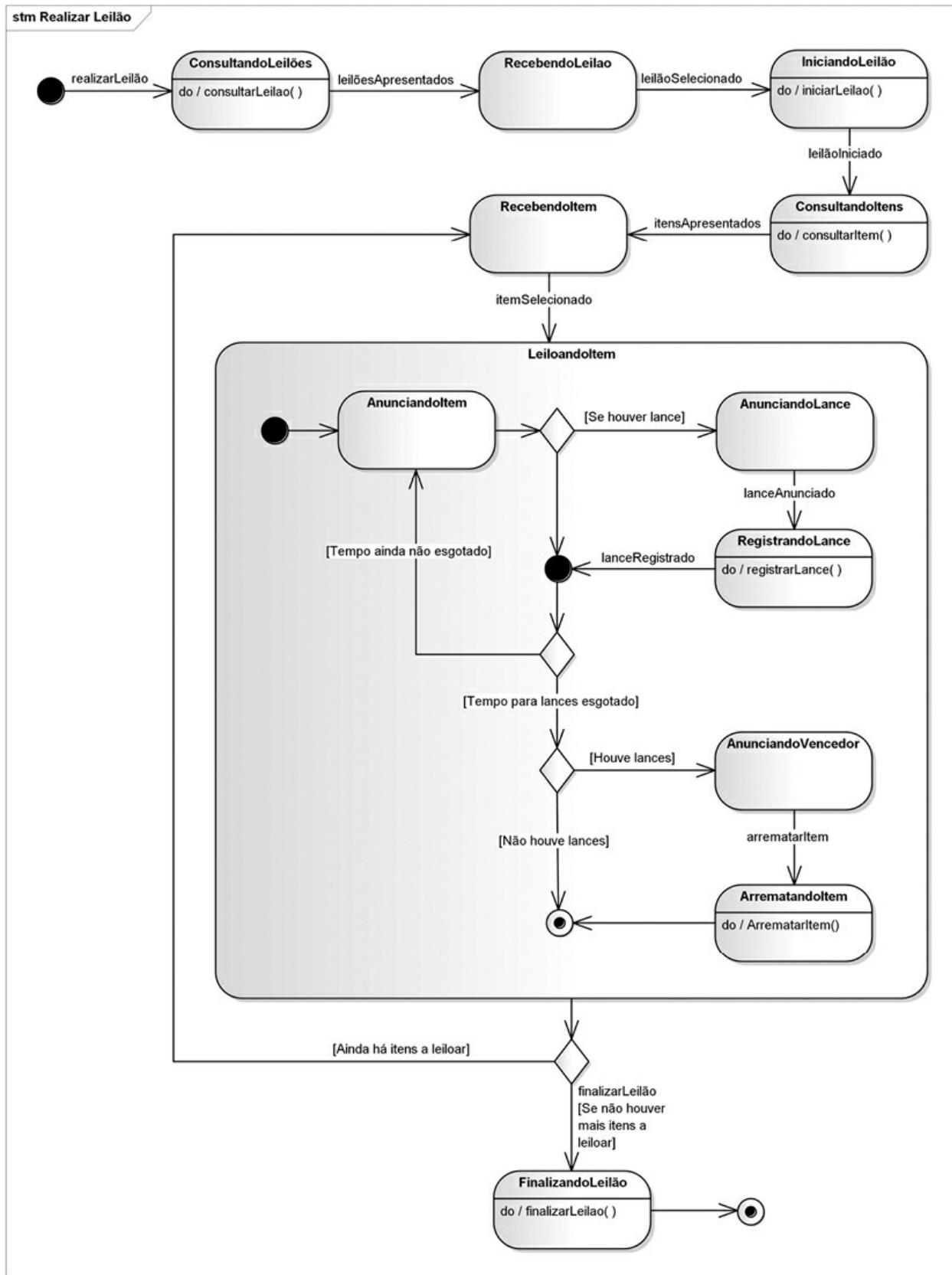
#### 9.24.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão

Esse processo inicia-se com o leiloeiro solicitando a execução do serviço de realizar leilões, como pode ser observado na figura 9.27. Isso causa uma transição que leva ao estado **ConsultandoLeilões**, que tem por função apresentar os leilões ainda não encerrados, executando o método

**consultarLeilao**. A partir dessa listagem, o leiloeiro deve selecionar qual leilão deseja iniciar, o que gera o estado **RecebendoLeilao**.

No momento em que um leilão é selecionado, passa-se ao estado **IniciandoLeilao**, em que é chamado o método **iniciarLeilao** e, após a conclusão desse estado, é gerada uma transição para o estado **ConsultandoItens**, cujo objetivo é carregar os itens a serem leiloados, no qual se executa o método **consultarItem**.

A partir da listagem dos itens a serem leiloados, o leiloeiro deve escolher um item a leiloar, o que é representado pelo estado **RecebendoItem**. A seleção de um item leva a um estado composto chamado **LeiloandoItem**. O primeiro subestado desse estado composto representa o anúncio do item a ser leiloadado. Durante esse estado, a página do leilão será atualizada para mostrar o referido item. A partir desse subestado, passa-se a um pseudoestado de escolha, no qual deve haver um teste. Se houver algum lance para o item anunciado, então se passa ao subestado que representa o anúncio dessa oferta e, em seguida, ao subestado que representa o registro dela, em que se executa o método **registrarLance**. Observe que o fluxo dividido pelo pseudoestado de escolha é unido novamente por um pseudoestado de junção.



*Figura 9.27 – Processo de Realizar Leilão.*

O pseudoestado de junção gera uma transição para um segundo pseudoestado de escolha, em que é preciso decidir se o tempo máximo de espera para que haja lances para o item foi atingido ou não. Se o tempo máximo não foi atingido, o processo volta ao estado em que o item está sendo anunciado e continua aguardando lances. Caso contrário, passa-se a um terceiro pseudoestado de escolha, em que se deve verificar se houve ofertas para o item, caso em que se gera uma transição para o subestado **AnunciandoVencedor** e, em seguida, para o subestado **ArrematandoItem**, em que é executado o método `arrematarItem`, encerrando-se os subestados do estado composto. Se não tiver ocorrido nenhuma oferta, o estado composto será simplesmente encerrado.

A conclusão do estado composto gera duas possíveis transições. A primeira verifica se ainda há itens a anunciar, caso em que o processo volta ao estado **RecebendoItem**, o que levará a um novo anúncio de item, como já foi explicado. Caso não haja mais itens a anunciar, gera-se uma transição para o último estado, no qual o leilão é encerrado, por meio do disparo do método `finalizarLeilao`, e a máquina de estados é concluída.

#### **9.24.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias**

A solução para esse exercício está representada na figura 9.28, cujos estados serão explicados a seguir.

Esse processo se inicia quando o hóspede informa o número do quarto do qual deseja quitar as diárias, o que gera uma transição para o estado composto **SelecionandoDiárias**. Nesse estado, existem quatro subestados. O primeiro é responsável por consultar o quarto informado, onde será disparado o método `selecionarDiarias`. Em seguida, passa-se ao segundo subestado, no qual é executado o método `consultarDiariasLocacao`, para fazer a ligação entre o quarto e o hóspede que o aluga, bem como para recuperar o período de locação. Depois, passa-se ao terceiro subestado, responsável por consultar o hóspede que está alugando o quarto, no qual é executado o método `consultarHospede`. Finalmente, passa-se para o último subestado, responsável por consultar as diárias devidas, disparando o método `consultarDiaria`.

O término desse estado composto gera uma transição para o estado **RecebendoDiariasParaPagamento**, em que são informadas quais diárias o hóspede quitará. Isto gera o estado **QuitandoDiárias**, no qual é executado o método **quitarDiaria**. A partir desse estado, uma transição encontra um pseudoestado de escolha, em que se deve testar se houve a solicitação de algum serviço no período em que o quarto estava ocupado, caso em que é gerada uma transição para um estado de submáquina referente ao processo de **Quitar Serviços**.

O fluxo dividido pelo pseudoestado de escolha é reunido por um pseudoestado de junção que gera uma transição para um segundo pseudoestado de escolha, em que é necessário testar se houve algum consumo de frigobar, caso em que é gerada uma transição para um segundo estado de submáquina que representa o processo de **Quitar Consumo**. O fluxo dividido é reunido por um segundo pseudoestado de junção e a máquina de estados é encerrada.

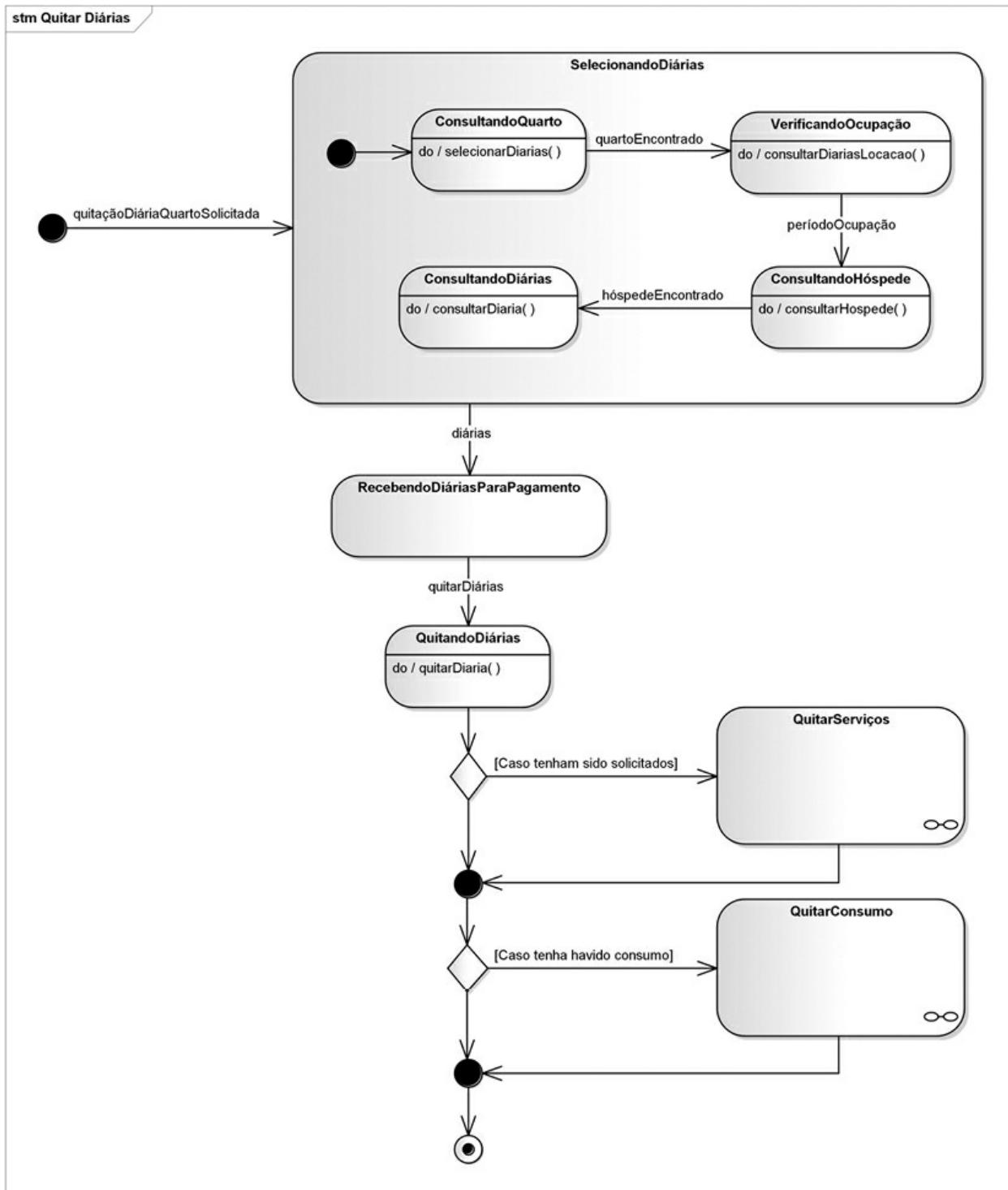


Figura 9.28 – Processo de Pagamento de Diárias.

### 9.24.6 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel

A figura 9.29 ilustra a solução para esse exercício.

Após o evento inicial, produz-se o estado **RecebendoTipoImóvel**, em que se aguarda que o tipo de imóvel a ser vendido seja informado. Quando a seleção do tipo de imóvel ocorre, gera-se o estado **ListandoImóveis**, em que são apresentados todos os imóveis do tipo selecionado. Depois, passa-se ao estado **ListandoPessoas**, que apresenta todas as pessoas registradas no sistema.



Figura 9.29 – Processo de Venda de Imóveis.

A partir dessa listagem é realizado um teste para determinar se o comprador se encontra na lista, caso contrário é executado o processo de **Gerenciar Clientes**, para cadastrá-lo. Esse processo é representado aqui por

uma máquina de subestados.

Depois disso, passa-se ao estado **RecebendoImóvel**, em que se aguarda que o imóvel desejado seja selecionado. Depois, passa-se ao estado em que se aguarda que o comprador seja escolhido e, então, ao estado em que os valores da transação são informados.

Nos estados seguintes são calculadas as comissões da imobiliária e do corretor, registrada a compra e, finalmente, registradas as taxas pagas, o que finaliza a máquina de estados.

## CAPÍTULO 10

# Diagrama de Atividade

O diagrama de atividade era considerado um caso especial do antigo diagrama de gráfico de estados, chamado diagrama de máquina de estados atualmente. A partir da versão 2.0 da UML, esse diagrama passou a ser totalmente independente, deixando até de se basear em máquinas de estados e passando a se basear em redes de Petri.

A modelagem de atividade enfatiza a sequência e condições para coordenar comportamentos de baixo nível. Dessa forma, o diagrama de atividade é o diagrama com mais ênfase no nível de algoritmo da UML e provavelmente um dos mais detalhistas. Esse diagrama apresenta muitas semelhanças com os antigos fluxogramas utilizados para desenvolver a lógica de programação e determinar o fluxo de controle de um algoritmo.

Esse diagrama é utilizado, como o próprio nome indica, para modelar atividades, as quais podem descrever:

- Computação procedural, que corresponde aos passos necessários para que uma operação (método) de uma classe seja concluída. Pode também incluir a chamada a outras atividades, como no caso de uma invocação de outro método.
- Modelagem organizacional para engenharia de processos de negócios e workflow (fluxo de trabalho), ou seja, definição das etapas necessárias para que um processo seja executado.
- Modelagem de sistemas de informação para especificar processos no nível de sistema.

Nas duas últimas situações, os diagramas de atividade podem ser empregados para modelar a lógica de um caso de uso, detalhando melhor suas etapas, ajudando a compreender sua real complexidade ou validando a definição do próprio caso de uso, ou seja, garantindo que o caso de uso realmente identifica a funcionalidade que ele representa.

Em alguns processos de desenvolvimento de software, como o RUP, os diagramas de atividade podem ser produzidos até antes dos casos de uso, como pode ocorrer durante a disciplina de Modelagem de Negócio. Nela, o designer de negócio pode utilizar diagramas de atividade para representar, de forma geral, os fluxos de trabalho dos setores onde o software virá a ser implantado, bem como os fluxos de trabalho gerais do próprio sistema. Os modelos de negócio produzidos podem auxiliar a identificar muitos casos de uso do software.

É mais comum que um diagrama de atividade represente uma única atividade, todavia é possível que um diagrama de atividade contenha mais de uma delas, se isto for considerado necessário. Uma atividade é composta de um conjunto de ações, ou seja, os passos necessários para que a atividade seja concluída, no entanto nem sempre essas ações estarão representadas dentro da atividade, como em situações em que é necessário fazer referência a uma atividade já modelada ou para poupar espaço no diagrama.

## 10.1 Atividade

Uma atividade especifica a coordenação de execuções de comportamentos subordinados usando um modelo de fluxo de controle e dados. Esses comportamentos subordinados podem ser iniciados em razão de outros comportamentos no modelo terminarem sua execução, por objetos e dados tornarem-se disponíveis ou pela ocorrência de eventos externos. Uma atividade é representada por um retângulo grande com as bordas arredondadas, conforme demonstra a figura 10.1.



*Figura 10.1 – Exemplo de Atividade.*

Esse exemplo representa a atividade referente ao processo de emissão de saldo de uma conta pertencente ao sistema de controle bancário que estamos modelando ao longo do livro. Durante este capítulo, detalharemos as etapas dessa atividade.

## 10.2 Nô de Ação

Nós de ação são os elementos mais básicos de uma atividade. Um nó de ação representa um passo, uma etapa que deve ser executada em uma atividade. Um nó de ação é atômico, não podendo ser decomposto. É representado por um pequeno retângulo com as bordas arredondadas, semelhante a uma atividade, porém o símbolo do nó de ação é menor. A figura 10.2 apresenta um exemplo de nó de ação.

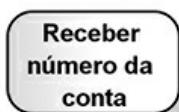


Figura 10.2 – Nô de Ação.

Esse exemplo representa a ação inicial da atividade de emissão de saldo, em que se deve receber o número da conta informada pelo cliente.

Atividades podem conter ações de vários tipos, como:

- ocorrências de funções primitivas, como funções aritméticas;
- invocação de comportamento, como outras atividades;
- ações de comunicação, como envio de sinais;
- manipulação de objetos, como leitura ou gravação de atributos ou mesmo instanciação ou destruição de objetos.

## 10.3 Fluxo de Controle

O fluxo de controle é um conector que liga dois nós, enviando sinais de controle de um nó para o outro. É representado por uma linha contendo uma seta apontando para o novo nó e partindo do antigo. A figura 10.3 apresenta um exemplo de fluxo de controle.

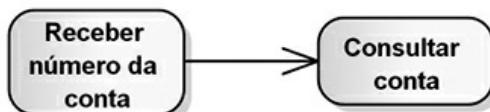


Figura 10.3 – Fluxo de Controle.

Nesse exemplo, que dá continuidade ao anterior, há dois nós de ação. A primeira ação representa o recebimento do número de uma conta, enquanto a segunda representa a consulta da conta informada. Observe

que a passagem de uma ação para outra é representada pela linha do fluxo de controle que une as duas ações. Note ainda que a seta do fluxo de controle demonstra a ordem em que as ações serão executadas.

Um fluxo de controle pode também conter uma descrição, uma condição de guarda e/ou uma restrição, chamada, nesse diagrama, de peso (weight), que determina, por exemplo, o número mínimo de sinais que devem ser transmitidos pelo fluxo. Um sinal (token) pode conter valores de controle, objetos ou dados, mas esses dois últimos somente podem ser transmitidos por meio de um fluxo de objetos, que será explicado neste capítulo.

## 10.4 Nó Inicial

Esse componente pertence ao grupo de nós de controle utilizados para o controle de fluxo da atividade. Esse nó é usado para representar o início do fluxo quando a atividade é invocada. É representado por um círculo preenchido. A figura 10.4 apresenta um exemplo de nó inicial.



Figura 10.4 – Nó Inicial.

Como o leitor pode observar, complementamos o exemplo anterior acrescentando um nó inicial à figura, indicando o início do diagrama.

## 10.5 Nó de Final de Atividade

Esse componente é também um nó de controle usado para representar o fim do fluxo de uma atividade. É representado por um círculo preenchido dentro de um círculo vazio. A figura 10.5 apresenta um exemplo de nó final.

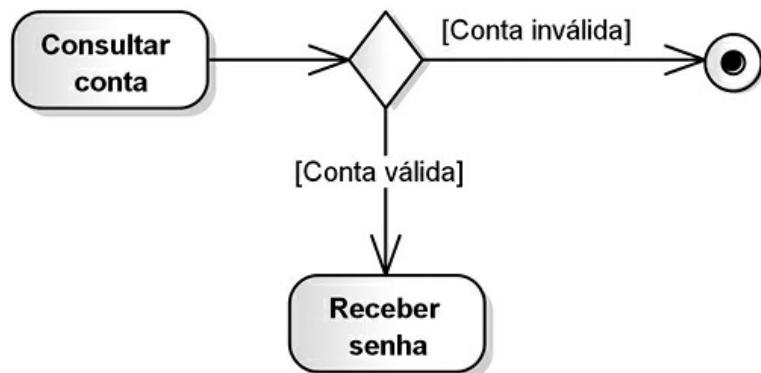


*Figura 10.5 – Nó de Final de Atividade.*

Aqui, apresentamos o final do fluxo iniciado nas figuras anteriores (obviamente faltam ações intermediárias), onde, depois de apresentar o saldo da conta, o fluxo da atividade é encerrado.

## 10.6 Nó de Decisão

Um nó de decisão é também um nó de controle, utilizado para representar uma escolha entre dois ou mais fluxos possíveis, em que um dos fluxos será escolhido em detrimento dos outros. Em geral, um nó de decisão é acompanhado por condições de guarda, ou seja, textos entre colchetes que determinam a condição para que um fluxo possa ser escolhido. Um nó de decisão pode ser utilizado também para unir um fluxo dividido por um nó de decisão anterior, quando passa a chamar-se nó de união. Na figura 10.6 é apresentado um exemplo de nó de decisão.



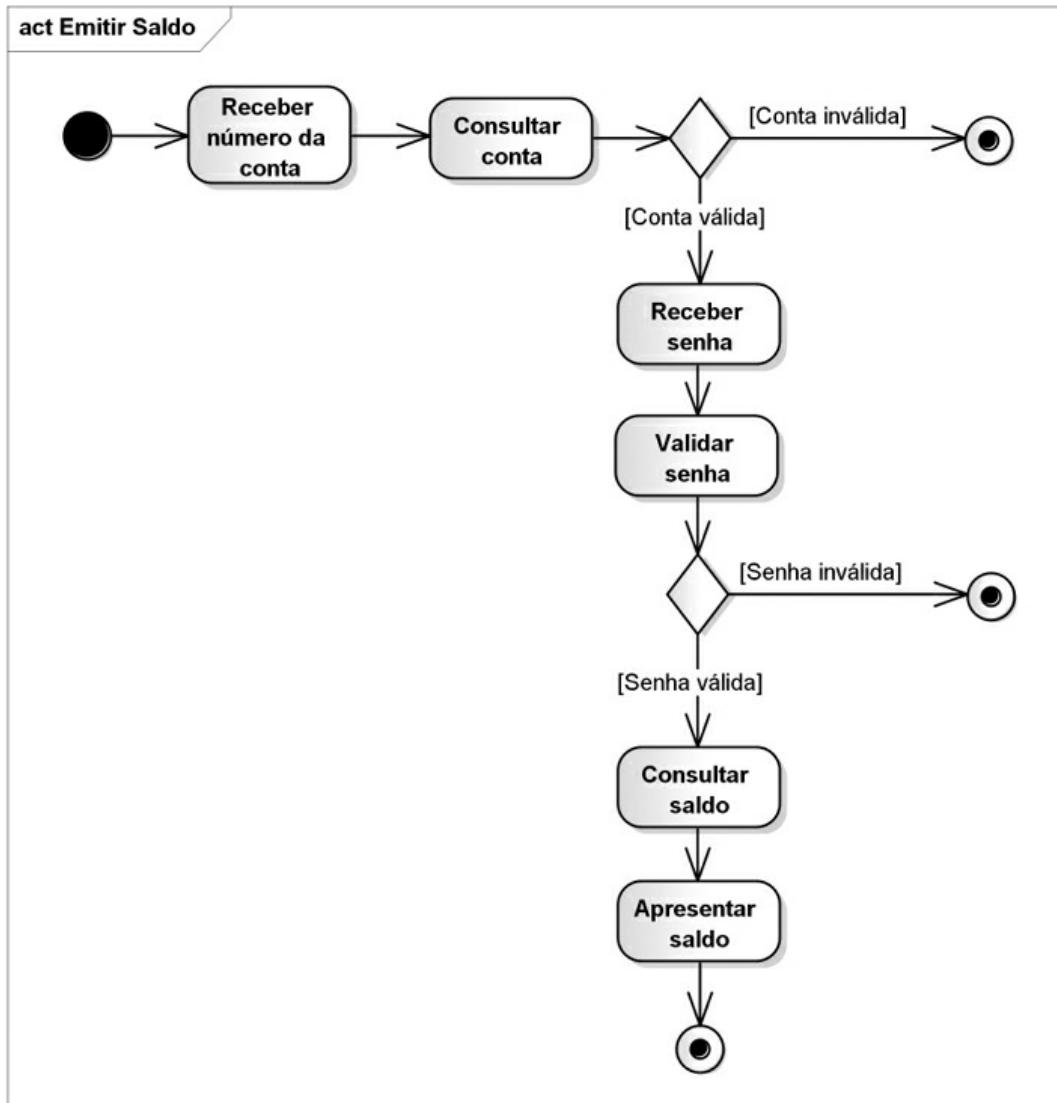
*Figura 10.6 – Nó de Decisão.*

Nesse exemplo, continuamos a atividade que representa o processo de emissão de saldo, em que, depois de consultar a conta, deve-se tomar uma decisão, representada por um losango. Caso a conta seja inválida, o processo deve ser encerrado, caso contrário, deve-se receber a senha da conta. Observe que as condições dessa decisão estão definidas por condições de guarda (texto entre colchetes), posicionadas sobre os dois fluxos alternativos.

## 10.7 Exemplo Simples de Diagrama de Atividade

Nesta seção demonstraremos um exemplo de diagrama de atividade, no

qual modelamos a atividade completa de emissão de saldo, que foi iniciada nas figuras anteriores. A atividade é apresentada pela figura 10.7.



*Figura 10.7 – Exemplo de Diagrama de Atividade – Processo de Emissão de Saldo.*

Como essa atividade foi parcialmente explicada nos exemplos anteriores, iniciaremos sua descrição pela ação seguinte ao nó de ação que recebe a senha da conta. Depois de a senha ter sido fornecida pelo cliente, esta deve ser validada, representada pelo nó de ação **Validar senha**, o que leva a um nó de decisão em que, se a senha for inválida, deve-se encerrar o fluxo da atividade, caso contrário, passa-se para a ação que consulta o saldo da conta. Quando essa ação é concluída, passa-se à ação que apresenta o

saldo na interface e encerra-se a atividade.

Como já foi dito, esse é um exemplo simples de diagrama de atividade. A seguir, demonstraremos outros componentes desse diagrama e apresentaremos exemplos mais complexos.

## 10.8 Nó de Bifurcação/União

Um nó de bifurcação/união é um nó de controle que pode tanto dividir um fluxo em dois ou mais fluxos concorrentes, quanto é chamado de nó de bifurcação, como mesclar dois ou mais fluxos concorrentes em um único fluxo de controle, quando é chamado de nó de união. Esse nó é representado por uma barra que pode estar tanto na horizontal como na vertical. A figura 10.8 apresenta um exemplo de nó de bifurcação/união.

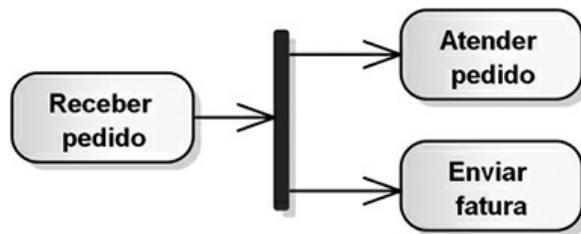


Figura 10.8 – Exemplo de Nó de Bifurcação/União.

Aqui, apresentamos um exemplo em que, a partir do nó de ação **Receber Pedido**, o fluxo de controle é dividido em dois fluxos paralelos, como demonstra o nó de bifurcação/união. A partir desse momento, os nós de ação **Atender pedido** e **Enviar fatura** serão executados paralelamente.

## 10.9 Final de Fluxo

Representa o encerramento de uma rotina representada pelo fluxo, mas não de toda a atividade. O símbolo de final de fluxo é representado por um círculo com um X, conforme demonstrado na figura 10.9, na qual assumimos uma situação em que muitos componentes podem ser construídos e instalados. Quando o último componente for construído, essa parte do fluxo estará concluída, como demonstra o símbolo de final de fluxo. No entanto, outras etapas deverão ser concluídas durante a atividade.

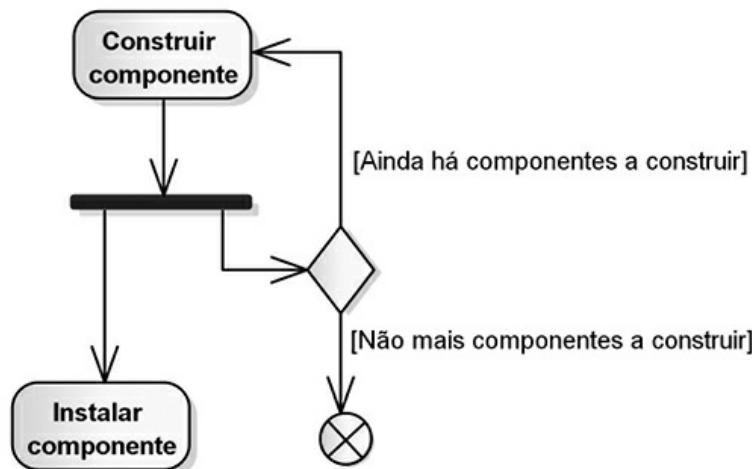


Figura 10.9 – Exemplo de Final de Fluxo.

## 10.10 Fluxo de Objetos

Um fluxo de objetos é um conector que pode ter objetos ou dados passando por ele. Representa o fluxo de valores (objetos ou dados) enviados a partir de um nó de objeto ou para um nó de objeto. O fluxo de objetos pode ser utilizado para modificar o estado de um objeto, definindo um valor para um de seus atributos ou mesmo instanciando ou destruindo o objeto. Um fluxo de objetos precisa conter ao menos um nó de objeto no início ou no fim de seu fluxo, podendo conter nós de objeto nas duas extremidades do fluxo. Na seção seguinte, apresentaremos alguns exemplos de fluxo de objetos.

## 10.11 Nó de Objeto

Um nó de objeto representa uma instância de uma classe, que pode estar disponível em um determinado ponto da atividade. Nós de objeto podem ser utilizados de diversas formas. Em sua forma mais tradicional, são representados como um retângulo, como demonstra a figura 10.10.

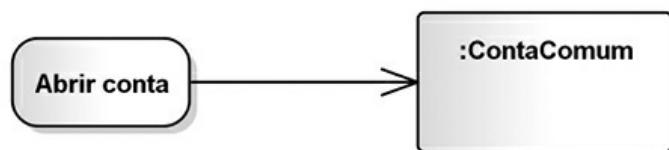


Figura 10.10 – Exemplo de Nó de Objeto e Fluxo de Objeto.

Neste exemplo, a ação de abrir conta gera um fluxo de objetos e cria um nó de objeto da classe **ContaComum**, representando a instanciação de um novo objeto dessa classe. Observe que a notação para nomear um nó de objeto é exatamente a mesma utilizada no diagrama de objetos, ou seja, primeiramente o nome do objeto (opcional, e neste exemplo não foi definido) seguido de dois-pontos (:) e o nome da classe. O leitor deve notar também que o fluxo de objetos é muito semelhante ao fluxo de controle. Na verdade, graficamente são idênticos. A diferença está nos tipos de valores transmitidos.

Um objeto pode apresentar multiplicidade, que, nesse caso, determina o número mínimo e o máximo de valores que o objeto aceita. Se existir um valor mínimo determinado, a ação só iniciará quando este for atingido. Um objeto pode apresentar também um “Upperbound” entre colchetes que determina o valor máximo de valores que o objeto pode conter. Em tempo de execução, quando esse valor é ultrapassado, o fluxo é interrompido.

## 10.12 Alfinetes (Pins)

Alfinetes são nós de objeto que representam uma entrada para uma ação ou uma saída de uma ação. Fornecem valores para as ações e recebem os valores resultantes delas. Os alfinetes podem ser representados também como pequenos retângulos e, em geral, são colocados ao lado das ações às quais estão ligados, conforme demonstra a figura 10.11.



Figura 10.11 – Exemplo de Alfinetes.

Nesse exemplo, após o atendimento de um pedido ter sido concluído, atualiza-se um objeto da classe **Pedido** para determinar que este foi concluído, passando-se em seguida para o nó de ação **Enviar pedido**. O nó de objeto **pedido** é uma informação de saída do nó de ação da esquerda e uma informação de entrada para o nó de ação da direita.

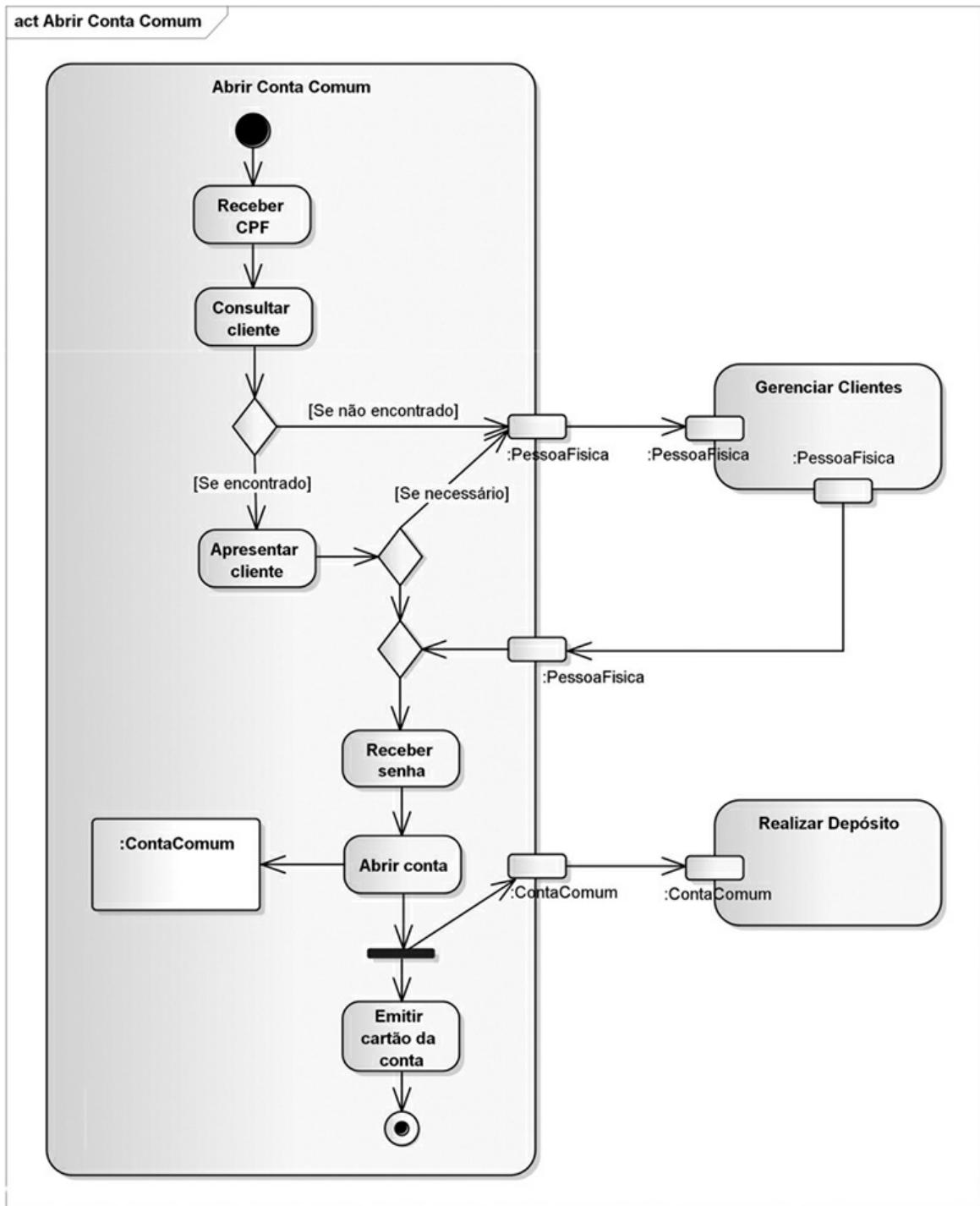
## 10.13 Nô de Parâmetro de Atividade

Um nó de parâmetro de atividade é um nó de objeto utilizado para

representar a entrada ou a saída de um fluxo de objetos em uma atividade. A figura 10.12 apresenta um exemplo de nó de parâmetro de atividade.

Esse exemplo se refere ao processo de abertura de uma conta comum, correspondente ao caso de uso **Abrir Conta Comum**, e contém alguns nós de parâmetro de atividade. Esse é um exemplo um pouco mais complexo que, além de conter o nó que desejamos exemplificar, contém diversos outros elementos do diagrama de atividade, como explicaremos a seguir.

O leitor notará que esse processo tem três atividades: a principal, que se refere ao processo de abertura de conta propriamente dito, detalha seus nós de ação, enquanto as atividades de **Gerenciar Clientes** e **Realizar Depósito** não o fazem, sendo apenas referenciadas. O leitor poderá observar também que a comunicação entre essas atividades é feita por meio de nós de parâmetro de atividade. A seguir, detalharemos esse diagrama, iniciando pela atividade de abertura de conta comum.



*Figura 10.12 – Exemplo de Parâmetro de Atividade – Processo de Abertura de Conta Comum.*

A atividade inicia-se com o processo de abertura de conta comum recebendo o CPF do cliente que deseja abrir uma conta. A ação seguinte consulta o CPF informado e passa-a a um nó de decisão, que determina

que o cliente deve ser registrado – se ele não for encontrado ou executar a ação que apresenta os dados do cliente, que passa em seguida a um novo nó de decisão, cuja função é estabelecer se é preciso atualizar os dados do cliente ou seguir para a ação que recebe a senha da nova conta a ser aberta.

Observe que nas duas situações em que se deve registrar ou alterar o cadastro do cliente a comunicação é feita por meio de nós de parâmetro de atividade, em que, primeiramente, é comunicada a informação do cliente para a atividade de **Gerenciar Clientes**, na qual este é cadastrado ou alterado. Em seguida, as informações do cliente são devolvidas atualizadas ao processo de abertura de conta por meio de outros nós de parâmetro atividade, mas que desempenham a mesma função e recebem o mesmo tipo de objeto. Note ainda que a comunicação com a atividade de **Gerenciar Clientes** é feita por meio de nós de parâmetro de atividade da classe **PessoaFisica**.

O leitor deve observar ainda que o fluxo dividido pelo segundo nó de decisão é unido por um nó de união, que apresenta o mesmo símbolo do de decisão. Utilizamos nós de parâmetro de atividade porque há suspensão temporária da atividade de abertura de conta para se executar outra atividade, e esta precisa de parâmetros iniciais para executar seu fluxo corretamente.

Depois de ter seu fluxo unido, a atividade de **abertura de conta comum** recebe a senha da nova conta a ser aberta e passa à sua abertura propriamente dita. Nessa ação, há um fluxo de objetos que representa a instanciação de um novo objeto da classe **ContaComum**. A seguir, a atividade passa a um nó de bifurcação que divide o fluxo em dois, em que o primeiro solicita a execução da atividade **Realizar Depósito** e o segundo emite o cartão da nova conta gerada, depois do que a atividade é encerrada. Observe que novamente usamos nós de parâmetro de atividade para enviar a conta gerada para a atividade de **Realizar Depósito**.

## 10.14 Nó de Buffer Central

Um nó de buffer central é um nó de objeto cuja função é gerenciar fluxos de múltiplas fontes e destinos. Age como um buffer de memória para múltiplos fluxos de entrada e fluxos de saída de outros nós de objeto. Não se conecta diretamente a ações. É representado como um nó de objeto com

o estereótipo <<centralBuffer>>.

## 10.15 Nó de Repositório de Dados (Data Store Node)

Esse nó é um tipo especial de nó de buffer central. Um nó de repositório de dados é um nó de objeto representado com o estereótipo de <<datastore>>. Um nó de repositório de dados é usado para armazenar dados ou objetos permanentemente. Os dados que entram em um nó de repositório de dados são armazenados permanentemente, atualizando os dados que já existiam. Os dados que vêm de um nó de repositório de dados são uma cópia dos dados originais. A figura 10.13 apresenta um exemplo de nó de repositório de dados.

Aqui, pegamos um trecho do exemplo da figura 10.12 referente à abertura de uma conta comum e acrescentamos um nó de repositório de dados. Nesse exemplo, demonstramos que a ação **Abrir conta** envia um objeto por meio de um fluxo de objetos para um repositório que armazenará a nova conta criada de forma permanente. Observe que o nome do repositório é **Conta**, que corresponde à tabela de mesmo nome, mapeada a partir da classe **ContaComum** constante no diagrama de domínio do sistema bancário, modelado no capítulo 4 sobre o diagrama de classes. Observe ainda que existe um alfinete (pin) ligado à ação de abertura de conta. Esse alfinete representa o objeto da classe **ContaComum**, criado pela ação e transmitido para o nó de repositório de dados. Uma vez que é transmitido, é um alfinete de saída.

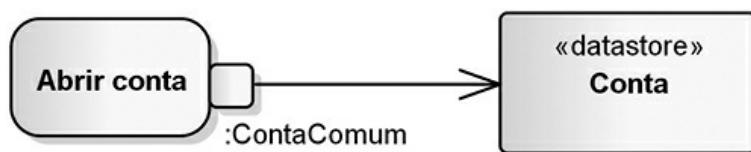
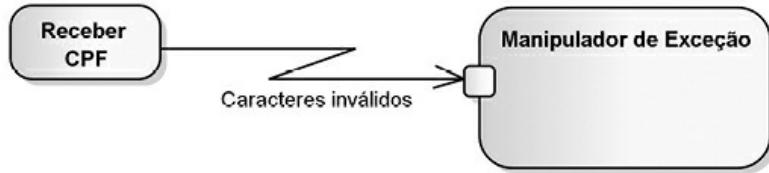


Figura 10.13 – Nô de Repositório de Dados.

## 10.16 Exceções

No diagrama de atividade, é possível detalhar a ocorrência de exceções, bastante comuns na maioria das linguagens de programação atuais. Para descrever a manipulação de uma exceção, o diagrama de atividade disponibiliza um fluxo, chamado fluxo de interrupção, representado por

uma seta em forma de raio que aponta para a rotina de tratamento da interrupção ou exceção. Na figura 10.14 é apresentado um exemplo de um tratamento de exceção.



*Figura 10.14 – Tratamento de Exceção.*

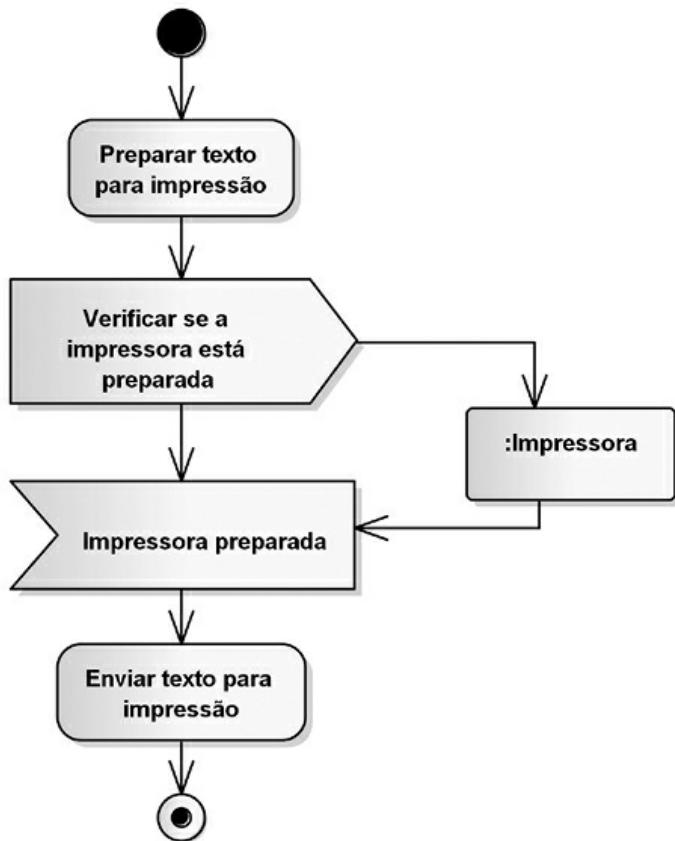
Nesse exemplo, a ação “Receber CPF” gera um fluxo de interrupção se forem informados caracteres inválidos (como letras ou outros símbolos que não números). Observe que a seta de exceção atinge um quadrado no Manipulador de Exceção (que é uma atividade – apenas referenciada aqui). Esse quadrado é também um nó de objeto – do tipo alfinete (pin), mais exatamente um alfinete de entrada – chamado nó de entrada de exceção.

## **10.17 Ação de Envio de Sinal (Ação de Objeto de Envio na versão 2.0)**

É uma ação que representa o envio de um sinal para um objeto ou ação. Uma ação de envio de sinal é representada por um retângulo com uma protuberância triangular no lado direito. Na seção 10.18, apresentaremos um exemplo de ação de envio de sinal.

## **10.18 Ação de Evento de Aceitação**

É uma ação que representa a espera de ocorrência de um evento de acordo com determinadas condições. Uma ação de evento de aceitação é representada por um retângulo com uma reentrância triangular no lado esquerdo. A figura 10.15 apresenta um exemplo de ações de envio de sinal e de evento de aceitação.

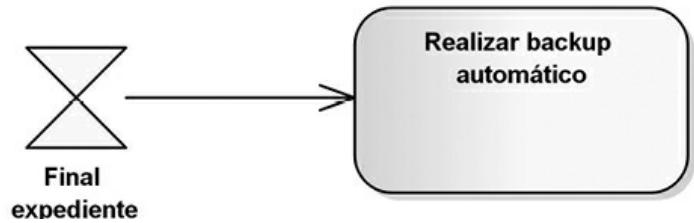


*Figura 10.15 – Ações de Envio de Sinal e de Evento de Aceitação.*

Nesse exemplo, utilizamos um fluxo de controle referente ao envio de um texto para impressão. Depois de preparar o texto, é enviado um sinal à impressora para verificar se esta está pronta para receber o material a ser impresso. Em seguida, a impressora envia um sinal em resposta, informando que pode imprimir o documento. Observe que a impressora é representada como um objeto e os sinais são enviados e recebidos por meio de um fluxo de objeto.

## 10.19 Ação de Evento de Tempo de Aceitação

É uma variação da ação de evento de aceitação que leva em consideração o tempo para que o evento possa ser disparado. Pode ser comparada com uma trigger. A figura 10.16 apresenta um exemplo de ação de evento de tempo de aceitação.

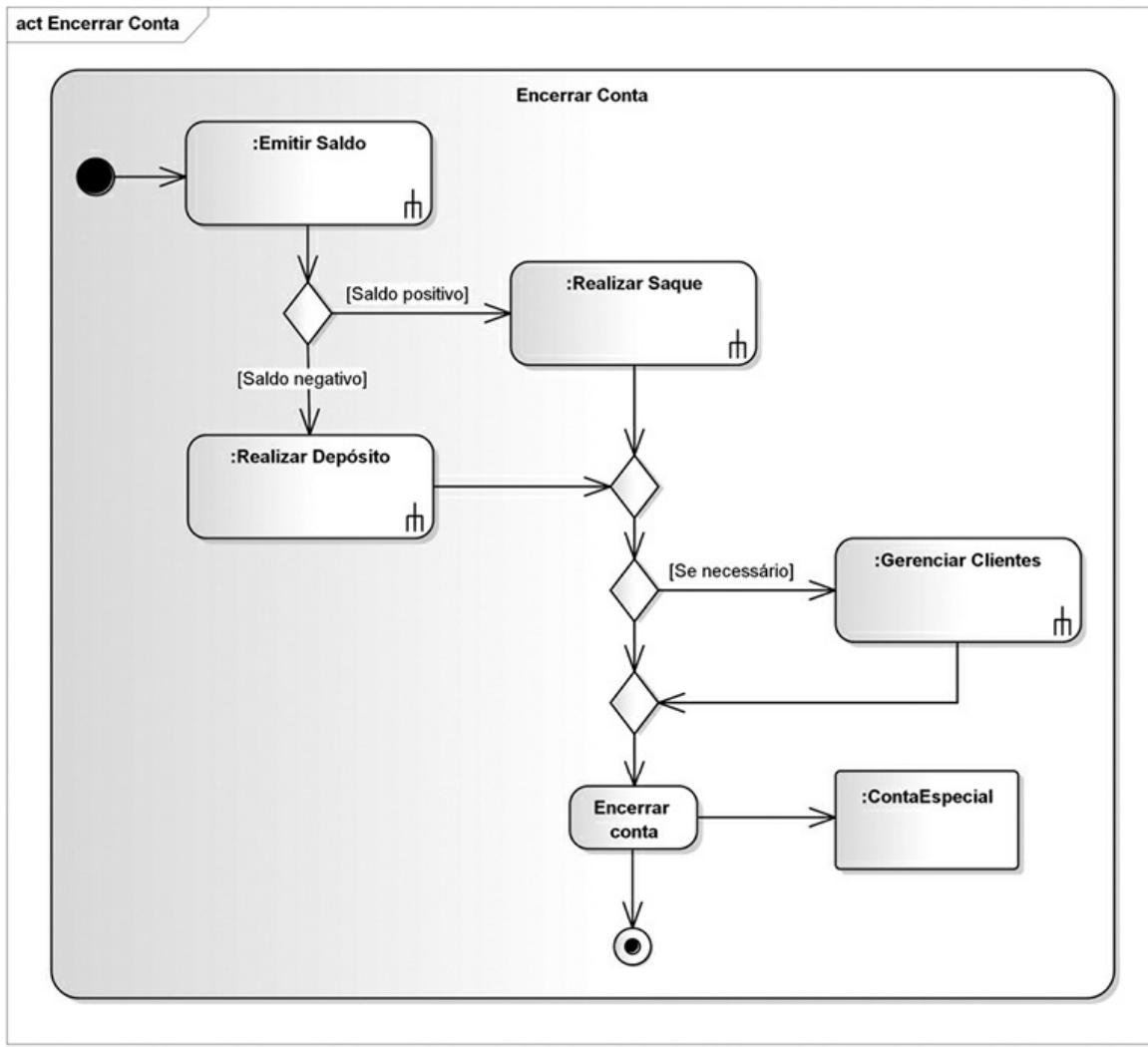


*Figura 10.16 – Ação de Evento de Tempo de Aceitação.*

Nesse exemplo, quando o horário de final de expediente for atingido, será disparada a atividade denominada **Realizar backup automático**.

## 10.20 Ação de Chamada de Comportamento

Este tipo de ação invoca a execução de um comportamento, sendo este, em geral, uma atividade. Esse tipo de ação apresenta um símbolo de ancinho apontando para baixo no canto inferior direito. A figura 10.17 apresenta um exemplo contendo várias ações desse tipo, enfocando o processo de encerramento de uma conta especial.



*Figura 10.17 – Exemplo de Ação de Chamada de Comportamento – Processo de Encerramento de Conta.*

O leitor perceberá que nesse processo utilizamos uma série de ações de chamada de comportamento que invocam atividades já modeladas previamente, por esse motivo não detalhamos suas ações internas, o que permite deixar o diagrama menor e mais fácil de compreender.

A primeira atividade a ser invocada pelo processo de encerramento de conta é a atividade de **Emitir Saldo**, uma vez que é necessário saber o saldo da conta antes de encerrá-la. Em seguida, é feito um teste: se o saldo da conta for positivo, será chamada a atividade **Realizar Saque**, caso contrário, será invocada a atividade **Realizar Depósito**. Observe que o fluxo dividido é unido por um nó de união.

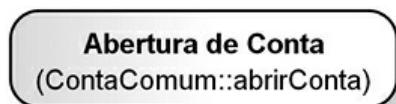
Em seguida, o fluxo cai novamente em um nó de decisão que verifica se

será necessário ou não efetuar manutenção no registro do cliente, se a conta a ser encerrada for a única por ele possuída. Em caso positivo, é chamada a atividade **Manter Cliente**. Os dois fluxos divididos são unidos por um novo nó de união. Finalmente, passa-se à ação de encerrar conta propriamente dita. Observe que há um fluxo de objetos entre essa ação e um objeto da classe **ContaEspecial**, que representa a alteração do atributo **situacao** desse objeto para inativo. Após a conclusão da ação, o processo é encerrado.

Observe que as ações de invocação de atividade não possuem um texto próprio, somente o nome da atividade que estão invocando. Poderíamos inserir um texto antes dos nomes das atividades, mas não consideramos isso necessário, por acreditarmos que os nomes das atividades são autoexplicativos.

## 10.21 Ação de Chamada de Operação

Enquanto a ação de chamada de comportamento é uma chamada direta a uma atividade completa, uma ação de chamada de operação é uma chamada indireta a um comportamento, na qual é invocada a execução de uma operação (método) em um objeto de uma classe. A figura 10.18 apresenta um exemplo desse tipo de ação.



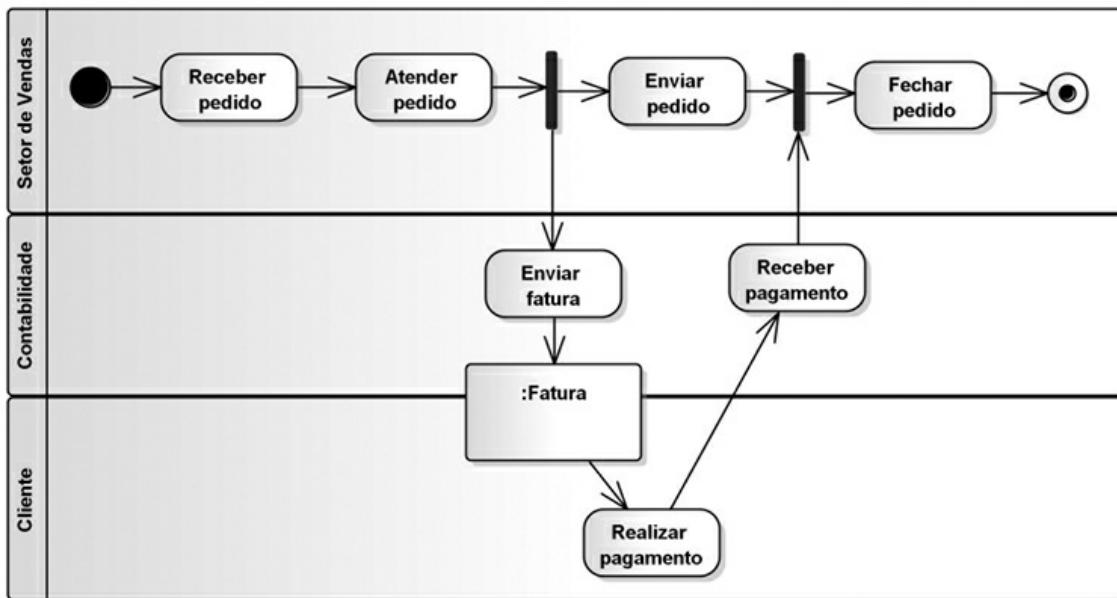
*Figura 10.18 – Exemplo de Ação de Chamada de Operação.*

Nesse exemplo, uma ação de chamada de operação nomeada **Abertura de Conta** solicita a execução do método **abrirConta**. Embaixo da solicitação, entre parênteses, é mostrada a classe que possui a operação em questão, seguida de dois símbolos de dois-pontos (:) e o nome da operação. O uso de uma ação de chamada de operação evita ter que descrever passo a passo todas as etapas da execução de um método em uma atividade.

## 10.22 Partição de Atividade

Partições de atividade são um tipo de Grupo de Atividade, ou seja, um

agrupamento de elementos de atividade como nós e fluxos. As partições de atividade permitem representar o fluxo de um processo que passa por diversos setores ou departamentos de uma empresa, ou mesmo um processo que é manipulado por diversos atores. As partições de atividade são formadas por retângulos, chamados raias de natação, representando divisões que identificam as zonas de influência de um determinado setor sobre parte de um determinado processo. As partições podem ser tanto horizontais como verticais. A figura 10.19 apresenta um exemplo de partições de atividade.



*Figura 10.19 – Partições de Atividade.*

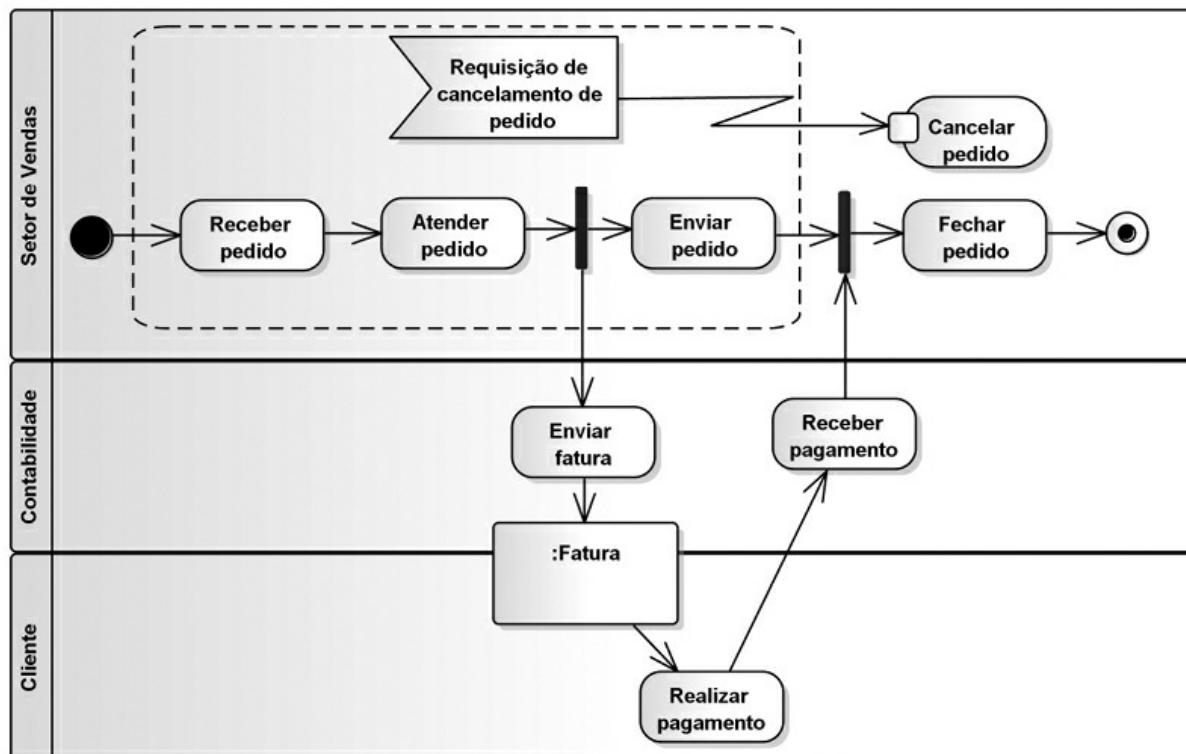
Nesse exemplo, existem três partições de atividade, representando o setor de vendas, o setor de contabilidade e o cliente, representando as pessoas que compram produtos da empresa. O início do processo é identificado pela ação que demonstra o recebimento de um pedido no setor de vendas. A ação seguinte representa o atendimento desse pedido e, quando esta for concluída, passa-se para um nó de bifurcação que divide o fluxo em dois, mantendo um fluxo no setor de vendas e passando o segundo fluxo para o setor de contabilidade. Assim, ao mesmo tempo que o pedido é enviado pelo setor de vendas, a fatura é enviada pelo setor de contabilidade.

O envio da fatura gera um objeto da classe **Fatura** que armazenará as informações da fatura gerada. O conteúdo desse objeto será enviado ao

cliente, que realizará a ação de **Realizar pagamento**. O recebimento do pagamento pelo setor de contabilidade reunirá novamente o fluxo dividido anteriormente no setor de vendas, como demonstra o nó de união, produzindo a ação **Fechar pedido** e concluindo o processo.

### 10.23 Região de Atividade Interrompível

Regiões de atividade interrompível são outro tipo de grupo de atividade. Uma região de atividade interrompível engloba certo número de elementos da atividade que podem sofrer uma interrupção. Assim, todo o processo envolvido pela região poderá ser interrompido, não importando em que elemento da atividade a interrupção ocorra. A região é delimitada por um retângulo tracejado com as bordas arredondadas. É necessário utilizar uma ação de envio de sinal para indicar a interrupção, além do uso de um fluxo de interrupção, conforme demonstra o exemplo da figura 10.20, onde a atividade representada na figura 10.19 foi melhorada para permitir a opção de cancelamento do pedido.



*Figura 10.20 – Exemplo de Região de Atividade Interrompível.*

Nesse exemplo, podemos perceber que a região envolvendo os nós de

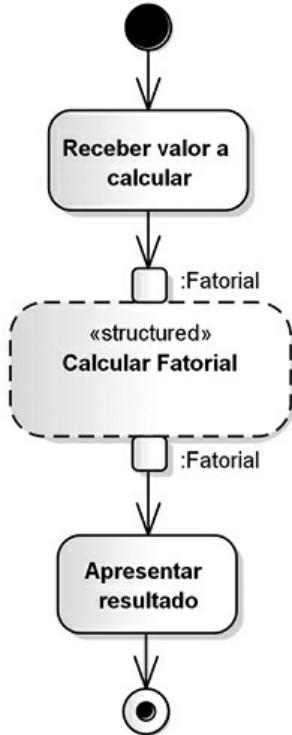
ação **Receber pedido**, **Atender pedido** e **Enviar pedido** é de atividade interrompível, ou seja, o pedido pode ser cancelado. Porém, a partir do momento que o pagamento for recebido, não será mais possível cancelar o pedido. Observe que a ação de envio de sinal gera um fluxo de exceção para o nó de ação **Cancelar pedido**, que encerra o processo, caso o cliente opte por essa alternativa.

## 10.24 Nó de Atividade Estruturada

Um nó de atividade estruturada é uma ação que também é um grupo de atividade e seu comportamento é especificado pelos nós e fluxos que ele contém. Porém, diferentemente dos outros grupos de atividade, os nós e fluxos desse tipo específico de grupo não podem ser compartilhados, pois pertencem exclusivamente ao grupo. Todavia, podem ser aninhados. Em razão da natureza concorrente da execução de ações dentro das atividades, pode ser difícil garantir o acesso e a modificação consistente da memória do objeto. Dessa forma, às vezes é necessário isolar os efeitos de um grupo de ações dos efeitos de ações fora do grupo.

Em sua forma mais simples, um nó de atividade estruturada é representado como um símbolo de atividade, mas contendo o estereótipo <<structured>> e possuindo as bordas tracejadas. A figura 10.21 apresenta um exemplo simples desse nó.

Nesse exemplo, a atividade estruturada contém os passos necessários para se calcular um número fatorial. O grupo de atividade que a atividade estruturada encapsula poderá ser executado recursivamente muitas vezes. O leitor perceberá que esse tipo de nó pode conter nós internos que não necessariamente são explicitamente declarados. Há outros tipos de nós de atividade estruturada mais especializados. Veremos alguns exemplos nas próximas subseções.



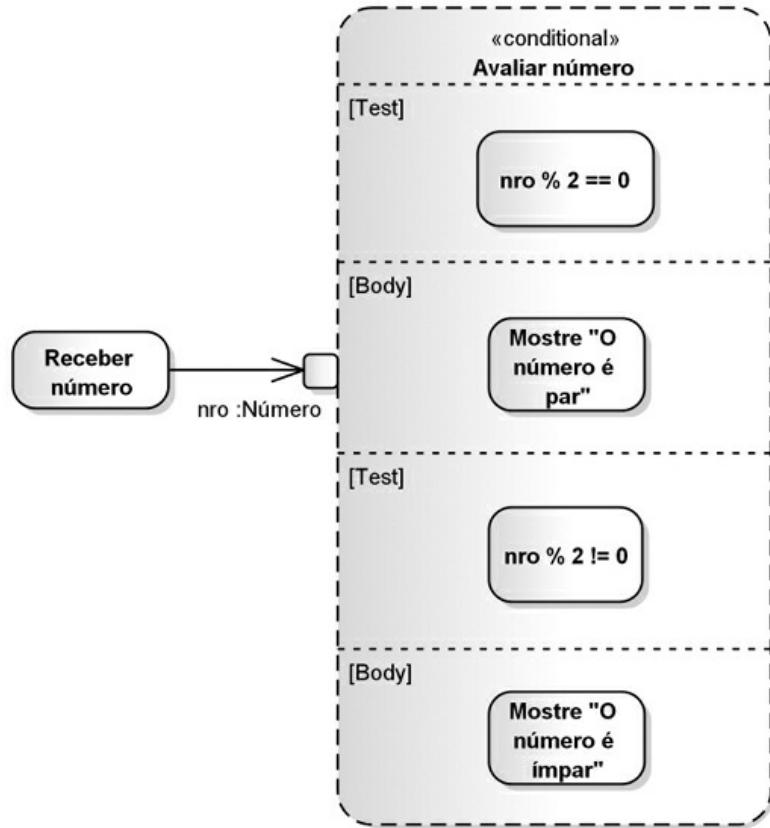
*Figura 10.21 – Exemplo de Nó de Atividade Estruturada.*

### 10.24.1 Nós Condicionais

São um tipo de nó de atividade estruturada que representa um conjunto de alternativas a serem escolhidas. Um nó condicional possui uma ou mais cláusulas e cada uma delas representa um dos possíveis caminhos que podem ser executados. Uma cláusula possui duas subseções: a primeira contém o teste que determina se a alternativa em questão deve ser executada e a segunda contém o corpo da alternativa, ou seja, um conjunto de nós e fluxos. A figura 10.22 apresenta um exemplo simples de nó condicional.

Como o leitor pode observar, um nó condicional possui o estereótipo <<conditional>>, para destacar que esse nó de atividade estruturada é do tipo nó condicional. Nesse exemplo, existem duas cláusulas, cada uma dividida nas subseções de teste e corpo. Essa atividade estruturada tenta determinar quando um número recebido como parâmetro é par ou ímpar. Assim, a condição da primeira alternativa é “nro % 2 == 0” para determinar se o número passado como parâmetro é par, enquanto a condição da segunda alternativa é “nro % 2 != 0” para determinar se o

número é ímpar. Se o teste da primeira alternativa for verdadeiro, serão executadas as ações associadas ao corpo da primeira alternativa, que se resumem a mostrar a mensagem de que o número é par. Caso contrário, serão executadas as ações contidas no segundo corpo, que basicamente mostram que o número é ímpar.



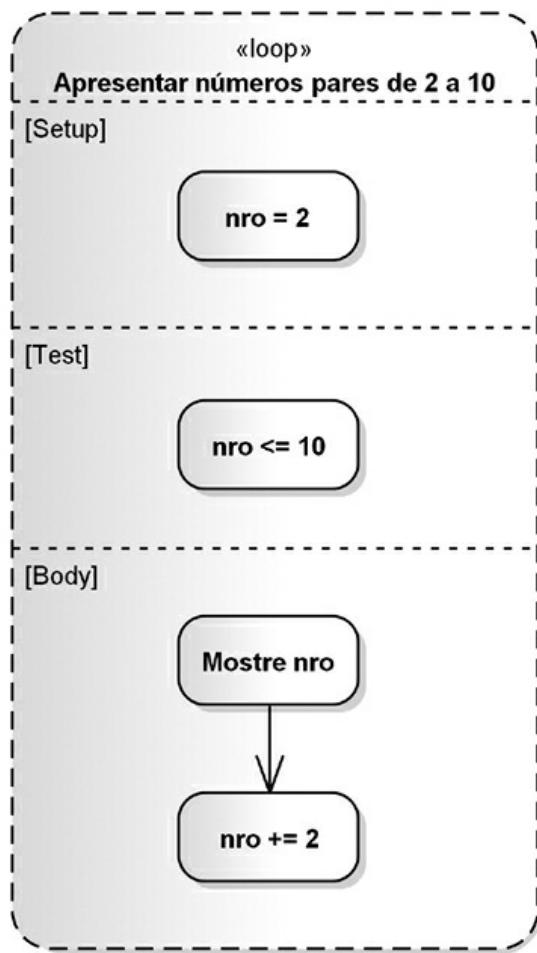
*Figura 10.22 – Exemplo de Nό Condisional.*

### 10.24.2 Nós de Laço

São um tipo de nó de atividade estruturada que representa um laço iterativo. Um nó de laço é dividido em três partes: iniciação (setup), teste e corpo. A primeira divisão inicia as possíveis variáveis necessárias ao laço. A segunda estabelece a condição para que o laço seja executado, enquanto a terceira contém as etapas que serão executadas se a condição for verdadeira. A figura 10.23 apresenta um exemplo desse tipo de nó de atividade estruturada.

Nesse exemplo, a atividade estruturada representa um laço que tem o

objetivo de mostrar os números pares de 2 a 10. Nessa atividade, na subseção de iniciação, a variável **nro** é iniciada com o valor 2. Na subseção teste é estabelecido que a condição para a execução do laço seja que o valor da variável **nro** armazene um valor inferior ou igual a 10. Finalmente, na subseção corpo, são representadas duas ações, que mostram o valor da variável **nro** e, em seguida, incrementam em 2 esse valor.



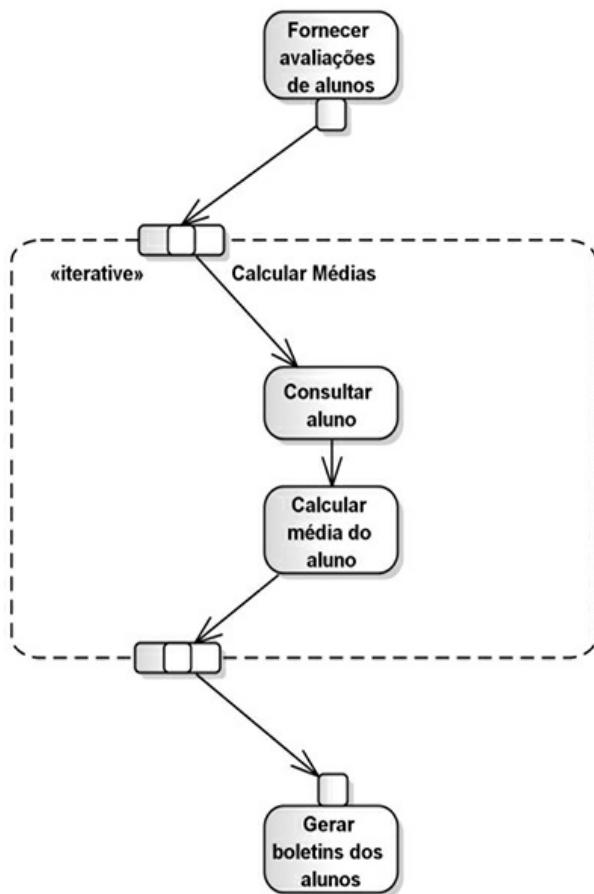
*Figura 10.23 – Exemplo de Nó de Laço.*

## 10.25 Região de Expansão

Uma região de expansão é um nó de atividade estruturada que toma como entrada uma ou mais coleções de valores e, para cada valor dessas coleções, executa os nós e fluxos contidos dentro da região. Isto resulta em coleções de saída (se algum resultado for efetivamente produzido), que não necessariamente são do mesmo número das coleções de entrada.

A cada execução da região, um valor de saída é inserido na coleção de saída, na mesma posição que os elementos de entrada. Os elementos de entrada e saída são nós de objeto, cada um representado por três pequenos quadrados juntos, colocados nas extremidades da região de expansão, aqui chamados de nós de expansão.

Uma região de expansão pode ser iterativa, onde as interações ocorrem na ordem dos elementos; paralela, onde todas as interações são independentes; ou de fluxo (stream), onde há uma única execução da região em que os valores na coleção de entrada são extraídos e colocados na execução da região de expansão como um fluxo. A figura 10.24 apresenta um exemplo simples de região de expansão.



*Figura 10.24 – Exemplo de Região de Expansão.*

Nesse exemplo, utilizamos uma região de expansão iterativa, conforme demonstra o estereótipo. Essa região de expansão recebe uma coleção de notas de diversos alunos. Assim, para cada elemento na coleção, é consultado o aluno a ele associado e calculada a sua média. A coleção

resultante é transmitida pelo nó de expansão de saída e utilizada para montar os boletins dos alunos.

## 10.26 Conectores

Conectores são basicamente atalhos para o fluxo, utilizados quando existe uma distância relativamente grande entre os nós que o fluxo precisa ligar. Um conector é representado por um círculo contendo uma letra, por exemplo. Deve haver sempre pares de conectores com a mesma nomenclatura, uma vez que um conector é um atalho. Assim, se houver um fluxo de entrada para o conector A, deve haver em outra parte do diagrama um fluxo de saída de um conector de mesmo nome. A figura 10.25 apresenta um exemplo simples de conectores.



Figura 10.25 – Exemplo de Conectores.

Outros exemplos de uso de conectores poderão ser examinados no capítulo 17.

## 10.27 Exemplo de Diagrama de Atividade – Emitir Extrato

Nesta e nas seções seguintes apresentaremos os diagramas de atividade referentes aos processos do sistema de controle bancário que estamos modelando ao longo deste livro, com exceção dos processos de emissão de saldo, abertura de conta e encerramento de conta que já foram apresentados. Aqui, apresentamos o diagrama de atividade para o processo de emissão de extrato, ilustrado pela figura 10.26.

A primeira ação desse processo é esperar que o cliente informe o número da conta da qual deseja emitir o extrato. Depois de receber o número da conta, passa-se à ação de consultá-la e, em seguida, a um nó de decisão, no qual é verificado se a conta é inválida – caso em que o processo é encerrado. Se a conta for válida, o processo fica aguardando que a senha da conta seja informada e, depois de esta ser fornecida, passa-se à ação de validação da senha, realizada por meio de um nó de decisão cuja função é determinar se o processo deve ser encerrado – caso a senha seja inválida – ou se é preciso passar à ação de solicitação dos períodos desejados para o

extrato.

Depois de receber os períodos, passa-se a um nó de decisão para determinar se estes são períodos corretos (o período inicial não pode ser maior que o final). Caso os períodos sejam inválidos, o processo deve ser encerrado, caso contrário passa-se ao nó de ação **Selecionar movimentos do período**. Observe que essa última ação tem um fluxo de objetos com um objeto da classe **Movimento**, significando que para selecionar os movimentos do período é necessário consultar objetos dessa classe. A rigor, deveríamos ter feito o mesmo quando consultamos a conta ou validamos a senha, mas achamos desnecessário esse nível de detalhe.

A partir dessa ação, passa-se a um novo nó de decisão que determina se o processo deve ser encerrado, caso não tenham sido encontrados movimentos no período, ou se é necessário passar à ação **Posicionar primeiro movimento**, em que o processo escolhe o primeiro movimento encontrado na seleção anterior. Depois, o movimento é apresentado na ação seguinte e passa-se a um novo nó de decisão, cuja função é determinar se ainda há movimentos, caso em que se passa à ação de posicionar no objeto **Movimento** seguinte e volta-se à ação de apresentar o movimento. Caso não haja mais movimentos, deve-se encerrar o processo.

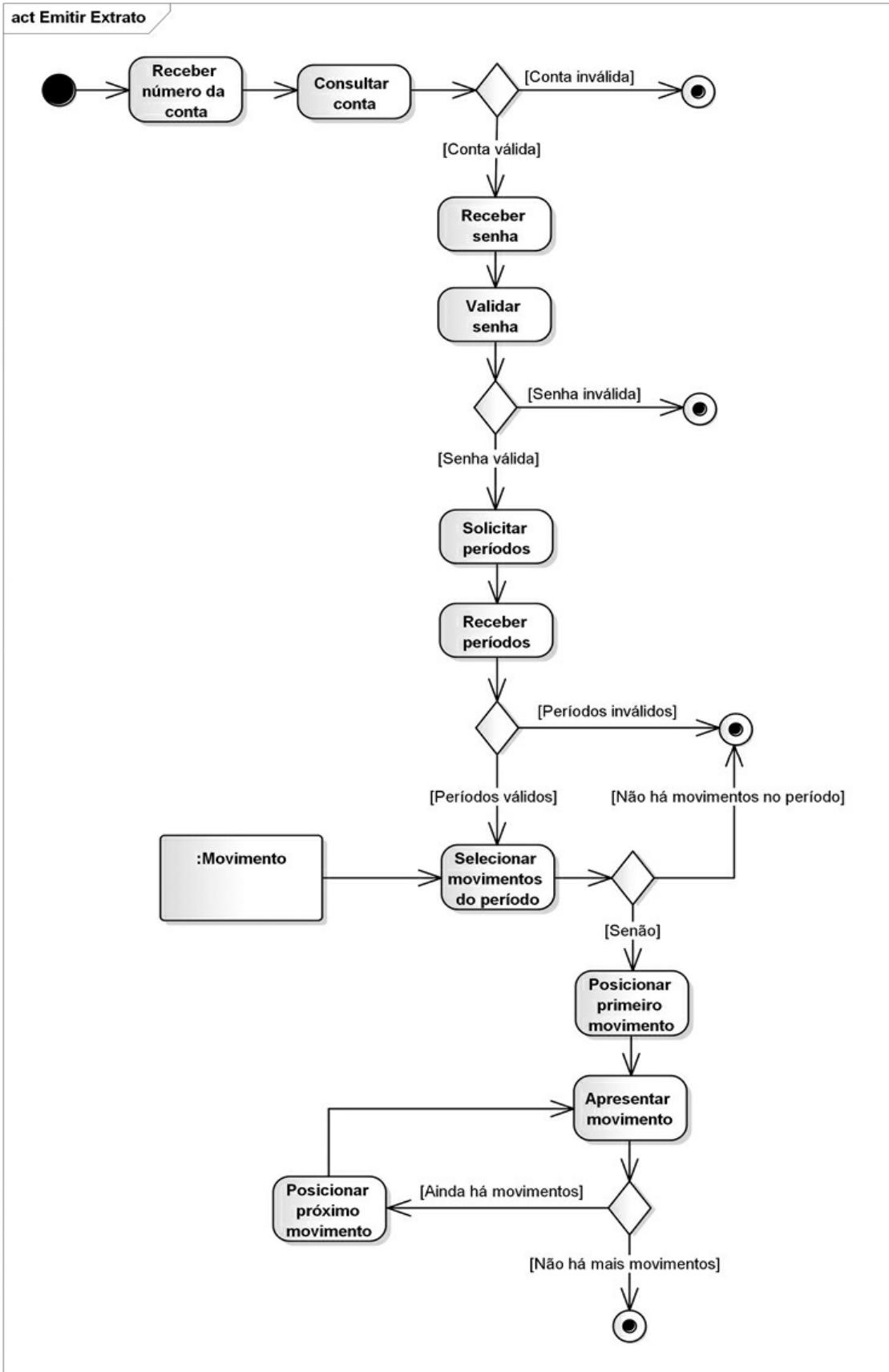


Figura 10.26 – Processo de Emissão de Extrato.

## 10.28 Exemplo de Diagrama de Atividade – Realizar Depósito

Nesta seção, enfocamos o processo de **Realizar Depósito** do sistema de controle bancário, conforme representado na figura 10.27.

As primeiras ações desse processo são idênticas à anterior, passando a ser diferentes a partir da ação que recebe o valor para o depósito. A ação seguinte soma o valor ao saldo da conta. Observe que há um fluxo de objetos para um objeto da classe **ContaComum** (herdado por objetos de suas subclasses) que recebe o valor transmitido por esse fluxo. Algo semelhante ocorre na ação seguinte, que registra o movimento e tem um fluxo de objetos para um objeto da classe **Movimento**, representando sua instanciação. Após essa ação, o processo é finalizado.

Poderíamos ter colocado a última ação em um diagrama separado, mas por esta ser apenas uma, preferimos inseri-la nesse diagrama.

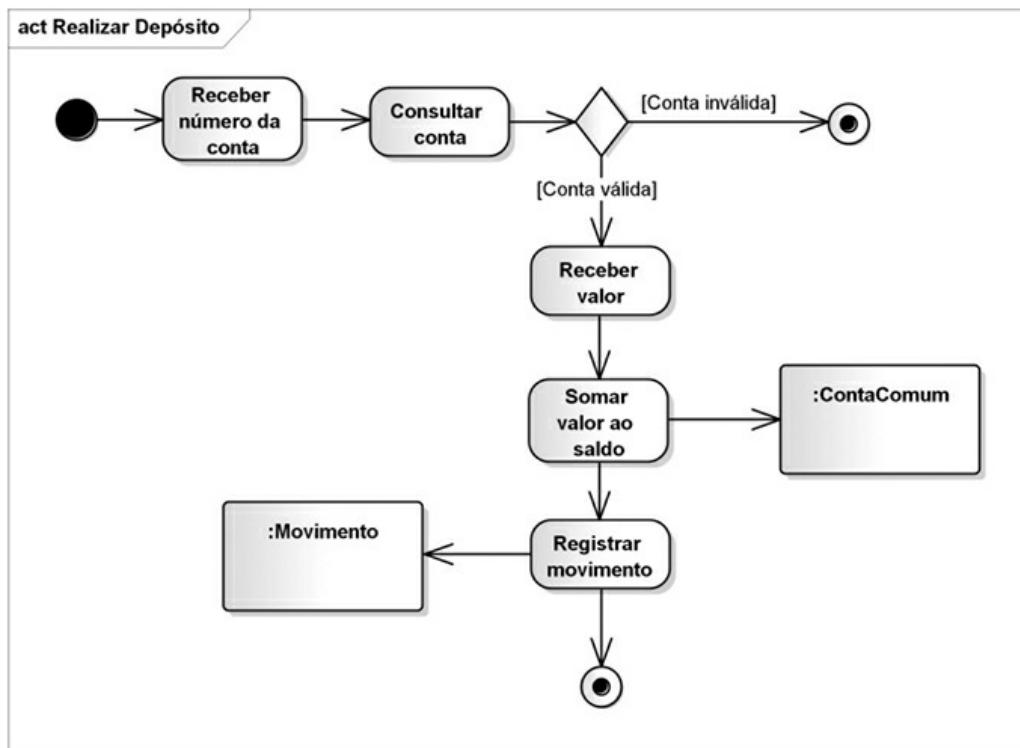


Figura 10.27 – Processo de Realizar Depósito.

## 10.29 Exemplo de Diagrama de Atividade – Realizar Saque

Passamos aqui ao processo de **Realizar Saque** do sistema de controle bancário, conforme pode ser visto na figura 10.28.

As primeiras ações desse processo são idênticas às do processo de **Emitir Extrato** até o momento em que a senha é validada. Depois disso, as ações são muito semelhantes ao processo de **Realizar Depósito**, exceto pela ação que diminui o valor do saldo da conta, a única diferente. O fluxo de objetos também representa ações semelhantes.

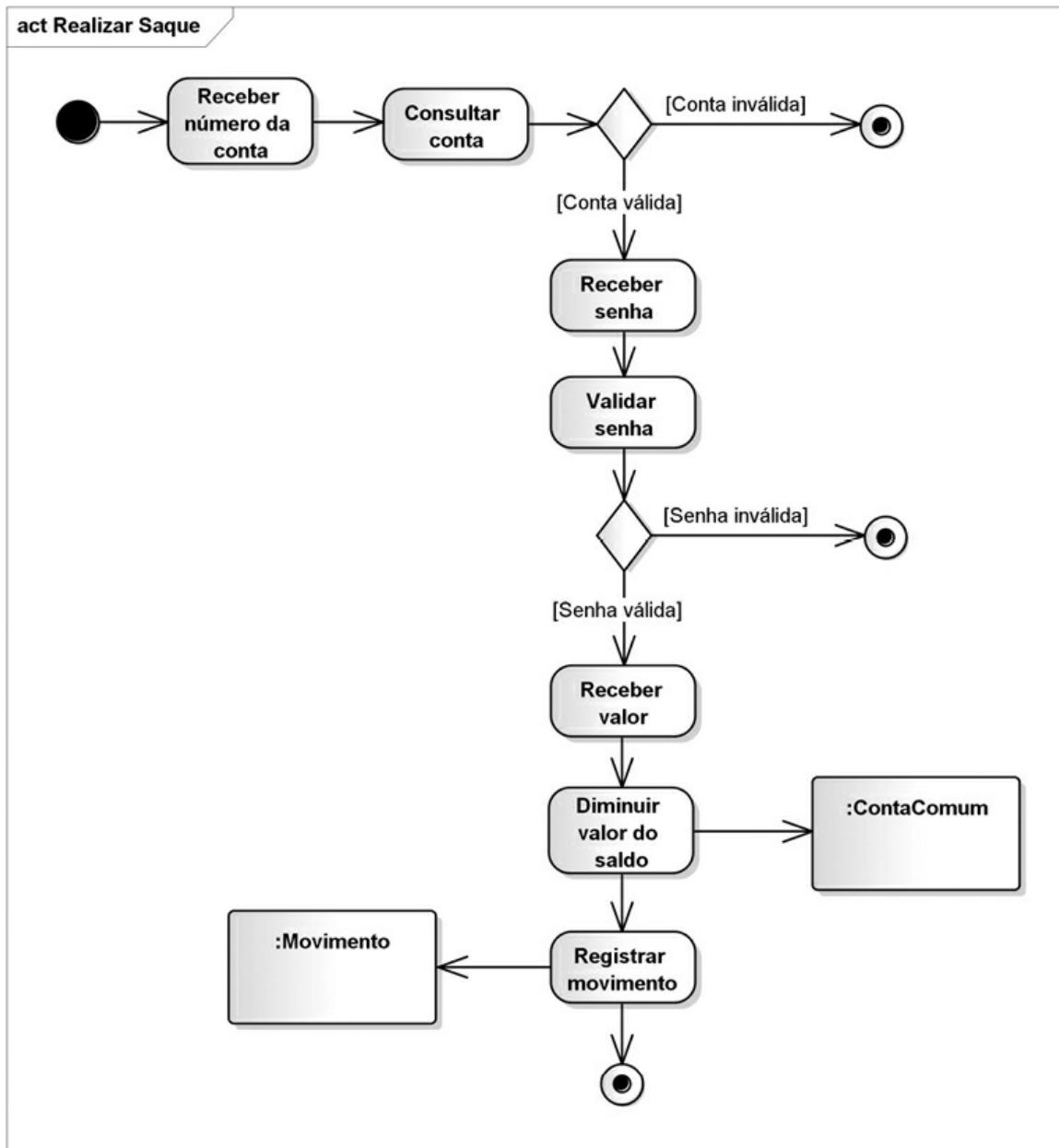


Figura 10.28 – Processo de Realizar Saque.

## **10.30 Exercícios Propostos**

Como tem ocorrido nos capítulos anteriores, continuaremos a modelagem dos sistemas já iniciados, enfocando, desta vez, o diagrama de atividade.

### **10.30.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos**

Desenvolva o diagrama de atividade referente ao processo de venda de ingressos para um sistema de controle de cinema, sabendo que:

- Ao selecionar a opção de venda de ingressos, o sistema deverá apresentar todas as sessões ainda não encerradas. Cada sessão deve informar o título do filme e a sala em que será apresentado.
- A partir da listagem apresentada, o funcionário deverá selecionar a sessão desejada pelo cliente.
- No momento em que a sessão for selecionada, o sistema deverá apresentar os assentos ainda disponíveis.
- Finalmente, o funcionário deverá selecionar os assentos desejados e os tipos de ingresso (se são ingressos inteiros ou meias-entradas) e confirmar a emissão dos ingressos referentes à sessão escolhida.

### **10.30.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade**

Desenvolva o diagrama de atividade referente ao processo de pagamento de mensalidade para um sistema de clube social, levando em consideração os seguintes fatos:

- Primeiramente, deve-se consultar o sócio que deseja pagar mensalidades.
- Após a consulta do sócio, deve-se consultar a(s) mensalidade(s) por ele devida(s).
- Se houver alguma mensalidade em atraso, deve-se calcular os juros referentes ao atraso do pagamento.
- Deve-se, então, apresentar a(s) mensalidade(s) devida(s) e aguardar que o sócio escolha quais deseja pagar.
- Finalmente, deve-se quitar a(s) mensalidade(s) escolhida(s).

### **10.30.3 Sistema de Locação de Veículos – Processo de Locação de Veículo**

Desenvolva o diagrama de atividade referente ao processo de locação de veículo para um sistema de aluguel de veículos, considerando que:

- Ao selecionar a opção de locação de veículos, o sistema deve carregar todos os clientes registrados.
- Em seguida, o sistema deve apresentar todos os veículos disponíveis. A listagem decorrente disso deve mostrar a descrição do automóvel, seu modelo e marca.
- A partir dessa listagem, o funcionário deve selecionar o cliente.
- Depois de o cliente ter sido selecionado, deve-se selecionar o automóvel.
- Depois de selecionar o veículo, o funcionário deve inserir os dados da locação, como o período de locação, a que se destina o veículo, para onde o cliente pretende ir etc.
- Finalmente, o funcionário poderá mandar gerar a locação do automóvel selecionado.

### **10.30.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão**

Desenvolva o diagrama de atividade referente ao processo de realizar leilão para um sistema de controle de leilão via internet, de acordo com os seguintes requisitos:

- Ao receber a solicitação do serviço de realizar leilões, o sistema deve carregar todos os leilões ainda não encerrados na interface.
- A partir dessa listagem, o leiloeiro deve selecionar qual leilão deseja iniciar.
- No momento em que um leilão é escolhido para ser iniciado, o sistema precisa carregar todos os itens a serem leiloados nele.
- A partir da listagem dos itens a serem leiloados, o leiloeiro deve escolher um item e anunciá-lo.
- Se houver algum lance para o item anunciado, o sistema deve anunciá-lo e, em seguida, registrá-lo.
- Existe um tempo máximo de espera para que haja lances. Enquanto esse

tempo não for atingido, o item permanece sendo anunciado. Sempre que houver um lance, esse cronômetro é reiniciado.

- Quando o tempo máximo de espera por um lance for atingido, o processo deve verificar se houve ofertas para o item, caso em que se deve anunciar o participante que ofereceu o maior lance como vencedor. Caso contrário, deve-se simplesmente encerrar o anúncio do item.
- Depois de ter sido encerrado o anúncio de um item, deve-se verificar se ainda há itens a anunciar, caso em que o processo passa a anunciar o novo item. Caso contrário, o leilão deve ser encerrado.

### **10.30.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias**

Desenvolva um diagrama de atividade referente ao processo de pagamento de diárias para um sistema de controle de hotelaria, de acordo com as seguintes definições:

- No momento em que o hóspede informa o número do quarto para quitar as diárias, o sistema deve consultar o hóspede e todas as diárias devidas pelo aluguel do quarto, apresentando-as ao funcionário.
- A partir dessa listagem, deve-se quitar as diárias apresentadas.
- Se tiver ocorrido a solicitação de quaisquer serviços no período em que o quarto estava ocupado, estes deverão ser quitados também;
- Isso também ocorre se houver quaisquer consumos de frigobar, sendo necessário também os quitar.

### **10.30.6 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel**

Desenvolva o diagrama de atividade referente ao processo de venda de imóvel para um sistema de controle de imobiliária, considerando as informações a saber:

- Ao ser escolhida a opção de venda de imóvel, o sistema deve receber o tipo de imóvel desejado e apresentar todos os imóveis do tipo selecionado. O sistema deve igualmente apresentar todos os clientes registrados.

- Caso o cliente em questão não esteja cadastrado, deve-se registrá-lo.
- Depois, o sistema deve receber os dados da compra, como o imóvel, o comprador, as taxas de transação e o valor pago.
- O sistema deve, então, calcular as comissões da imobiliária e do corretor, registrar a compra e todas as taxas pagas.

## 10.31 Solução dos Exercícios

### 10.31.1 Sistema de Controle de Cinema – Processo de Venda de Ingressos

A figura 10.29 ilustra a solução para esse exercício. A seguir, explicaremos cada um dos passos dessa atividade.

Essa atividade inicia-se com a seleção de todas as sessões de cinema ainda não encerradas. Observe que essa ação apresenta um fluxo de objetos com um objeto da classe **Sessao**, o que significa que a ação seleciona objetos dessa classe. Após a seleção, a atividade passa à ação que representa o posicionamento sobre o primeiro objeto **Sessao** selecionado. A seguir são consultados a sala da sessão e, em seguida, o filme apresentado nela. Essas ações poderiam apresentar fluxos de objetos para representar a busca por essas informações, mas preferimos identificar somente as operações mais importantes. Depois de concluídas as consultas, as informações da sessão são apresentadas.

Em seguida, a atividade passa a um nó de decisão, onde se deve determinar se ainda há sessões a apresentar, caso em que se passa à ação **Posicionar próxima sessão** e repetem-se as consultas de sala e filme, apresentando a nova sessão. Se não houver mais sessões, a atividade passa à ação em que se aguarda que o funcionário informe a sessão desejada pelo cliente. Após a escolha da sessão, a atividade deve selecionar os ingressos já vendidos para a sessão selecionada, por meio de uma consulta a objetos da classe **Ingresso**. Isso é necessário para determinar quais assentos ainda estão disponíveis.

A seguir, a atividade passa à ação em que os assentos desejados pelo cliente e os tipos de ingresso (inteiros ou meias-entradas) são selecionados. Depois, passa-se à ação **Gerar ingresso**, em que um novo objeto da classe **Ingresso** será instanciado, como demonstra o fluxo de objetos entre a

ação e o objeto. Caso o cliente tenha selecionado mais de um assento, essa ação se repetirá para cada assento escolhido por ele, o que é controlado pelo último nó de decisão desse diagrama. Quando o último ingresso for gerado, a atividade será encerrada.

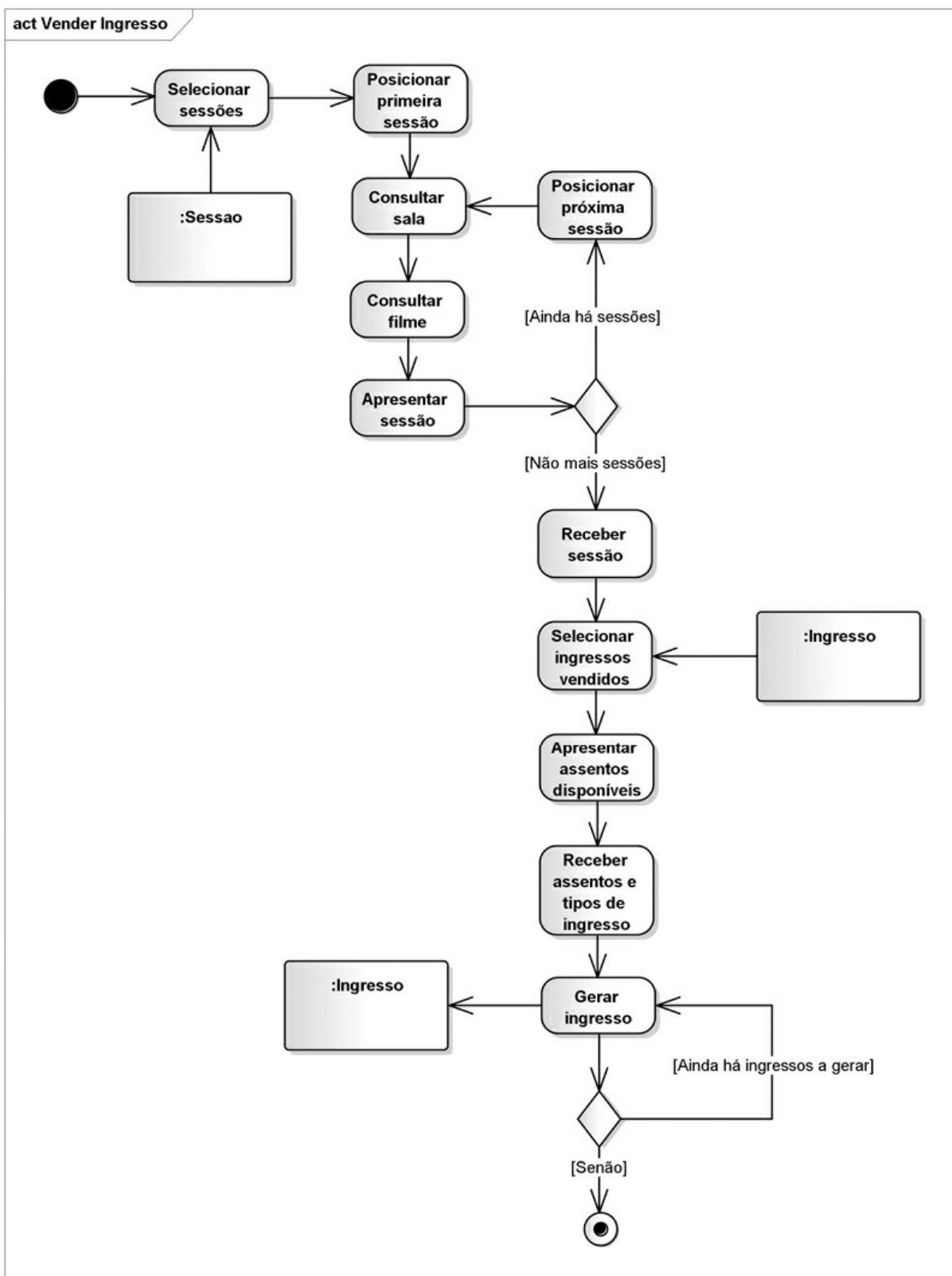


Figura 10.29 – Processo de Venda de Ingressos.

## **10.31.2 Sistema de Controle de Clube Social – Processo de Pagamento de Mensalidade**

A seguir, por meio da figura 10.30, apresentamos a solução para esse exercício.

A primeira ação dessa atividade é caracterizada pelo recebimento do número do cartão do sócio, seguida pela ação em que esse número é consultado. Depois, a atividade passa para um nó de decisão que testa se o número informado é válido e, caso não seja, passa a uma ação que informa que o sócio não foi encontrado e a atividade é encerrada.

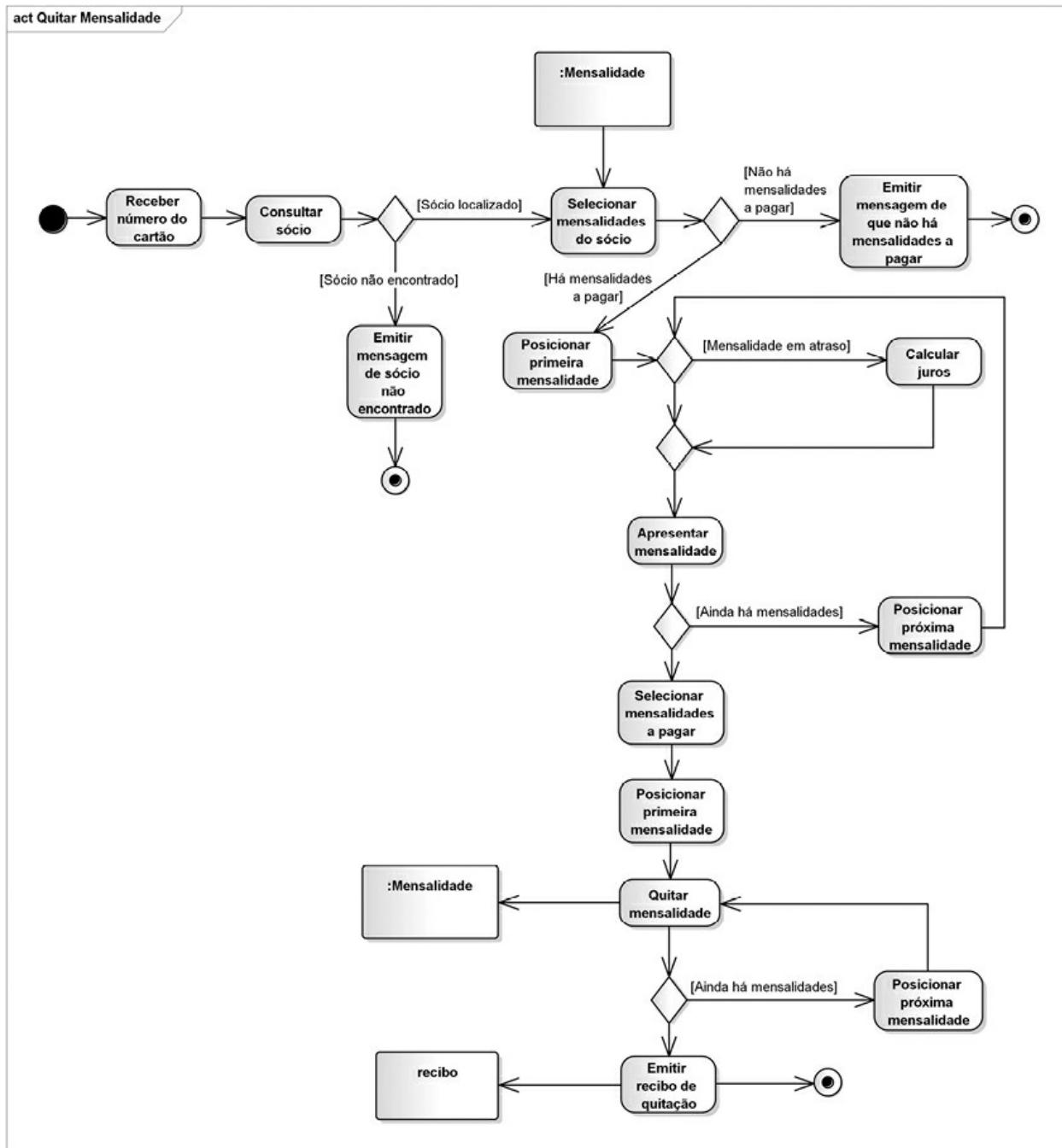


Figura 10.30 – Processo de Pagamento de Mensalidade.

Se o sócio for encontrado, passa-se à ação **Selecionar mensalidades do sócio**, em que as mensalidades ainda não pagas serão pesquisadas. Observe que existe um fluxo de objetos entre essa ação e um objeto da classe **Mensalidade**, que indica que a consulta será feita sobre objetos dessa classe. Como vimos procedendo anteriormente, só identificamos os fluxos de objeto mais importantes, para não deixar o diagrama poluído demais, já

que, a rigor, deveria haver um fluxo de objetos durante a ação de consulta de sócio também.

A partir dessa seleção, a atividade atinge um novo nó de decisão, que verifica se foi encontrada alguma mensalidade a pagar. Caso não tenha sido encontrada nenhuma mensalidade, deve-se informar isso ao sócio, por meio de um nó de ação, e encerrar a atividade.

Se forem encontradas mensalidades, então a atividade passará ao nó de ação **Posicionar primeira mensalidade** e, em seguida, para um novo nó de decisão, que tem a função de verificar se a mensalidade em questão está atrasada, caso em que deverá ser executada a ação **Calcular juros**.

O fluxo dividido pelo nó de decisão é reunido por um nó de união e passa-se à ação que apresenta a mensalidade consultada. Em seguida, é necessário tomar outra decisão, representada por um novo nó de decisão, que verifica se ainda existem mensalidades a apresentar, caso em que se executa a ação na qual a atividade se posiciona em um novo objeto da classe **Mensalidade** e volta ao nó de decisão que verifica se este está em atraso, repetindo o processo já descrito.

Depois de apresentar as mensalidades, a atividade entra em uma ação em que se aguarda que o sócio selecione as mensalidades que quer pagar. No momento em que o sócio selecioná-las, passa-se à ação **Posicionar primeira mensalidade** e, em seguida, à ação em que a referida mensalidade é quitada. Nessa ação existe um fluxo de objetos para um objeto da classe **Mensalidade**, já que sua situação será alterada, sendo preciso atualizá-lo.

Em seguida, a atividade encontra um novo nó de decisão, que verifica se existem mais mensalidades a quitar, caso em que é executada a ação **Posicionar próxima mensalidade** e volta-se à ação de quitar mensalidade. Caso contrário, a atividade executa a ação **Emitir recibo de quitação** e, após o término desta, encerra-se a atividade. Observe que há novamente um fluxo de objetos, atingindo um objeto chamado **recibo**. Esse objeto não existe no modelo de classes e representa apenas o recibo físico que o sistema deve emitir para o cliente.

### 10.31.3 Sistema de Locação de Veículos – Processo de Locação de Veículo

Apresentaremos aqui a atividade referente à solução desse exercício, ilustrado pela figura 10.31.

O primeiro passo dessa atividade é representado pela ação **Selecionar clientes**, que, como o leitor pode perceber, apresenta um fluxo de objetos com um objeto da classe **Cliente**, determinando que a busca por clientes será feita sobre objetos dessa classe. Em seguida, a atividade passa à ação **Posicionar primeiro cliente**, para depois o apresentar na interface.

A seguir, a atividade encontra um nó de decisão, o qual verifica se ainda há clientes a apresentar e, em caso positivo, executa a ação **Posicionar próximo cliente** e volta à ação que apresenta o cliente, repetindo esse laço enquanto houver clientes a apresentar.

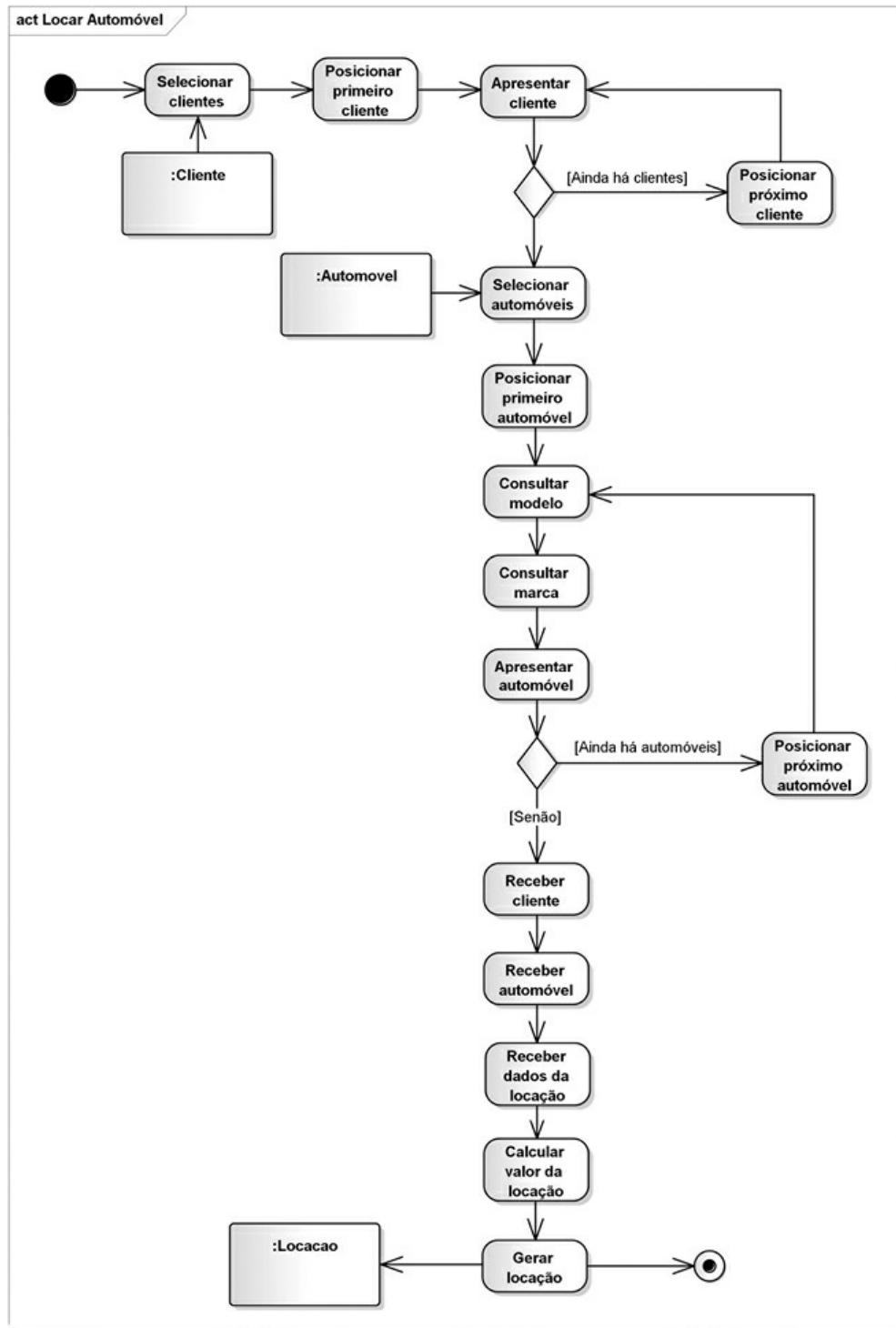


Figura 10.31 – Processo de Locação de Veículo.

Quando não houver mais clientes, a atividade seleciona todos os automóveis disponíveis. Essa consulta é feita em objetos da classe **Automovel**, como demonstra o fluxo de objetos. A atividade passa, então, para a ação **Posicionar primeiro automóvel**, ou seja, selecionará um dos

objetos da classe **Automóvel** pesquisados. Depois disso, a atividade executará as ações para consultar o modelo e a marca do automóvel em questão, o que poderia gerar fluxo de objetos, mas não achamos necessário detalhar tanto.

Em seguida, a atividade passa a um novo nó de decisão, que testa se ainda há automóveis e, caso positivo, posiciona sobre o próximo automóvel, repetindo a consulta de modelo e marca, apresentando-o. Esse laço será repetido enquanto houver automóveis.

A partir dessa listagem, o funcionário deve selecionar o cliente, o que é representado pela ação **Receber cliente**. Em seguida, deve-se selecionar o automóvel, o que está representado pela ação **Receber automóvel** e, depois, o funcionário deve inserir os dados da locação, como demonstra a ação **Receber dados da locação**.

Finalmente, a atividade passa à ação final, em que é gerada a locação do veículo selecionado. Essa ação causa um fluxo de objetos para um objeto da classe **Locacao**, representando sua instanciação. Após a conclusão dessa ação, a atividade é encerrada.

#### **10.31.4 Sistema para Controle de Leilão Via Internet – Processo de Realizar Leilão**

A atividade que soluciona esse exercício está contida na figura 10.32. Essa atividade inicia-se com a ação **Selecionar leilões**, que, como demonstra o fluxo de objetos, seleciona, a partir de objetos da classe **Leilao**, todos os leilões ainda não encerrados. Em seguida, passa-se à atividade **Posicionar primeiro leilão**, em que é selecionado o primeiro objeto **Leilao** pesquisado. O leilão em questão é apresentado na ação seguinte e depois há um teste, representado por um nó de decisão, que verifica se ainda há leilões. Em caso positivo, a atividade posiciona-se sobre o próximo leilão, apresenta-o e testa novamente se ainda há leilões, ficando nesse laço enquanto a condição for verdadeira.

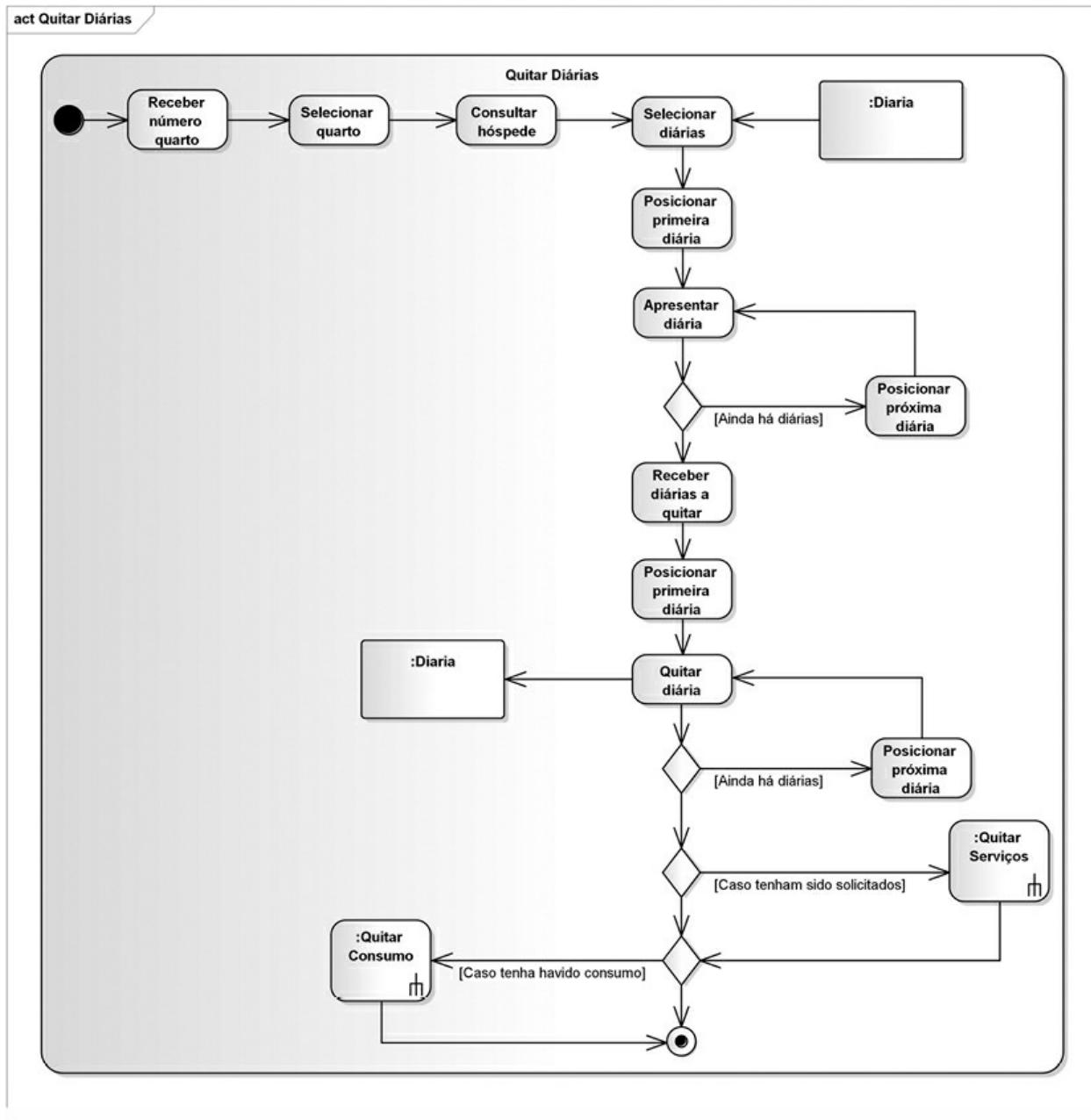
Quando não houver mais leilões, a atividade aguarda que o leiloeiro escolha um dos leilões apresentados. No momento em que um leilão é escolhido, é executada a ação **Iniciar leilão**, que, como o nome indica, inicia um novo leilão. Essa ação leva à ação seguinte, em que são selecionados os itens a serem leiloados no leilão escolhido. O fluxo de

objetos associados a essa última ação demonstra que é feita uma pesquisa nos objetos da classe `ItemLeilão` associados ao leilão selecionado.

Em seguida, a atividade envolve-se em um laço, em que serão anunciados os itens referentes ao leilão. Primeiramente é executada a ação **Posicionar primeiro item**, em que o primeiro dos itens pesquisados é selecionado. Em seguida, a atividade anuncia o item em questão e, depois, inicia um cronômetro que servirá para determinar o tempo máximo em que o item ficará sob anúncio.

A seguir, a atividade verifica se um lance foi recebido e, caso isso seja verdadeiro, o lance é registrado e um novo objeto da classe Lance é gerado. Depois disso, o lance é anunciado, o cronômetro, reiniciado e volta-se a testar se algum lance foi feito.

Caso um lance não tenha sido recebido, então a atividade verifica se o tempo máximo para anúncio do item em questão se esgotou. Se isto for falso, então o cronômetro será incrementado e volta-se a testar se algum lance foi recebido. Todavia, se o tempo de anúncio se esgotou, a atividade passa a verificar se o item anunciado recebeu alguma oferta.



*Figura 10.32 – Processo de Realizar Leilão.*

Se algum lance foi registrado, a atividade passa à ação em que é declarado o vencedor e, em seguida, à ação que define o item como arrematado. Essa ação altera a situação do objeto **Itemleilao** selecionado, por esse motivo há um fluxo de objetos entre a ação e esse objeto. A seguir, a atividade passa a um novo nó de decisão, que verifica se ainda há itens a leiloar. Esse nó de decisão também é atingido caso não tenham ocorrido lances para o item sob anúncio.

Se ainda houver itens a leiloar, a atividade anunciará o item seguinte e repetirá todo o processo já explicado. Caso não haja mais itens a leiloar, a atividade executará a última ação, que é responsável por finalizar o leilão. Essa ação marca o leilão selecionado como encerrado, por esse motivo há um fluxo de objetos entre ela e um objeto da classe **Leilão**. Após a execução dessa ação, a atividade é concluída.

### **10.31.5 Sistema de Controle de Hotelaria – Processo de Pagamento de Diárias**

Aqui, apresentamos a solução para esse exercício por meio de um diagrama de atividade representado na figura 10.33.

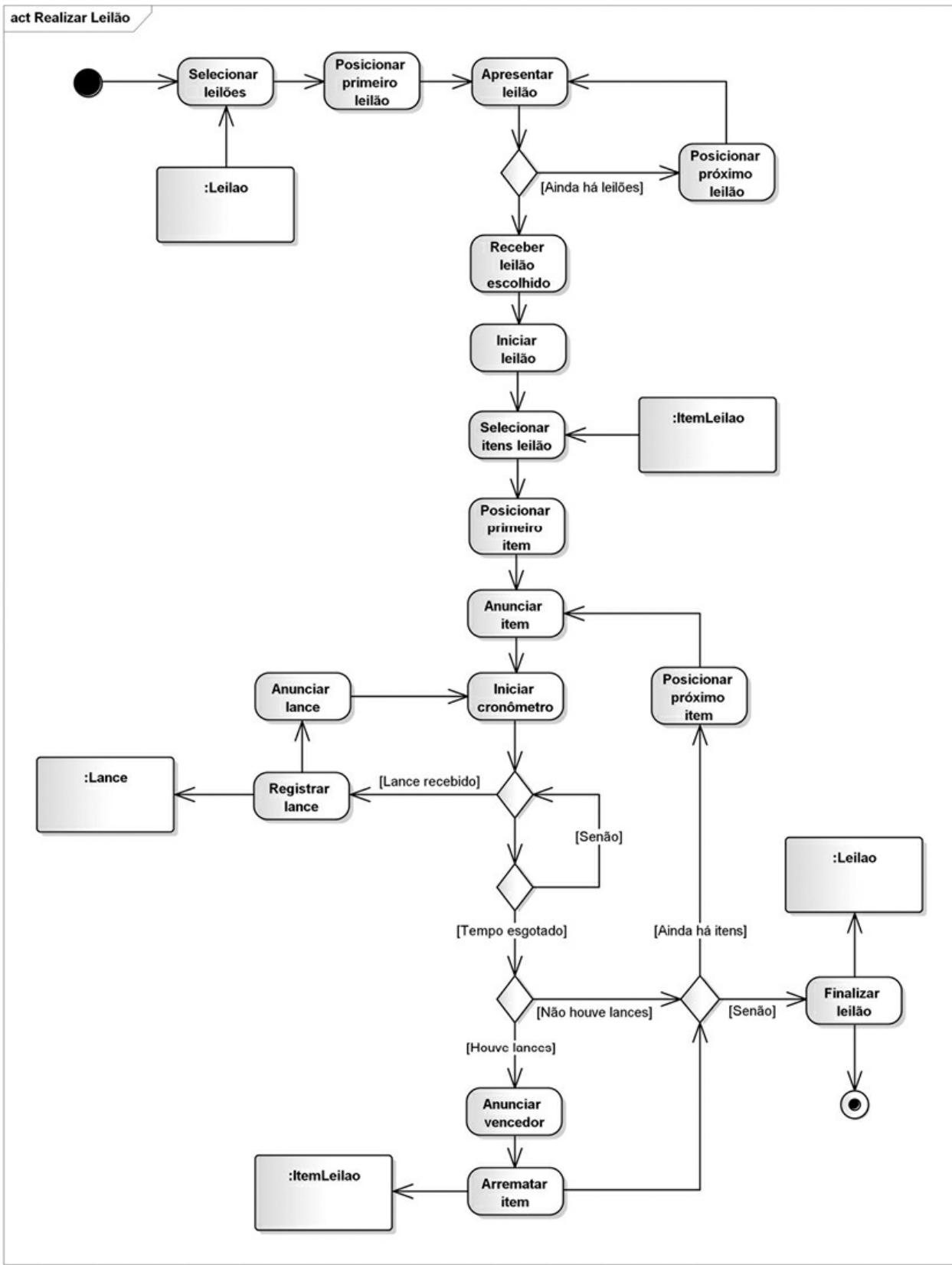


Figura 10.33 – Processo de Pagamento de Diárias.

Essa atividade inicia-se no momento em que o funcionário fornece o número do quarto a ser consultado. Depois dessa ação, a atividade passa à ação de consultar o quarto informado, passando depois à ação de consultar o hóspede que o aluga e, em seguida, à ação de selecionar as diárias devidas. Observe que essa consulta é realizada sobre objetos da classe **Díaria**, como demonstra o fluxo de objetos entre a ação e o objeto da classe.

Depois, a atividade executa a ação **Posicionar primeira diária**, em que é selecionada a primeira diária pesquisada. A ação seguinte apresenta a referida diária e, a seguir, passa-se a um nó de decisão que deve verificar se ainda há diárias a apresentar, caso em que a atividade executa a ação **Posicionar próxima diária** e volta à ação que apresenta a diária, ficando nesse laço enquanto houver diárias a apresentar.

Quando não houver mais diárias a apresentar, a atividade passa à ação de **Receber diárias a quitar**, onde o funcionário informa quais diárias deverão ser quitadas. Assim, a atividade executa a ação de **Posicionar primeira diária** e, em seguida, entra em um segundo laço no qual será executada a ação que quitará a diária selecionada, definindo esse objeto como quitado, como mostra o fluxo de objetos. A seguir, é feito um teste por meio de um nó de decisão que verifica se ainda há diárias a quitar. Em caso positivo, a atividade se posiciona na próxima diária e repete a operação de quitação; caso contrário, a atividade é encerrada.

Depois, é realizado um teste para determinar a necessidade de invocação da atividade **Quitar Serviços**, caso algum dos serviços oferecidos pelo hotel tenha sido solicitado. Após esse teste, é realizada outra verificação para determinar se o hóspede consumiu algo do frigobar, caso em que deverá ser invocada a atividade **Quitar Consumo**. Ao concluir-se esses últimos testes, o processo é finalizado.

## 10.31.6 Sistema de Controle de Imobiliária – Processo de Venda de Imóvel

Essa atividade, como ilustra a figura 10.34, inicia-se com o recebimento do tipo de imóvel que será vendido. A ação seguinte seleciona todos os imóveis do tipo escolhido, a partir de um fluxo de objetos da classe **Imóvel**, e passa a um laço onde todos os imóveis selecionados serão apresentados.

Após o término desse laço, a ação seguinte seleciona todas as pessoas registradas no sistema, também por meio de um fluxo de objetos vindo da classe **Pessoa**, e envolve-se em um laço para apresentá-las. Ao fim desse laço é verificado se o cliente encontra-se na listagem apresentada e, caso isso não seja verdadeiro, faz-se uma invocação à atividade de **Gerenciar Clientes**, por meio da qual ele será cadastrado.

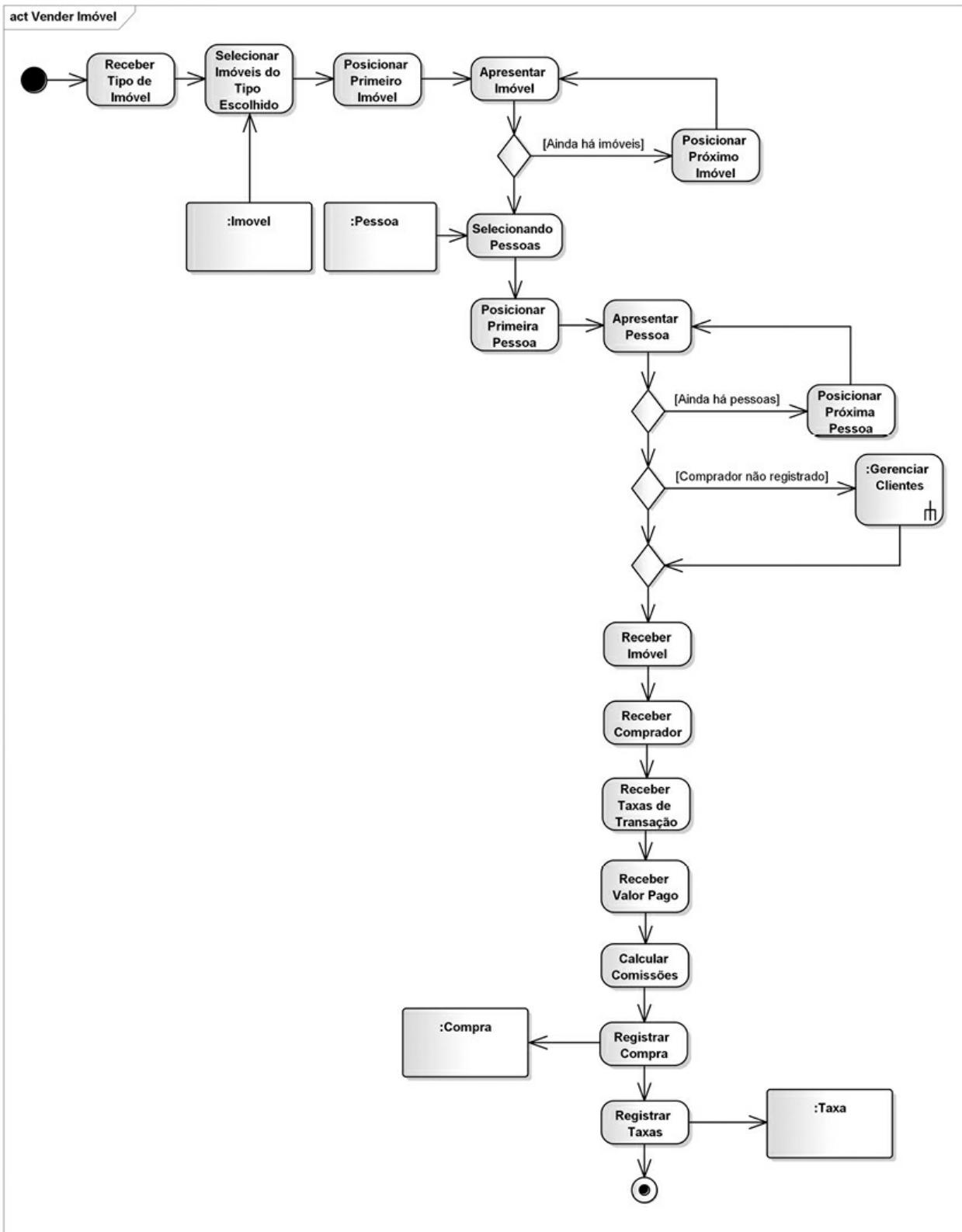


Figura 10.34 – Processo de Venda de Imóveis.

Depois, são executadas as ações em que o imóvel desejado e o comprador

são recebidos pelo sistema e passa-se para as ações em que as taxas da transação e o valor pago são informados. Nas ações seguintes são calculadas as comissões da venda em andamento e registradas a compra e as taxas pagas. Essas ações geram objetos das classes **Compra** e **Taxa**, conforme demonstra o fluxo de objetos disparados pelas duas últimas ações. Depois de as taxas terem sido registradas, a atividade é concluída.

## CAPÍTULO 11

# Diagrama de Visão Geral de Interação

Este diagrama é uma variação do diagrama de atividade. Seu objetivo é fornecer uma visão geral do controle de fluxo. O diagrama utiliza quadros no lugar dos nós de ação, embora símbolos como o nó de decisão e o nó inicial sejam também utilizados. Existem basicamente dois tipos de quadros: quadros de interação, que contêm qualquer tipo de diagrama de interação da UML, e quadros de ocorrência de interação, que normalmente fazem uma referência a um diagrama de interação, mas não apresentam seu detalhamento. A figura 11.1 apresenta um pequeno exemplo de diagrama de visão geral de interação representando o processo geral de encerramento de conta especial referente ao sistema de controle bancário que temos modelado neste livro.

O leitor notará que esse diagrama é muito semelhante ao diagrama de sequência referente ao mesmo processo, já explicado no capítulo 7, porém apresenta uma visão um pouco diferente. Isso ocorre porque o sistema de controle bancário não apresenta muitos processos interligados em um processo geral, sendo o processo de encerramento de conta o que mais se adapta aos objetivos desse diagrama.

Nesse exemplo estão representados três quadros de ocorrência de interação, referentes aos processos de **Emissão de Saldo**, **Realizar Saque** e **Realizar Depósito**, e um quadro de interação referente ao processo de **Encerrar Conta**, que contém um diagrama de sequência. Na verdade, os quadros de ocorrência de interação também se referem a diagramas de sequência. No entanto, optamos por apenas os referenciar para não tornar o diagrama grande demais, o que é muito comum nesse tipo de caso. Quando se inserem quadros de interação, também é recomendável procurar representar somente o fluxo mais geral ou importante, bem como os objetos mais essenciais, para não deixar o diagrama grande demais.

Nesse diagrama, o fluxo se inicia com a execução do processo de emissão

de saldo. Depois de este ter sido concluído, passa-se a um nó de decisão que verifica se o saldo da conta é positivo ou negativo. Caso seja positivo, é executado o processo de **Realizar Saque**, e se for negativo, o processo de **Realizar Depósito**. Um nó de união une os dois fluxos divididos anteriormente. Em seguida é necessário registrar o movimento, tenha sido referente a um saque ou a um depósito, conforme demonstra o quadro de ocorrência de interação **Registrar Movimento**.

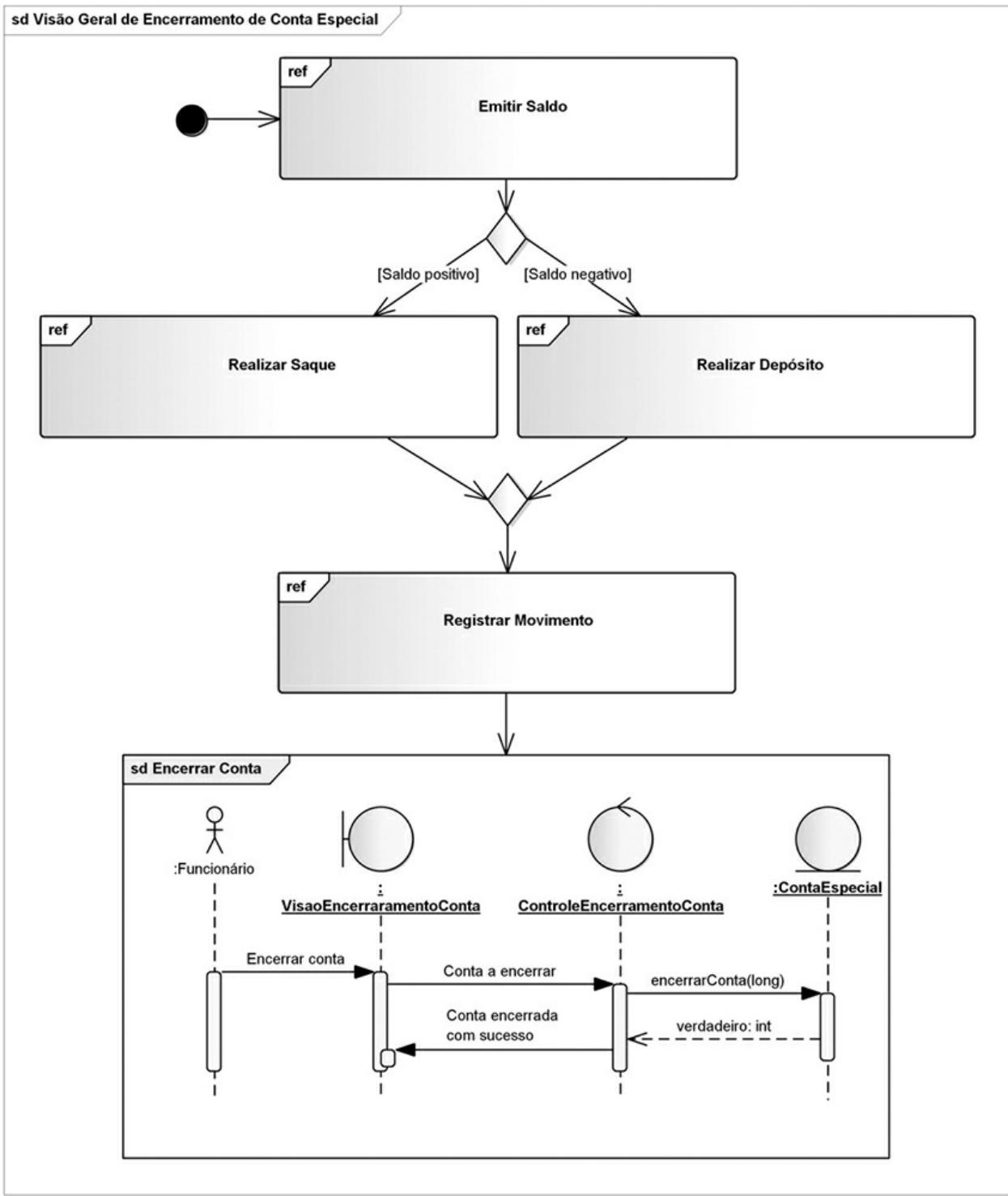


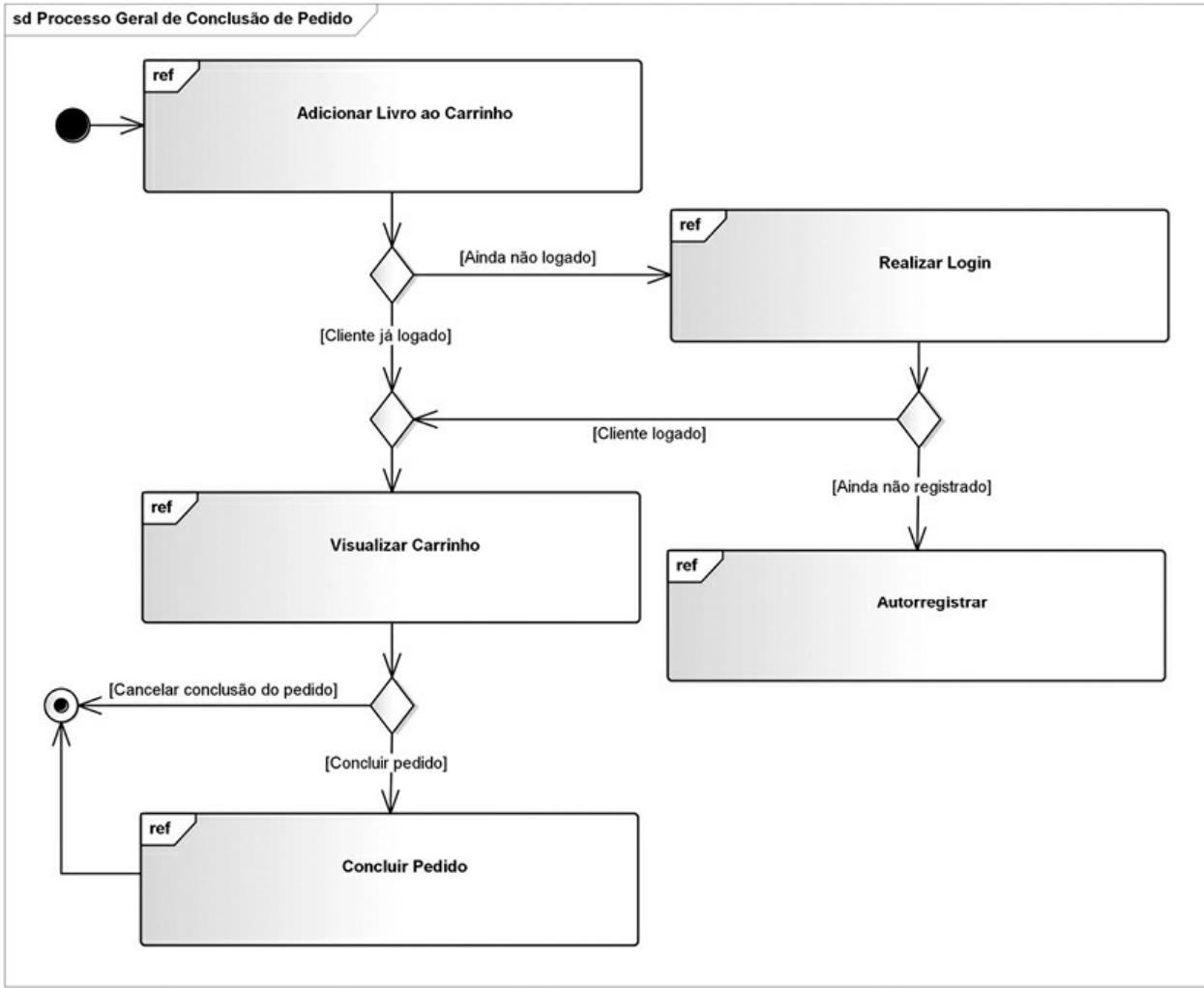
Figura 11.1 – Diagrama de Visão Geral de Interação – Processo Geral de Encerramento de Conta Especial.

O processo se encerra após a execução do quadro de interação **Encerrar Conta**, em que o funcionário solicita o encerramento da conta à interface, que repassa a solicitação à controladora e, por sua vez, dispara o método

**encerrarConta** em um objeto da classe **ContaEspecial**. A execução desse método retornará verdadeiro se o método for concluído com sucesso, o que fará a controladora mandar a interface exibir a mensagem de “Conta encerrada com sucesso”.

## **11.1 Exemplo de Diagrama de Visão Geral de Interação – Processo Geral de Conclusão de Pedido – Sistema de Livraria Digital**

Esse exemplo ilustrado na figura 11.2 baseia-se em um sistema de livraria virtual, cujo diagrama de casos de uso pode ser visto na figura 3.8. Nesse exemplo, um cliente pode adicionar livros, logar, visualizar o carrinho e concluir o pedido, sendo esse o processo final para concluir a compra. Todas essas funcionalidades são representadas aqui como quadros de ocorrência de interação. Aqui, demonstramos o processo geral que um cliente deve percorrer para comprar livros. Esse exemplo está um pouco simplificado, mas o objetivo desse diagrama é dar uma visão geral do processo.



*Figura 11.2 – Processo Geral de Conclusão de Pedido – Sistema de Livraria Digital.*

Nesse exemplo, o cliente inicia pelo processo de **Adicionar Livro ao Carrinho**, em que escolherá e adicionará quantos livros quiser. Quando o cliente desejar concluir o pedido, o sistema deverá antes realizar um teste, representado aqui por um nó de decisão, em que se deve verificar se o cliente já está logado.

Se o cliente estiver logado, passa-se ao processo de **Visualização do Carrinho**, porém, se não tiver se logado ainda, será necessário executar o processo de **Realizar Login**, em que o cliente se logará no sistema. Pode-se notar que há um segundo teste representado por um segundo nó de decisão, que verifica se o cliente está registrado no sistema. Se estiver registrado, passa-se ao processo de **Visualização de Carrinho** (observe que o fluxo dividido anteriormente é unido por um nó de união), mas se o

cliente não estiver registrado, deve-se executar o processo de **Autorricular**, em que o cliente poderá se cadastrar no sistema.

Durante o processo de **Visualização de Carrinho**, será apresentada ao cliente a lista de todos os livros que ele adicionou, bem como suas quantidades, e lhe será permitido aumentar ou diminuir a quantidade de qualquer livro, bem como excluir qualquer um deles.

Após a conclusão desse processo, deve-se realizar um último teste, uma vez que o sistema deve perguntar se o cliente quer realmente concluir o pedido. Nesse momento, o cliente pode cancelar a conclusão do pedido ou, se realmente desejar, executar o processo **Concluir Pedido**.

## 11.2 Exercícios Propostos

Aqui, daremos continuidade à modelagem dos sistemas que estamos projetando como exercício ao longo do livro, porém nem todos os sistemas farão parte desta seção, uma vez que nem todos se enquadram no escopo desse diagrama.

### 11.2.1 Sistema de Controle de Clube Social – Processo Geral de Associação

Desenvolva o diagrama de visão geral de interação referente ao processo geral de associação para um sistema de clube social, sabendo que:

- Primeiramente, o candidato a sócio deve percorrer os passos do processo para apresentar um pedido de aceitação.
- Caso o pedido seja recusado, o candidato não será aceito, porém, se for aprovado, então passará ao processo por meio do qual se associará ao clube.
- Se o sócio possuir dependentes, deverá passar pelo processo de associação de dependentes.

### 11.2.2 Sistema de Controle de Hotel – Processo Geral de Encerramento de Estada

Desenvolva o diagrama de visão geral de interação referente ao processo geral de encerramento de estada para um sistema de controle de hotel, levando em consideração os seguintes requisitos:

- O processo de encerramento de estada exige, antes de mais nada, que todas as diárias ainda devidas pelo hóspede sejam pagas.
- É necessário também verificar se houve solicitação de algum dos serviços oferecidos pelo hotel, que deverão ser pagos também.
- Igualmente, é preciso verificar se houve consumo de frigobar, que deverá ser quitado da mesma forma.
- Somente depois de essas etapas serem concluídas, será possível executar o processo de encerramento de conta propriamente dito.

## 11.3 Solução dos Exercícios

### 11.3.1 Sistema de Controle de Clube Social – Processo Geral de Associação

A solução para esse exercício é apresentada na figura 11.3. Nesse exercício são representados dois quadros de ocorrência de interação referentes aos processos de **Apresentar Pedido de Aceitação** e **Associar Dependentes** e um quadro de interação referente ao processo de **Associar Sócio**, que contém um diagrama de sequência.

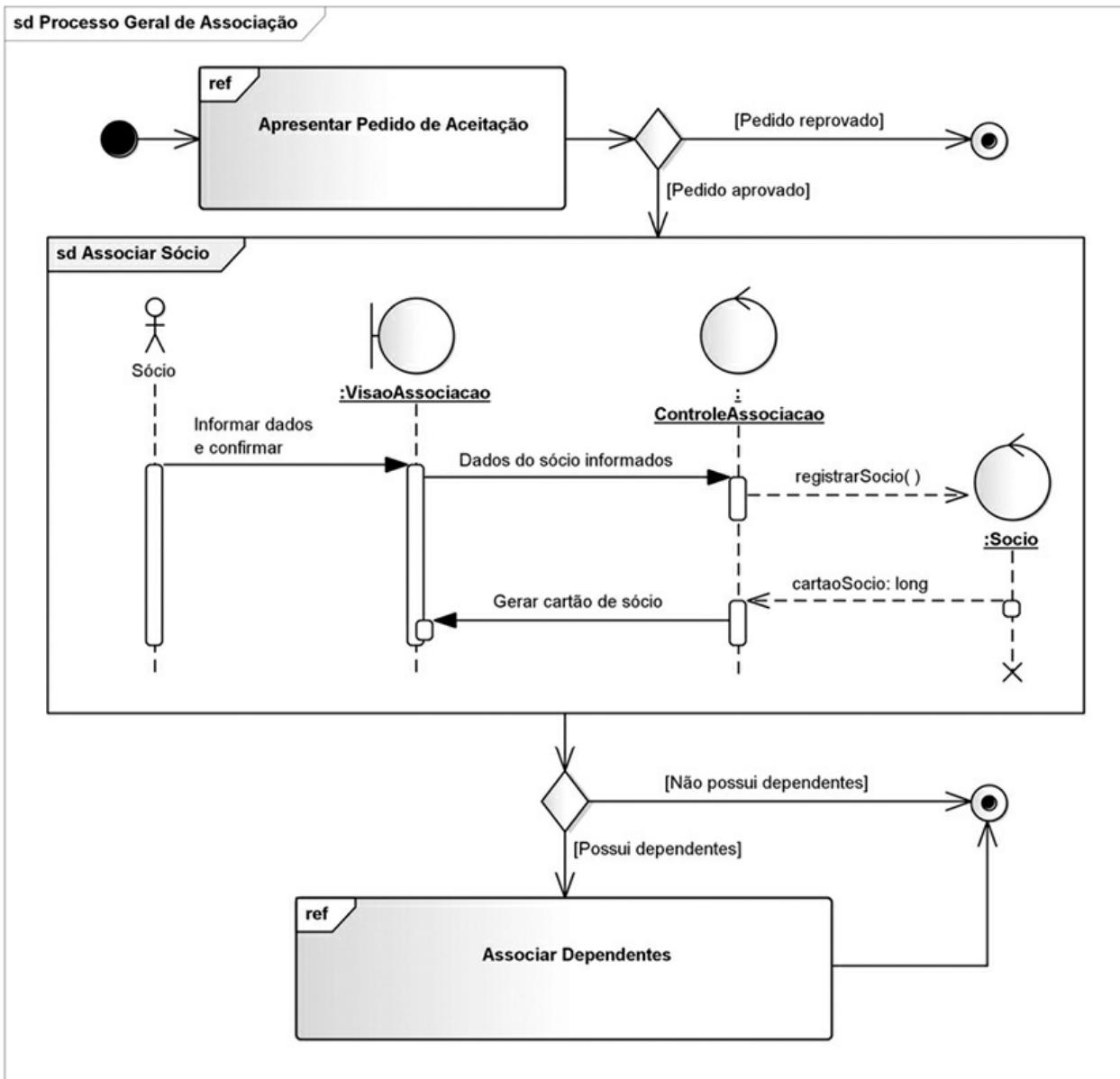


Figura 11.3 – Diagrama de Visão Geral de Interação – Processo Geral de Associação.

O diagrama se inicia com a execução do processo **Apresentar Pedido de Aceitação**, em que um candidato a sócio apresentará formalmente um pedido para ser aceito no clube, e este verificará se o candidato é adequado para ser sócio. A seguir, passa-se a um teste, representado por um nó de decisão, em que, caso o pedido tenha sido recusado, encerra-se o processo.

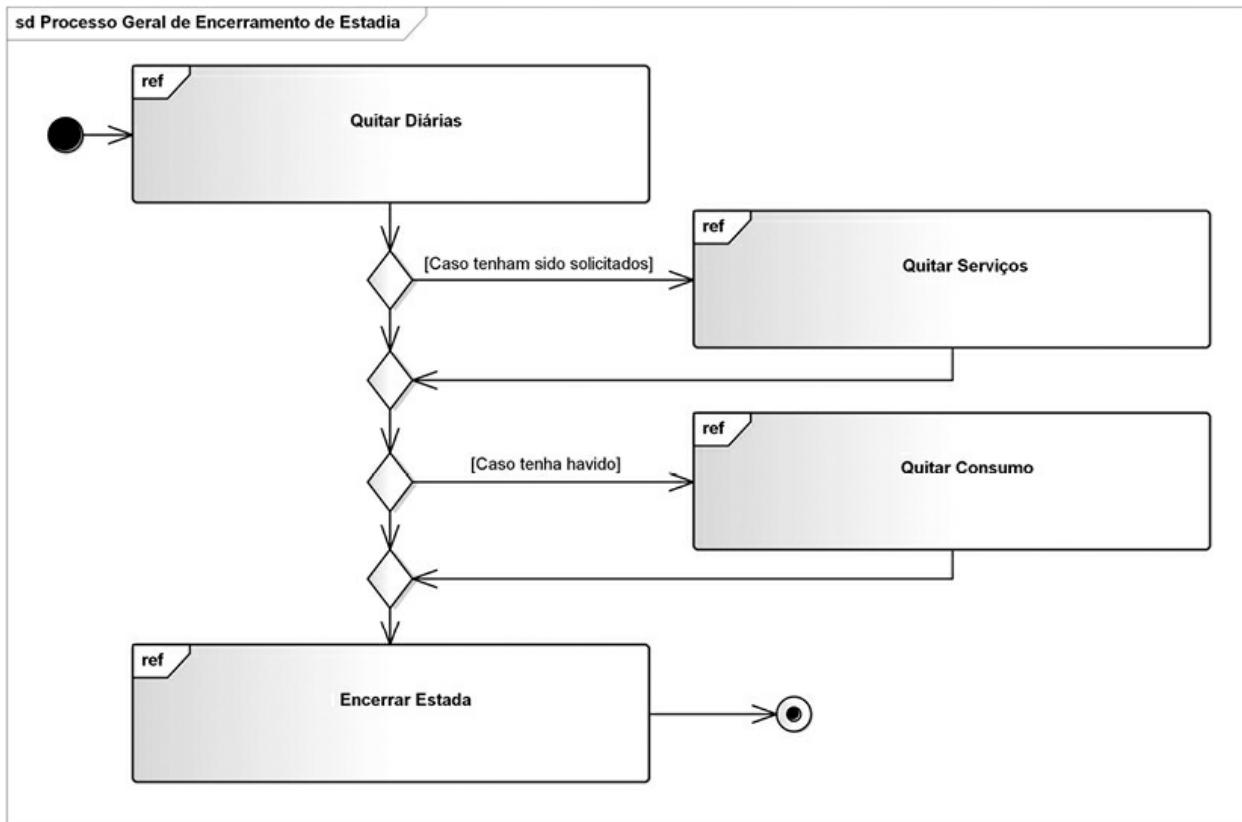
Caso o pedido seja aceito, passa-se ao processo de **Associar Sócio**, representado por um quadro de interação, em que o sócio informa seus dados pessoais através da interface e confirma. Essas informações são

repassadas à controladora, que dispara o método **registrarSocio** em um objeto da classe **Socio**, instanciando-o. Esse método retorna um long (caso seja concluído com sucesso), representando o número do cartão do novo sócio. De posse desse número, a controladora solicita à interface que emita o cartão do novo sócio.

Após o término desse processo, é feito um novo teste, no qual é necessário verificar se o sócio possui dependentes. Caso não possua, o processo geral encerra-se nesse momento. Porém, caso haja dependentes, é necessário, ainda, executar o processo **Associar Dependentes** antes da conclusão do processo geral.

### **11.3.2 Sistema de Controle de Hotel – Processo Geral de Encerramento de Estada**

A figura 11.4 apresenta a solução para esse exercício. É recomendável examinar o diagrama de casos de uso da figura 3.26, referente ao sistema de controle de hotel, em que o leitor notará a existência de uma associação do tipo <<include>> entre os casos de uso **Encerrar Estada** e **Quitar Diárias**.



*Figura 11.4 – Processo Geral de Encerramento de Estadia – Sistema de Controle de Hotel.*

Existem ainda duas associações de extensão entre o caso de uso **Quitar Diárias** e os casos de uso **Quitar Serviços** e **Quitar Consumo**. Sendo assim, antes de encerrar a estada, é preciso quitar todas as diárias ainda devidas e, se tiver ocorrido a solicitação de serviços ou consumo do frigobar, estes deverão também ser pagos para somente, então, poder encerrar a estada. Essas associações serão representadas, sob outra visão, nessa solução.

Pelos motivos já explicados, o primeiro processo a ser iniciado nesse diagrama é o de **Quitar Diárias**. Quando este for finalizado, será necessário fazer um teste para verificar se houve solicitação de algum serviço. Em caso positivo, deve-se executar o processo de **Quitar Serviços** antes de passar ao teste seguinte.

Observe que o fluxo dividido pelo nó de decisão anterior é reunido por um nó de união, o qual conduz a outro nó de decisão, cuja função é verificar se houve consumo do frigobar. Se isso for verdadeiro, deverá ser executado o processo de **Quitar Consumo**.

Depois de terem sido quitados as diárias e, possivelmente, os serviços e

consumos, finalmente se passa ao processo de encerramento propriamente dito, em que a situação do quarto será definida como liberado e o hóspede estará quite com o hotel. Após a conclusão desse último processo, o processo como um todo será encerrado.

## CAPÍTULO 12

# Diagrama de Componentes

O diagrama de componentes, como seu próprio nome indica, identifica os componentes que fazem parte de um sistema, um subsistema ou mesmo os componentes ou classes internas de um componente individual. Um componente pode representar tanto um componente lógico (um componente de negócio ou de processo) ou físico como arquivos contendo código-fonte, arquivos de ajuda (help), bibliotecas, arquivos executáveis etc.

O diagrama de componentes pode ser utilizado para documentar como estão estruturados os arquivos físicos de um sistema, permitindo, assim, uma melhor compreensão deste, além de facilitar a reutilização de código. Esse diagrama também pode identificar os componentes utilizados no desenvolvimento de sistemas baseado em componentes.

A utilização de diagramas de componentes pode ser particularmente útil em atividades de gerenciamento de configuração e mudanças, com o objetivo de modelar baselines, ou seja, todo o conjunto necessário de arquivos (módulos de software, versões de bibliotecas, arquivos de ajuda etc.) para produzir uma versão específica de um sistema. Considerando que um software pode possuir muitas versões em uso, é muito importante saber os componentes específicos necessários à geração de cada versão.

### 12.1 Componente

Um componente pode sempre ser considerado uma unidade autônoma dentro de um sistema ou subsistema. Pode conter uma ou mais interfaces fornecidas e requeridas (potencialmente expostas via portas) e seus interiores são transparentes e inacessíveis por outro meio que não seja fornecido por suas interfaces. Embora possa ser dependente de outros elementos em termos de interfaces requeridas, um componente é encapsulado e suas dependências são projetadas de tal forma que possa ser

tratado o mais independentemente possível.

Na UML 1.x, um componente era representado por um retângulo com dois retângulos menores sobressaindo à sua esquerda. A partir da UML 2, esse símbolo foi substituído por um retângulo contendo internamente o antigo símbolo, conforme demonstra a figura 12.1.



*Figura 12.1 – Exemplo de Componente.*

O componente **GerenciadorContas** apresentado na figura 12.1 representa um módulo executável, responsável pelo gerenciamento e pela manutenção de contas bancárias. Isso pode ser melhor explicitado pelo uso de estereótipos, os quais, como explicado anteriormente, atribuem características extras a um determinado item de um diagrama. A UML prevê diversos estereótipos prontos para o diagrama de componentes, como:

- **Executável** (executable) – determina que o componente em questão é um arquivo executável, ou seja, um arquivo já compilado que pode executar uma série de instruções. O termo executável é genérico, podendo se referir a módulos compilados em Java, que, às vezes, não são executáveis no sentido clássico, sendo interpretados pela máquina virtual.
- **Biblioteca** (library) – refere-se a bibliotecas contendo funções e subrotinas que podem ser compartilhadas por diversos componentes executáveis, podendo estas ser fornecidas pela própria linguagem em que o sistema será desenvolvido ou ser criadas pelos desenvolvedores do sistema.
- **Tabela** (table) – refere-se a repositórios físicos de dados onde as informações produzidas pelo sistema deverão ser armazenadas.
- **Documento** (document) – faz referência a arquivos-texto utilizados por outros componentes do sistema, como arquivos de ajuda (help), por exemplo.

- **Arquivo** (file) – pode se referir a qualquer outro arquivo que componha o sistema, como arquivos de código-fonte dos módulos do sistema, por exemplo.

A figura 12.2 apresenta exemplos do uso de estereótipos em componentes.

Pelo exame dos estereótipos representados na figura, podemos perceber que o componente **GerenciadorContas** refere-se a um módulo executável, que o componente **Conta** refere-se a uma tabela de um banco de dados e que o componente **Biblioteca** representa uma biblioteca cujos recursos são utilizados pelos módulos que compõem outros softwares.

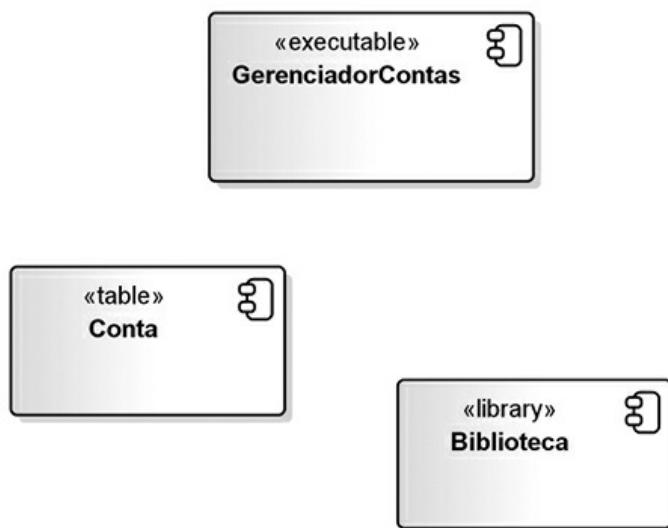
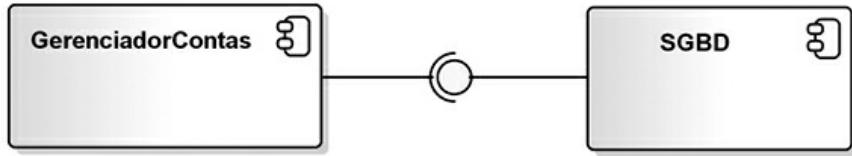


Figura 12.2 – Componentes com Estereótipos.

## 12.2 Interfaces Fornecidas e Requeridas

Na UML 1.x, uma das principais formas de representar comunicação entre os componentes era realizada por meio do relacionamento de dependência, já explicado no capítulo 4 sobre o diagrama de classes. No entanto, a partir da UML 2, embora esse tipo de relacionamento continue sendo válido e utilizado, passou-se a utilizar com muito mais frequência interfaces fornecidas e requeridas, as mesmas descritas no diagrama de classes, conforme demonstra a figura 12.3.



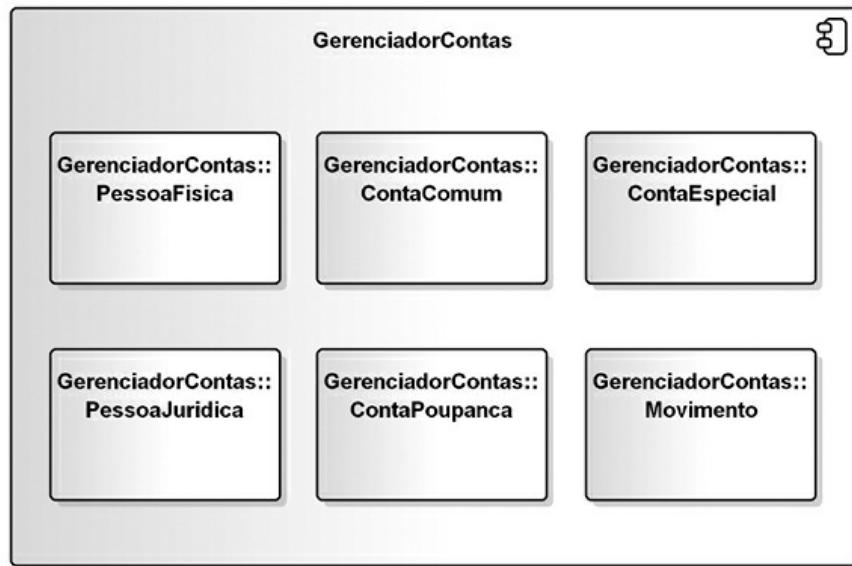
*Figura 12.3 – Relacionamento entre Componentes por meio de Interfaces Fornecidas e Requeridas.*

Nesse exemplo, podemos perceber que o componente **GerenciadorContas** tem um relacionamento com o componente SGBD, que representa um sistema gerenciador de banco de dados. Observe que o componente **GerenciadorContas** tem uma interface requerida com o componente SGBD, e este, uma interface fornecida com o componente **GerenciadorContas**, uma vez que esse último componente solicita frequentemente a gravação e recuperação de informações no SGBD.

### 12.3 Classes e Componentes Internos

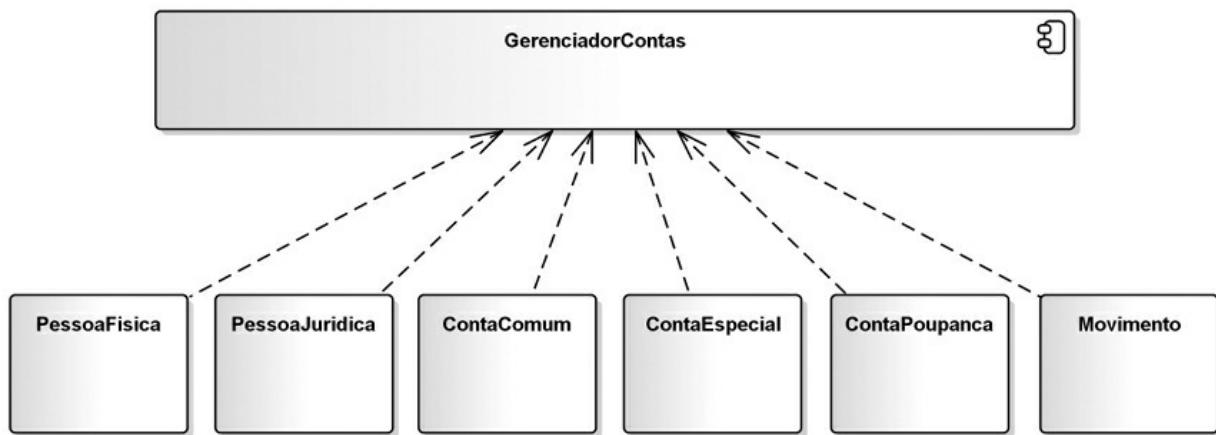
Um componente pode conter ou implementar uma ou mais classes internas, podendo conter também componentes internos. A representação de um componente sem apresentar seus componentes ou classes internas é chamada de visão de caixa preta. Já a representação de um componente com suas classes ou componentes internos é chamada visão de caixa branca. A contenção de classes por um componente significa que este as implementa de alguma maneira.

A figura 12.4 demonstra um exemplo de componente com classes internas. O leitor notará que o componente **GerenciadorContas** implementa as classes necessárias para a manutenção de contas do sistema bancário. É possível também definir associações entre os componentes internos, mas isso pode deixar o componente exageradamente grande.



*Figura 12.4 – Componente com Classes Internas.*

Outra forma de representar as classes internas de um componente é por meio do relacionamento de dependência, conforme demonstra a figura 12.5.

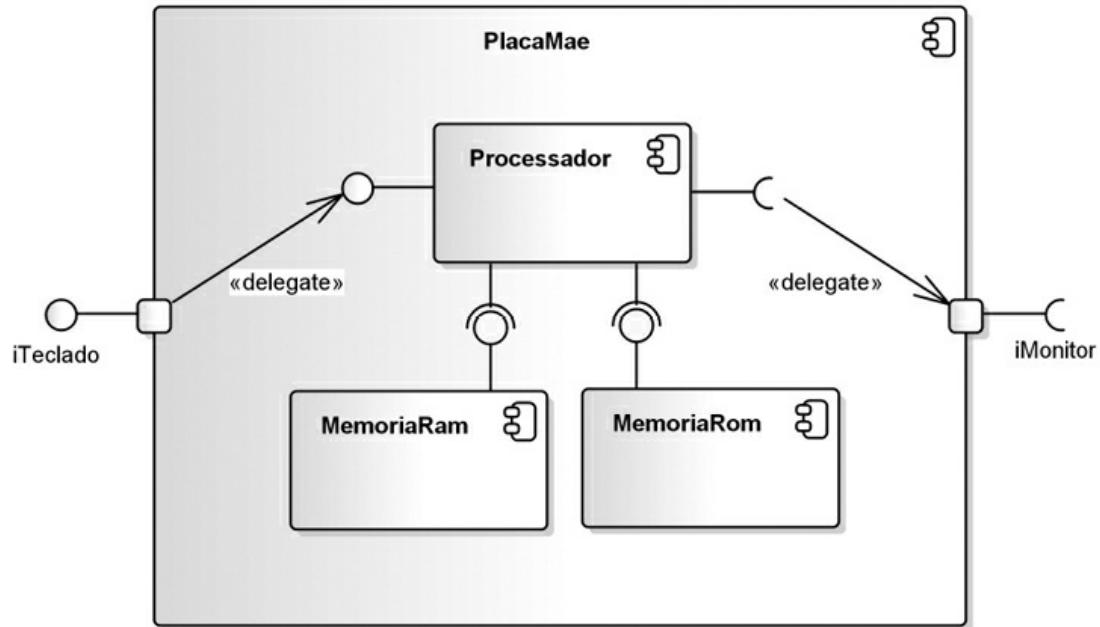


*Figura 12.5 – Classes Internas relacionadas a um componente por meio de Dependências.*

### 12.3.1 Portas

O conceito de Portas foi explicado no capítulo 4 sobre o diagrama de classes. Seu uso é bastante comum em componentes para comunicar os elementos internos de um componente com o ambiente externo quando este solicita ou executa algum serviço. A figura 12.6 apresenta um exemplo

de componente utilizando Portas.



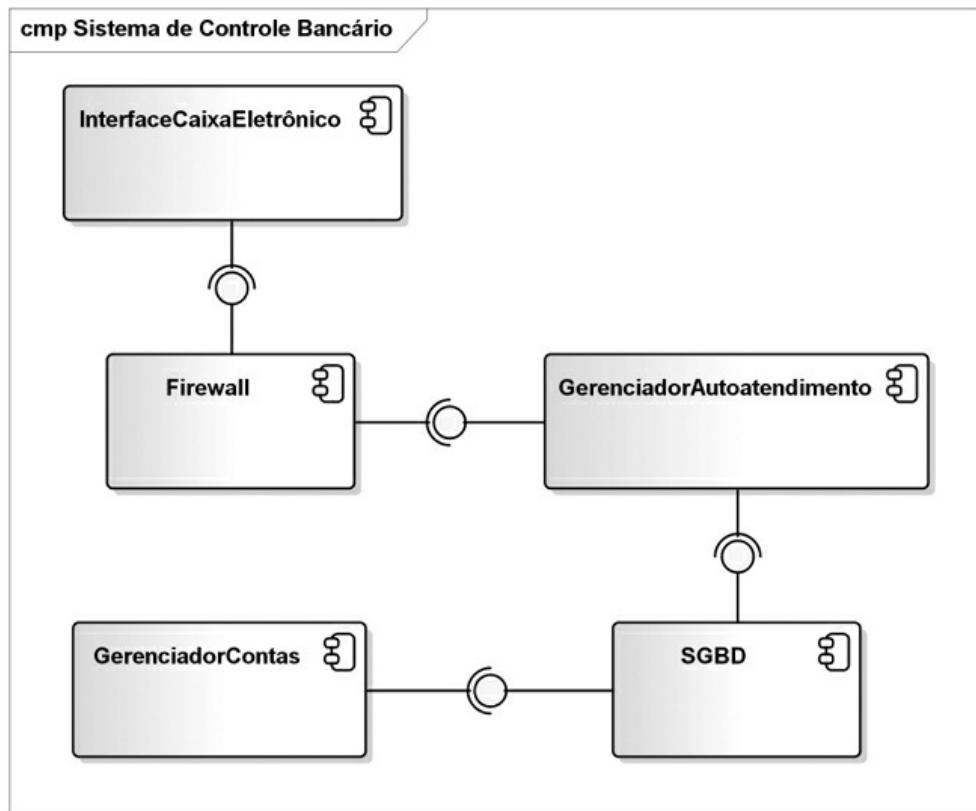
*Figura 12.6 – Visão de Caixa Branca de um Componente.*

Nesse exemplo, representamos um componente chamado **PlacaMae** com seus componentes internos. Observe que o componente **Processador** solicita serviços dos componentes **MemoriaRam** e **MemoriaRom**, além de se comunicar com o ambiente externo por meio de portas. Note também o relacionamento de dependência com o estereótipo <<delegate>> entre as interfaces do componente e as portas. Esse estereótipo passa a mensagem de que o serviço atribuído ao componente **PlacaMae** está sendo delegado a seu componente interno **Processador**. Essa mesma abordagem pode ser também utilizada com as classes internas de um componente.

## 12.4 Exemplo de Diagrama de Componentes – Sistema de Controle Bancário

Na Figura 12.7, apresentaremos o diagrama de componentes equivalentes aos módulos executáveis do sistema de controle bancário que estamos modelando ao longo deste livro. Alguns módulos não são exatamente executáveis, como o componente que representa a interface do caixa eletrônico, e outros podem não pertencer exclusivamente ao sistema, como os componentes que representam o firewall e o sistema gerenciador de

banco de dados, mas são indispensáveis ao funcionamento dele.



*Figura 12.7 – Diagrama de Componentes – Sistema de Controle Bancário.*  
Esse diagrama representa os seguintes componentes:

- **InterfaceCaixaEletrônico** – Esse componente representa simplesmente a interface do caixa eletrônico, por meio da qual o cliente pode solicitar serviços como emissão de saldo e extrato, saque ou depósito. Observe que esse componente apresenta uma interface requerida com o componente firewall, uma vez que solicita ao firewall a transmissão dos eventos ocorridos sobre a interface ao **GerenciadorAutoatendimento**. Alternativamente, poderíamos modelar um componente de interface mais genérico que representasse também uma interface via web, por exemplo.
- **Firewall** – Esse componente representa um software que implementa um sistema de firewall, ou seja, é um software de segurança que tenta impedir que usuários não autorizados invadam o sistema. Esse componente não faz realmente parte do sistema bancário, no entanto, por medida de segurança, todo o tráfego externo deve passar por ele.

antes de se comunicar com o sistema propriamente dito.

- **GerenciadorAutoatendimento** – Esse componente representa o módulo do sistema responsável por gerenciar as solicitações realizadas pelos clientes na interface do caixa eletrônico. Deve interpretar os eventos ocorridos na interface e solicitar as execuções adequadas a eles.
- **SGBD** – Esse componente representa o sistema gerenciador de banco de dados com o qual o sistema interage. É responsável por persistir e recuperar os dados do sistema. Não faz realmente parte do sistema, mas é indispensável a ele.
- **GerenciadorContas** – Esse é o módulo responsável por gerir as contas mantidas pela instituição bancária. Por meio dele, é possível abrir, encerrar contas ou emitir relatórios gerenciais. É o núcleo do sistema propriamente dito. Observe que tanto esse módulo como o **GerenciadorAutoatendimento** contêm interfaces requeridas com o SGBD, uma vez que este executa os pedidos de gravação e recuperação das informações do sistema.

## 12.5 Exercícios Propostos

Como tem ocorrido nos capítulos anteriores, continuaremos a modelagem dos sistemas já iniciados, enfocando, desta vez, o diagrama de componentes. Como a maioria dos sistemas estudados é simples, os diagramas de componentes serão pequenos, alguns até bastante semelhantes entre si, por existirem poucos módulos de software na maioria das vezes.

### 12.5.1 Sistema de Controle de Cinema

Desenvolva o diagrama de componentes para um sistema de controle de cinema, sabendo que:

- É preciso existir um módulo para gerir a venda de ingresso aos clientes. Esse módulo deve gerar os ingressos e emiti-los por meio da interface (que pode ser física também) para os clientes do cinema.
- O sistema necessita de um SGBD para persistir suas informações.
- Finalmente, é necessário um módulo de manutenção do sistema, onde basicamente serão mantidos os cadastros de sessões, salas, filmes,

atores, gêneros etc.

### **12.5.2 Sistema de Controle de Clube Social**

Desenvolva o diagrama de componentes referente ao sistema de clube social, levando em consideração os seguintes fatos:

- O clube necessita de um módulo para gerenciar os pedidos de associação dos candidatos a sócio.
- O clube precisa também de um módulo para manter o cadastro de seus sócios, dependentes e suas categorias.
- É preciso existir um SGBD para persistir esses dados.
- Finalmente, é necessário haver um módulo para a emissão e quitação das mensalidades dos sócios.

### **12.5.3 Sistema de Locação de Veículos**

Desenvolva o diagrama de componentes para um sistema de aluguel de veículos, considerando que:

- O sistema precisa de um módulo para gerenciar o cadastro dos clientes, automóveis, modelos e marcas.
- É preciso existir um SGBD para gravar e recuperar os dados do sistema.
- É necessário ainda haver um módulo para realizar as locações dos automóveis da empresa, bem como registrar as devoluções dessas mesmas locações.

### **12.5.4 Sistema para Controle de Leilão Via Internet**

Desenvolva o diagrama de componentes relativo a um sistema de controle de leilão via internet, de acordo com os seguintes requisitos:

- O sistema necessita de uma página por meio da qual os participantes poderão se logar ou se autorregular. Essa página será útil também para apresentar os itens anunciados pelo leiloeiro e receber as ofertas dos participantes, bem como para anunciar um item como arrematado.
- O sistema precisa de um módulo que gerencie o login dos participantes. É recomendável existir um módulo associado ao módulo de login para permitir o autorregistro das pessoas interessadas em participar do leilão.

- É necessário haver um módulo que permita registrar novos leilões, bem como cadastrar os itens que serão anunciados neles.
- Também é necessário um módulo responsável por gerenciar cada leilão, que permita a um leiloeiro abrir um leilão, anunciar os itens deste, receber lances, anunciar vencedores e arrematar itens.
- Finalmente, é necessário um sistema gerenciador de banco de dados para persistir e recuperar as informações necessárias ao sistema.

### **12.5.5 Sistema de Controle de Hotelaria**

Desenvolva o diagrama de componentes para um sistema de controle de hotelaria, de acordo com as seguintes definições:

- Como de praxe, é necessário haver um módulo de manutenção para gerenciar os cadastros de funcionários, categorias, quartos etc.
- É necessário também implementar um módulo para realizar as reservas de quartos pelos hóspedes.
- Da mesma forma, é preciso existir um módulo para gerenciar o aluguel dos quartos.
- O aluguel de um quarto pode implicar a solicitação de serviços ou o consumo de itens do frigobar, assim é necessário haver alguma forma de registrar esses pedidos.

### **12.5.6 Sistema de Controle de Imobiliária**

Desenvolva o diagrama de componentes para um sistema de controle de imobiliária, de acordo com as seguintes afirmações:

- Como nos outros exercícios, é preciso haver um módulo de manutenção dos cadastros do sistema.
- O software também necessita de um módulo que permita aos usuários se autenticarem.
- O sistema deve conter ainda um módulo para gerenciar os contratos que autorizam a imobiliária a vender ou alugar imóveis.
- Finalmente, o sistema deve conter um módulo para gerenciar as vendas de imóveis realizadas pela imobiliária, bem como um módulo que gerencie as locações de um imóvel por inquilinos e os pagamentos

mensais relativos às locações.

## 12.6 Solução dos Exercícios

### 12.6.1 Sistema de Controle de Cinema

A figura 12.8 apresenta a solução para esse exercício.

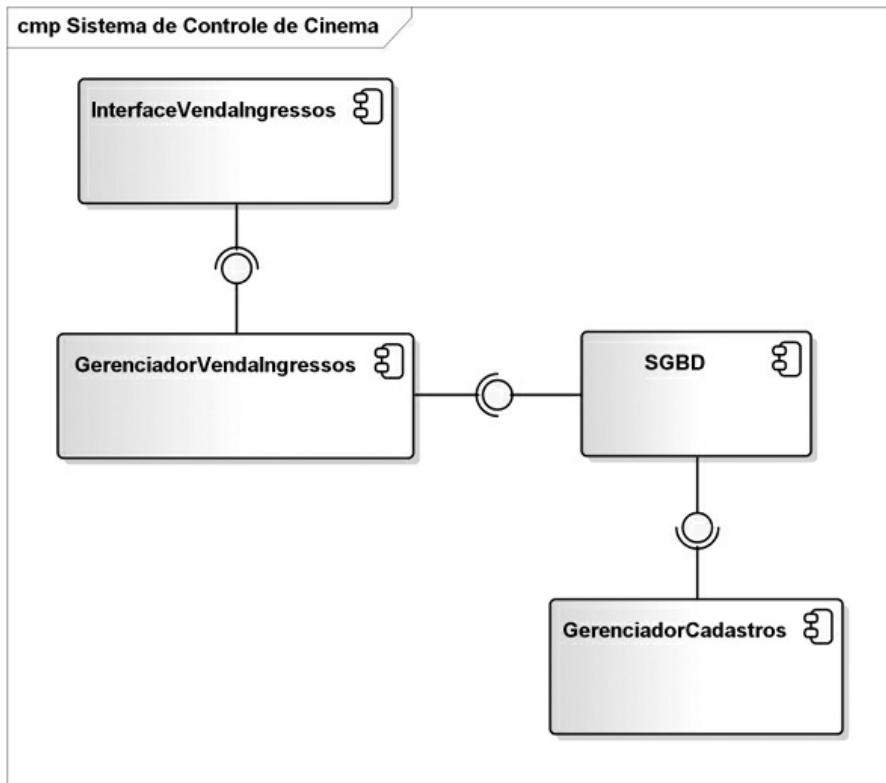


Figura 12.8 – Diagrama de Componentes – Sistema de Controle de Cinema.

Esse diagrama representa os seguintes componentes:

- **InterfaceVendaIngressos** – Esse componente representa a interface de vendas de ingressos, abrangendo não somente a interface do sistema, mas também o hardware necessário para a impressão de ingressos.
- **GerenciadorVendaIngressos** – Esse módulo é responsável por gerenciar o processo de venda de ingressos para o cinema.
- **SGBD** – Esse componente representa o sistema gerenciador de banco de dados responsável por manter as informações do sistema.
- **GerenciadorCadastros** – Esse módulo é responsável pela manutenção dos cadastros do sistema, como filmes, sessões, salas, atores, gêneros

etc.

## 12.6.2 Sistema de Controle de Clube Social

A solução para esse exercício é apresentada na figura 12.9.

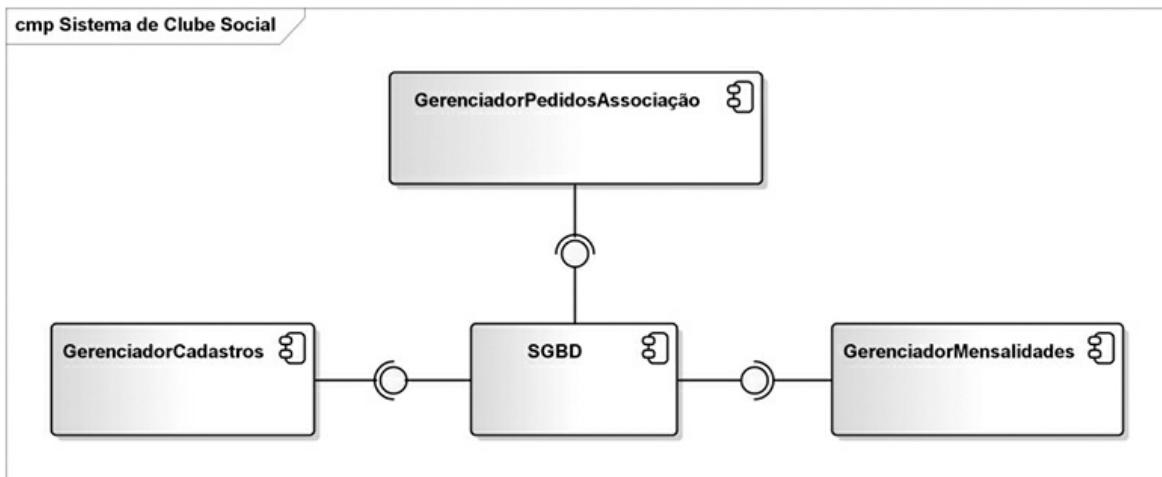


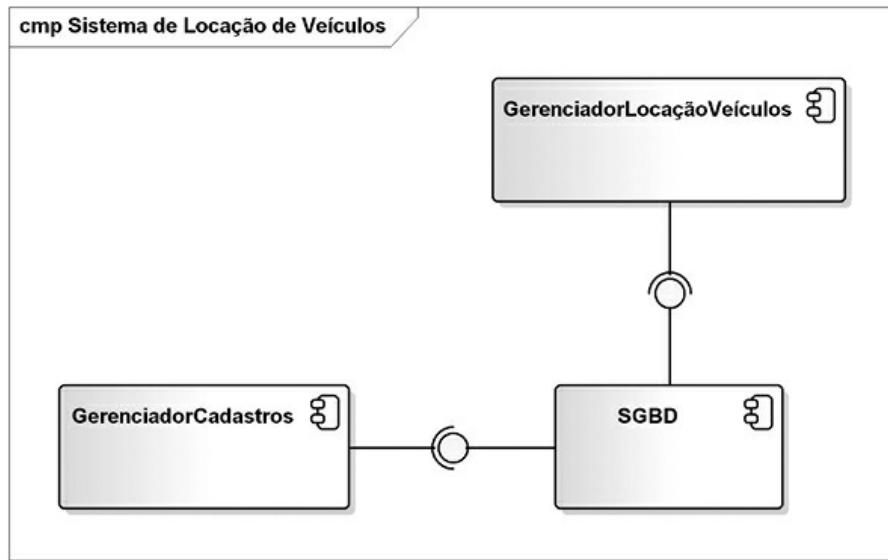
Figura 12.9 – Diagrama de Componentes – Sistema de Controle de Clube Social.

Esse diagrama representa os seguintes componentes:

- **GerenciadorPedidosAssociação** – Esse módulo tem por função gerenciar os pedidos de pessoas que desejam tornar-se sócias do clube, bem como gerenciar a avaliação desses pedidos pelo clube e notificar os solicitantes da aprovação ou não de seus pedidos.
- **GerenciadorCadastrados** – Esse módulo é responsável pela manutenção dos cadastros do sistema, como sócios, dependentes e categorias.
- **GerenciadorMensalidades** – Esse módulo é responsável por gerar as mensalidades devidas pelos sócios, bem como realizar a quitação delas.
- **SGBD** – Esse componente representa o sistema gerenciador de banco de dados responsável por manter as informações do sistema. Observe que os dois módulos anteriores apresentam interfaces requeridas com esse componente, uma vez que solicitam operações de gravação e recuperação de dados no SGBD.

## 12.6.3 Sistema de Locação de Veículos

A solução desse exercício está ilustrada na figura 12.10.

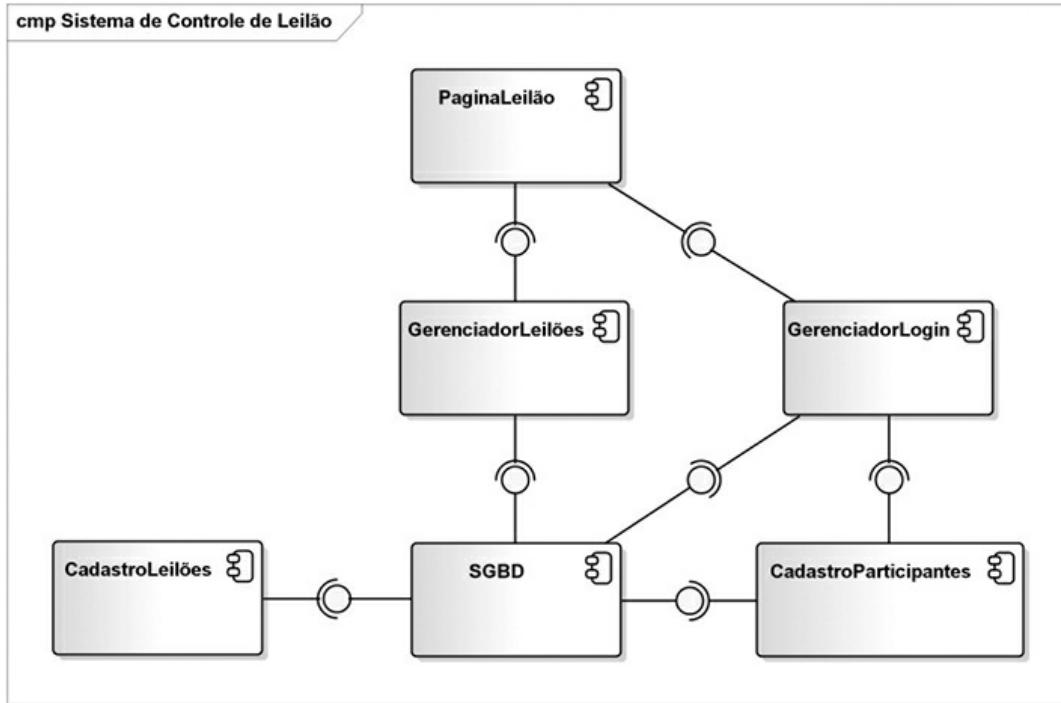


*Figura 12.10 – Diagrama de Componentes – Sistema de Locação de Veículos.*

Esse diagrama é muito semelhante ao apresentado na solução anterior. Por esse motivo, explicaremos apenas o **GerenciadorLocaçãoVeículos**, cuja função é gerenciar as locações de veículos da empresa, bem como as devoluções dela.

#### 12.6.4 Sistema para Controle de Leilão Via Internet

A seguir, explicaremos os componentes do diagrama que representa a solução desse exercício apresentado na figura 12.11.



*Figura 12.11 – Diagrama de Componentes – Sistema para Controle de Leilão Via Internet.*

Esse diagrama representa os seguintes componentes:

- **PáginaLeilão** – Esse módulo representa a interface do sistema por meio do qual os participantes do leilão se cadastram, logam e submetem lances. Observe que essa página tem interfaces requeridas com os módulos **GerenciadorLogin** e **GerenciadorLeilões**, posto que são eles que realmente implementam os serviços oferecidos na página.
  - **GerenciadorLogin** – Esse módulo gerencia o login dos participantes no sistema, não permitindo que pessoas não registradas participem do leilão.
  - **CadastroParticipantes** – Esse módulo está associado ao anterior, sendo responsável por permitir o autocadastro de novos participantes.
  - **CadastroLeilões** – Esse módulo permite cadastrar novos leilões, bem como os itens a eles associados.
  - **GerenciadorLeilões** – Este é o módulo principal do sistema, por meio do qual um leilão é iniciado, seus itens são anunciados, os lances são recebidos e anunciados e os itens, arrematados.
  - **SGBD** – Como de praxe, esse componente representa o sistema

gerenciador de banco de dados que mantém as informações do sistema.

### 12.6.5 Sistema de Controle de Hotelaria

Os componentes representados na figura 12.12, que representa a solução desse exercício, serão explicados a seguir.

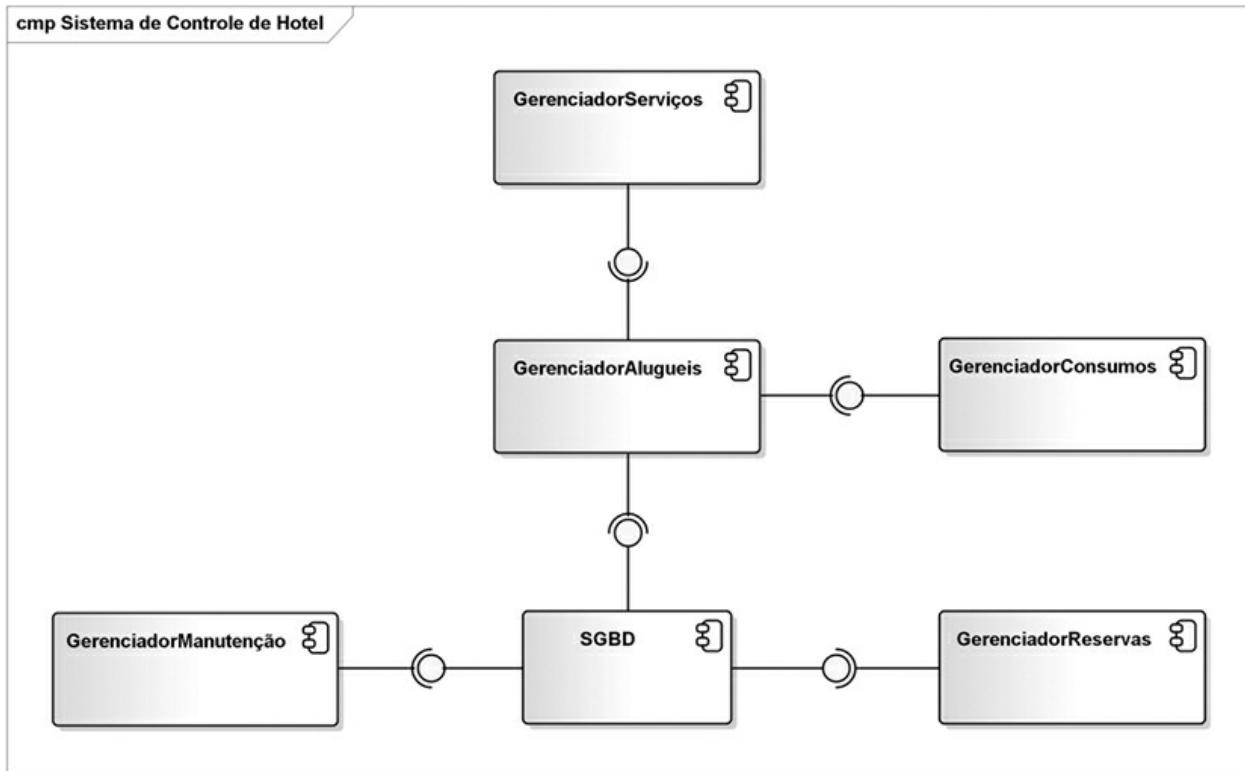


Figura 12.12 – Diagrama de Componentes – Sistema de Controle de Hotelaria.

Esse diagrama representa os seguintes componentes:

- **GerenciadorManutenção** – Esse módulo é responsável pela manutenção dos cadastros do sistema, como funcionários, categorias, quartos, itens de frigobar, serviços oferecidos etc.
- **GerenciadorReservas** – Esse módulo tem como função permitir que um funcionário reserve um quarto para um hóspede ou mesmo que um hóspede faça sua reserva diretamente por meio da página do hotel.
- **GerenciadorAlugueis** – Este é um dos módulos mais importantes do sistema, sendo responsável por permitir o aluguel de quartos por um funcionário do hotel para um hóspede. Observe que este módulo contém interfaces requeridas com os dois módulos seguintes, uma vez

que solicita o registro dos serviços pedidos por um hóspede e o registro do consumo de frigobar para esses dois módulos.

- **GerenciadorServiços** – Esse módulo está associado ao módulo anterior e permite o registro de pedido de serviços por um hóspede.
- **GerenciadorConsumos** – Esse módulo também está associado ao **GerenciadorAlugueis** e sua função é registrar o consumo de frigobar por um hóspede.
- **SGBD** – Esse módulo é responsável pelo armazenamento e recuperação das informações necessárias ao sistema.

## 12.6.6 Sistema de Controle de Imobiliária

A resolução deste exercício está representada na figura 12.13. Explicaremos cada um dos componentes nela contidos.

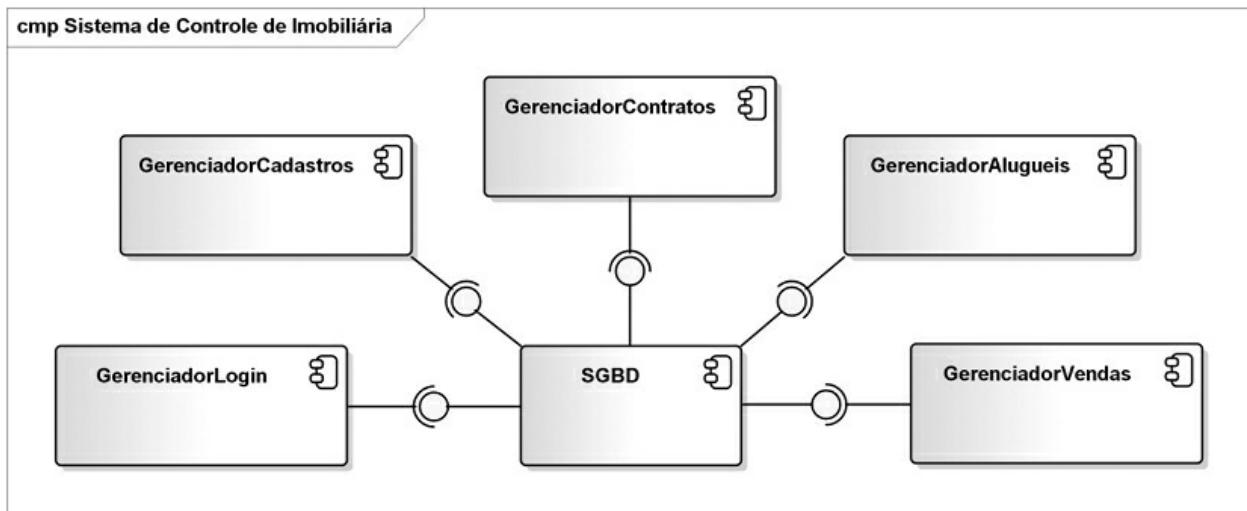


Figura 12.13 – Diagrama de Componentes – Sistema de Controle de Imobiliária.

- **GerenciadorLogin** – Este é um módulo simples que permite ao administrador e aos corretores da imobiliária se autenticarem no sistema.
- **GerenciadorCadastros** – Esse é igualmente um módulo simples, responsável pela manutenção dos cadastros do sistema, como os de corretores, clientes e imóveis.
- **GerenciadorContratos** – Por meio deste módulo, é possível registrar e controlar todos os contratos de compra ou aluguel de imóveis

intermediados pela imobiliária.

- **GerenciadorAlugueis** – Esse componente permite controlar a locação de imóveis, incluindo o controle de pagamento dos aluguéis referentes a cada locação, bem como possíveis multas, quebras de contrato etc., além das comissões devidas à imobiliária.
- **GerenciadorVendas** – Esse componente possibilita controlar todas as vendas de imóveis intermediadas pela empresa, incluindo quem vendeu e quem comprou, o valor final pago pelo imóvel, os valores de impostos e taxas, o corretor que intermediou a venda e as comissões devidas à imobiliária e ao corretor.
- **SGBD** – Finalmente, esse módulo representa um sistema gerenciador de banco de dados por meio do qual os dados do sistema são preservados e recuperados.

## CAPÍTULO 13

# Diagrama de Implantação

O diagrama de implantação é o diagrama com a visão mais física da UML. Enfoca a questão da organização da arquitetura física sobre a qual o software será implantado e executado em termos de hardware, ou seja, as máquinas (computadores pessoais, servidores etc.) que suportarão o sistema, além de definir como estas estarão conectadas e por meio de quais protocolos se comunicarão e transmitirão informações. Esse diagrama também permite demonstrar como se dará a distribuição dos módulos do sistema, em situações em que estes sejam executados em mais de um servidor.

Um diagrama de implantação obviamente só é útil quando o sistema modelado for executado sobre múltiplas máquinas que executem determinados módulos do software ou armazenem arquivos necessários a este. Se o software for projetado para ser executado sobre uma única máquina individual que não se comunique com outro hardware, não será necessário modelar um diagrama de implantação.

### 13.1 Nós

Nós são os componentes básicos de um diagrama de implantação. Um nó pode representar um item de hardware, como um servidor onde um ou mais módulos do software são executados ou que armazene arquivos consultados pelos módulos do sistema. Um nó pode representar também um ambiente de execução, ou seja, um ambiente que suporta o sistema de alguma forma. Nós podem conter outros nós, sendo possível encontrar um nó que representa um item de hardware contendo outro nó que representa um ambiente de execução, embora um nó que represente um item de hardware possa conter outros nós representando outros itens de hardware e um nó que represente um ambiente de execução possa conter outros ambientes de execução.

Quando um nó representa um hardware, este pode apresentar o estereótipo <<device>>; já quando um nó representa um ambiente de execução pode utilizar o estereótipo <<ExecutionEnvironment>>. Exemplos de ambientes de execução são sistemas operacionais ou sistemas de banco de dados. Um nó é representado por um cubo contendo um texto indicando o nome do item de hardware ou ambiente de execução que ele representa. A figura 13.1 demonstra um exemplo de nó, representando um item de hardware.



*Figura 13.1 – Exemplo de Nó.*

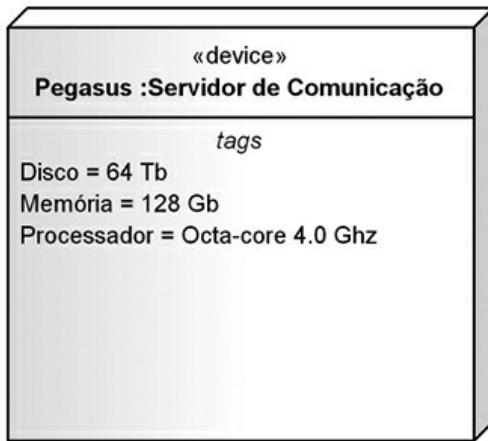
Essa figura apresenta um exemplo de nó que se refere a um caixa eletrônico qualquer, sem especificar um em especial. Podemos, no entanto, definir um nome específico para a máquina que o nó representa, como é comum encontrar nas organizações que têm grandes redes com muitos servidores espalhados, conforme demonstra a figura 13.2.



*Figura 13.2 – Nó com Denominação.*

Nesse exemplo, determinamos que o nó representa um servidor de firewall e que o nome pelo qual esse servidor é conhecido na organização é Kerberos. Um nó pode ainda conter detalhes de configuração do hardware que ele representa. Os detalhes de configuração podem ser definidos por

meio de etiquetas (tags), que são uma forma de estabelecer propriedades extras para um elemento. A figura 13.3 mostra um exemplo de nó com sua configuração detalhada.



*Figura 13.3 – Nó com Detalhes de Configuração.*

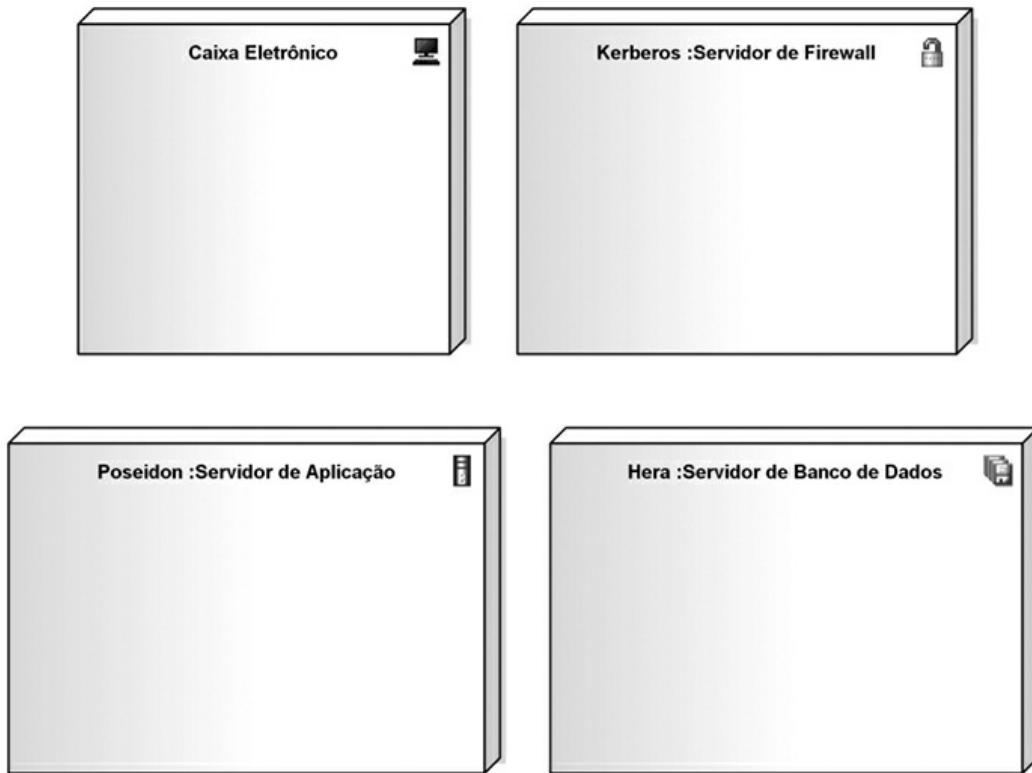
Aqui, utilizamos outro nó que representa um servidor de comunicação denominado Pegasus. A configuração desse servidor é definida como tendo 64 terabytes de disco, 128 gigabytes de memória RAM e um processador quad-core com velocidade de 4.0 megahertz.

## 13.2 Estereótipos

Existem outros estereótipos que podem ser aplicados a nós, além do estereótipo genérico <<device>>, que podem ser aplicados a itens de hardware, como <<computer>>, que representa um computador mais simples; <<secure>>, que representa um hardware de segurança responsável por impedir invasões à rede interna da organização; <<server>>, que representa um servidor; ou <<storage>>, que representa um hardware cuja função é armazenar informações, como um servidor de banco de dados ou de arquivos.

A figura 13.4 apresenta exemplos desses quatro estereótipos (existem outros), na ordem em que foram citados, da esquerda para direita e de cima para baixo. Assim, o nó Caixa Eletrônico representa um computador simples, o nó Kerberos identifica um hardware de segurança (no caso, um firewall), o nó Poseidon representa um servidor de aplicação e o nó Hera, um servidor de banco de dados. Observe que esses estereótipos são todos

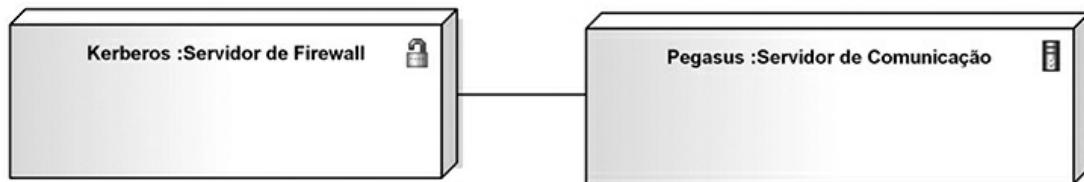
gráficos, uma vez que modificam de alguma forma o desenho-padrão do elemento.



*Figura 13.4 – Nós com Estereótipos.*

### 13.3 Associações

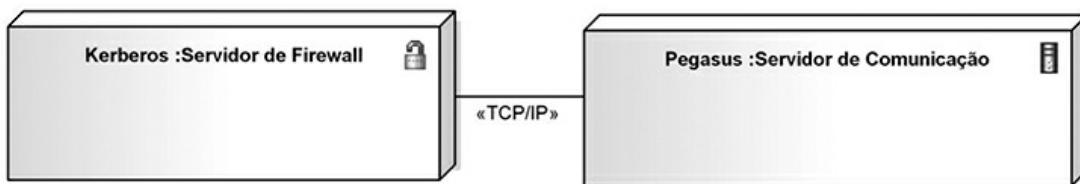
Os nós podem ter ligações físicas entre si para se comunicar e trocar informações. Tais ligações denominam-se associações e são representadas por linhas ligando um nó a outro. A figura 13.5 apresenta um exemplo de associação entre nós.



*Figura 13.5 – Associação entre Nós.*

Como podemos observar nessa figura, o servidor de firewall Kerberos está associado ao servidor de comunicação Pegasus. As associações, além de

determinarem uma ligação física entre os nós, podem também determinar a forma como se comunicam, ou seja, o protocolo de comunicação utilizado pelo uso de estereótipos. Um exemplo de nós associados utilizando estereótipos é apresentado na figura 13.6.



*Figura 13.6 – Associação entre Nós com Estereótipo.*

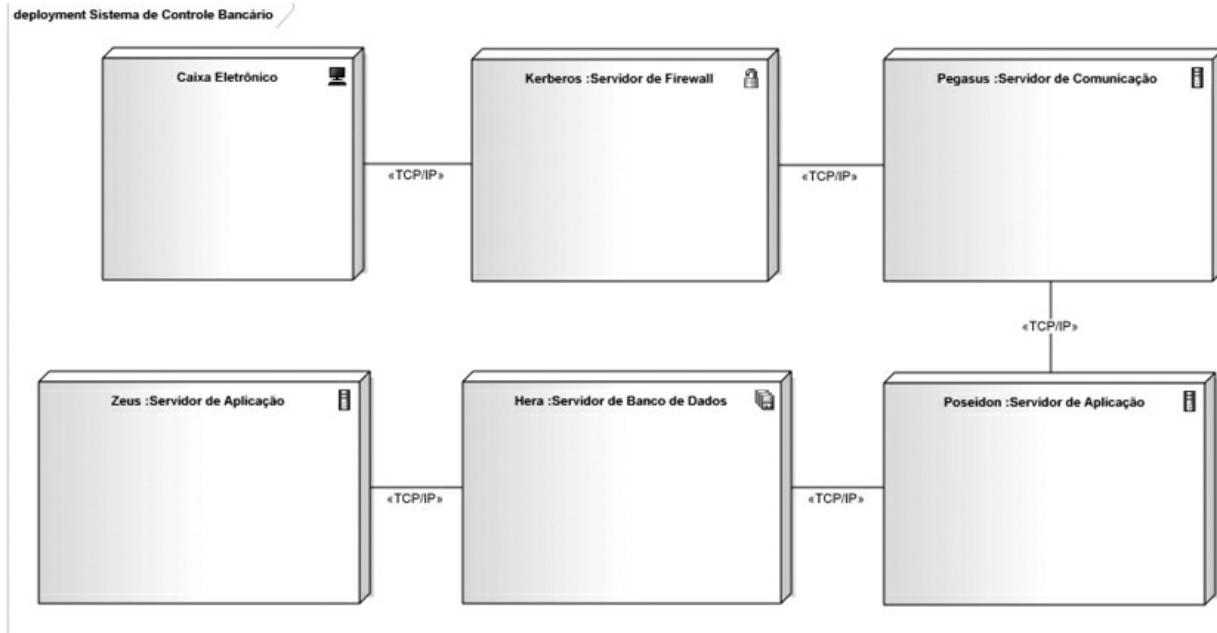
Ao observarmos essa figura, percebemos que os nós **Kerberos** e **Pegasus** estão associados e comunicam-se por meio do protocolo TCP/IP.

### 13.4 Exemplo de Diagrama de Implantação

O exemplo apresentado na figura 13.7 enfoca o ambiente físico onde será executado o sistema de controle bancário que vem sendo desenvolvido ao longo de diversos capítulos deste livro.

Nesse exemplo, existe um grande número de terminais de caixa eletrônico, muitos deles distantes geograficamente, em conexão com o sistema bancário. Obviamente, não há como detalhar cada um deles. Assim, o nó **Caixa Eletrônico** (cujo estereótipo indica que é um computador simples) representa todos os terminais de caixa eletrônico disponibilizados pela instituição bancária.

O nó **Caixa Eletrônico** comunica-se com o servidor de firewall Kerberos, responsável por impedir a invasão de pessoas não autorizadas na empresa, como demonstra seu estereótipo. O servidor Kerberos repassa as mensagens recebidas (se estas forem consideradas válidas e seguras) para o servidor de comunicação Pegasus. Esse servidor é necessário, uma vez que o sistema deve ser capaz de suportar uma quantidade muito grande de conexões que precisarão ser gerenciadas.



*Figura 13.7 – Diagrama de Implantação – Sistema de Controle Bancário.*

O servidor Pegasus repassa as mensagens para o servidor de aplicação Poseidon, onde o módulo de gerenciamento de autoatendimento está sendo executado, sendo este responsável por interpretar os eventos ocorridos no caixa eletrônico. Por sua vez, o servidor Poseidon comunica-se com o servidor de banco de dados Hera, que é um servidor de armazenamento, como se pode notar por seu estereótipo, no qual estará executando algum tipo de sistema gerenciador de banco de dados para armazenar e recuperar as informações necessárias ao sistema. Observe que existe ainda um segundo servidor de aplicação, chamado Zeus, onde são executados outros módulos do sistema de controle bancário, como o gerenciador de contas.

## 13.5 Artefatos

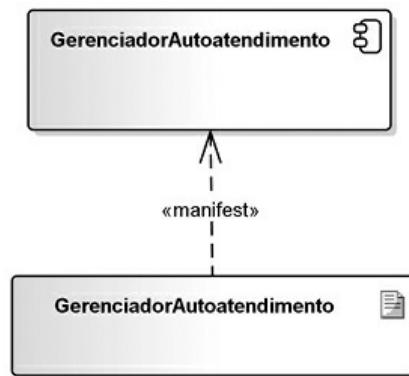
Um artefato é uma entidade física, um elemento concreto que existe realmente no mundo real, assim como os nós que o suportam. Um artefato pode ser um arquivo fonte, um arquivo executável, um arquivo de ajuda, um documento de texto etc. Um artefato deve estar implementado em um nó. A figura 13.8 apresenta um exemplo de artefato implementado em um nó.

O leitor notará que o artefato denominado **GerenciadorAutoatendimento** tem a mesma denominação que um dos componentes apresentados no

capítulo 12, sobre o diagrama de componentes. Na verdade, um artefato é muitas vezes uma “manifestação” no mundo real de um componente. No entanto, não necessariamente existirá um artefato para cada componente, sendo possível existirem diversos artefatos manifestados a partir de um único componente. A figura 13.9 apresenta um exemplo de artefato instanciado a partir de um componente. Observe que existe um relacionamento de dependência entre o componente e o artefato, contendo o estereótipo <<manifest>>, significando que o artefato é uma representação do componente no mundo real.



*Figura 13.8 – Artefato implementado em um Nô.*



*Figura 13.9 – Artefato manifestado a partir de um Componente.*

Ao examinarmos essa figura, podemos perceber que o artefato **GerenciadorAutoatendimento** é uma manifestação do componente de mesmo nome.

Outra forma de demonstrar que um artefato está contido em um nó é também por meio de um relacionamento de dependência, contendo o estereótipo <<deploy>>, entre o nó e os artefatos, conforme demonstra a

figura 13.10.

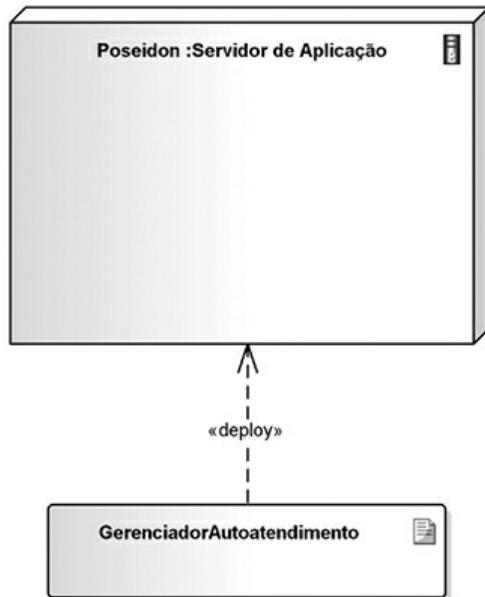


Figura 13.10 – Artefato implementado em um nó.

### 13.6 Especificação de Implantação

Especifica um conjunto de propriedades que determinam parâmetros de execução de um artefato implementado em um nó, conforme demonstra a figura 13.11.

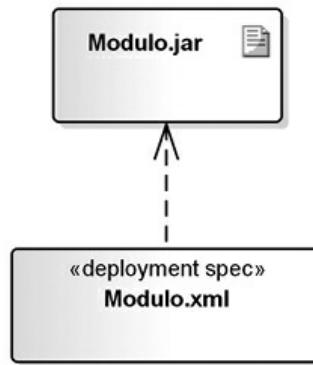
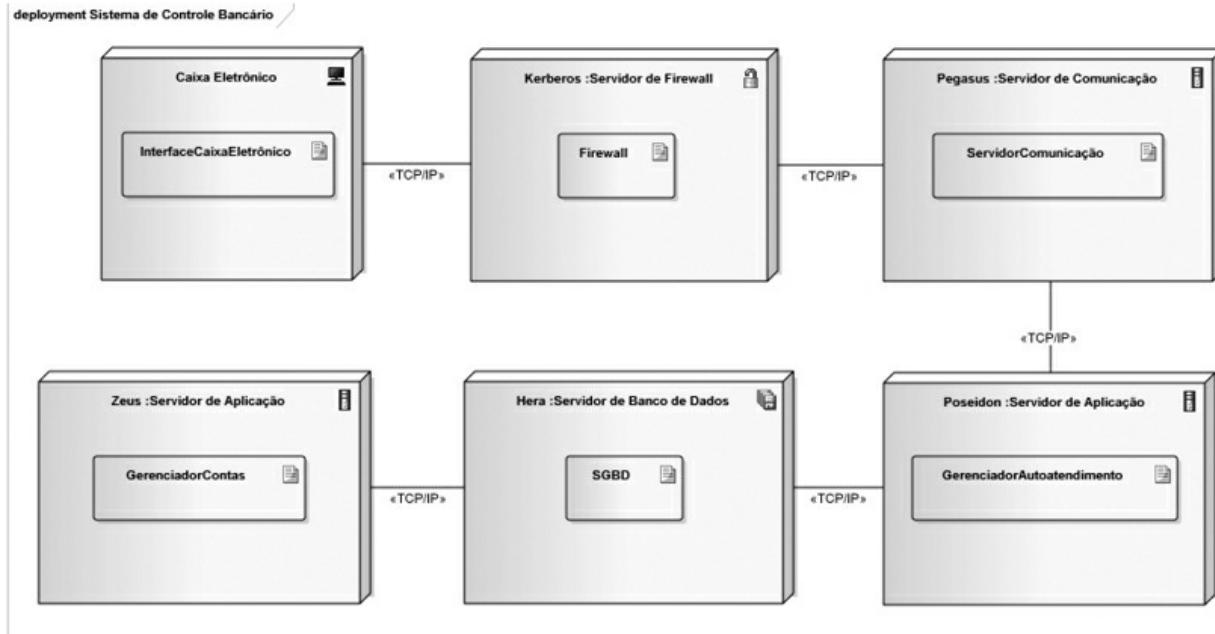


Figura 13.11 – Especificação de Implantação.

### 13.7 Exemplo de Diagrama de Implantação contendo Artefatos

Nesta seção, ilustraremos um diagrama de implantação contendo artefatos, como pode ser visto na figura 13.12.

Ao examinar essa figura, o leitor perceberá que tomamos o primeiro exemplo de diagrama de implantação e o complementamos com artefatos. Esses artefatos são manifestações dos componentes apresentados no diagrama de componentes do sistema de controle bancário estudado no capítulo 12.



*Figura 13.12 – Diagrama de Implantação Contendo Artefatos.*

Dessa forma, o nó **Caixa Eletrônico** suporta o módulo denominado **InterfaceCaixaEletrônico**, cuja função é repassar os eventos ocorridos sobre a interface para o sistema de controle bancário. Do mesmo modo, o nó **Kerberos** suporta um software de firewall, responsável por impedir que pessoas não autorizadas invadam o sistema. Seguindo a mesma lógica, o nó **Pegasus** suporta um software responsável por estabelecer a comunicação externa do sistema.

O servidor Poseidon é, por sua vez, responsável por suportar o módulo de software **GerenciadorAutoatendimento** e o servidor Zeus, por suportar o módulo de software **GerenciadorContas**. Finalmente, o nó **Hera** suporta um sistema gerenciador de banco de dados que armazena e recupera as informações necessárias ao sistema de controle bancário.

Aqui, só representamos a ligação física entre os nós, mas os artefatos podem ter ligações lógicas entre si, como conexões de banco de dados, por exemplo.

## 13.8 Nós Contendo Pacotes

Um nó pode suportar subsistemas ou mesmo sistemas inteiros. Esses sistemas podem ser representados como pacotes, como demonstra a figura 13.13.

Nesse exemplo existem dois nós, o **Servidor de Aplicação Departamento de Contabilidade** e o **Servidor de Aplicação Departamento de Vendas**. O primeiro suporta dois sistemas representados pelos pacotes **Sistema de Contabilidade** e **Sistema de Folha de Pagamento**, enquanto o segundo suporta o sistema representado pelo pacote **Sistema de Controle de Estoque**. Uma vez que esses sistemas estão integrados, há um relacionamento de dependência entre eles.

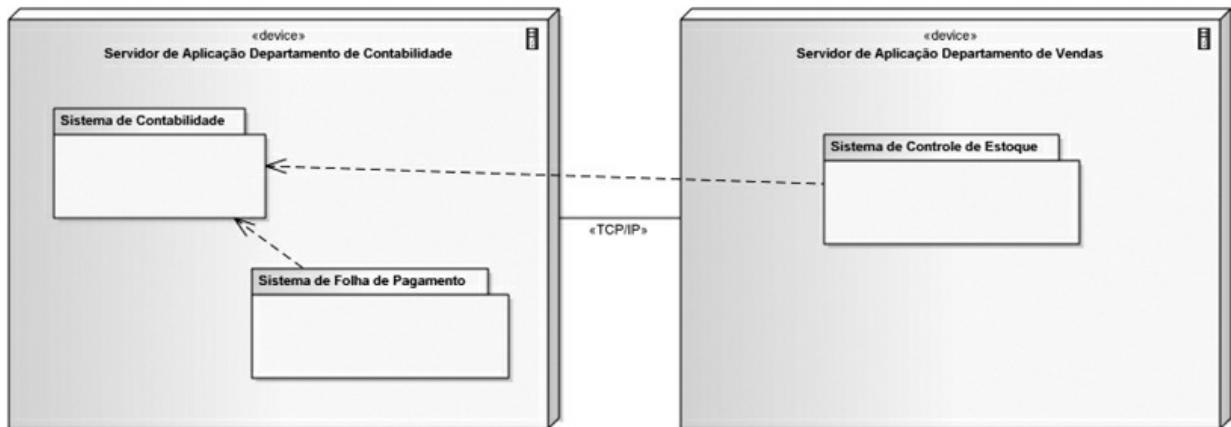


Figura 13.13 – Diagrama de Implantação Contendo Pacotes.

## 13.9 Exercícios Propostos

Aqui será finalizada a modelagem dos sistemas que temos exercitado desde o capítulo 3, referente ao diagrama de casos de uso. Porém, a maioria dos sistemas que temos modelado, por se tratar de sistemas relativamente simples, não necessita de um grande número de servidores, portanto consideramos válido exercitar somente o sistema de controle de leilão.

### **13.9.1 Sistema para Controle de Leilão Via Internet**

Desenvolva o diagrama de implantação para o sistema de controle de leilão via internet, sabendo que:

- Os participantes acessam o sistema de leilão de suas próprias máquinas

pessoais, de forma que é necessário representá-las.

- Uma vez que esse sistema é acessado externamente, é necessário estabelecer uma linha de segurança para impedir invasões.
- O sistema precisa suportar uma quantidade potencialmente grande de conexões. Assim, é necessário existir um servidor de comunicação.
- Toda a aplicação pode rodar em um único servidor, não havendo necessidade de dividi-la.
- No entanto, deve haver um servidor de banco de dados para armazenar as informações do sistema, bem como as recuperar quando solicitado. Embora este pudesse estar instalado no mesmo hardware, poderia ser útil se fosse executado em outra máquina.

## 13.10 Solução dos Exercícios

### 13.10.1 Sistema para Controle de Leilão Via Internet

A figura 13.14 apresenta a solução para esse exercício. Nela, representamos os artefatos que manifestam os componentes relativos ao mesmo sistema definidos no capítulo 12.

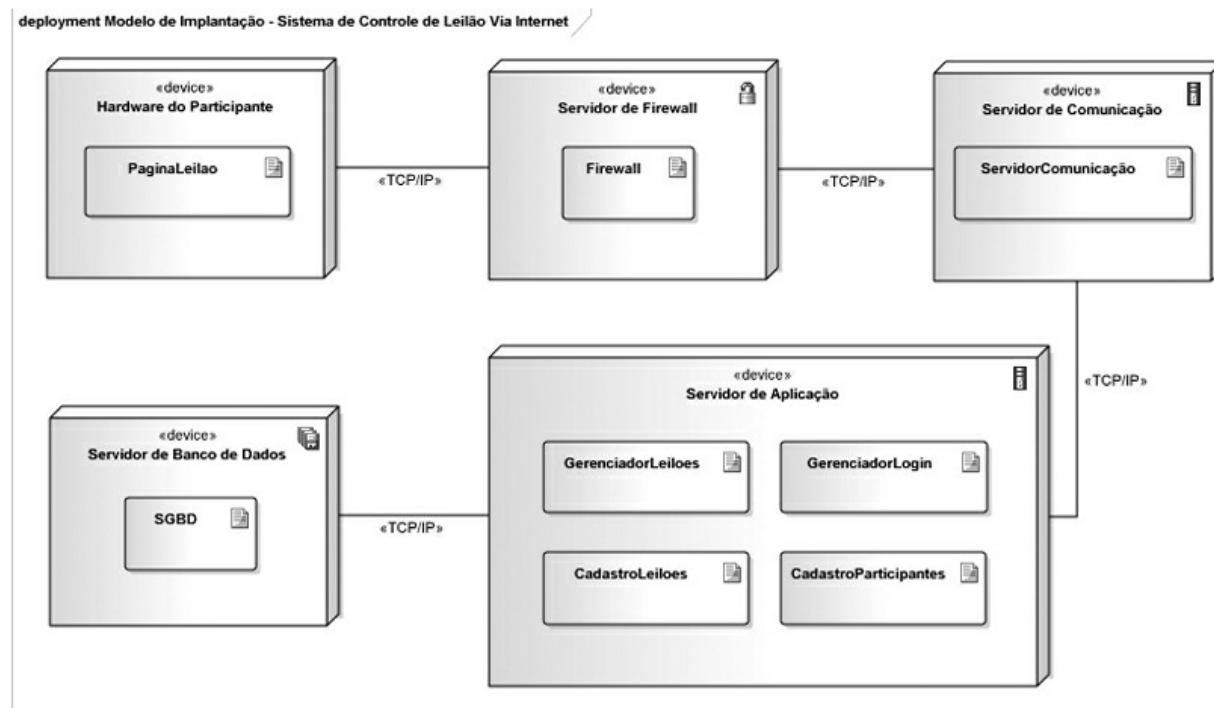


Figura 13.14 – Diagrama de Implantação – Sistema para Controle de Leilão

*Via Internet.*

Esse diagrama representa os seguintes nós:

- **Hardware do Participante** – Esse nó representa as máquinas das pessoas que se registraram para participar do leilão. O nó suporta o artefato que representa a página do leilão. Essa página será atualizada sempre que um novo item for anunciado ou um item receber um lance. Por meio dela, o participante pode fazer ofertas para um item, se assim o desejar.
- **Servidor de Firewall** – Esse nó representa o hardware que suporta o sistema de segurança da empresa. Nele é executado o módulo denominado **Firewall**, que, como seu nome indica, representa um software de firewall responsável por impedir invasões de usuários não autorizados.
- **Servidor de Comunicação** – Levando-se em consideração a possibilidade de existir um grande número de conexões externas relativas aos participantes de um leilão, considerou-se válido representar esse nó, que suporta um artefato representando um software de comunicação.
- **Servidor de Aplicação** – Esse nó é responsável por suportar o sistema de leilão propriamente dito. Nele são executados os módulos **GerenciadorLogin**, **CadastroParticipantes**, **GerenciadorLeiloes** e **CadastroLeiloes**, que são manifestações dos componentes desse sistema, já explicados no capítulo 12.
- **Servidor de Banco de Dados** – Finalmente, esse nó suporta um sistema gerenciador de banco de dados responsável pela gravação e recuperação das informações do sistema.

## CAPÍTULO 14

# Diagrama de Estrutura Composta

Este é um dos novos diagramas que passaram a existir a partir do lançamento da UML 2. O diagrama de estrutura composta pode ser utilizado para modelar colaborações. Uma colaboração descreve uma visão de um conjunto de entidades cooperativas interpretadas por instâncias que cooperam entre si para executar uma função específica. O termo estrutura desse diagrama refere-se a uma composição de elementos interconectados, representando instâncias de tempo de execução colaborando por meio de vínculos de comunicação para atingir algum objetivo comum. Esse diagrama também pode ser utilizado para descrever a estrutura interna de um classificador.

O diagrama de estrutura composta é semelhante ao diagrama de classes, porém esse último apresenta uma visão estática da estrutura de classes, enquanto o primeiro tenta expressar arquiteturas de tempo de execução, padrões de uso e os relacionamentos dos elementos participantes, o que nem sempre pode ser representado por diagramas estáticos.

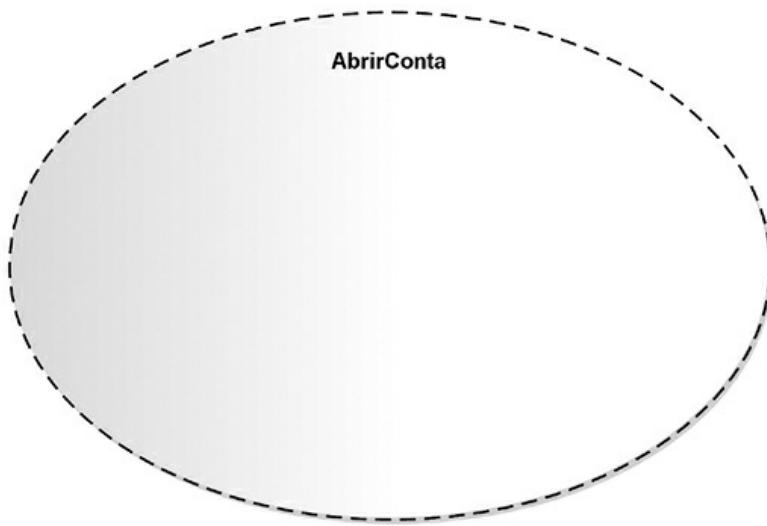
### 14.1 Colaborações

Uma colaboração define um conjunto de entidades cooperativas que serão interpretadas por instâncias que representarão o papel das entidades, bem como um conjunto de conectores que definem caminhos de comunicação entre as instâncias participantes. As entidades cooperativas são as propriedades da colaboração.

Colaborações são geralmente utilizadas para explicar como um conjunto de instâncias cooperando entre si realiza uma tarefa conjunta ou um conjunto de tarefas. O propósito primário de uma colaboração é explicar como um sistema funciona, portanto apresenta somente os aspectos extremamente necessários à explicação em questão, suprimindo quaisquer outros detalhes. Um mesmo objeto pode estar interpretando

simultaneamente diversos papéis em diferentes colaborações, mas cada colaboração deverá representar somente os aspectos do objeto relevantes ao seu propósito. Uma colaboração é representada por uma elipse tracejada contendo uma descrição, conforme pode ser observado na figura 14.1.

Essa colaboração representa a tarefa de abertura de conta do sistema de controle bancário que temos modelado no decorrer do livro.



*Figura 14.1 – Exemplo de Colaboração – Abertura de Conta.*

## 14.2 Papéis

Os “papéis” de uma colaboração são interpretados por instâncias que cooperam entre si para concluir uma tarefa. Os relacionamentos entre as instâncias considerados relevantes para a tarefa em questão são representados por meio da utilização de linhas ligando uma instância a outra, chamadas de conectores.

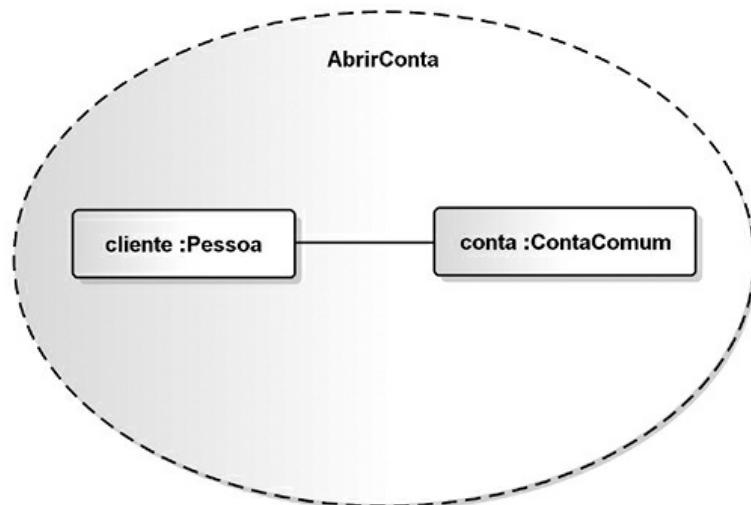
Os papéis de colaborações definem um uso de instâncias, enquanto os classificadores que definem esses papéis especificam todas as propriedades requeridas dessas instâncias. Assim, uma colaboração especifica quais propriedades uma instância deve ter habilitadas. Nem todas as características, nem todo o conteúdo das instâncias participantes nem todas as ligações dessas instâncias são sempre requeridos em uma colaboração particular, motivo pelo qual uma colaboração é muitas vezes descrita em termos de papéis definidos por interfaces, uma vez que estes

são descrições de um conjunto de propriedades (características observáveis externamente) fornecidas ou requeridas por uma instância. A figura 14.2 demonstra um exemplo de colaboração com sua estrutura interna, ou seja, papéis e conectores.

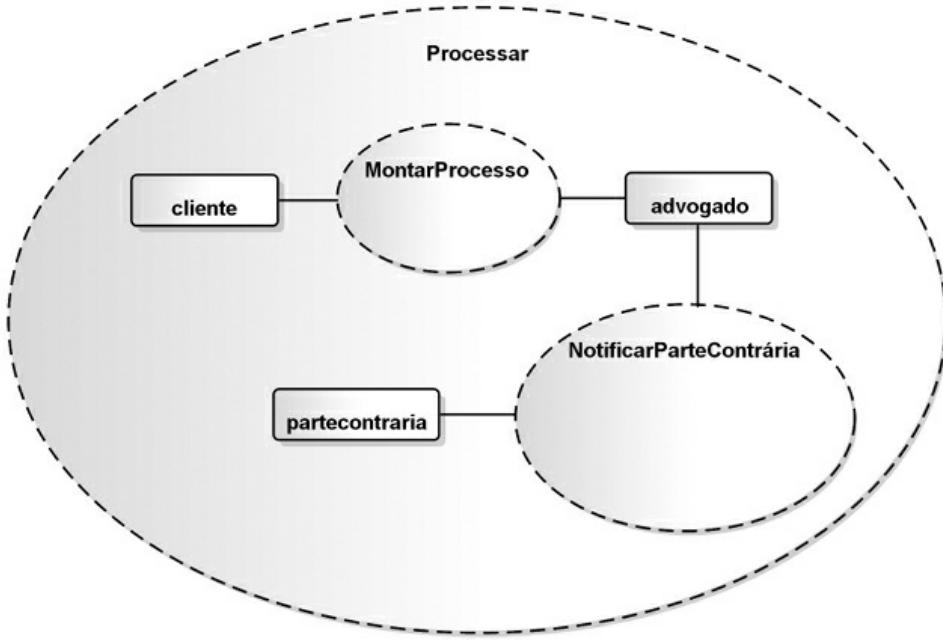
Nesse exemplo, uma instância da classe **Pessoa**, ao interpretar o papel de cliente, colabora (interage) com uma instância da classe **ContaComum**, que interpreta o papel de conta. O objetivo dessa interação é permitir que uma nova conta possa ser aberta. A definição do classificador da instância a que pertence o papel não é obrigatória, mas às vezes facilita a compreensão da colaboração.

Uma colaboração pode conter outras colaborações dentro de si, conforme é demonstrado pela figura 14.3.

Nesse exemplo, a colaboração **Processar** contém duas colaborações internas, as colaborações **MontarProcesso** e **NotificarParteContrária**. Na primeira, um cliente colabora com um advogado para montar um processo e, na segunda, um advogado interage com uma parte contrária para notificá-la do processo.



*Figura 14.2 – Colaboração Contendo Papéis e Conectores.*



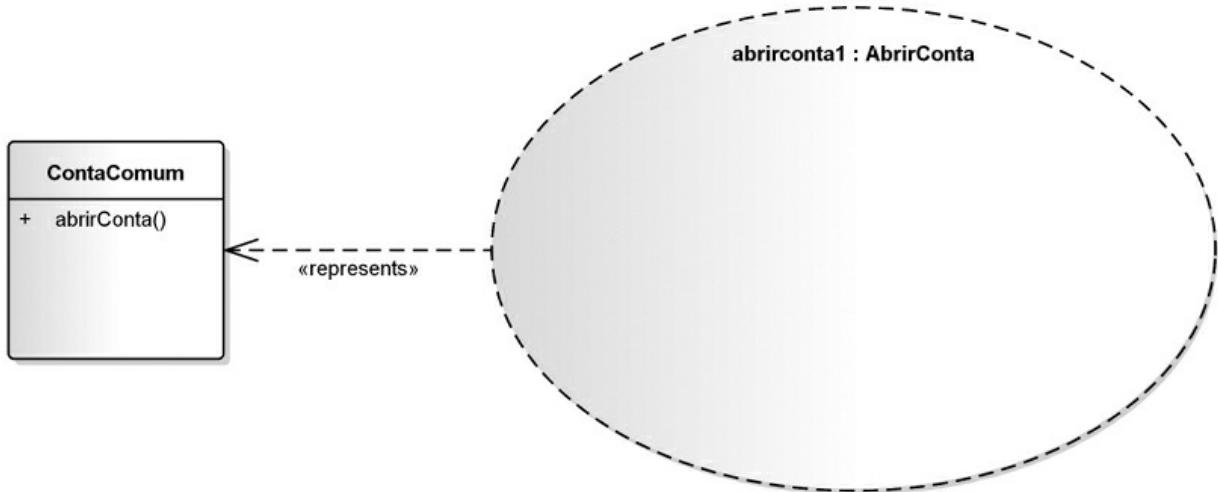
*Figura 14.3 – Colaboração Contendo Colaborações.*

### 14.3 Ocorrência de Colaboração

Uma ocorrência de colaboração representa a aplicação do padrão descrito por uma colaboração a uma determinada situação envolvendo classes ou instâncias que executam papéis específicos da colaboração.

A denominação de uma ocorrência de colaboração apresenta a mesma notação utilizada na denominação de um objeto (uma vez que uma ocorrência de colaboração é uma instância de uma colaboração), ou seja, o nome da ocorrência seguido de dois-pontos (:) e o nome da colaboração ou, caso não se queira dar um nome para a ocorrência, simplesmente dois-pontos (:) e o nome da colaboração.

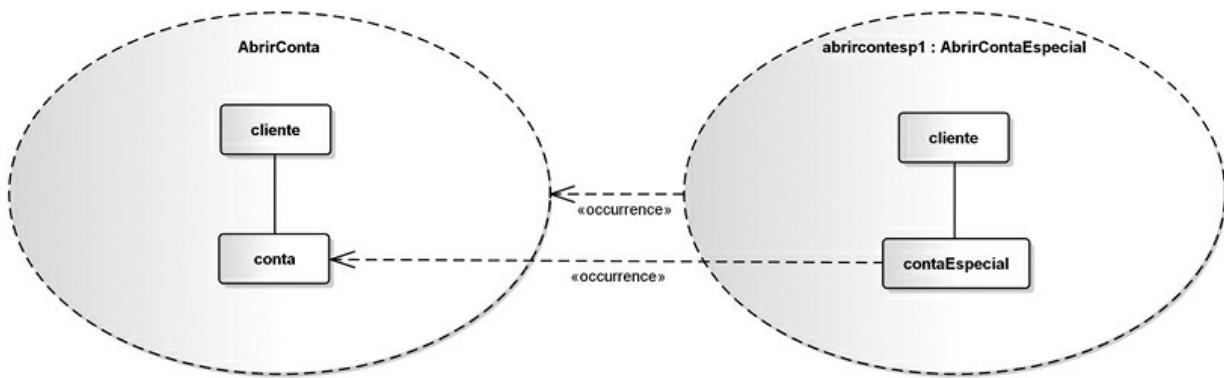
Uma ocorrência de colaboração pode ser relacionada a uma classe por meio de uma associação de dependência contendo o estereótipo <<represents>>, conforme demonstra a figura 14.4.



*Figura 14.4 – Ocorrência de Colaboração Relacionada a uma Classe.*

Esse exemplo determina que a ocorrência de colaboração **abrirconta1** está relacionada a um classificador, nesse caso a classe **ContaComum**. Essa colaboração representa a atividade de abertura de conta, na qual o método **abrirConta** definido na classe **ContaComum** desempenha uma função muito importante.

Pode-se também representar uma ocorrência de colaboração a partir de uma colaboração por meio do mesmo relacionamento de dependência, dessa vez contendo o estereótipo <>occurrence<>, conforme demonstra a figura 14.5.



*Figura 14.5 – Ocorrência de Colaboração.*

Nesse exemplo, percebemos que **AbrirContaEspecial** é uma ocorrência de colaboração da colaboração **AbrirConta** e que a instância **contaEspecial** da ocorrência **abrircontesp1** interpreta o papel de conta do elemento da colaboração **AbrirConta**.

## 14.4 Portas

Como foi introduzido no diagrama de classes, portas são características estruturais de um classificador e representam pontos de interação distintos entre um classificador e seu ambiente ou entre o classificador e suas partes internas. Portas são conectadas às propriedades de um classificador por meio de conectores, por meio dos quais requisições podem ser feitas para invocar as características comportamentais do classificador.

Uma porta é utilizada para representar os serviços que um classificador fornece a seu ambiente ou os serviços que requer deste. Portas são representadas por quadrados sobrepostos à borda de um classificador ou de suas partes internas. Uma porta pode ter um nome ou não. A figura 14.6 apresenta um exemplo de porta.

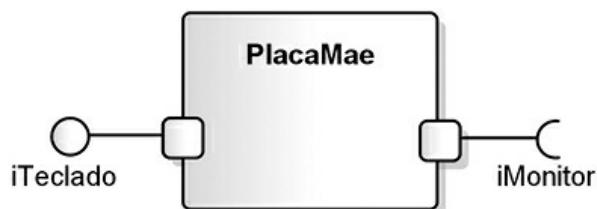


Figura 14.6 – Exemplo de Portas.

Nesse exemplo, a classe **PlacaMae** tem duas portas, uma para cada interface com seu ambiente externo. A classe está totalmente encapsulada, ou seja, não possuímos nenhum conhecimento a respeito de sua estrutura interna.

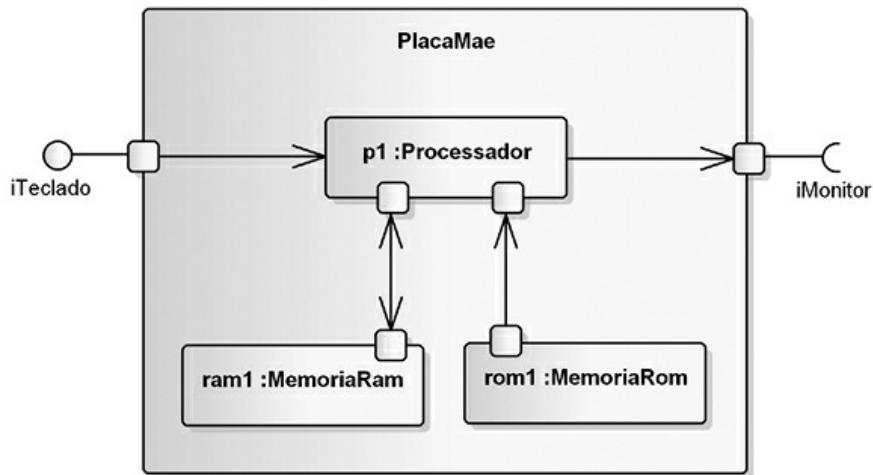
## 14.5 Propriedades e Partes

Uma propriedade representa um conjunto de instâncias internas que são possuídas por uma instância de um classificador contêiner. Quando uma instância de um classificador é criada, um conjunto de instâncias correspondente às suas propriedades pode ser criado também. Uma propriedade especifica que um conjunto de instâncias pode existir. Esse conjunto é um subconjunto do conjunto total de instâncias especificado pelo classificador que define a propriedade.

Uma Parte declara que uma instância desse classificador pode conter um conjunto de instâncias por composição, ou seja, essas instâncias não

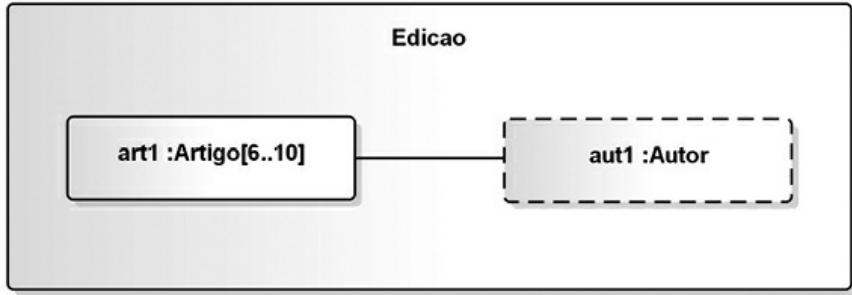
podem relacionar-se com outro objeto, pois pertencem exclusivamente à instância da classe contêiner e serão destruídas quando essa instância for destruída. A figura 14.7 apresenta um exemplo de Partes.

Nesse exemplo, apresentamos a estrutura interna de uma placa-mãe, composta nesse exemplo de partes que representam processador, memória RAM e memória ROM. As linhas que ligam as Partes entre si e ao ambiente externo por meio das portas são chamadas conectores e podem ser unidirecionais, se a seta aponta somente em uma direção; ou bidirecionais, se apontam para as duas direções, o que significa que podem receber e enviar informações. O conector que liga o processador à memória ROM é unidirecional, porque a memória é somente leitura, assim a informação é unidirecional.



*Figura 14.7 – Exemplo de Partes.*

Uma propriedade que especifique uma instância que não pertença por composição à instância do classificador contêiner é apresentada como um retângulo tracejado e dita referenciada. Partes podem conter a definição de sua multiplicidade, podendo esta ser fixa, quando se sabe o número exato de instâncias contidas na instância do classificador contêiner, ou determinar o número mínimo e o máximo da multiplicidade, conforme demonstra o exemplo da figura 14.8.



*Figura 14.8 – Exemplo de Parte com Multiplicidade e Propriedade Referenciada.*

Aqui, tomamos o exemplo da figura 4.11, onde uma edição relaciona-se por composição a diversos artigos, significando que uma instância da classe **Artigo** relaciona-se única e exclusivamente com uma instância da classe **Edicao**. Como se pode observar, a Parte que representa os artigos da edição apresenta uma multiplicidade que determina que devem existir, no mínimo, seis instâncias da classe **Artigo** dentro da instância da classe **Edicao** e, no máximo, dez. Ao mesmo tempo, existe uma propriedade referenciada, como podemos observar pelo tracejado de seu retângulo, uma vez que instâncias da classe **Autor** devem estar associadas a um artigo, mas não se relacionam à edição por composição.

## CAPÍTULO 15

# Diagrama de Tempo ou de Temporização

Esse diagrama apresenta algumas semelhanças com o diagrama de máquina de estados. No entanto, enfoca as mudanças de estado de um objeto ao longo do tempo. Esse diagrama terá pouca utilidade para modelar aplicações comerciais, contudo pode ser utilizado na modelagem de sistemas de tempo real; sistemas que utilizem recursos de multimídia/hipermídia, onde o tempo em que um objeto executa algo é muitas vezes importante; ou, ainda, para modelar processos de rede em que o sincronismo entre os elementos eventos é essencial em algumas situações. A figura 15.1 apresenta um exemplo de diagrama de tempo.

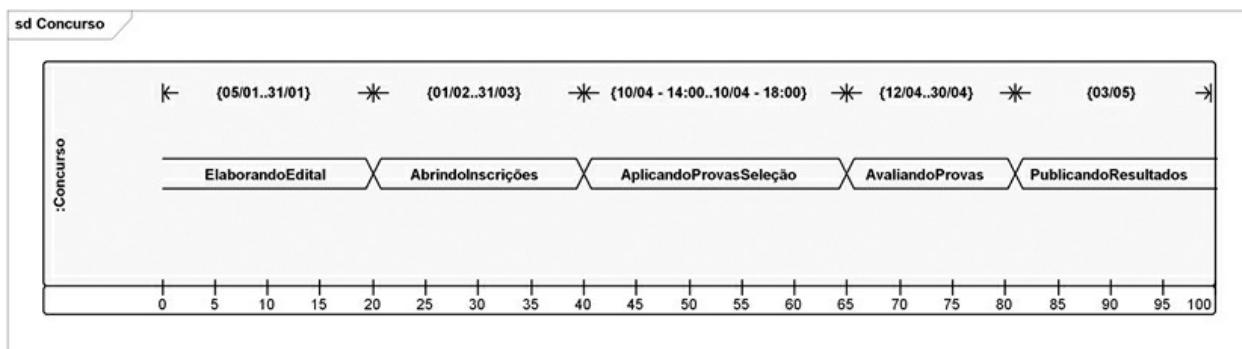


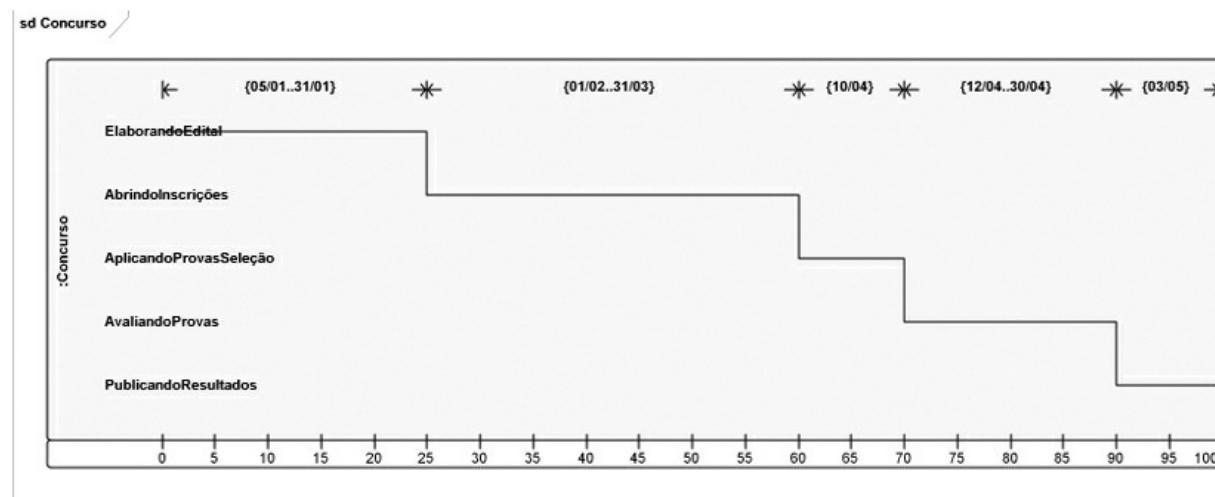
Figura 15.1– Diagrama de Tempo – Linha de Vida de Valor.

É importante destacar que o diagrama de tempo tem duas notações ou formas de representação: uma notação conhecida como concisa, mais simples, chamada linha de vida de valor, adotada no exemplo da figura 15.1, e uma notação considerada mais robusta, em que as etapas são apresentadas em forma semelhante a um gráfico, chamada linha de vida de estado. No diagrama de tempo, o termo linha de vida (lifeline) refere-se ao caminho percorrido por um objeto durante um determinado tempo.

No exemplo da figura 15.1, utilizamos a forma concisa, ou seja, a linha de vida de valor. Nesse exemplo, descrevemos as etapas pelas quais passa um concurso (o objeto da linha de vida), da elaboração de seu edital à

apresentação de seus resultados. Cada etapa ou estado do objeto da classe **Concurso** é apresentada por meio de um hexágono e o primeiro e último estados se encontram abertos. Acima de cada etapa, entre barras verticais, se encontram as restrições de duração que determinam o tempo em que transcorrem as etapas. No caso do estado **AbrindoInscrições**, o período vai de 1º de fevereiro a 31 de março.

A figura 15.2 apresenta o mesmo diagrama, dessa vez utilizando sua forma robusta, linha de vida de estado, onde as transições de estado são determinadas por mudanças em um gráfico, podendo estas conter descrições que determinam o evento que causou a mudança, se isso for considerado necessário.



*Figura 15.2 – Diagrama de Tempo – Linha de Vida de Estado.*

Um diagrama de tempo pode ter linhas de vida de múltiplos objetos utilizando a mesma notação ou notações diferentes.

## CAPÍTULO 16

# Diagrama de Perfil

O diagrama de perfil é um diagrama mais abstrato que os descritos anteriormente. Tal diagrama permite adaptar a UML a uma plataforma (como a J2EE ou .NET) ou domínio (como sistemas de tempo real, processos de negócios, sistemas multiagente, sistemas hipermídia etc.) para o qual a linguagem UML não foi projetada originalmente, portanto não possui recursos para modelar as características particulares da plataforma, tecnologia ou domínio em questão. Sendo assim, por meio da criação de perfis, é possível estender a linguagem, criando-se novas metaclasses e estereótipos que permitam a modelagem desses novos domínios.

Quando criamos um perfil, produzimos uma extensão conservadora da UML, sem alterar muito o metamodelo original, uma vez que adicionamos principalmente estereótipos, restrições e valores rotulados (ou etiquetados – tags) a ele, apenas adaptando um conceito já existente para que este possa ser aplicado a um domínio para o qual não foi projetado originalmente. Assim, o diagrama de perfis é um mecanismo leve de extensão da UML.

### 16.1 Conceitos Básicos: Modelos, Metamodelos e Metaclasses

Um modelo captura uma visão de um sistema físico. É uma abstração do sistema com um certo propósito, como descrever aspectos estruturais ou comportamentais do software. Esse propósito determina o que deve ser incluído no modelo e o que é irrelevante. Assim o modelo descreve completamente aqueles aspectos do sistema físico relevantes ao propósito do modelo, no nível apropriado de detalhe.

Um metamodelo é um modelo que define uma linguagem para expressar modelos. Seu papel é definir a semântica para modelar elementos dentro de um modelo que está sendo instanciado. Dessa forma, um modelo é uma instância de um metamodelo.

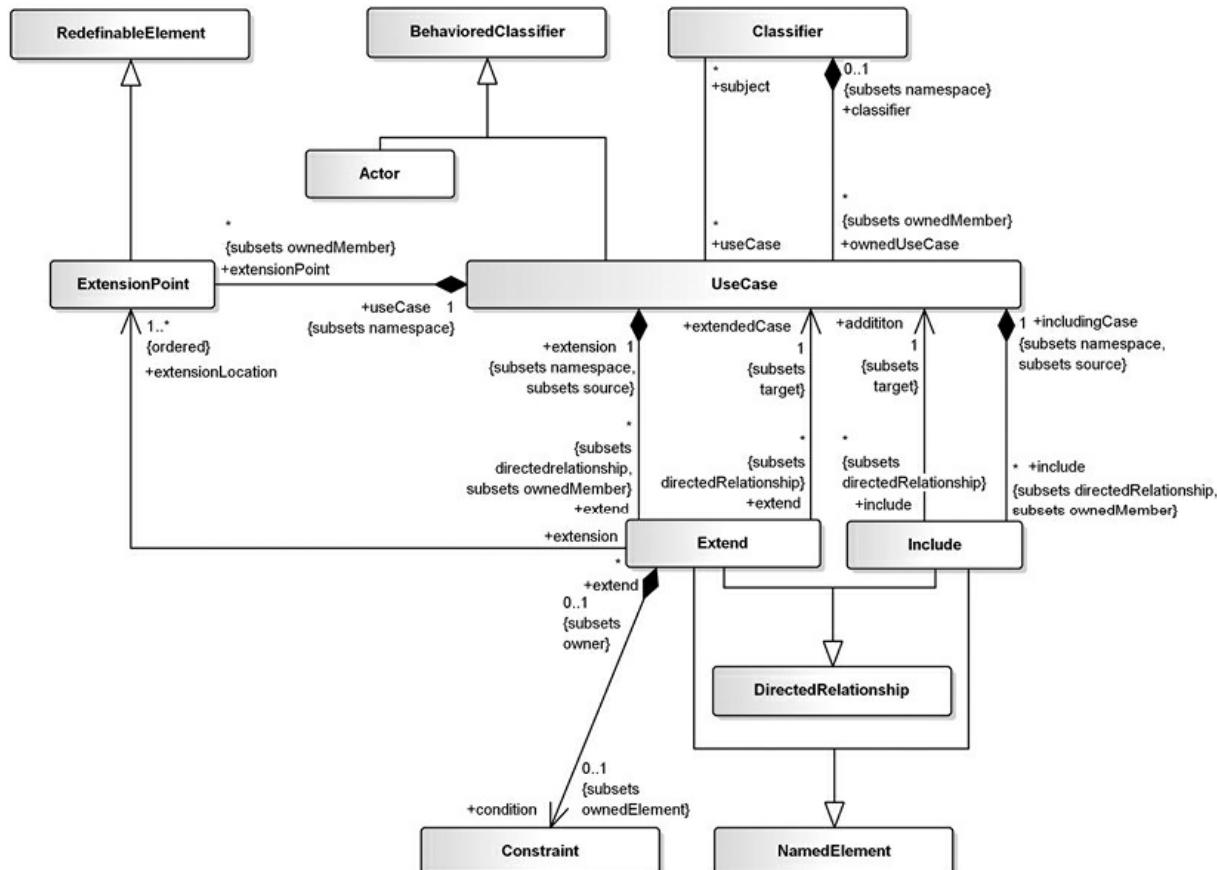
Uma metaclass é uma classe em um metamodelo cujas instâncias são elementos concretos da UML, como classes, atores ou casos de uso. Metaclasses são usadas para construir metamodelos. A figura 16.1 apresenta um exemplo de metaclass.



*Figura 16.1 – Metaclass Actor.*

Esse exemplo representa a metaclass Actor. Essa metaclass representa o conceito de Ator explicado no capítulo 3 sobre Casos de Uso. Assim, um ator representado em um modelo nada mais é que uma instância da metaclass **Actor**. O estereótipo **metaclass** é opcional, mas ajuda a destacar que o componente em questão é uma metaclass.

Para ilustrar um exemplo de metamodelo, na figura 16.2 apresentamos o metamodelo utilizado para definir os conceitos empregados para a modelagem de casos de uso.



*Figura 16.2 – Exemplo de Metamodelo.*

Ao observarmos essa figura, percebemos que as metaclasses centrais **Actor**, **UseCase**, **ExtensionPoint**, **Extend** e **Include** correspondem aos conceitos explicados no capítulo 3 sobre atores, casos de uso, pontos de extensão e associações de extensão e inclusão. Dessa forma, ao modelarmos esses conceitos em um modelo de casos de uso, estamos representando instâncias das metaclasses definidas nesse metamodelo. A seguir, explicaremos sucintamente as outras metaclasses contidas nesse metamodelo.

### 16.1.1 Metaclasses Classifier

Um classificador (classifier) é uma metaclasses abstrata que representa uma classificação de instâncias. Ele descreve um conjunto de instâncias com características em comum. É um espaço de nome cujos membros podem incluir características. Um classificador é um tipo e pode ter generalizações, tornando possível definir relacionamentos de generalização para outros

classificadores. Essa metaclasses pode especificar uma hierarquia de generalização por meio de referência aos seus classificadores gerais. É um elemento redefinível, o que significa que é possível redefinir classificadores aninhados.

Um classificador é um mecanismo que descreve características comportamentais e estruturais, porém as instâncias de um classificador não são objetos como as instâncias de uma classe, mas elementos UML concretos (que incluem classes). Não pode ser utilizado em um modelo, somente suas instâncias o podem sê-lo. Classificadores são utilizados apenas em metamodelos.

### **16.1.2 Metaclasses BehavioredClassifier**

A metaclasses **BehavioredClassifier** é um classificador com especificações de comportamento definidas em seu espaço de nomes (namespace). É a partir da metaclasses **BehavioredClassifier** que são derivadas as metaclasses **Actor** e **UseCase**.

### **16.1.3 Metaclasses NameSpace**

A metaclasses **NameSpace** (espaço de nome) é uma metaclasses abstrata que representa um elemento em um modelo que contém um conjunto de elementos nomeados que podem ser identificados pelo nome.

### **16.1.4 Metaclasses NamedElement**

Um **NamedElement** (elemento nomeado) é uma metaclasses abstrata que representa elementos que podem ter um nome. O nome é usado para identificar o elemento nomeado dentro do espaço de nome no qual está definido. A partir desta metaclasses foram especializadas as metaclasses **Extend** e **Include**.

### **16.1.5 Metaclasses DirectedRelationship**

Um **DirectedRelationship** (relacionamento direcionado) é uma metaclasses abstrata que representa um relacionamento entre uma coleção de elementos de modelo fonte e uma coleção de elementos de modelo destino. Essa metaclasses também foi utilizada para especializar as metaclasses **Extend** e **Include**, denotando uma herança múltipla.

### **16.1.6 Metaclasses Constraint**

Uma **Constraint** (restrição) é uma metaclasses abstrata que representa uma condição ou restrição expressada em texto em linguagem natural ou linguagem de programação para o propósito de declarar algumas das semânticas de um elemento. Constraint é uma condição que restringe a extensão do elemento associado além do que é imposto por outras construções de linguagem aplicadas naquele elemento. Observe que existe uma associação de composição entre essa metaclasses e a metaclasses **Extend**, uma vez que esta é uma associação que necessita que uma condição seja satisfeita e tal condição complementa a informação da extensão.

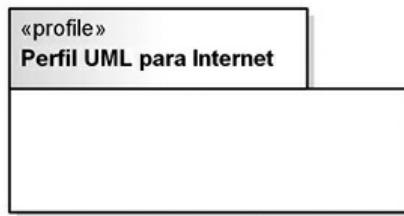
### **16.1.7 Metaclasses RedefinableElement**

Um **RedefinableElement** (elemento redefinível) é uma metaclasses abstrata que representa um elemento nomeado que pode ser redefinido no contexto de uma generalização. A partir dessa metaclasses foi derivada a metaclasses **ExtensionPoint**.

## **16.2 Criação de Perfis**

Um Perfil é um tipo de Pacote que estende um metamodelo de referência. A construção de extensão primária é o **Stereotype** (estereótipo), que é definido como parte dos Perfis. Um perfil introduz diversas restrições sobre a metamodelagem ordinária por meio do uso das metaclasses definidas neste pacote.

Como foi dito anteriormente, um perfil é um mecanismo leve e conservador de extensão da UML, dessa forma não é permitido a um perfil modificar um metamodelo. Por conseguinte, não é permitido a um perfil retirar quaisquer restrições aplicadas a um metamodelo, porém é possível adicionar novas restrições específicas ao perfil em questão. Assim, um perfil é uma adaptação, uma customização de um metamodelo original que deve estar contida em um pacote separado. A figura 16.3 apresenta um exemplo de um pacote contendo um novo perfil.



*Figura 16.3 – Exemplo de Pacote Contendo um Novo Perfil.*

Esse pacote contém as metaclasses que definem novos conceitos necessários à modelagem de sistemas para Internet. O estereótipo <<profile>> deixa claro que o pacote contém um novo perfil.

## 16.3 Estereótipos

Um estereótipo é um tipo limitado de metaclass que não pode ser usada diretamente, mas deve sempre ser usada em conjunção com uma das metaclasses que ele estende. Cada estereótipo pode estender uma ou mais classes por meio de extensões como parte de um perfil. Similarmente, uma metaclass pode ser estendida por um ou mais estereótipos.

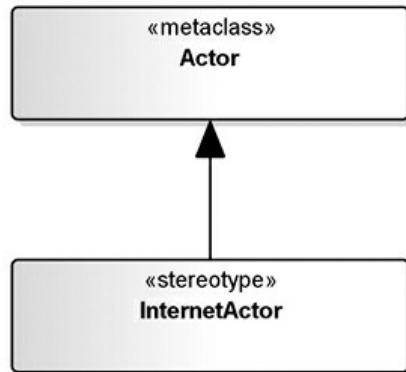
Dentro do contexto de modelo, um estereótipo permite atribuir novas características a um elemento UML. Já dentro do contexto de metamodelos e perfis, um estereótipo define como uma metaclass pode ser estendida, atribuindo-lhe novas características e/ou restrições, sem, no entanto, retirar quaisquer características ou restrições já possuídas pela metaclass em questão. Ao longo deste livro, foram demonstrados alguns exemplos de estereótipos, como os estereótipos boundary, control e entity apresentados no capítulo 4, sobre o diagrama de classes.

No momento em que uma nova metaclass é derivada de uma metaclass anterior e utiliza o estereótipo denominado “**stereotype**”, essa nova metaclass permite modelar instâncias da metaclass original contendo um estereótipo com o nome da nova metaclass, ou seja, a partir deste momento, pode-se modelar elementos estereotipados em um modelo com características que os diferenciem de alguma forma dos elementos originais.

## 16.4 Extensão

No diagrama de perfis, uma extensão é um tipo de associação usada para

indicar que as propriedades de uma metaclassse são estendidas por meio de um estereótipo. A figura 16.4 apresenta um exemplo de aplicação da associação de extensão na criação de um estereótipo a partir de uma metaclassse.



*Figura 16.4 – Exemplo Simples de Perfil – Estereótipo InternetActor.*

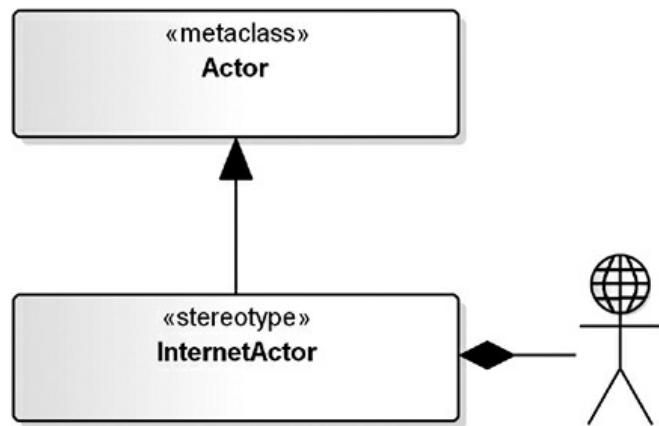
Nessa figura, criamos a metaclassse **InternetActor** especializada a partir da metaclassse **Actor**, aplicando restrições que declarem que um **InternetActor** deve representar somente atores que accessem o sistema remotamente através da Internet. Observe que aplicamos o estereótipo **stereotype** nessa metaclassse, o que significa que deverá ser utilizada associada à metaclassse **Actor** como um estereótipo.

Embora este seja um exemplo bastante simples e, em geral, perfis englobem a criação de vários estereótipos para um determinado domínio, não deixa de ser um exemplo de perfil no qual a metaclassse **Actor** foi adaptada para o domínio da Internet. Assim, se aplicássemos esse estereótipo em atores em um diagrama de casos de uso, estes apresentariam o estereótipo de texto “**InternetActor**”, conforme mostra a figura 16.5.



*Figura 16.5 – Ator com o Estereótipo InternetActor.*

É possível também criar estereótipos gráficos. Para isso, é preciso associar uma imagem ao estereótipo por meio de uma associação de composição, que, como foi explicado no diagrama de classes, indica uma complementação das informações de uma classe. A figura 16.6 apresenta um exemplo de como associar uma imagem (ou ícone) a um estereótipo.



*Figura 16.6 – Estereótipo Associado a um Ícone por meio de uma Associação de Composição.*

A partir do momento que associamos um ícone a um estereótipo, quando este for aplicado, o desenho-padrão da metaclasses que ele estende será substituído pela imagem associada ao estereótipo, conforme demonstra a figura 16.7.



*Figura 16.7 – Ator com o Estereótipo Gráfico InternetActor.*

## CAPÍTULO 17

# Estudo de Caso – Sistema de Pizzaria Online – PizzaNet

Neste capítulo, modelaremos um sistema de pizzaria online fictício cujos pedidos poderão ser feitos pela internet. Deve ficar claro, porém, que os diagramas apresentados ao longo deste capítulo não estão atrelados a nenhuma linguagem de programação específica e alguns dos processos aqui modelados talvez venham a ser simplificados ou até alguns métodos poderão ser considerados desnecessários, sendo possível ser incluídos dentro das instruções de outros métodos. Nosso objetivo foi produzir diagramas de fácil compreensão que possam ser entendidos por qualquer leitor, sem ser necessário conhecer uma determinada linguagem de programação para entendê-los.

O sistema será modelado por meio de quase todos os diagramas da UML, excetuando-se o diagrama de estrutura composta, que consideramos desnecessário modelar, além do diagrama de tempo, que não se aplica a essa situação, e o diagrama de perfil, cujo objetivo é adaptar a UML a outros domínios para os quais não foi projetada anteriormente e, portanto, possui um escopo diferente do software aqui modelado.

O leitor que desejar exercitar seus conhecimentos poderá tentar modelar o sistema a partir do enunciado do problema proposto, descrito na seção 17.1 e, depois, corrigi-lo com base na solução apresentada ao longo deste capítulo.

### 17.1 Descrição do Problema

Uma empresa necessita de um sistema de pizzaria online, por meio do qual seus clientes possam solicitar pizzas pela Internet. Para modelar esse sistema, devemos levar em consideração os seguintes requisitos apresentados pela empresa:

- A tela inicial do site da pizzaria deve apresentar duas divisões verticais. Na divisão da esquerda, devem aparecer os ícones intitulados **Logar**, **Pizzas**, **Bebidas**, **Visualizar Pedido**, **Sabores Mais Pedidos**, **Pedidos Anteriores**, **Concluir Pedido** e **Opinar**.
- Já a divisão da direita carregará o módulo escolhido pelo usuário quando este selecionar um dos botões já descritos. Por padrão, deverá ser carregado o módulo para escolha de pizzas, que poderá ser chamado também por meio do botão **Pizzas**. Nesse módulo, a tela deverá ser subdividida em duas divisões horizontais: na superior, os clientes poderão escolher entre três ícones que representam os tamanhos que podem ser escolhidos para a pizza (pequena – com 4 pedaços, média – com 6 pedaços e grande – com 8 pedaços). Ao selecionar o tamanho da pizza, o sistema deve permitir que o cliente escolha, na segunda divisão horizontal, tantos sabores quanto os permitidos pelo tamanho da pizza (1, 2 ou 4). Essa divisão contém ainda um ícone para adicionar a pizza escolhida ao pedido, permitindo que o cliente escolha outras pizzas depois. Assim, ao terminar de escolher os tamanhos da pizza, basta que o cliente pressione o botão **Adicionar** para que o sistema adicione a pizza ao pedido e permita ao cliente escolher outra pizza, caso deseje. O valor da pizza é calculado com base no preço do sabor mais caro multiplicado pelo número de pedaços da pizza.
- O botão **Logar** permitirá que o cliente se autentique no sistema, o que é necessário para que este possa concluir o pedido. Esse módulo solicita ao cliente que informe seu **nome-login** e sua senha para logá-lo. Caso o cliente não esteja cadastrado, esse módulo deverá permitir que o cliente solicite a execução do módulo **Autorregisterar**, onde poderá se cadastrar.
- O botão **Bebidas** deverá apresentar um formulário um pouco semelhante ao módulo de escolha de pizzas, com uma divisão horizontal na qual, na parte superior, deverão ser apresentados os tipos de bebida oferecida (suco, refrigerante e cerveja) e, após a escolha do tipo desejado, o sistema deverá apresentar, na segunda divisão, todas as bebidas do tipo escolhido disponíveis, permitindo ao cliente selecionar e adicionar ao pedido quantas bebidas quiser.
- O botão **Visualizar Pedido** deverá apresentar todos os itens escolhidos pelo cliente (pizzas e bebidas) até o momento, bem como o valor total

do pedido, permitindo que ele exclua algum item, se assim o desejar.

- O botão **Fechar Pedido** deverá permitir que o cliente conclua o pedido. Nesse processo, o cliente deverá obrigatoriamente se logar, caso ainda não o tenha feito, podendo alterar seus dados, se desejar, ou se cadastrar no sistema, caso esta seja a primeira vez em que realiza um pedido na PizzaNet. Após essa verificação, o sistema deverá executar o módulo **Visualizar Pedido** para apresentar os itens escolhidos pelo cliente. Em seguida, o sistema deverá apresentar ainda o endereço de entrega (que poderá ser modificado), o tempo de preparo (levando em consideração o item mais demorado) e o tempo médio de entrega e solicitar a confirmação. Caso o cliente confirme o pedido, o sistema o marcará como concluído e dará baixa no estoque em todos os itens necessários à execução do pedido, incluindo os ingredientes necessários à produção de cada pizza.
- O botão **Sabores Mais Pedidos** deverá apresentar os sabores de pizzas mais solicitadas na PizzaNet como sugestão ao cliente.
- O botão **Pedidos Anteriores** deverá apresentar uma lista de todos os pedidos já solicitados pelo cliente, permitindo que este solicite novamente um pedido já realizado, podendo realizar modificações no novo pedido, se assim o desejar.
- O botão **Opinar** só poderá ser utilizado caso o cliente tenha se autenticado. Essa opção deverá permitir que o cliente dê a sua opinião sobre o atendimento da pizzaria, referindo-se tanto à qualidade da pizza como à da entrega. Dessa forma, é necessário manter informações referentes a qual funcionário fez a pizza e qual a entregou.
- Durante o registro do pedido, o sistema deverá salvar também todos os seus itens, ou seja, as pizzas e bebidas solicitadas.
- Um cliente poderá realizar muitos pedidos, no entanto um pedido será exclusivo para um único cliente.
- Cada pedido deverá armazenar, entre outras informações, a data e a hora em que o pedido foi feito e a hora provável de sua entrega, calculada de acordo com o tempo de preparo da pizza mais demorada e o tempo médio de entrega na cidade.
- Um pedido deverá ser composto de, no mínimo, um item, podendo

conter muitos itens. Cada item é relativo a uma pizza ou bebida, independentemente da quantidade.

- Há uma grande quantidade de sabores e cada um tem um valor específico. No entanto, caso o cliente opte por pedir uma pizza com mais de um sabor, o valor desta será calculado com base no sabor mais caro multiplicado pelo número de pedaços da pizza.
- Cada pizza consome diversas quantidades de diversos itens de estoque. Sempre que uma determinada pizza é produzida, essas quantidades devem ser diminuídas de seus respectivos itens no estoque.
- Quando qualquer pedido for entregue, deverão ser conferidos os produtos e as quantidades solicitados.
- Cada pedido é produzido por um funcionário específico, podendo ser entregue por ele ou por outro funcionário. Deverá haver uma opção na página que permita aos clientes darem sua opinião sobre a qualidade do pedido e a rapidez da entrega, bem como outras sugestões. Um cliente só poderá opinar sobre pedidos feitos por ele próprio, ou seja, terá que estar logado. Caso haja reclamações, a empresa precisa saber quem deve cobrar.
- Sempre que um item de estoque estiver com a quantidade abaixo ou perto da quantidade mínima, deverá ser montado um pedido para o fornecedor que vende esse tipo de produto.
- A empresa necessita de relatórios que permitam saber quais sabores de pizza são mais pedidos e em que épocas do ano ou dias da semana, para ter uma expectativa de consumo, oferecer promoções e antecipar compras de produtos de estoque.
- A empresa necessita também de relatórios que informem quais clientes consomem mais e em que bairros eles se encontram.
- A empresa precisa saber ainda qual o consumo médio diário de cada produto, para ter uma base das quantidades a serem pedidas de cada item específico. Por exemplo, dois itens de uma mesma pizza podem consumir quantidades de itens de estoque diferentes.

A partir dessa lista de requisitos, elaboramos dois formulários, à guisa de protótipo, de caráter meramente ilustrativo, para o leitor ter uma ideia de como será a interface do sistema. As figuras 17.1 e 17.2 ilustram os

formulários de escolha de pizzas e de bebidas, respectivamente.

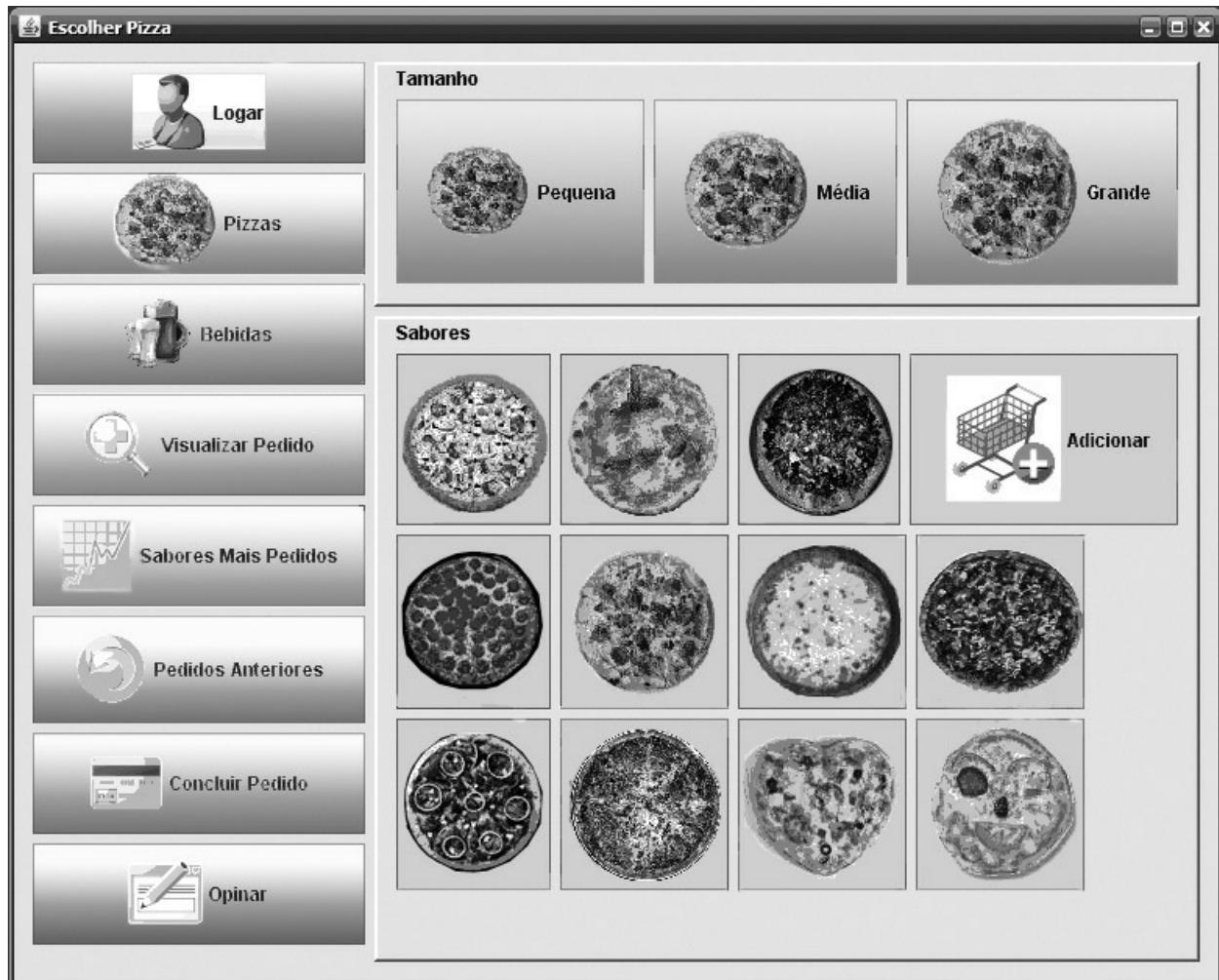


Figura 17.1 – Formulário para Escolha de Pizzas do Sistema PizzaNet.

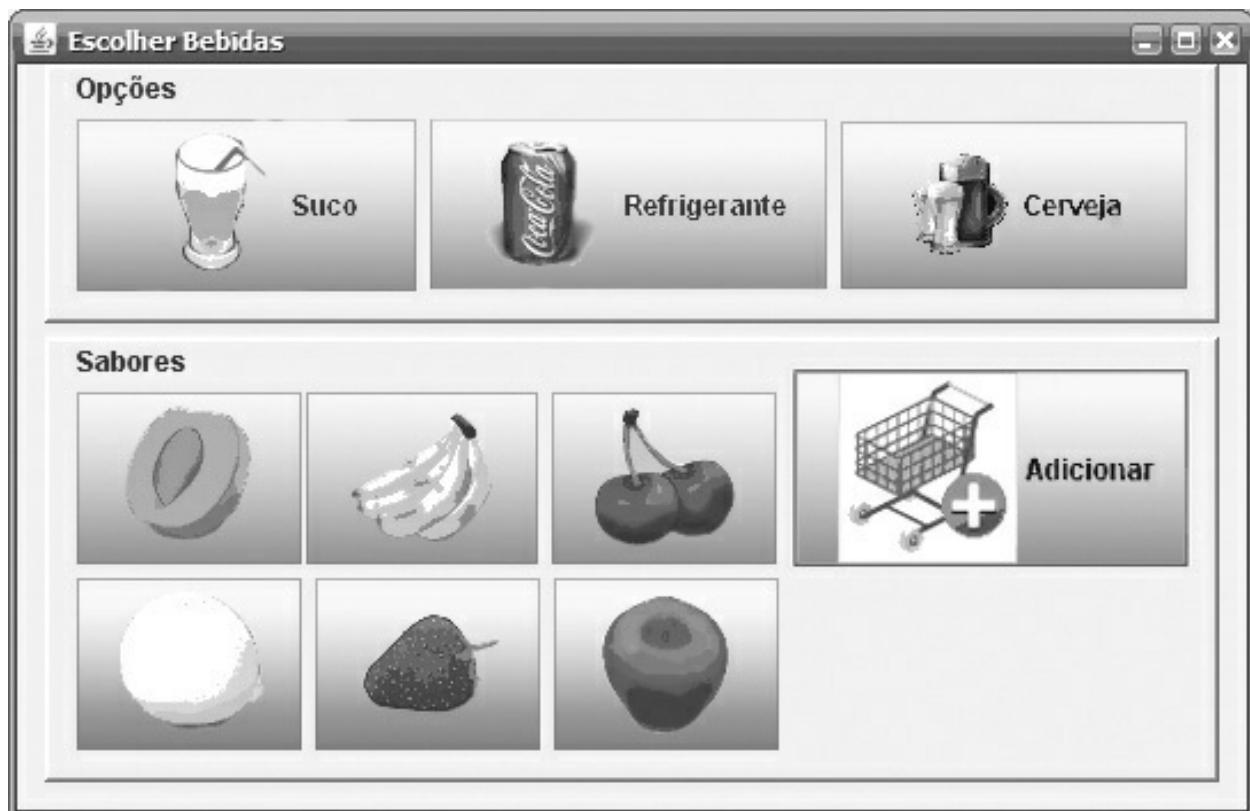


Figura 17.2 – Formulário para Escolha de Bebidas do Sistema PizzaNet.

## 17.2 Solução do Problema

Nas seções seguintes, modelaremos uma solução para o problema apresentado na seção 17.1, por meio dos diversos diagramas da UML já ensinados, sempre que estes se mostrarem úteis à modelagem desse sistema. Os diagramas serão explicados de maneira a esclarecer o que cada um deles representa.

### 17.2.1 Diagramas de Casos de Uso

Nesse estudo de caso, decidimos modelar o sistema por meio de dois diagramas de caso de uso independentes pelo fato de existirem dois grupos de funcionalidades diferentes, representados pelos serviços que podem ser utilizados externamente pelos clientes via Internet e os serviços internos que só podem ser manipulados pelos funcionários da empresa. Dessa forma, decidimos dividir o sistema PizzaNet em dois subsistemas: o subsistema de vendas e o subsistema administrativo.

*Primeiro Diagrama – Subsistema de Vendas*

Na figura 17.3, apresentamos o primeiro diagrama de casos de uso ilustrando os atores e funcionalidades do subsistema de vendas, que pode ser utilizado pelos clientes da PizzaNet.

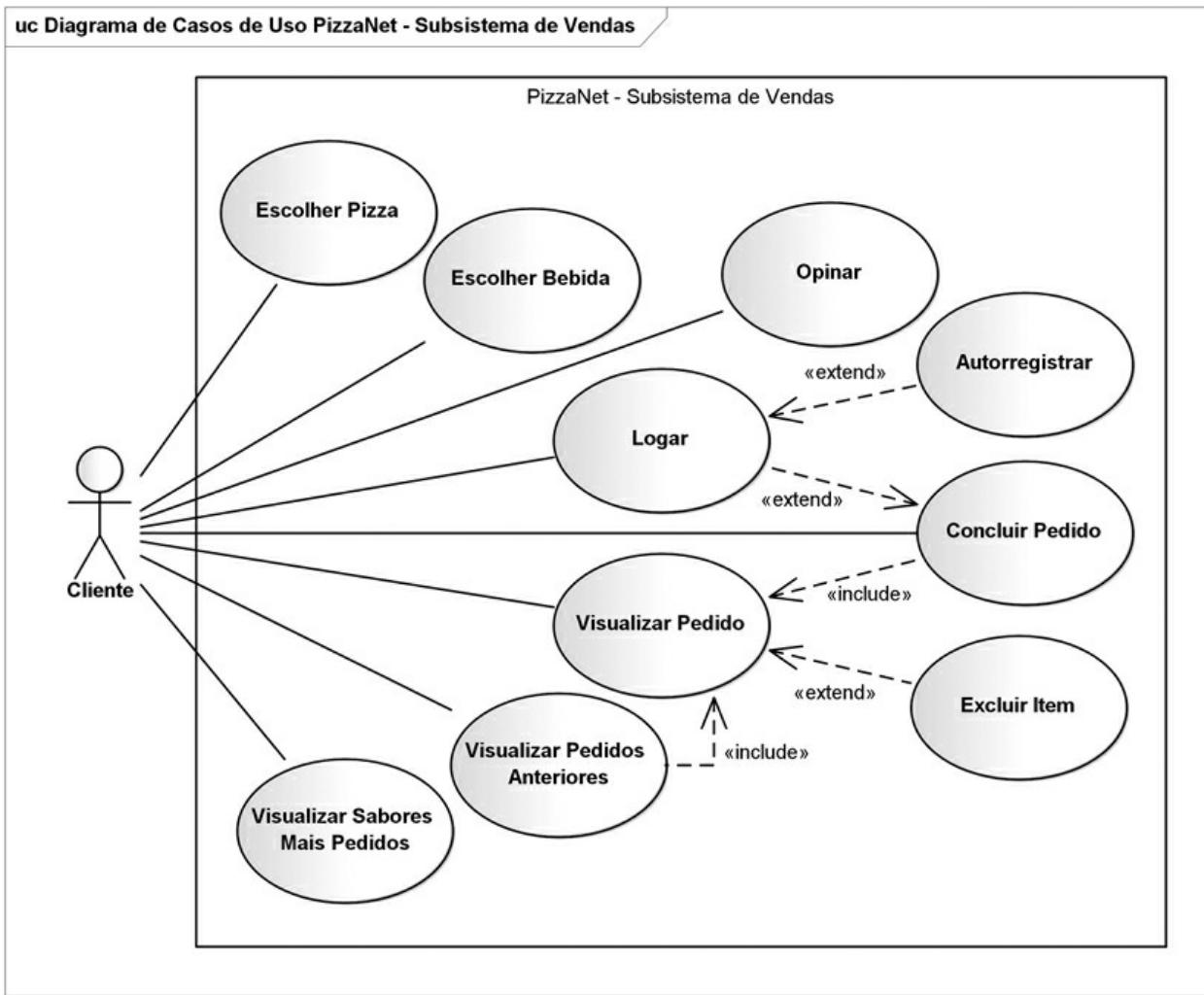


Figura 17.3 – Diagrama de Casos de Uso – Subsistema de Vendas.

O diagrama de casos de uso apresentado nessa figura contém apenas um ator denominado **Cliente**. Esse ator representa os usuários externos, ou seja, os clientes propriamente ditos, que, por meio da internet, acessam a página da PizzaNet e realizam pedidos contendo pizzas e/ou bebidas. O diagrama representa, ainda, as seguintes funcionalidades oferecidas ao ator **Cliente**:

- **Escolher Pizza** – Este caso de uso representa o processo por meio do qual um cliente pode escolher uma pizza. Ao ter essa opção selecionada, o sistema deverá apresentar um formulário contendo duas divisões: a

primeira apresentará os tamanhos de pizzas que a pizzaria oferece (pequeno, médio e grande) e a segunda, os sabores disponíveis. Por meio desse formulário, primeiramente o cliente deverá escolher o tamanho da pizza que deseja. Após o tamanho escolhido, o cliente poderá escolher tantos sabores quanto os permitidos pelo tamanho. Quando tiver concluído a escolha da pizza, bastará selecionar o botão **Adicionar** para adicionar a pizza ao pedido. O cliente poderá repetir esse processo quantas vezes quiser.

- **Escolher Bebida** – Representa o processo pelo qual um cliente escolhe uma bebida, sendo um pouco semelhante ao processo de escolha de pizza. O processo é disparado quando o cliente seleciona (clica sobre) o botão **Bebidas**, o que faz o sistema apresentar um formulário contendo duas divisões. Na primeira divisão são apresentados os tipos de bebidas oferecidos pela PizzaNet (suco, refrigerante ou cerveja) e na segunda são mostradas as bebidas pertencentes ao tipo escolhido. Por exemplo, se o cliente escolher o tipo de bebida suco, o sistema apresentará todos os sabores de suco disponíveis. Assim, primeiramente o cliente deve escolher o tipo de bebida que deseja e, em resposta, o sistema apresentará as bebidas desse tipo disponíveis, permitindo que ele escolha uma delas. Depois de ter escolhido o tipo e a bebida desejada, o cliente deve selecionar o botão **Adicionar** para acrescentar a bebida a seu pedido. Esse processo poderá igualmente ser repetido tantas vezes quanto ele desejar.
- **Logar** – Por meio dessa opção, o cliente poderá autenticar-se na PizzaNet, informando seu **nome-login**, bem como sua senha. Se tais dados estiverem corretos, o sistema logará o cliente. Isso é necessário para que o cliente possa emitir opiniões sobre pedidos anteriores, concluir um pedido ou visualizar pedidos feitos anteriormente por ele. No entanto, caso o cliente não esteja registrado no sistema, será preciso primeiro executar o caso de uso **Autorregisterar**, que será explicado a seguir.
- **Autorregisterar** – Está associado ao caso de uso **Logar** por meio de uma associação de extensão, o que significa que esse caso de uso poderá ser chamado a partir do caso de uso **Logar** quando uma condição for satisfeita. Aqui, a condição é que o cliente não esteja registrado. Quando

o cliente escolher se registrar, o sistema apresentará um formulário solicitando os dados do cliente, como seu nome e endereço. Depois de informar seus dados e confirmar, o cliente será cadastrado na PizzaNet.

- **Opinar** – Esse serviço permite que o cliente emita opiniões sobre os pedidos feitos anteriormente por ele. Quando essa alternativa for selecionada, o sistema apresentará um formulário onde o cliente escreverá sua opinião e, se esta for confirmada, o sistema a registrará. Esse caso de uso só pode ser utilizado depois que o cliente autenticar-se no sistema.
- **Visualizar Pedido** – Por meio dessa opção, o cliente pode visualizar os itens escolhidos (pizzas e bebidas) para seu pedido. Essa funcionalidade apresenta todas as pizzas e bebidas escolhidas pelo cliente. Eventualmente, um cliente pode querer excluir algum dos itens selecionados. Essa funcionalidade está descrita no caso de uso **Excluir Item**.
- **Excluir Item** – Representa a situação em que o cliente, a partir do caso de uso **Visualizar Pedido**, deseja excluir um item de seu pedido. Para isso, deverá selecionar o item que deseja eliminar e escolher a opção **Excluir Item**. Quando essa opção for escolhida, o item selecionado será eliminado do pedido do cliente. Observe que existe uma associação de extensão entre o caso de uso **Excluir Item** e o **Visualizar Pedido**, o que demonstra que o primeiro caso de uso será chamado pelo segundo somente quando uma condição for satisfeita, ou seja, quando o cliente selecionar um item e pressionar o botão **Excluir Item**.
- **Visualizar Pedidos Anteriores** – Por meio desse caso de uso é possível ao cliente visualizar todos os pedidos já feitos por ele. Observe que esse caso de uso tem uma associação de inclusão com o caso de uso **Visualizar Pedido**, uma vez que esse processo seleciona todos os pedidos já feitos pelo cliente, mas para apresentar cada um deles chama a rotina já descrita no caso de uso **Visualizar Pedido**.
- **Visualizar Sabores Mais Pedidos** – Esse processo apresenta todos os sabores da pizzaria em ordem de sua maior predileção, ou seja, os sabores mais solicitados pelos clientes são apresentados primeiro.
- **Concluir Pedido** – Representa o último passo para solicitar um pedido.

O processo apresenta uma associação de extensão com o caso de uso **Logar**, o que significa que, caso o cliente não tenha se autenticado ainda, o sistema deverá executar esse caso de uso para que o cliente se logue. Observe também que o caso de uso **Concluir Pedido** tem relação de inclusão com o caso de uso **Visualizar Pedido**, já que é obrigatório que o cliente visualize seu pedido uma última vez antes de concluir-lo. Somente depois de o cliente estar logado e ter visualizado o pedido, o sistema definirá o pedido como concluído. Quando isso ocorre, é dada baixa no estoque de todos os produtos necessários à produção do pedido.

#### *Segundo Diagrama – Subsistema Administrativo*

A seguir, descreveremos o segundo diagrama de casos de uso referente ao subsistema administrativo, apresentado na figura 17.4, onde são modeladas as funcionalidades utilizadas internamente pelos funcionários da empresa.

Esse diagrama contém dois atores. O primeiro representa os funcionários da empresa responsáveis por atender aos pedidos dos clientes. Esse ator tem permissão para executar somente os casos de uso **Realizar Login Funcionário**, **Visualizar Pedidos em Aberto** e **Finalizar Pedido Cliente**.

Já o segundo é um ator derivado do primeiro, uma vez que se trata também de um funcionário, porém com um nível de permissão maior chamado **Administrador**. Esse ator pode executar todos os casos de uso aqui representados, incluindo os já descritos. Os outros casos de uso, que serão explicados nesta seção, têm um caráter de cunho mais administrativo e, por esse motivo, só podem ser utilizados pelo ator **Administrador**.

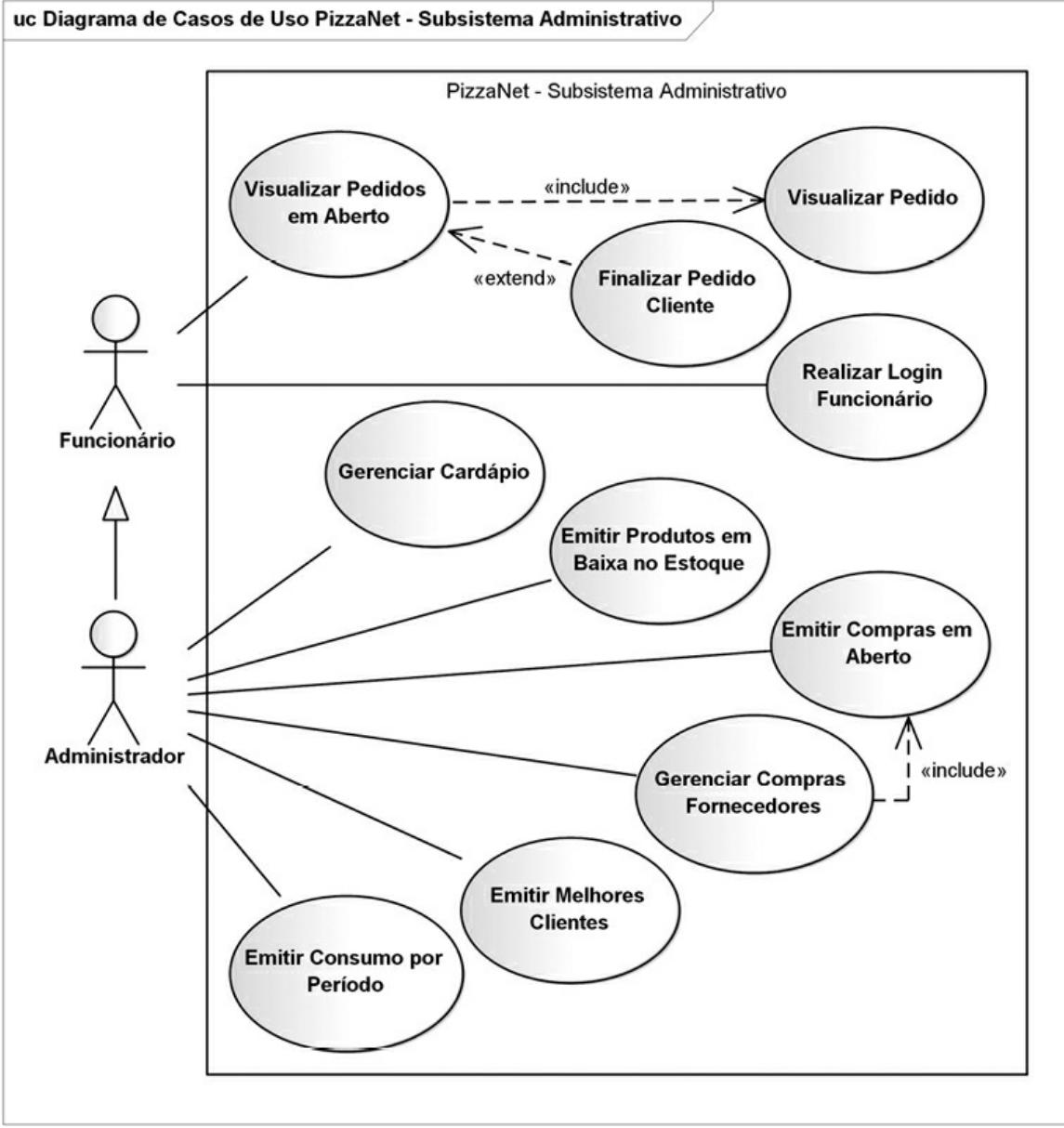


Figura 17.4 – Diagrama de Casos de Uso – Subsistema Administrativo.

A seguir, descreveremos os casos de uso desse diagrama, primeiramente enfocando os casos de uso que podem ser solicitados pelos funcionários comuns.

- **Realizar Login Funcionário** – Representa o processo para que um funcionário se autentique no sistema. Por esse processo ser extremamente semelhante ao caso de uso Logar do subsistema de vendas, este será o único caso de uso não detalhado por meio de outros diagramas.

- **Visualizar Pedidos em Aberto** – Este processo permite que um funcionário obtenha uma listagem de todos os pedidos ainda não atendidos. Observe que existe uma associação de inclusão entre esse caso de uso e o caso de uso **Visualizar Pedido**, descrito na sessão anterior sobre o sistema de vendas. Assim, ao executar esse processo, para cada pedido em aberto é chamado o processo **Visualizar Pedido** para apresentar seus detalhes. Observe que existe ainda uma relação de extensão entre esse caso de uso e o caso de uso **Finalizar Pedido Cliente**, que representa a situação em que o pedido foi terminado pelo funcionário e este deseja defini-lo como concluído. Como essa é uma situação que só ocorre mediante a condição descrita anteriormente, é representada por uma extensão.
- **Finalizar Pedido Cliente** – Define um pedido como finalizado, determinando o funcionário que o preparou e o que o entregou. Apresenta uma associação de extensão com o caso de uso **Visualizar Pedidos em Aberto**, pelo fato de ser chamado a partir deste, mediante a condição de que o funcionário tenha finalizado um dos pedidos listados. Quando isso ocorre, o funcionário deve selecionar o pedido na listagem e selecionar a opção finalizar pedido, o que disparará esse processo.

Os próximos casos de uso que descreveremos só podem ser solicitados pelo Administrador:

- **Gerenciar Cardápio** – Este é um caso de uso secundário que permite ao **Administrador** consultar, incluir, alterar ou excluir pizzas do cardápio da PizzaNet.
- **Emitir Produtos em Baixa no Estoque** – Esse processo gera um relatório apresentando todos os produtos em baixa no estoque.
- **Emitir Compras em Aberto** – Apresenta uma listagem ao **Administrador** contendo todas as compras solicitadas a fornecedores que ainda não foram entregues.
- **Gerenciar Compras Fornecedores** – permite ao **Administrador** efetuar a manutenção das compras da empresa. Por meio desse processo, é possível consultar, incluir, alterar ou excluir compras. Observe que existe uma associação de inclusão entre esse caso de uso e o caso de uso anterior, determinando que, no momento em que esse caso de uso for

executado, o caso de uso **Emitir Compras em Aberto** será também executado e, a partir da listagem por ele fornecida, o **Administrador** poderá definir uma compra como concluída ou incluir uma nova compra.

- **Emitir Melhores Clientes** – Representa o processo de emissão de um relatório de clientes ordenado pelos clientes que mais consomem na PizzaNet.
- **Emitir Consumo por Período** – Representa a emissão do relatório que apresenta o consumo dos itens do estoque em um determinado período informado pelo **Administrador**.

### 17.2.2 Documentação dos Diagramas de Casos de Uso da PizzaNet

Nesta seção, o leitor poderá examinar as documentações de cada caso de uso descrito na seção 17.2.1.

*Tabela 17.1 – Documentação do Caso de Uso Escolher Pizza*

Nome do caso de uso	UC01 – Escolher Pizza
Ator principal	Cliente
Atores secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para escolher uma pizza
Pré-condições	
Pós-condições	
Cenário principal	Ações do sistema
Ações do ator	
1. Selecionar opção Escolher Pizza (o simples acesso à página já caracteriza essa escolha, pois essa é a opção default, mas o cliente pode selecioná-la também durante sua interação com o sistema, caso anteriormente tenha selecionado outra opção)	2. Apresentar tamanhos e sabores disponíveis
3. Selecionar o tamanho da pizza	4. Permitir a escolha de sabores de acordo com o tamanho selecionado
5. Selecionar tantos sabores quantos desejados até o limite do tamanho	

escolhido

6. Adicionar pizza ao carrinho de pizzas

7. Acrescentar a pizza escolhida ao pedido

Cenário alternativo – Primeiro item do pedido	
Ações do ator	Ações do sistema
	1. Caso a pizza seja o primeiro item, registrar o pedido

Restrições/validações

*Tabela 17.2 – Documentação do Caso de Uso Escolher Bebida*

Nome do caso de uso	UC02 – Escolher Bebida
Ator principal	Cliente
Atores secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para adicionar bebida ao seu pedido
Pré-condições	
Pós-condições	
Cenário principal	
Ações do ator	Ações do sistema
1. Selecionar opção Escolher Bebida	2. Apresentar tela para escolha de bebida
3. Escolher o tipo de bebida (suco, refrigerante ou cerveja)	4. Apresentar todas as bebidas disponíveis, com seus valores, para o tipo selecionado
5. Selecionar a bebida, informando a quantidade desejada, e adicioná-la ao pedido	6. Acrescentar a bebida escolhida ao pedido
Cenário alternativo – Primeiro item do pedido	
Ações do ator	Ações do sistema
	1. Caso a bebida seja o primeiro item, registrar o pedido

Restrições/validações

*Tabela 17.3 – Documentação do Caso de Uso Logar*

Nome do caso de uso	UC03 – Logar
Autor principal	Cliente

<b>Atores secundários</b>	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para logar-se ao sistema
Pré-condições	É preciso estar cadastrado
Pós-condições	
<b>Cenário principal</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar opção Logar	2. Apresentar o formulário de login
3. Informar login e senha	4. Autenticar cliente
<b>Cenário alternativo – Manutenção do cadastro de cliente</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
	1. Caso o cliente não esteja cadastrado, executar o módulo de UC04 – Autorregisterar
<b>Restrições/validações</b>	

*Tabela 17.4 – Documentação do Caso de Uso Autorregisterar*

<b>Nome do caso de uso</b>	<b>UC04 – Autorregisterar</b>
Ator principal	Cliente
Atores secundários	
Resumo	Este caso de uso apresenta os passos para que o cliente se cadastre
Pré-condições	
Pós-condições	
<b>Cenário principal</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar opção Autorregisterar	2. Apresentar módulo de cadastro
3. Fornecer dados e confirmar	4. Registrar cliente
<b>Restrições/validações</b>	

*Tabela 17.5 – Documentação do Caso de Uso Opinar*

<b>Nome do caso de uso</b>	<b>UC05 – Opinar</b>
Autor principal	Cliente
Atores secundários	

Resumo	Este caso de uso descreve as etapas percorridas por um cliente para expressar uma opinião sobre o atendimento e a qualidade da Pizzanet
Pré-condições	É preciso estar logado
Pós-condições	
<b>Cenário principal</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar opção Opinar	2 Apresentar a tela para submeter uma opinião
2. Informar opinião	4. Registrar opinião
<b>Restrições/validações</b>	

*Tabela 17.6 – Documentação do Caso de Uso Visualizar Pedido*

<b>Nome do caso de uso</b>	<b>UC06 – Visualizar Pedido</b>
Ator principal	Cliente
Atores secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para visualizar os itens de seu pedido
Pré-condições	
Pós-condições	
<b>Cenário principal</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar opção Visualizar Pedido	2. Apresentar todas as pizzas, com seus respectivos sabores, e as bebidas escolhidas pelo cliente
<b>Cenário alternativo – Excluir item</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar item e solicitar sua exclusão	2. Executar o caso de uso UC07 – Excluir Item
<b>Restrições/validações</b>	

*Tabela 17.7 – Documentação do Caso de Uso Excluir Item*

<b>Nome do caso de uso</b>	<b>UC07 – Excluir Item</b>
Ator principal	Cliente
Atores secundários	
Resumo	Este caso de uso descreve as etapas percorridas para excluir um item do pedido
Pré-condições	

Pós-condições	Cenário principal	
Ações do ator	Ações do sistema	
1. Selecionar item a excluir e confirmar	<b>Cenário alternativo I – Excluir pizza</b>	
Ações do ator	Ações do sistema	
	1. Excluir pizza 2. Excluir itens de sabor da pizza	
<b>Cenário alternativo II – Excluir bebida</b>		
Ações do ator	Ações do sistema	
	1. Excluir bebida	
Restrições/validações		

*Tabela 17.8 – Documentação do Caso de Uso Visualizar Pedidos Anteriores*

Nome do caso de uso	UC08 – Visualizar Pedidos Anteriores
Ator principal	Cliente
Atores secundários	
Resumo	Este caso de uso apresenta todos os pedidos já realizados pelo cliente
Pré-condições	O cliente precisa estar logado
Pós-condições	
<b>Cenário principal</b>	
Ações do ator	Ações do sistema
1. Selecionar opção Visualizar Pedidos Anteriores	2. Para cada pedido, executar o módulo UC06 – Visualizar Pedido 3. Apresentar pedidos
<b>Cenário alternativo – Pedir um pedido anterior</b>	
Ações do ator	Ações do sistema
1. Opcionalmente, o cliente poderá selecionar um dos pedidos já realizados para pedi-lo novamente	2. Registrar o novo pedido
Restrições/validações	

*Tabela 17.9 – Documentação do Caso de Uso Visualizar Sabores Mais Pedidos*

Nome do caso de uso	UC09 – Visualizar Sabores Mais Pedidos
Ator principal	Cliente
Atores secundários	

Resumo	Este caso de uso apresenta os sabores de pizzas mais pedidos pelos clientes
Pré-condições	
Pós-condições	
	<b>Cenário principal</b>
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar opção Sabores Mais Pedidos	
	2. Selecionar os sabores solicitados nos pedidos e somá-los
	3. Apresentar sabores por ordem dos mais solicitados
Restrições/validações	

*Tabela 17.10 – Documentação do Caso de Uso Concluir Pedido*

Nome do caso de uso	UC10 – Concluir Pedido
Ator principal	Cliente
Atores secundários	
Resumo	Este caso de uso descreve as etapas percorridas por um cliente para concluir seu pedido
Pré-condições	1. O cliente necessita estar logado 2. É necessário ter adicionado ao menos um item (pizza ou bebida) ao pedido
Pós-condições	
	<b>Cenário principal</b>
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar opção Fechar Pedido	
	2. Executar o caso de uso Visualizar Pedido
	3. Apresentar o endereço de entrega e permitir que este seja alterado
4. Confirmar o pedido	
	5. Dar baixa nos itens do estoque necessários ao atendimento do pedido
	6. Concluir o pedido
	<b>Cenário alternativo I – Logar cliente</b>
<b>Ações do ator</b>	<b>Ações do sistema</b>
	1. Caso o cliente não esteja logado, executar o módulo de Logar
	<b>Cenário alternativo II – Atualizar endereço</b>
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Caso o cliente deseje, fornecer novo endereço para entrega e confirmar	
	2. Atualizar o endereço para entrega do cliente

Restrições/validações	Caso o cliente exclua todos os itens do pedido, este deve ser cancelado
-----------------------	---

*Tabela 17.11 – Documentação do Caso de Uso Realizar Login Funcionário*

Nome do caso de uso	UC11 – Realizar Login Funcionário	
Autor principal	Funcionário	
Atores secundários		
Resumo	Este caso de uso descreve as etapas percorridas por um funcionário para logar-se ao sistema	
Pré-condições	É preciso estar registrado	
Pós-condições		
Cenário principal		
Ações do ator	Ações do sistema	
1. Selecionar opção Realizar Login Funcionário	2. Apresentar o formulário para logar funcionários	
3. Informar login e senha	4. Autenticar funcionário	
Restrições/validações		

*Tabela 17.12 – Documentação do Caso de Uso Visualizar Pedidos em Aberto*

Nome do caso de uso	UC12 – Visualizar Pedidos em Aberto	
Autor principal	Funcionário	
Atores secundários		
Resumo	Este caso de uso apresenta os passos para que um funcionário verifique quais pedidos se encontram em aberto	
Pré-condições		
Pós-condições		
Cenário principal		
Ações do ator	Ações do sistema	
1. Selecionar opção Visualizar Pedidos em Aberto	2. Para cada pedido em aberto, executar o caso de uso UC06 – Visualizar Pedido	
Restrições/validações		
Cenário alternativo I – Finalizar pedido		
Ações do ator	Ações do sistema	
1. Escolher um pedido e selecionar a opção Finalizar Pedido	2. Executar o caso de uso UC13 – Finalizar Pedido Cliente	

**Restrições/validações**

*Tabela 17.13 – Documentação do Caso de Uso Finalizar Pedido Cliente*

<b>Nome do caso de uso</b>	<b>UC13 – Finalizar Pedido Cliente</b>
Ator principal	Funcionário
Atores secundários	
Resumo	Este caso de uso apresenta os passos para que um funcionário finalize um pedido em aberto
Pré-condições	
Pós-condições	
<b>Cenário principal</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
	1. Apresentar os funcionários disponíveis
2. Informar funcionários que prepararam e entregaram o pedido	3. Finalizar pedido
Restrições/validações	É necessário haver funcionários cadastrados

*Tabela 17.14 – Documentação do Caso de Uso Gerenciar Cardápio*

<b>Nome do caso de uso</b>	<b>UC14 – Gerenciar Cardápio</b>
Ator principal	Administrador
Atores secundários	
Resumo	Este caso de uso detalha os passos para que o administrador possa inserir, alterar ou excluir sabores oferecidos no cardápio da pizzaria
Pré-condições	
Pós-condições	
<b>Cenário principal</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar opção Manter Cardápio	2. Carregar todos os sabores oferecidos pela pizzaria 3. Carregar todos os ingredientes utilizados pela pizzaria
Restrições/validações	É necessário haver ingredientes
<b>Cenário alternativo I – Incluir sabor</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar a opção inclusão de sabor	
2. Informar sabor e ingredientes	3. Registrar sabor
Restrições/validações	

<b>Cenário alternativo II – Consultar sabor</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar e escolher a opção de consulta de sabor	2. Apresentar ingredientes do sabor
<b>Restrições/validações</b>	
	Deve-se selecionar um sabor
<b>Cenário alternativo III – Alterar sabor</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Informar alterações no sabor e/ou ingredientes	2. Registrar alterações
<b>Restrições/validações</b>	
<b>Cenário alternativo IV – Excluir sabor</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar a exclusão do sabor	2. Excluir sabor
<b>Restrições/validações</b>	

*Tabela 17.15 – Documentação do Caso de Uso Emitir Produtos em Baixa no Estoque*

<b>Nome do caso de uso</b>	<b>UC15 – Emitir Produtos em Baixa no Estoque</b>
Ator principal	Administrador
Atores secundários	
Resumo	Este caso de uso detalha os passos para que o administrador visualize os produtos cujas quantidades estão baixas no estoque
Pré-condições	
Pós-condições	
<b>Cenário principal</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar opção Emitir Produtos em Baixa no Estoque	2. Apresentar ingredientes cujas quantidades estejam baixas 3. Apresentar bebidas cujas quantidades estejam baixas
<b>Restrições/validações</b>	

*Tabela 17.16 – Documentação do Caso de Uso Emitir Compras em Aberto*

<b>Nome do caso de uso</b>	<b>UC16 – Emitir Compras em Aberto</b>
Ator principal	Administrador
Atores secundários	
<b>Ações do ator</b>	<b>Ações do sistema</b>

Resumo	Este caso de uso detalha os passos para que o funcionário obtenha uma listagem das compras ainda não atendidas pelo fornecedor
Pré-condições	
Pós-condições	
<b>Cenário principal</b>	
<b>Ações do ator</b>	<b>Ações do sistema</b>
1. Selecionar opção Emitir Compras em Aberto	2. Selecionar compras cuja situação esteja em aberto 3. Apresentar compras em aberto detalhando os itens solicitados
<b>Restrições/validações</b>	

*Tabela 17.17 – Documentação do Caso de Uso Gerenciar Compras Fornecedores*

Nome do caso de uso		UC17 – Gerenciar Compras Fornecedores
Atores principal		Administrador
Atores secundários		
Resumo		Este caso de uso detalha os passos para solicitar pedidos de compras a fornecedores
Pré-condições		
Pós-condições		
<b>Cenário principal</b>		
<b>Ações do ator</b>	<b>Ações do sistema</b>	
1. Selecionar opção Manter Compras Fornecedor	2. Executar Caso de Uso UC16 – Emitir Compras em Aberto 3. Apresentar tela de manutenção de compras, contendo todas as compras em aberto, permitindo ao usuário finalizar uma compra em aberto ou gerar uma nova compra	
<b>Cenário alternativo I – Finalizar compra</b>		
<b>Ações do ator</b>	<b>Ações do sistema</b>	
1. Selecionar uma compra em aberto e solicitar seu fechamento	2. Finalizar compra	
Restrições/validações		É preciso selecionar uma compra
<b>Cenário alternativo II – Gerar nova compra</b>		
<b>Ações do ator</b>	<b>Ações do sistema</b>	
1. Escolher a opção para		

gerar uma nova compra	2. Carregar os ingredientes utilizados 3. Carregar as bebidas comerciadas 4. Carregar os fornecedores registrados
5. Selecionar fornecedor	
6. Selecionar ingredientes e quantidades a comprar	
7. Selecionar bebidas e quantidades a comprar	
8. Confirmar compra	9. Registrar compra

**Restrições/validações** Deve-se selecionar ao menos um ingrediente ou bebida

*Tabela 17.18 – Documentação do Caso de Uso Emitir Melhores Clientes*

Nome do caso de uso	UC18 – Emitir Melhores Clientes
Ator principal	Administrador
Atores secundários	
Resumo	Este caso de uso detalha os passos para apresentar os melhores clientes, considerando-se o maior valor gasto por eles
Pré-condições	
Pós-condições	
Cenário principal	
Ações do ator	Ações do sistema
1. Selecionar opção Emitir Melhores Clientes	2. Selecionar clientes 3. Selecionar os pedidos de cada cliente 4. Totalizar os valores das pizzas dos pedidos 5. Totalizar os valores das bebidas dos pedidos 6. Apresentar os clientes por ordem de maior valor gasto com o bairro em que residem
<b>Restrições/validações</b>	

*Tabela 17.19 – Documentação do Caso de Uso Emitir Consumo por Período*

Nome do caso de uso	UC19 – Emitir Consumo por Período
Ator principal	Administrador
Atores secundários	
Resumo	Este caso de uso apresenta os passos para que o administrador verifique o consumo dos ingredientes em um determinado período
Pré-condições	
Pós-condições	
Ações do ator	Ações do sistema

Cenário principal	
Ações do ator	Ações do sistema
1. Selecionar opção Emitir Consumo por Período	2. Solicitar períodos desejados
3. Informar períodos	4. Selecionar todas as pizzas dos pedidos realizados no período 5. Totalizar os ingredientes de cada sabor pedido 6. Apresentar consumo no período solicitado
Restrições/validações	

### 17.2.3 Diagrama de Classes – Modelo de Domínio

Nesta seção, apresentaremos na figura 17.5 o modelo de domínio da solução para o sistema de pizzaria online.

Descreveremos as classes que compõem esse diagrama com suas associações, métodos e atributos. Embora já tenhamos afirmado isso no capítulo 4, sobre o diagrama de classes, acreditamos ser útil salientar novamente que, a rigor, deve haver um método **get** e um método **set** para cada atributo de uma classe. No entanto, essa modelagem enfoca o sistema em um nível mais alto. Assim, consideramos impraticável e mesmo desnecessário inserir um conjunto de métodos óbvios em classes que contenham um grande número de atributos. Dessa forma, definimos métodos como “registrar” ou “consultar” para inserir ou retornar conjuntos de atributos sempre que o processo não exigir métodos exclusivos para um atributo específico. Por esse motivo, muitos métodos aqui descritos retornam um atributo do tipo **String**, que poderá conter os valores de um conjunto de atributos necessários ao processo.

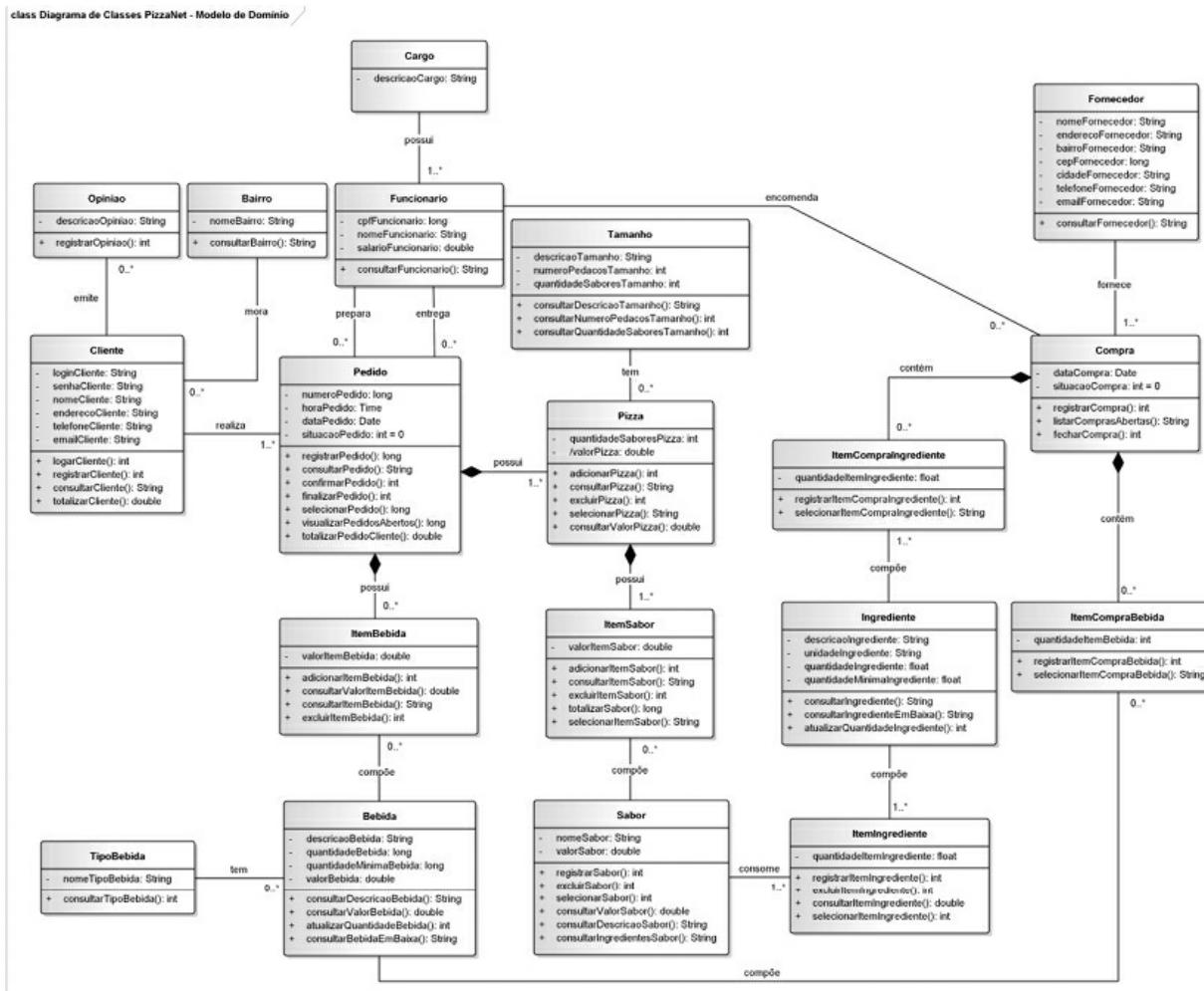


Figura 17.5 – Diagrama de Classes da PizzaNet – Modelo de Domínio.

- Cliente** – Essa classe tem como função armazenar as informações relativas aos clientes da pizzaria, ou seja, pessoas que já solicitaram ao menos um pedido. Os atributos necessários a essa classe são **login**, **senha**, **nome**, **endereço**, **telefone** e **e-mail**, todos do tipo **String**. A classe contém, ainda, os seguintes métodos:

Método	Descrição
<b>logarCliente</b>	Permite autenticar um cliente na página do software, recebendo como parâmetros o login e a senha do cliente. Se estes estiverem corretos, o método retornará verdadeiro, informando que o cliente foi autenticado com sucesso, caso contrário, retornará falso.
<b>registrarCliente</b>	Utilizado para registrar um novo cliente. Recebe como parâmetros todos os atributos definidos na classe <b>Cliente</b> e retorna verdadeiro, se o cliente foi registrado com sucesso, ou falso, caso contrário.
	Permite pesquisar um determinado cliente. Esse método retorna uma

`consultarClienteString` contendo o nome do cliente consultado.

Possui como objetivo totalizar todos os pedidos já realizados por um cliente para determinar quais os melhores clientes da pizzaria. O método `totalizarCliente` retorna um `double` contendo os valores totais gastos por um cliente. É utilizado em conjunto com outros métodos, definidos em outras classes, como `totalizarPedido`, que serão explicados ao longo desta seção.

- **Opiniao** – Essa classe armazena as opiniões registradas pelos clientes relativas a seu nível de satisfação quanto aos pedidos solicitados. Tem como únicos atributos a descrição da opinião fornecida, do tipo `String`, e o método `registrarOpiniao`, que instancia um novo objeto da classe, retornando verdadeiro quando é bem-sucedido. Observe que um objeto da classe `Opiniao` está vinculado a somente um objeto da classe `Cliente`, mas um objeto dessa última classe pode estar vinculado a muitas instâncias da classe `Opiniao`.
- **Bairro** – Essa classe contém as informações relativas aos bairros onde estão localizados os clientes da pizzaria. Seu único atributo é o nome do bairro, do tipo `String`. Aqui, identificamos somente o método `consultarBairro`, usado para consultar o bairro do cliente, retornando uma `String` com o nome do bairro consultado. Obviamente, é necessário existir um método para registrar novos bairros, no entanto, uma vez que o diagrama tem um conjunto muito grande de informações, procuramos definir somente os métodos realmente importantes. Muitos clientes podem morar em um determinado bairro, como demonstra a associação `mora`, mas um cliente específico só pode morar em um único bairro.
- **Cargo** – Essa classe armazena os cargos assumidos pelos funcionários da empresa. Seu único atributo é a descrição do cargo do tipo `String`.
- **Funcionario** – A classe `Funcionario` armazena as informações dos funcionários que trabalham ou trabalharam na empresa. Seus atributos contêm o CPF do funcionário, do tipo `long`, o nome, do tipo `String`, e o salário, do tipo `double`. O único método aqui descrito é o `consultarFuncionario`, que serve para consultar um funcionário, retornando uma `String` contendo seu nome. Observe que essa classe

tem uma associação com a classe anterior, que determina que um funcionário deve ter um único cargo, mas um mesmo cargo pode ser atribuído a muitos funcionários.

- **Pedido** – Essa classe armazena as informações relacionadas aos pedidos solicitados pelos clientes. Observe que a classe tem duas associações com a classe **Funcionario**, na qual um funcionário pode preparar e/ou entregar muitos pedidos, mas um pedido só pode ser preparado por um funcionário específico e, da mesma forma, só pode ser entregue por um determinado funcionário (não necessariamente o mesmo). A classe **Pedido** tem também uma associação com a classe **Cliente**, que determina que um cliente pode solicitar muitos pedidos, mas um pedido deve estar associado somente a um cliente. Há, ainda, outras associações dessa classe com outras classes, que serão explicadas ao longo desta seção.

A classe **Pedido** contém os atributos número do pedido, do tipo **long**, a hora em que o pedido foi solicitado, do tipo **Time**, a data em que este foi realizado, do tipo **Date**, e a situação do pedido, do tipo **int**, que determina se o pedido ainda não foi atendido, se já foi finalizado ou já foi entregue. É preciso destacar que os tipos **Time** e **Date** referem-se a classes que devem ser implementadas pela linguagem ou estar contidas no sistema. Essa classe tem ainda os seguintes métodos:

- **Tamanho** – Essa classe armazena os tamanhos disponibilizados pela PizzaNet. Seus atributos são a descrição do tamanho, do tipo **String**, o número de pedaços e a quantidade de sabores, ambos do tipo **int**. Essa classe tem ainda os seguintes métodos:

Método	Descrição
<code>consultarDescricaoTamanho</code>	Retorna uma <b>String</b> contendo a descrição do tamanho (se é pequeno, médio ou grande).
<code>consultarNumeroPedacosTamanho</code>	Retorna um inteiro contendo o número de pedaços do tamanho.
<code>consultarQuantidadeSaboresTamanho</code>	Retorna um inteiro contendo o número de sabores que uma pizza desse tamanho pode conter.

- **Pizza** – Essa classe armazena as informações das pizzas solicitadas em um determinado pedido. Observe que existe uma associação de

composição entre a classe **Pedido** e a classe **Pizza**, o que significa que os objetos da classe **Pedido** são **objetos-todo** cujas informações devem ser complementadas pelas informações contidas em **objetos-parte** da classe **Pizza** (essas informações devem ser complementadas também por **objetos-parte** da classe **ItemBebida**, que será explicada mais adiante). Dessa maneira, como podemos observar pela associação de composição entre essas duas classes, um pedido pode conter muitas pizzas, mas uma pizza deve estar contida única e exclusivamente em um pedido.

É preciso notar também que existe uma associação entre essa classe e a classe tamanho, determinando que uma pizza pode estar associada a somente um tamanho, mas um tamanho pode estar relacionado a muitas pizzas. Existe ainda outra associação que será explicada quando abordarmos a próxima classe.

A classe **Pizza** contém os atributos quantidade de sabores, do tipo **int**, e valor da pizza, do tipo **double**. O leitor poderá se perguntar o porquê do atributo quantidade de sabores, uma vez que essa informação pode ser puxada da classe **Tamanho** relacionada à classe **Pizza**. Isso estaria correto, mas pode acontecer de um cliente solicitar uma pizza grande com um único sabor. Por esse motivo, optamos por inserir esse atributo nessa classe. O leitor poderá inquirir também por que existe o atributo valor da pizza, uma vez que este é calculado, como demonstra a barra ao lado de sua visibilidade. Optamos por inserir esse atributo para facilitar os processos que pesquisarão as pizzas dos pedidos, diminuindo o número de operações a ser realizadas e as classes a ser consultadas.

Essa classe contém ainda os seguintes métodos:

Método	Descrição
<b>adicionarPizza</b>	Serve para adicionar uma pizza ao pedido. Retorna um inteiro que determina se o método foi executado com sucesso ou não.
<b>consultarPizza</b>	Permite consultar uma pizza de um pedido e retorna uma <b>String</b> contendo os dados da pizza consultada.
<b>excluirPizza</b>	Por meio desse método é possível excluir uma pizza de um pedido. O método retorna um inteiro que conterá um valor verdadeiro (1), caso o método consiga excluir a pizza, ou falso (0), caso contrário.
<b>selecionarPizza</b>	Seleciona uma pizza de um pedido, retornando uma <b>String</b> contendo os totais dos ingredientes consumidos em cada pizza.
	Permite consultar o valor de uma pizza específica, retornando um

`consultarValorPizza` contendo seu valor.

- `ItemSabor` – Essa classe armazena as informações sobre os sabores escolhidos para uma pizza. Observe que existe uma associação de composição entre a classe `Pizza` e a classe `ItemSabor`. Isso significa que os objetos da classe `Pizza` são também **objetos-todo** em relação aos objetos da classe `ItemSabor`, que desempenham o papel de **objetos-parte** nessa associação, uma vez que estes devem complementar as informações de um objeto da classe `Pizza`, fornecendo os dados de cada sabor escolhido para uma pizza específica.

Essa classe contém o atributo `valorItemSabor`, do tipo `double`. Alguém poderia se perguntar se esse atributo é realmente necessário, uma vez que pode ser acessado a partir da classe `Sabor`, que explicaremos a seguir. Esse atributo é realmente necessário porque o valor de um sabor pode ser alterado, o que deturparia as informações relativas ao valor do sabor na época em que o pedido foi feito.

A classe `ItemSabor` contém ainda o seguintes métodos:

Método	Descrição
<code>adicionarItemSabor</code>	Permite adicionar um sabor a uma pizza. Retorna um inteiro determinando o sucesso ou não do método.
<code>consultarItemSabor</code>	Consulta um sabor específico de uma pizza, retornando uma <code>String</code> contendo seus dados.
<code>excluirItemSabor</code>	Permite excluir um sabor de uma pizza, retornando um inteiro que determina o sucesso ou não dessa operação.
<code>totalizarSabor</code>	Utilizado para totalizar quantas vezes um sabor foi pedido. O método retorna um <code>long</code> contendo esse total.
<code>selecionarItemSabor</code>	Chamado pelo método <code>selecionarPizza</code> para selecionar todos os objetos da classe <code>ItemSabor</code> associados à pizza. O método retorna uma <code>String</code> com os dados desses objetos.

- `Sabor` – Essa classe armazena as informações relacionadas aos sabores oferecidos pela pizzaria. Seus atributos são o nome do sabor, do tipo `String`, e o valor, do tipo `double`. Observe que essa classe tem uma associação com a classe `ItemSabor`, que determina que um objeto da classe `Sabor` pode estar associado a muitos objetos da classe `ItemSabor`, mas um objeto da classe `ItemSabor` só pode estar relacionado a um

objeto da classe **Sabor**. A classe contém ainda os métodos:

Método	Descrição
<b>registrarSabor</b>	Permite o registro de um novo sabor. Retorna um inteiro que, se contiver verdadeiro (1), indicará que foi possível registrar o novo sabor, caso contrário, determinará que algum erro ocorreu.
<b>excluirSabor</b>	Permite a exclusão de um sabor. Retorna um inteiro para determinar o sucesso ou não da operação.
<b>selecionarSabor</b>	Usado durante o processo de conclusão de um pedido para auxiliar a dar baixa nos ingredientes utilizados na produção dos sabores. Sua função é selecionar todos os sabores solicitados no pedido. O método retorna um inteiro determinando se o método foi concluído com sucesso ou não.
<b>consultarValorSabor</b>	Permite consultar o valor de um sabor, retornando um <b>double</b> contendo seu valor.
<b>consultarDescricaoSabor</b>	Permite consultar a descrição do sabor, retornando uma <b>String</b> contendo sua descrição.
<b>consultarIngredientesSabor</b>	Permite consultar os ingredientes necessários ao preparo de um sabor. Retorna uma <b>String</b> contendo a listagem dos ingredientes necessários.

O leitor talvez se pergunte por que é necessária a classe **ItemSabor** e se sua função não poderia ser assumida pela classe **Sabor**. Ocorre, porém, que uma pizza pode ter muitos sabores e um sabor pode estar contido em muitas pizzas. Sendo assim, é necessário haver uma classe intermediária entre as classes **Pizza** e **Sabor** que armazene as informações dos sabores de uma pizza específica.

- **Ingrediente** – Essa classe armazena as informações referentes aos ingredientes necessários para preparar os sabores oferecidos pela PizzaNet. Seus atributos são a descrição do ingrediente, a unidade do ingrediente, ambos do tipo **String**, além da quantidade em estoque e a quantidade mínima do produto (se a quantidade do ingrediente estiver igual ou abaixo do mínimo, deverá ser feita solicitação de compra a um fornecedor), ambos do tipo **double**.

Método	Descrição
<b>consultarIngrediente</b>	Permite consultar um ingrediente específico, retornando uma <b>String</b> com sua descrição. Retorna uma <b>String</b> contendo as informações de

`consultarIngredientEmBaixa` todos os ingredientes cuja quantidade em estoque estiver igual ou abaixo do mínimo.

`atualizarQuantidadeIngrediente` Diminui a quantidade necessária à produção de um sabor da quantidade em estoque de um ingrediente. Retorna um inteiro que determina o sucesso ou não da operação.

- **ItemIngrediente** – Esta é uma classe intermediária que contém as informações de cada ingrediente necessário à produção de um sabor. Essa classe justifica-se uma vez que um sabor pode estar relacionado a muitos ingredientes e um ingrediente pode estar associado a muitos sabores. Assim, cada objeto dessa classe está associado a um objeto da classe **Ingrediente** e a um objeto da classe **Sabor**, mas um objeto da classe **Sabor** pode estar associado a muitos objetos da classe **ItemIngrediente**, o mesmo ocorrendo em relação aos objetos da classe **Ingrediente**.

Essa classe armazena o atributo `quantidade` do tipo `double`, que estabelece a quantidade necessária de um ingrediente específico para um sabor específico. A classe tem ainda os seguintes métodos:

Método	Descrição
<code>registrarItemIngrediente</code>	Permite registrar um novo item de ingrediente.
<code>excluirItemIngrediente</code>	Permite excluir um item de ingrediente.
<code>consultarItemIngrediente</code>	Permite consultar um item de ingrediente, retornando um <code>double</code> contendo sua quantidade.
<code>selecionarItemIngrediente</code>	Permite selecionar os itens de ingrediente de um determinado sabor.

- **TipoBebida** – Essa classe armazena os tipos de bebida oferecidos pela PizzaNet, ou seja, suco, refrigerante e cerveja. Seu único atributo é o nome, do tipo `String`, e seu único método, `consultarTipoBebida`, permite consultar um tipo de bebida, retornando uma `String` contendo seu nome.
- **Bebida** – Essa classe armazena os dados de todas as bebidas comercializadas pelo sistema. Note que esta classe está associada à classe **TipoBebida**. A multiplicidade dessa associação determina que uma bebida deve estar associada a somente um tipo de bebida, no entanto um tipo de bebida pode estar associado a muitas bebidas. Os atributos dessa classe são a descrição da bebida, do tipo `String`, a quantidade em

estoque e a quantidade mínima da bebida, ambos do tipo `long`, e o valor da bebida, do tipo `double`.

Essa classe contém os seguintes métodos:

Método	Descrição
<code>consultarDescricaoBebida</code>	Permite consultar a descrição de uma bebida, retornando uma <code>String</code> contendo a descrição.
<code>consultarValorBebida</code>	Permite consultar o valor de uma bebida, retornando um <code>double</code> contendo o valor em questão.
<code>atualizarQuantidadeBebida</code>	Diminui a quantidade solicitada em um pedido da quantidade em estoque de uma bebida. Retorna verdadeiro, se a operação pôde ser realizada, e falso, caso contrário.
<code>consultarBebidaEmBaixa</code>	Consulta todas as bebidas com a quantidade em estoque abaixo da quantidade mínima, retornando uma <code>String</code> com os dados de cada bebida em baixa.

- **ItemBebida** – Essa é uma classe intermediária posicionada entre a classe **Pedido** e a classe **Bebida**. É necessária para armazenar as bebidas solicitadas em um pedido específico, pelo fato de que um pedido pode incluir muitas bebidas e uma bebida pode estar presente em muitos pedidos, não havendo espaço em nenhuma das duas classes para armazenar as informações das bebidas do pedido, já que não é possível saber quantas bebidas terá um pedido, nem quantos pedidos solicitarão uma bebida. Mesmo que se reservasse um espaço grande em qualquer das duas classes, este poderia ser atingido e, enquanto isso não ocorresse, um grande número de atributos seria deixado em branco.

Observe que a classe **Pedido** tem uma associação de composição com a classe **ItemBebida**. Isso significa que as informações de um **objeto-todo** da classe **Pedido** precisam ser complementadas pelas informações contidas nos **objetos-parte** da classe **ItemBebida**. Além disso, essa associação nos passa a informação de que um objeto **ItemBebida** está associado a um único objeto da classe **Pedido** e só deverá ser excluído também se o pedido a que estiver associado for destruído.

A classe **ItemBebida** também está associada à classe **Bebida** e a multiplicidade dessa associação passa a informação de que uma bebida pode estar associada a muitos itens de bebida, mas um item de bebida tem que estar associado a somente uma bebida.

Essa classe tem como atributo somente o valor do item de bebida do tipo **double**. A exemplo da classe **ItemSabor**, armazenamos o valor da bebida na época em que o pedido foi feito, já que o valor armazenado na classe **Bebida** pode ser atualizado. Assim, se puxássemos o valor da classe bebida, o total apresentado em um pedido consultado poderia ser diferente do real, se tivesse ocorrido algum aumento no valor da bebida.

A classe **ItemBebida** contém ainda os métodos:

Método	Descrição
<b>adicionarItemBebida</b>	Permite adicionar uma nova bebida a um pedido, retornando verdadeiro, se o método foi executado com sucesso, ou falso, caso contrário.
<b>consultarValorBebida</b>	Consulta o valor de uma bebida, retornando um <b>double</b> contendo seu valor.
<b>consultarItemBebida</b>	Consulta um item de bebida, retornando uma <b>String</b> com seus dados. É usado em conjunto com o método <b>consultarDescricaoBebida</b> , já explicado na classe anterior.
<b>excluirItemBebida</b>	Permite excluir um item de bebida do pedido. O método retorna verdadeiro (1), se for finalizado com sucesso, ou falso (0), caso contrário.

- **Fornecedor** – Essa classe armazena as informações dos fornecedores dos quais a pizzaria compra ingredientes e bebidas. Seus atributos são nome, endereço, bairro, cidade, telefone e e-mail do tipo **String**, além do CEP do tipo **long**. A classe contém, ainda, o método **consultarFornecedor** que permite consultar um fornecedor, retornando uma **String** com seus dados.
- **Compra** – Essa classe contém as informações relativas aos pedidos de compras solicitados aos fornecedores. A classe contém os atributos data, do tipo **Date** (que se refere a uma classe), que armazena a data em que a compra foi solicitada, e situação, do tipo **int**, que armazena 0 (seu valor inicial), caso a compra ainda não tenha sido entregue, ou 1, caso contrário.

Essa classe tem uma associação com a classe **Fornecedor**, que informa que a um fornecedor podem ter sido solicitadas muitas compras, mas uma compra está associada a somente um fornecedor. Há também uma associação com a classe **Funcionario**, que determina que um

funcionário pode encomendar muitas compras, mas uma compra só pode estar associada a um funcionário. Há, ainda, outras associações que serão explicadas quando da explanação de outras classes.

Essa classe contém os seguintes métodos:

Método	Descrição
<code>registrarCompra</code>	Permite registrar uma nova compra, retornando 1 (verdadeiro), se for possível registrar a nova compra, ou falso (0), caso contrário.
<code>listarComprasAbertas</code>	Retorna uma <code>String</code> contendo os dados de todas as compras ainda não entregues pelos fornecedores.
<code>fecharCompra</code>	Permite definir uma compra como concluída, ou seja, uma compra que foi entregue pelo fornecedor, de acordo com o que foi solicitado. Esse método muda o valor do atributo <code>situacao</code> para 1, significando que a compra está fechada.

- **ItemCompraIngrediente** – Esta é uma classe intermediária que armazena as informações dos ingredientes solicitados em uma determinada compra. Observe que existe uma associação de composição entre a classe `Compra` e a classe `ItemCompraIngrediente`, determinando que um objeto da classe `Compra` é um **objeto-todo** cujas informações precisam ser complementadas pelos **objetos-parte** da classe `ItemCompraIngrediente`. Assim, um objeto dessa classe precisa estar associado a um único objeto da classe `Compra`, embora este possa estar associado a muitos objetos da classe `ItemCompraIngrediente`.

Há, ainda, uma associação dessa classe com a classe `Ingrediente`, que determina que um objeto da classe `Ingrediente` pode estar associado a muitos objetos da classe `ItemCompraIngrediente`, mas um objeto da classe `ItemCompraIngrediente` precisa estar vinculado, nessa associação, a somente um objeto da classe `Ingrediente`.

Essa classe tem como atributo unicamente a quantidade do ingrediente solicitada em cada compra, do tipo `double`. Além disso, apresenta ainda os seguintes métodos:

Método	Descrição
<code>registrarItemCompraIngrediente</code>	Permite instanciar um novo objeto dessa classe, retornando verdadeiro, se for executado com sucesso, ou falso, caso contrário.
<code>selecionarItemCompraIngrediente</code>	Seleciona cada ingrediente solicitado em uma determinada compra, retornando uma <code>String</code>

com seus dados.

- **ItemCompraBebida** – Esta é também uma classe intermediária, posicionada entre as classes **Compra** e **Bebida**, com uma função bastante semelhante à classe **ItemCompraIngrediente**. Essa classe armazena as informações das bebidas solicitadas em uma determinada compra. O leitor notará que existe também uma associação de composição entre a classe **Compra** e a classe **ItemCompraBebida**, o que significa que as informações dos **objetos-parte** dessa classe devem também complementar as informações dos **objetos-todo** da classe **Compra**. Essa associação também informa que um objeto da classe **ItemCompraBebida** necessita estar vinculado, nessa associação, a somente um objeto da classe **Compra**, mas um objeto da classe **Compra** pode estar associado a muitos objetos da classe **ItemCompraBebida**.

Essa classe tem também uma associação com a classe **Bebida** e tal associação informa que um objeto da classe **Bebida** pode estar associado a muitos objetos da classe **ItemCompraBebida**, mas um objeto da classe **ItemCompraBebida** precisa estar vinculado, nessa associação, a somente um objeto da classe **Bebida**.

Essa classe tem como atributo unicamente a quantidade da bebida solicitada em cada compra, do tipo **double**. Além disso, contém ainda os seguintes métodos:

Método	Descrição
<code>registrarItemCompraBebida</code>	Permite instanciar um novo objeto dessa classe, retornando verdadeiro, se for executado com sucesso, ou falso, caso contrário.
<code>selecionarItemCompraBebida</code>	Seleciona cada bebida solicitada em uma determinada compra, retornando uma <b>String</b> com seus dados.

## 17.2.4 Diagrama de Objetos

Nesta seção apresentaremos, na figura 17.6, um diagrama de objetos referente ao modelo de domínio apresentado na seção 17.2.3. O objetivo desse diagrama é meramente ilustrativo, com o intuito de aclarar como estarão organizados os objetos do sistema quando instanciados, como, aliás, é o objetivo de todo diagrama de objetos.

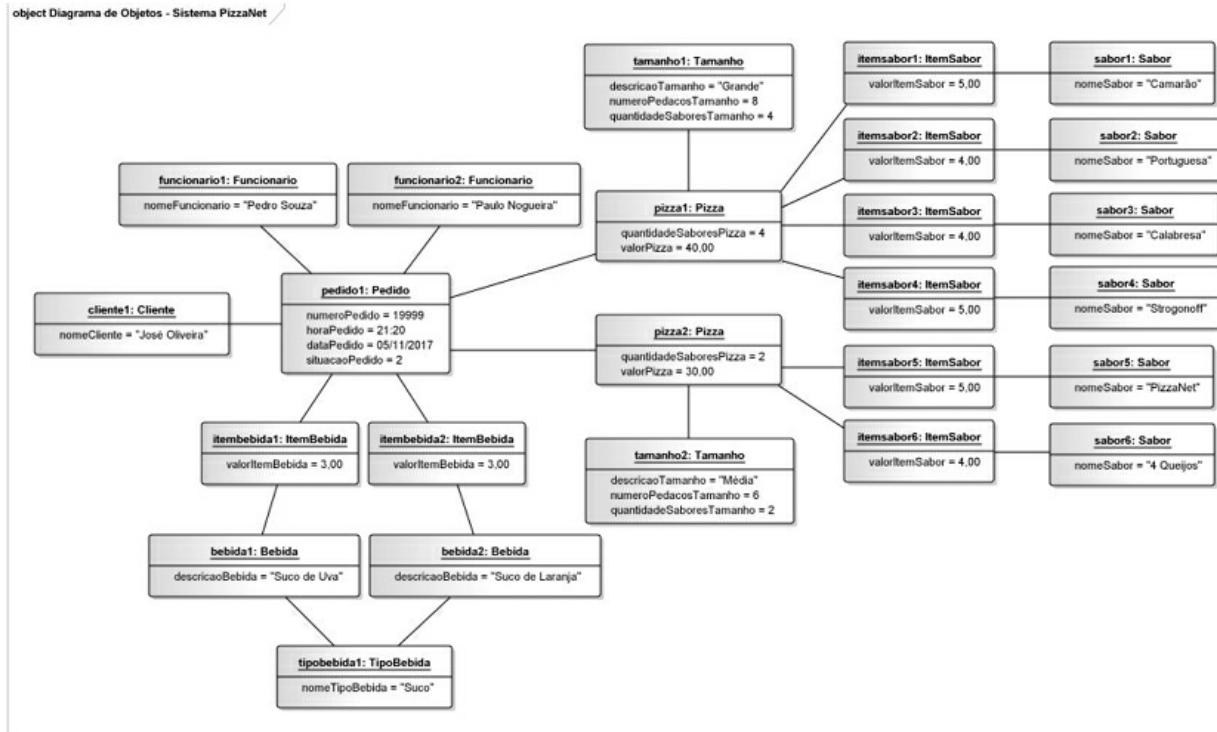


Figura 17.6 – Diagrama de Objetos PizzaNet.

Nesse exemplo, ilustramos um pedido que foi finalizado, como podemos observar pelo valor 2 do atributo **situacaoPedido**. O pedido em questão inclui duas pizzas e duas bebidas. A primeira pizza tem quatro sabores e a segunda, dois. As duas bebidas são do tipo suco, mas cada uma se refere a um suco diferente. Finalmente, o pedido foi atendido por dois funcionários (um o preparou e outro o entregou).

O leitor notará que muitos objetos não possuem todos os atributos contidos nas classes a partir da qual foram instanciados. Optamos por essa abordagem para não deixar o diagrama extenso demais, definindo o valor somente dos atributos mais importantes.

Talvez o leitor se pergunte por que o valor do objeto **pizza1** é 40,00 quando a soma dos valores dos objetos **ItemSabor** a ele associados não chega a esse valor. Isso ocorre porque o valor da pizza é calculado pela multiplicação do valor do sabor mais caro pelo número de pedaços da pizza (que, neste caso, são, respectivamente, 5,00 e 8,00).

## 17.2.5 Diagrama de Pacotes da PizzaNet

Nesta seção apresentaremos, na figura 17.7, o diagrama de pacotes do

sistema PizzaNet, no qual podemos perceber que este é composto de dois subsistemas, o subsistema de vendas e o subsistema administrativo. Podemos perceber também que o subsistema administrativo tem uma dependência com relação ao subsistema de vendas, uma vez que é necessário haver pedidos para que os módulos administrativos possuam dados que justifiquem sua execução.

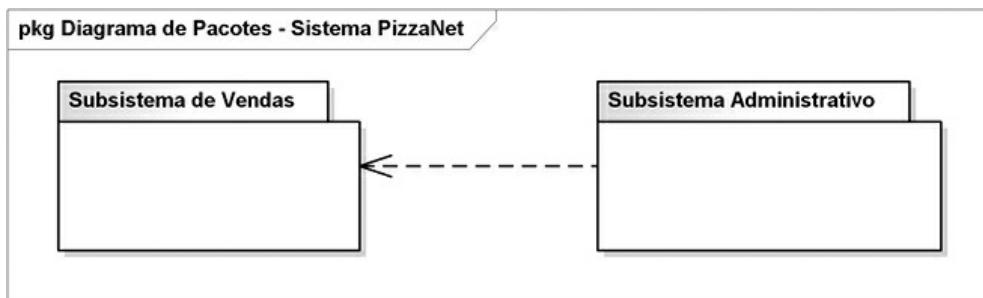


Figura 17.7 – Diagrama de Pacotes PizzaNet.

### 17.2.6 Diagramas de Sequência da PizzaNet

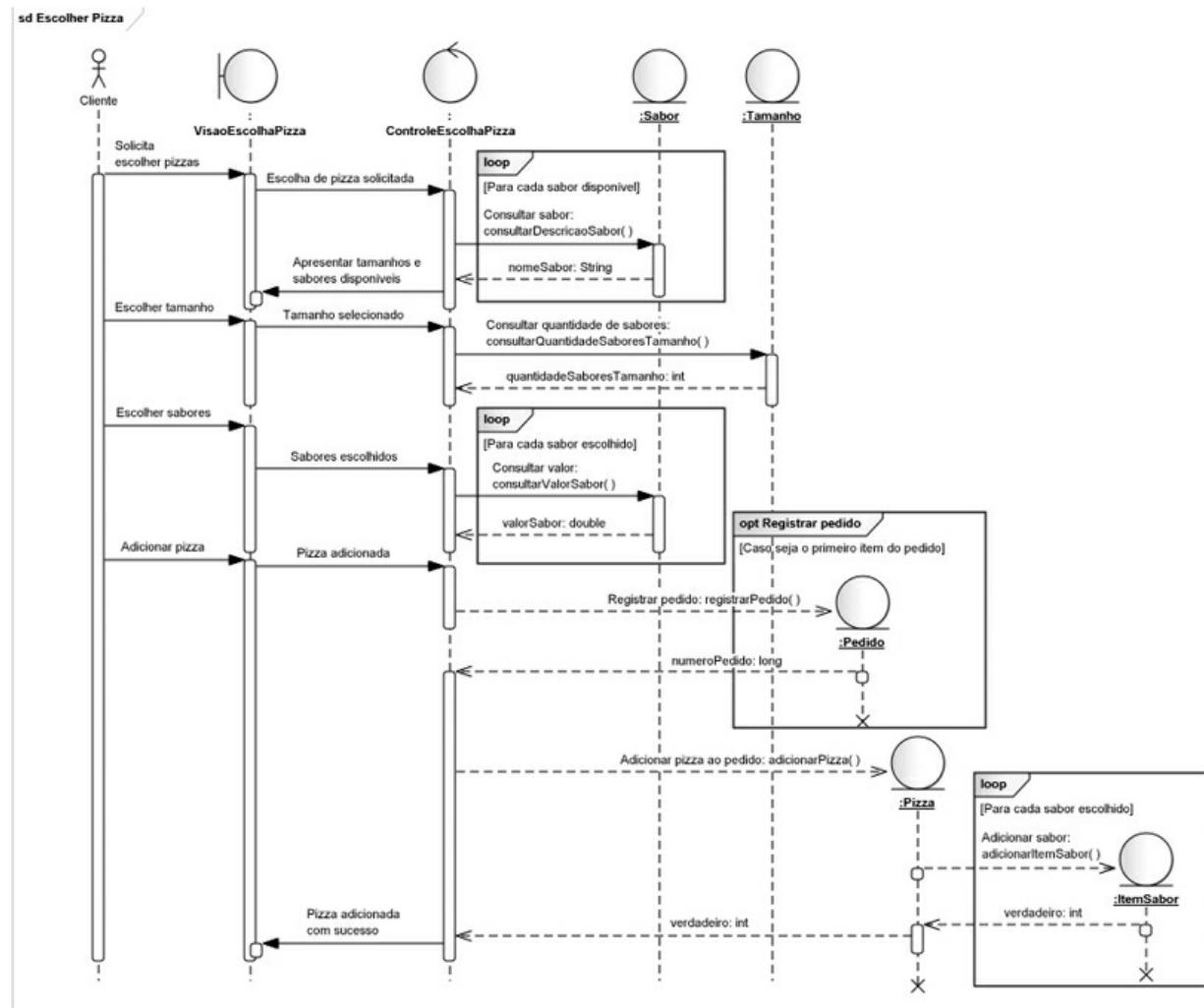
Nesta seção, apresentaremos os diagramas de sequência referentes a esse estudo de caso, que também são uma forma de documentar cada caso de uso apresentado no início deste capítulo. Talvez alguns dos métodos apresentados nos diagramas a seguir pudessem ser simplificados e algumas vezes poderia não ser necessária a chamada de outros métodos para complementá-los, podendo tais métodos ser absorvidos em alguns casos pelos métodos que os chamaram. No entanto, optamos por uma abordagem mais detalhada para facilitar a compreensão dos diagramas, mesmo porque os diagramas não estão baseados em nenhuma linguagem de programação específica, o que, aliás, é o correto.

Além disso, como foi dito anteriormente, a rigor deve haver um método **get** e um método **set** para cada atributo de uma classe, mas isso é impraticável em processos que contenham um número grande de atributos, além de desnecessário e mesmo não recomendado, uma vez que os diagramas são modelos de nível mais alto. Assim, utilizamos algumas vezes métodos como “registrar”, para instanciar um objeto, ou “consultar”, para retornar todas as informações de uma instância.

#### *Diagrama de Sequência Escolher Pizza*

A figura 17.8 apresenta o detalhamento do processo **Escolher Pizza** por

meio de um diagrama de sequência.



*Figura 17.8 – Diagrama de Sequência Escolher Pizza.*

Esse processo inicia-se quando o cliente acessa a página da PizzaNet, por esta ser a opção default; ou quando clica o botão Pizzas da interface. Ao ser avisado da ocorrência desse evento, o controlador dispara o método `consultarDescricaoSabor` em cada objeto da classe `Sabor`, para retornar sua descrição. Observe que existe um fragmento combinado do tipo `loop`, que demonstra que isso ocorre por meio de um laço. A partir das descrições retornadas pelas chamadas desse método, o controlador manda carregar na página os tamanhos e os sabores disponíveis.

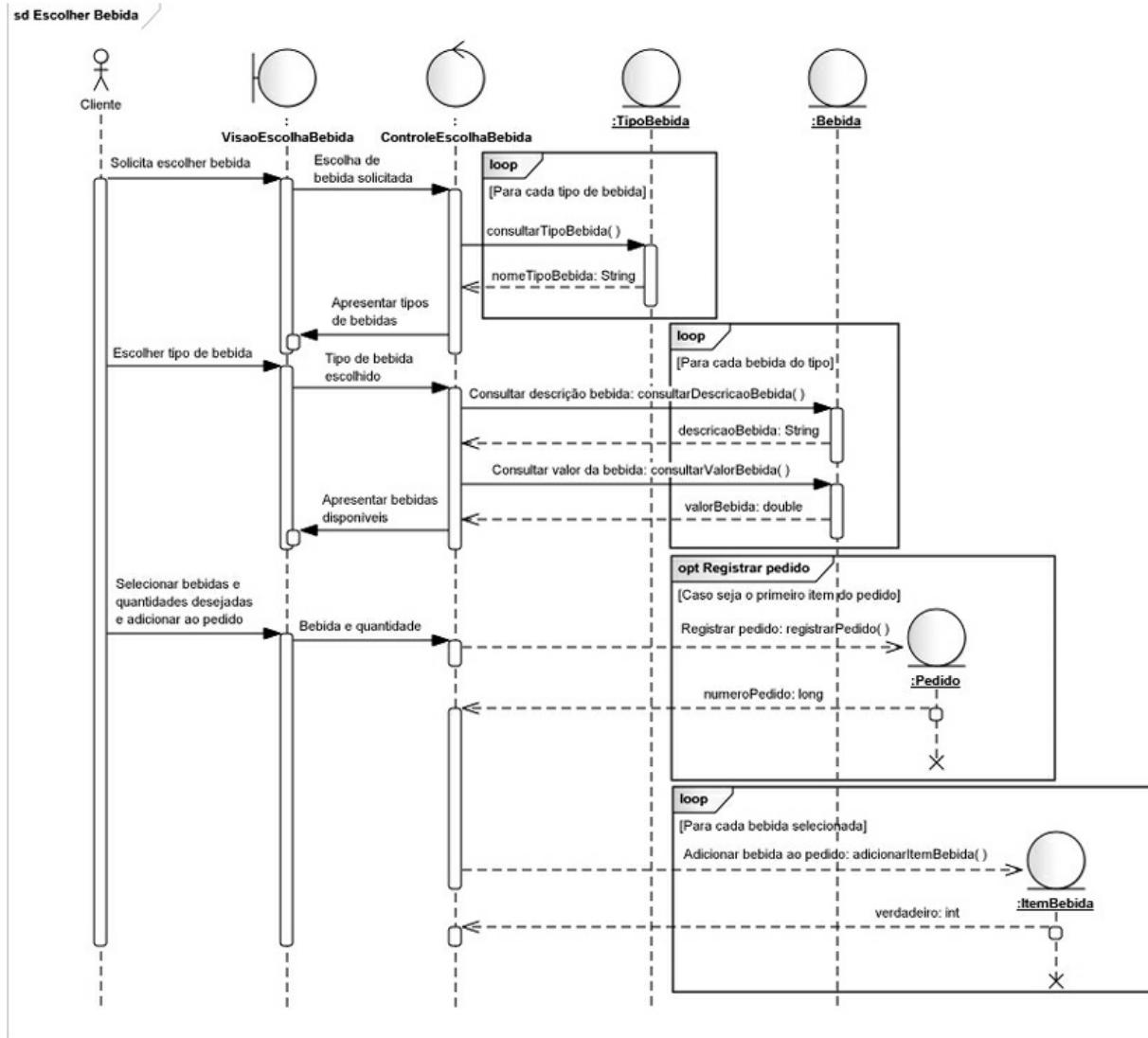
O cliente deve, então, selecionar o tamanho desejado da pizza, que é repassado para o controlador que dispara o método `consultarQuantidadeSaboresTamanho` para retornar a quantidade de

sabores que podem ser pedidos para o tamanho escolhido. Depois disso, o cliente poderá escolher os sabores desejados para a pizza, simplesmente clicando sobre o sabor. A escolha desses sabores é repassada para o controlador que executará um laço, disparando o método `consultarValorSabor` para cada sabor escolhido, recuperando o valor do sabor. Quando o cliente tiver terminado de escolher os sabores da pizza, bastará pressionar o botão adicionar. Esse evento obrigará o controlador a realizar um teste para determinar se a pizza escolhida é o primeiro item do pedido, conforme demonstra o fragmento combinado do tipo `opt`. Caso o teste seja positivo, o controlador disparará o método `registrarPedido` que instanciará um novo objeto da classe `Pedido` e retornará o número deste.

O controlador, então, executará o método `adicionarPizza` para adicionar a nova pizza ao pedido. Este é um método construtor, que instanciará um novo objeto da classe `Pizza` e chamará o método `adicionarItemSabor` para instanciar os objetos da classe `ItemSabor` associados à pizza. O método será disparado tantas vezes quantos forem os sabores escolhidos, como demonstra o fragmento combinado do tipo `loop`. Esse método é também um método construtor. Se o retorno desses métodos for verdadeiro, será apresentada a mensagem de “Pizza adicionada com sucesso”.

#### *Diagrama de Sequência Escolher Bebida*

Esse processo se inicia quando o cliente pressiona o botão **Bebidas** na página da PizzaNet. Esse evento é repassado pela página à controladora que ordena à página que seja apresentado o formulário para escolha de bebidas, contendo os tipos de bebidas oferecidos pela pizzaria. Atualmente, os tipos são suco, refrigerante e cerveja, mas, futuramente, poderão ser acrescentados novos tipos de bebidas. Por esse motivo, todos os objetos da classe `TipoBebida` são recuperados e apresentados (Figura 17.9).



*Figura 17.9 – Diagrama de Sequência Escolher Bebida.*

Nesse formulário, o cliente deve escolher o tipo de bebida que deseja. Esse evento faz o controlador consultar todas as bebidas, com seus respectivos valores, do tipo selecionado. Isso é feito por meio de um laço, representado pelo fragmento combinado do tipo **loop**, no qual são executados os métodos **consultarDescricaoBebida** e **consultarValorBebida** para retornar a descrição e o valor de cada bebida do tipo escolhido. Com essas informações, o controlador mandará atualizar a página, apresentando todas as bebidas disponíveis ao cliente.

Em seguida, o cliente deverá selecionar a bebida e a quantidade desejada e pressionar o botão **Adicionar**, para acrescentar a bebida ao pedido. Esse evento fará o controlador realizar novamente o teste descrito no processo

anterior, para determinar se este é o primeiro item do pedido do cliente, conforme demonstra o fragmento combinado do tipo **loop**. Caso isso seja verdadeiro, o controlador disparará o método `registrarPedido` em um objeto da classe `Pedido`, instanciando-o e retornando o número do novo pedido.

Após esse teste, será executado outro laço, no qual o controlador chamará o método `adicionarItemBebida` para instanciar um novo objeto da classe `ItemBebida`, relativo a cada bebida que foi adicionada ao pedido. O retorno verdadeiro (na verdade, o valor 1) significará que a bebida em questão foi adicionada com sucesso.

### *Diagrama de Sequência Logar*

Esse processo pode ser chamado pelo cliente quando este clicar o botão **Logar** ou pode ser chamado pelo processo **Concluir Pedido**, caso o cliente não tenha se autenticado ainda. A chamada a esse processo faz a controladora solicitar a apresentação do formulário de login na página da PizzaNet (Figura 17.10).

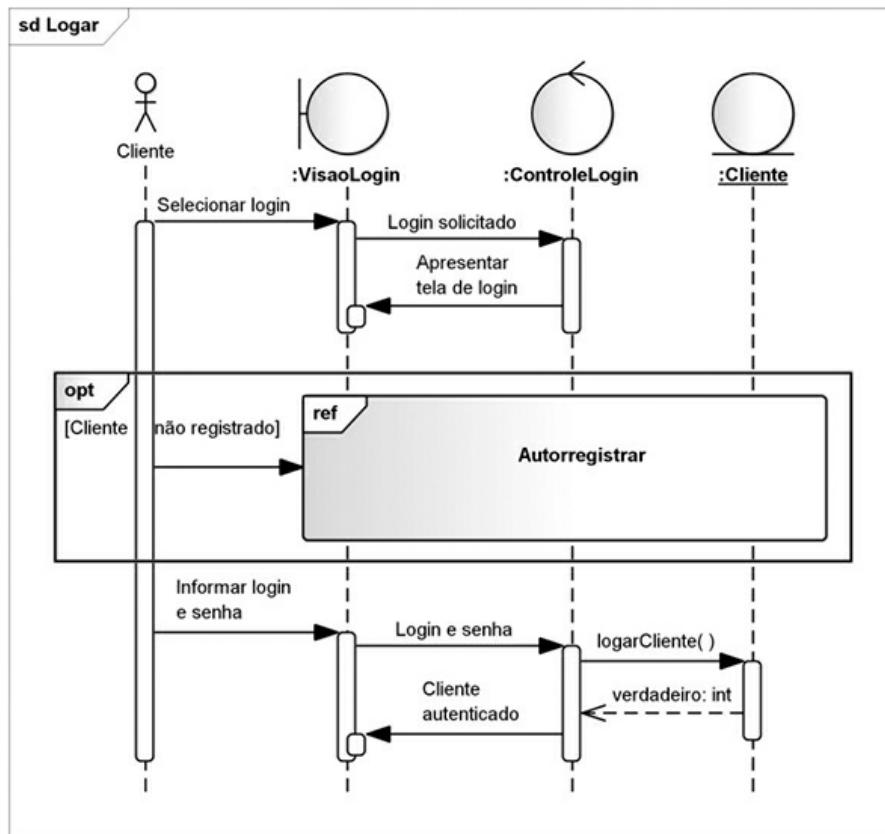


Figura 17.10 – Diagrama de Sequência Logar.

Caso o cliente não esteja cadastrado no sistema, ele terá a opção de se autorregular, conforme é demonstrado pelo fragmento combinado do tipo **opt**, que envolve o comportamento que será executado caso o cliente não esteja registrado ainda no sistema, ou seja, o processo de **Autorregular**, que será chamado pelo uso de interação que referencia esse processo.

Se já estiver registrado, o cliente informará, então, seu login e senha, fazendo o controlador disparar o método **logarCliente**. O retorno do valor verdadeiro para o controlador fará que este ordene a apresentação da mensagem “Cliente autenticado”.

#### *Diagrama de Sequência Autorregular*

Este é um processo bastante simples. No momento em que o cliente solicita a execução desse processo, a controladora, em resposta, solicita a apresentação do formulário de registro, onde o cliente deverá inserir os seus dados e confirmar. Ao receber a confirmação, o controlador disparará o método **registrarCliente** e instanciará um novo objeto da classe **Cliente** (Figura 17.11)..

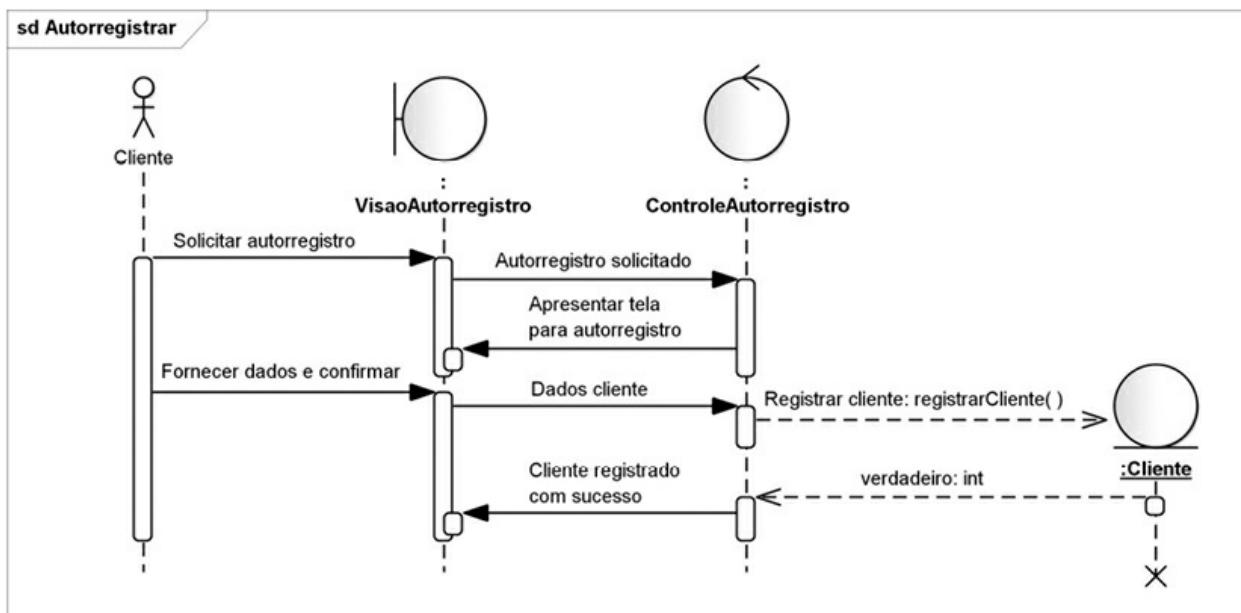


Figura 17.11 – Diagrama de Sequência Autorregular.

#### *Diagrama de Sequência Opinar*

Este é também um processo muito simples, em que, ao receber a solicitação de execução dessa funcionalidade, a controladora ordena que

seja apresentado o formulário para registro de opiniões. Ao receber a opinião do cliente, a controladora executará o método `registrarOpiniao`, para instanciar um novo objeto da classe `Opiniao`, e o processo estará finalizado (Figura 17.12).

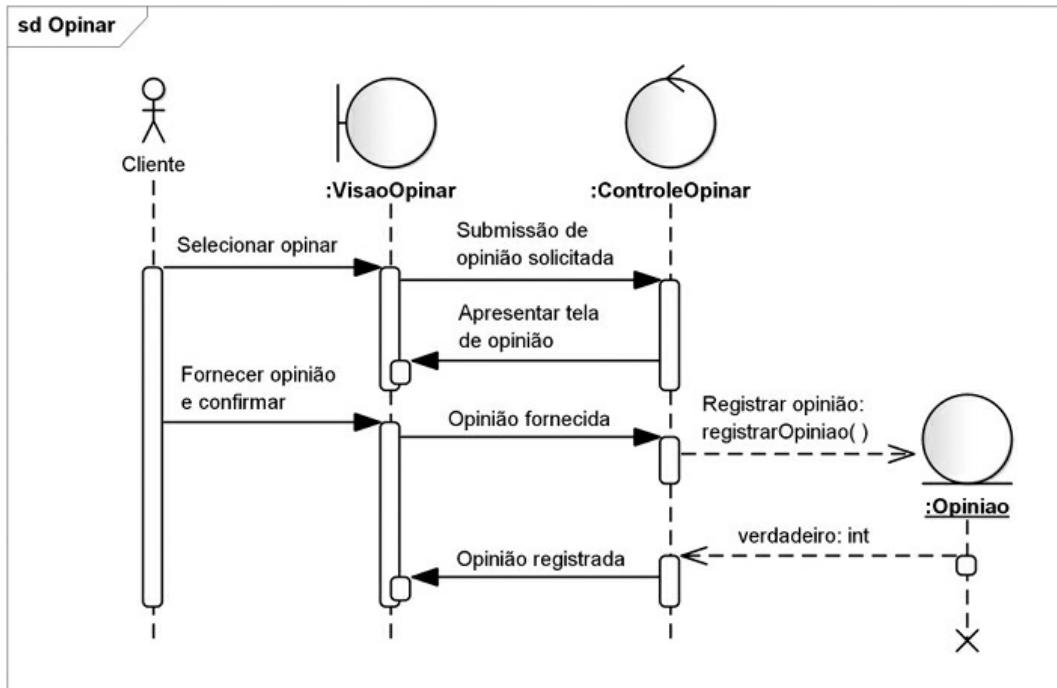


Figura 17.12 – Diagrama de Sequência Opinar.

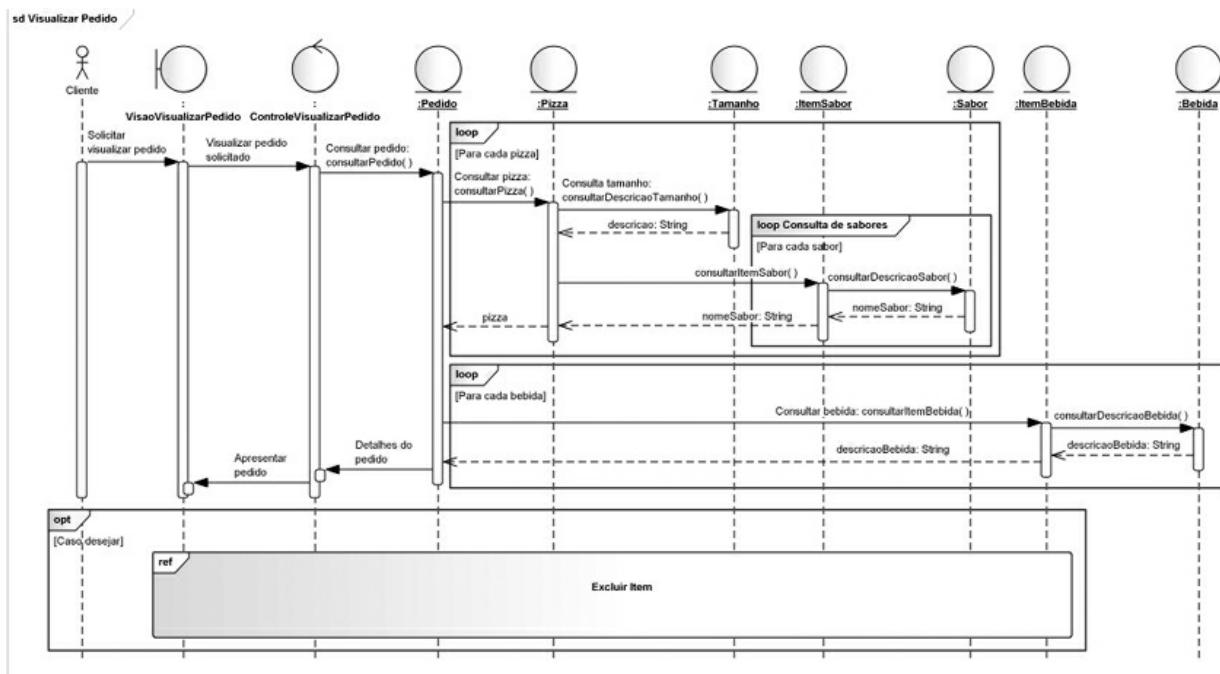
#### *Diagrama de Sequência Visualizar Pedido*

Quando o cliente solicita a visualização dos itens de seu pedido, o controlador dispara o método `consultarPedido` em um objeto da classe `Pedido`, para retornar as informações do pedido feito pelo cliente, ou seja, as pizzas e bebidas solicitadas.

Como os objetos da classe `Pedido` precisam que suas informações sejam complementadas por objetos da classe `Pizza` e `ItemBebida`, o método `consultarPedido` dispara o método `consultarPizza` para consultar cada pizza associada ao pedido, como demonstra o fragmento combinado do tipo `loop`. Por sua vez, esse método dispara o método `consultarDescricaoTamanho` em um objeto da classe `Tamanho`, para retornar o tamanho da pizza consultada e, em seguida, chama o método `consultarItemSabor` em cada objeto da classe `ItemSabor` associado à pizza, como demonstra o segundo fragmento combinado do tipo `loop`,

uma vez que um objeto da classe **Pizza** também precisa que suas informações sejam complementadas pelos objetos dessa classe. Ainda dentro desse laço, o método **consultarItemSabor** chama o método **consultarDescricaoSabor** para retornar a descrição do sabor relacionado ao objeto **ItemSabor** em cada iteração do laço. Os resultados da consulta de cada pizza são retornados na forma de uma **String** para o método **consultarPedido** (Figura 17.13).

O método **consultarPedido**, então, inicia um segundo laço, demonstrado por mais um fragmento combinado do tipo **loop**, disparando o método **consultarItemBebida** em cada objeto da classe **ItemBebida** relacionado ao pedido. Por sua vez, esse método chama o método **consultarDescricaoBebida** para retornar a descrição da bebida. As informações de cada bebida são, então, retornadas ao método **consultarPedido** que, em posse de todas essas informações, retorna-as ao controlador que as manda apresentar na página da PizzaNet.



*Figura 17.13 – Diagrama de Sequência Visualizar Pedido.*

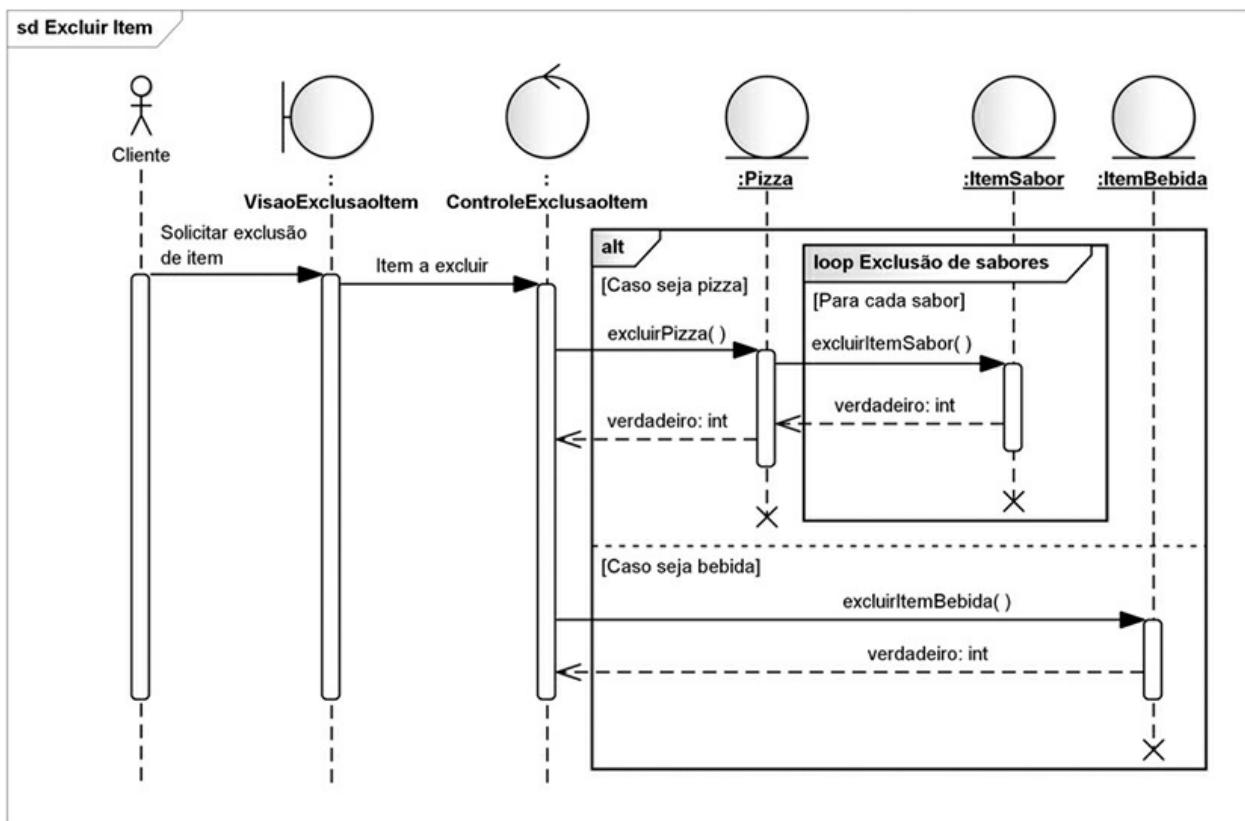
A partir dessa listagem, o cliente pode excluir qualquer um dos itens apresentados. Como isso é opcional, esse comportamento é apresentado por meio de um fragmento combinado do tipo **opt**, que contém um uso de interação que chama o processo de **Excluir Item**.

### Diagrama de Sequência Excluir Item

Esse processo é chamado a partir do processo anterior, e quando se inicia, a controladora precisa escolher entre dois comportamentos possíveis, como demonstra o fragmento combinado do tipo **alt**.

Assim, se o item a excluir for uma pizza, a controladora dispara o método **excluirPizza** para excluir a pizza em questão. Esse método, por sua vez, dispara o método **excluirItemSabor** em cada objeto da classe **ItemSabor** associado à pizza. Isso é necessário porque existe uma associação de composição entre a classe **Pizza** e a classe **ItemSabor** e, por esse motivo, quando um objeto da classe **Pizza** for excluído, devem ser excluídos também todos os objetos **ItemSabor** a ele associados. Observe que esses dois métodos são destrutores, como demonstra o “X” que interrompe a linha de vida dos objetos (Figura 17.14).

Já se o item a excluir for uma bebida, o controlador chamará o método **excluirItemBebida** para destruir o objeto da classe **ItemBebida**. Este também é um método destrutor, como também demonstra o “X” que interrompe a linha de vida do objeto da classe **ItemBebida**.



*Figura 17.14 – Diagrama de Sequência Excluir Item.*

#### *Diagrama de Sequência Visualizar Pedidos Anteriores*

Essa funcionalidade só pode ser solicitada se o cliente estiver autenticado. Quando o cliente seleciona essa opção, a controladora entra em um laço, demonstrado por um fragmento combinado do tipo **loop**, em que disparará o método **visualizarPedidosAbertos** para cada objeto da classe **Pedido** associado ao cliente. Esse método retornará o número de um pedido e, com esse número, chama-se o processo **Visualizar Pedido** aqui referenciado por um uso de interação, que apresentará os detalhes do pedido (Figura 17.15).

A partir da listagem dos pedidos anteriores, o cliente tem a opção de solicitar um pedido novamente. Como isso é opcional, esse comportamento é representado por um fragmento combinado do tipo **opt**. Nesse caso, o cliente deverá selecionar um pedido, o que fará a controladora disparar o método **registrarPedido** para instanciar um novo pedido.

Após obter o número do novo pedido, a controladora dispara o método **adicionarPizza** para instanciar um novo objeto da classe **Pizza**. Isso é feito para cada pizza contida no pedido anterior escolhido, como demonstra o fragmento combinado do tipo **loop**. Por sua vez, esse método inicia um novo laço representado por um segundo fragmento combinado do tipo **loop**, disparando o método **adicionarItemSabor** para instanciar um novo objeto da classe **ItemSabor**. Esse laço é executado tantas vezes quantos forem os sabores da pizza.

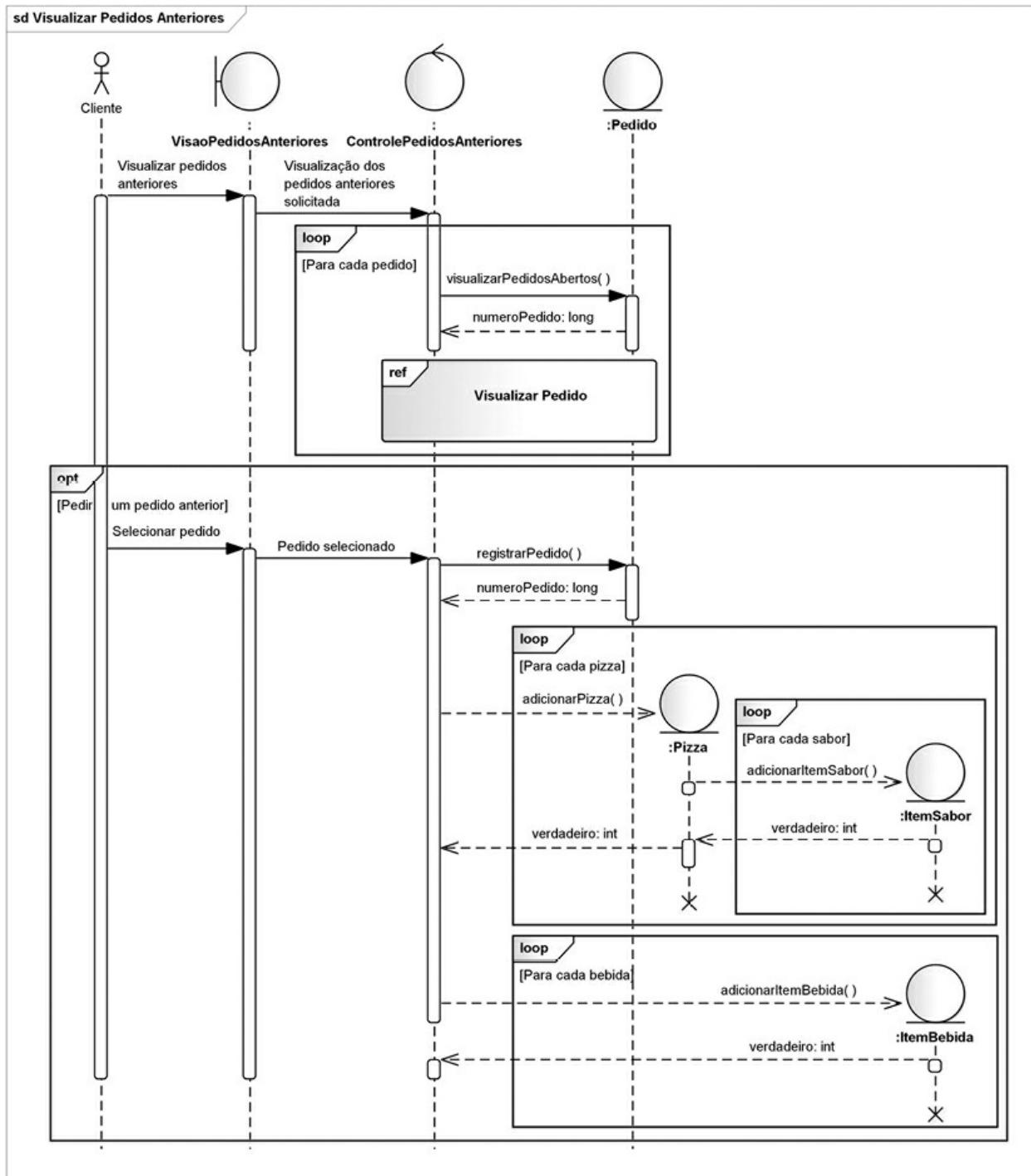
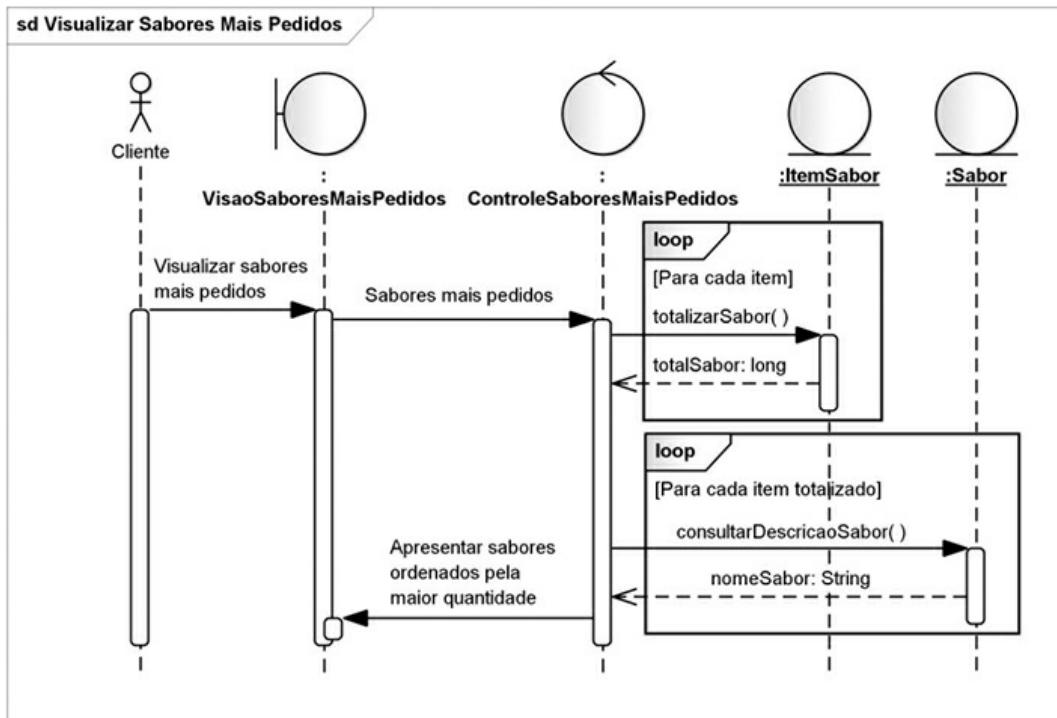


Figura 17.15 – Diagrama de Sequência Visualizar Pedidos Anteriores.

Após instanciar as pizzas do novo pedido, a controladora inicia um novo laço, representado por mais um fragmento combinado do tipo loop, onde é disparado o método `adicionarItemBebida`, instanciando um objeto da classe `ItemBebida` para cada bebida contida no pedido anterior.

### *Diagrama de Sequência Visualizar Sabores Mais Pedidos*

Nesse processo, a controladora executa um laço representado por um fragmento combinado do tipo **loop**, onde é chamado o método **totalizarSabor**. Esse método soma todos os objetos da classe **ItemSabor** associados a um determinado sabor, retornando seu total quando o sabor muda e reiniciando a contagem (Figura 17.16).



*Figura 17.16 – Diagrama de Sequência Visualizar Sabores Mais Pedidos.*

Ao receber esses totais, a controladora inicia outro laço para consultar a descrição de cada sabor totalizado, por meio do método **consultarDescricaoSabor**, que retorna uma **String** contendo a descrição do sabor consultado.

### *Diagrama de Sequência Concluir Pedido*

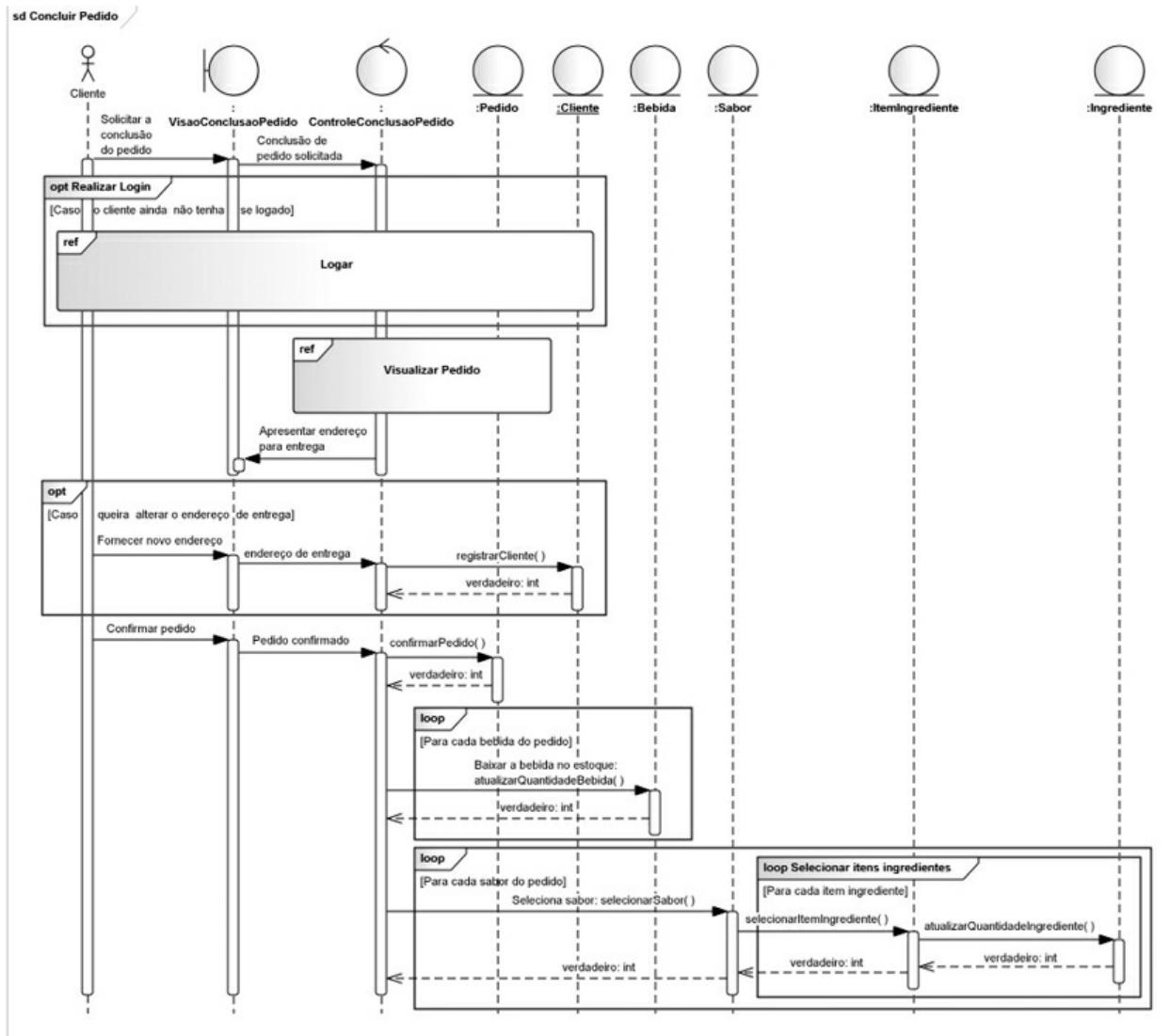
Ao receber a solicitação de conclusão do pedido do cliente, a controladora primeiramente necessita verificar se o cliente já se encontra autenticado, conforme demonstra o fragmento combinado do tipo **opt**. Caso o cliente ainda não tenha se logado, é necessário executar o processo de **Logar**, referenciado por meio do uso de interação contida dentro do fragmento (Figura 17.17).

A seguir, é necessário executar o processo de **Visualizar Pedido**,

conforme demonstra a ocorrência de interação que o referencia, para ter certeza de que é isso mesmo que o cliente deseja pedir. Depois, a controladora ordena à página que apresente o endereço de entrega do cliente.

Em seguida, é preciso realizar outro teste, demonstrado pelo fragmento combinado do tipo **opt**, no qual se deve verificar se o cliente acha necessário modificar o endereço de entrega. Se o cliente assim o desejar, deverá informar o novo endereço e confirmar. Quando isso ocorrer, a controladora disparará o método **registrarCliente** para atualizar os dados do cliente.

Depois disso, quando o cliente confirmar o pedido pela última vez, o controlador disparará o método **confirmarPedido** que definirá o pedido como confirmado, alterando o valor da situação do pedido para 1, o que significa que o cliente confirmou o pedido e este deve ser atendido.



*Figura 17.17 – Diagrama de Sequência Concluir Pedido.*

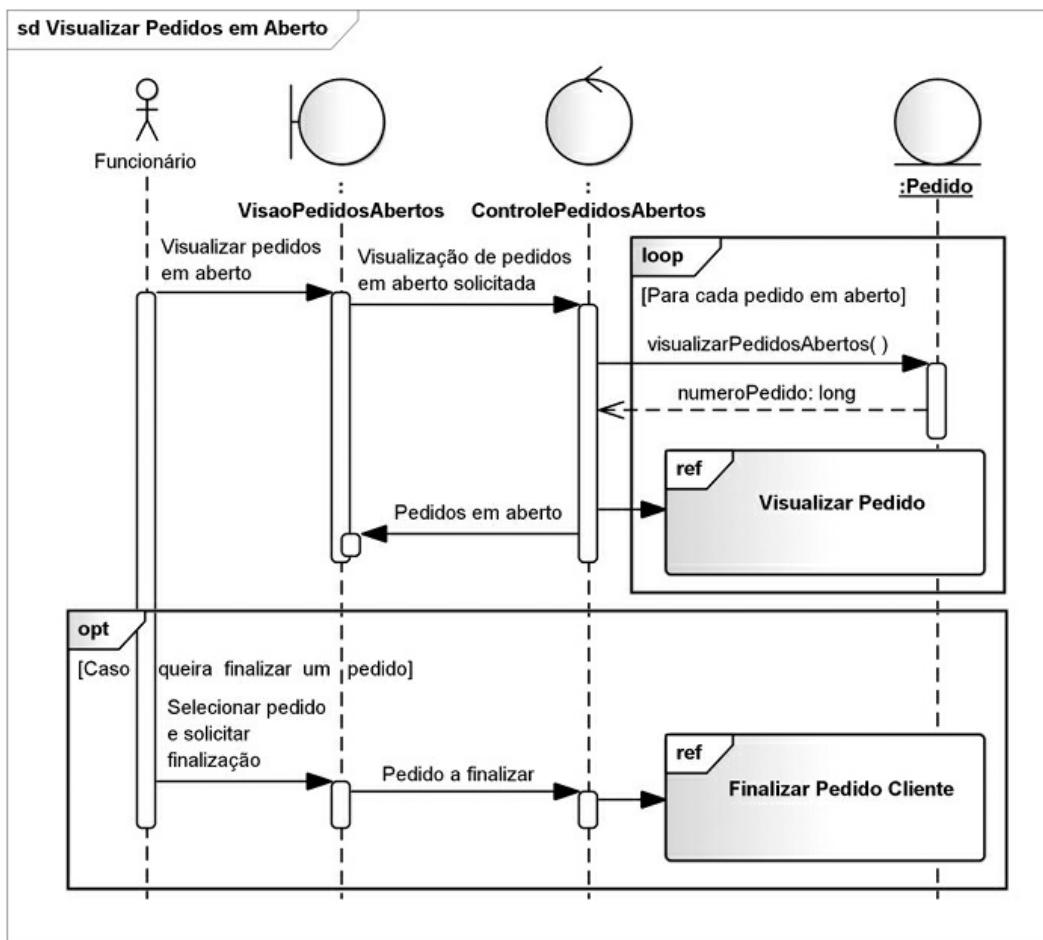
A seguir, a controladora inicia um laço, representado por um fragmento combinado do tipo **loop**, em que a quantidade solicitada de cada bebida será diminuída do estoque da bebida em questão, por meio do disparo do método **atualizarQuantidadeBebida**.

Ao terminar essa operação, a controladora inicia um novo laço, disparando o método **selecionarSabor** para cada sabor solicitado no pedido. Por sua vez, esse método executa o método **selecionarItemIngrediente**, selecionando cada um dos objetos da classe **ItemIngrediente** associados ao objeto da classe **Sabor** selecionado. Finalmente, para cada objeto da classe **ItemIngrediente** selecionado, o método **selecionarItemIngrediente** disparará o método

`atualizarQuantidadeIngrediente` que diminuirá a quantidade contida no objeto da classe `ItemIngrediente` da quantidade contida no objeto da classe `Ingrediente` a ele associado.

### *Diagrama de Sequência Visualizar Pedidos em Aberto*

Este é um processo disparado pelo funcionário, uma vez que agora começaremos a modelar os processos do subsistema administrativo. Quando o funcionário solicita esse serviço, o controlador inicia um laço, representado por um fragmento combinado do tipo **loop**, em que, para cada pedido em aberto, será disparado o método `visualizarPedidosAbertos`. Esse método retornará o número de cada pedido em aberto e, na posse dele, a controladora solicitará a execução do processo **Visualizar Pedido**, descrito anteriormente, que apresentará os detalhes do pedido, como demonstra o uso de interação de mesmo nome (Figura 17.18).



*Figura 17.18 – Diagrama de Sequência Visualizar Pedidos em Aberto.*

A seguir, o funcionário pode, opcionalmente, finalizar um pedido em aberto, como demonstra o fragmento combinado do tipo **opt**. Nessa situação, o funcionário deverá selecionar o pedido em questão e solicitar sua finalização. Isso fará a controladora ordenar a execução do processo **Finalizar Pedido Cliente**, conforme demonstra o uso de interação. Esse novo processo será descrito na próxima seção.

*Diagrama de Sequência Finalizar Pedido Cliente*

Por esse processo ser chamado a partir do processo de **Visualizar Pedidos em Aberto**, ele se inicia com a controladora executando um laço, como demonstra o fragmento combinado do tipo **loop**, em que é disparado o método **consultarFuncionario** para consultar todos os funcionários da empresa, retornando seus nomes para serem apresentados no formulário de finalização de pedido (Figura 17.19).

Por meio desse formulário, o funcionário deve informar os funcionários que prepararam e entregaram o pedido, fazendo a controladora disparar o método **finalizarPedido** em um objeto da classe **Pedido**, definindo-o como finalizado, por meio da mudança do valor da situação do pedido para 2.

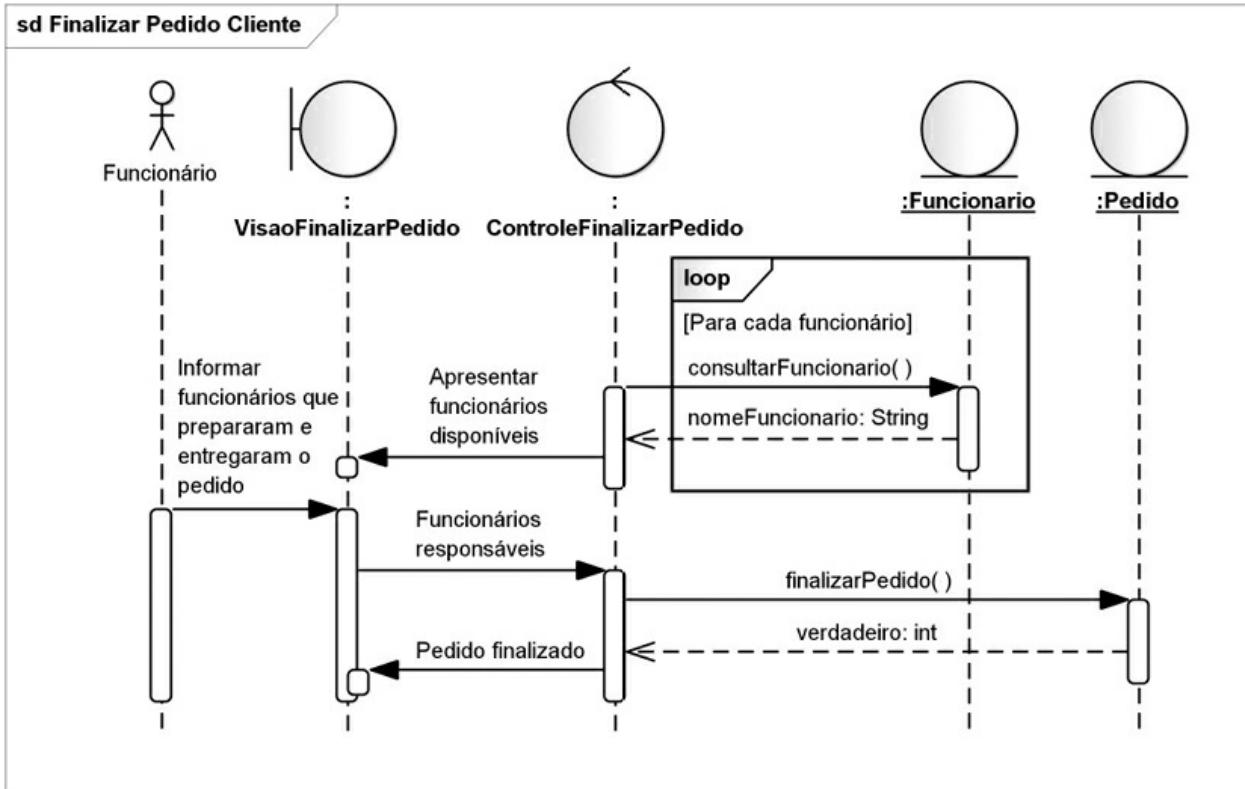


Figura 17.19 – Diagrama de Sequência Finalizar Pedido Cliente.

#### Diagrama de Sequência Gerenciar Cardápio

Esta é uma funcionalidade que só pode ser solicitada pelo administrador. Quando este a solicita, a controladora, em resposta, inicia um laço demonstrado pelo fragmento combinado do tipo **loop**, em que é disparado o método **consultarDescricaoSabor** para recuperar a descrição de cada sabor registrado no sistema.

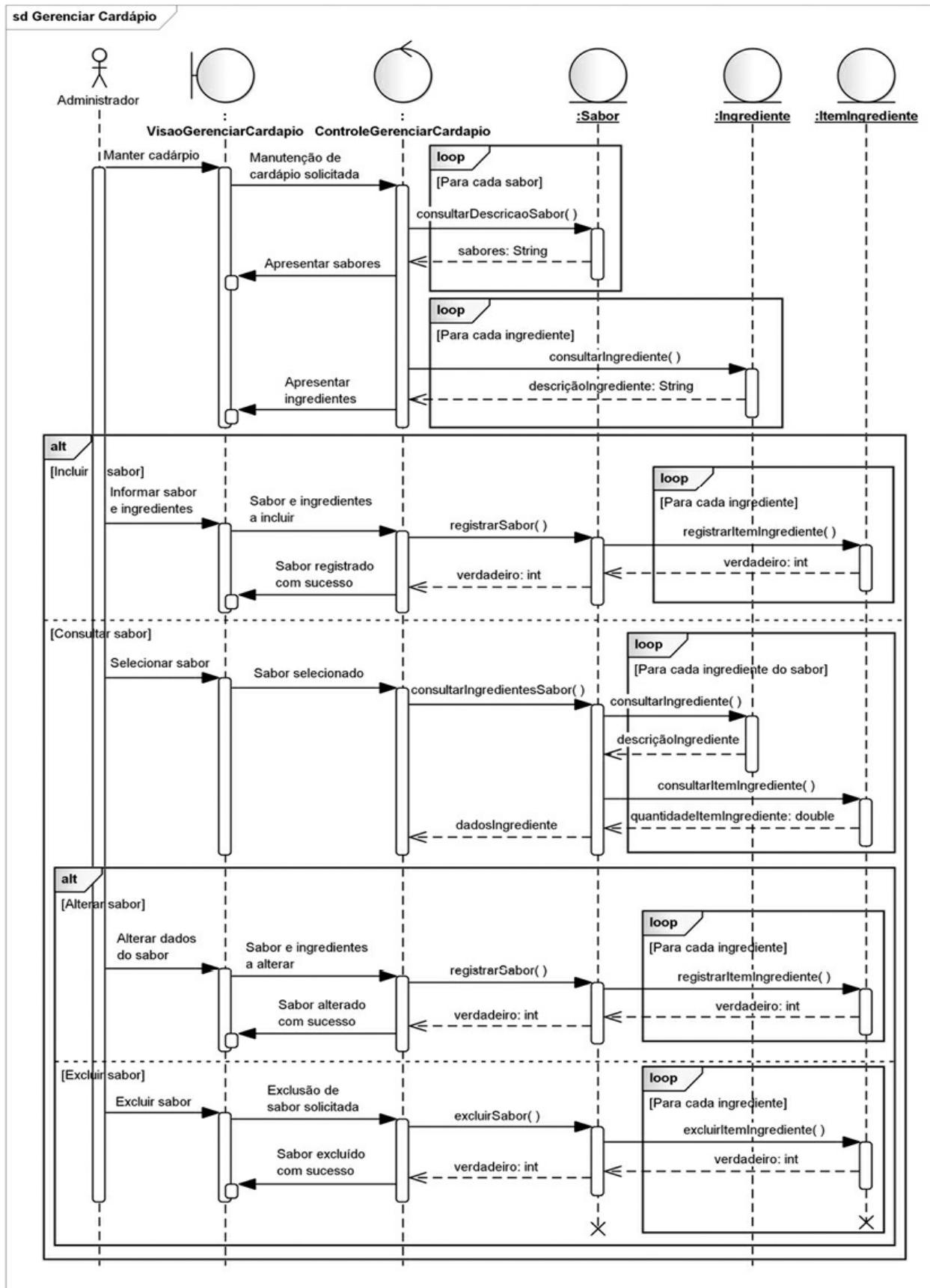
A seguir, a controladora inicia um segundo laço, onde dispara o método **consultarIngrediente** para retornar a descrição de cada ingrediente cadastrado. Com essas informações, a controladora solicita à interface que apresente o formulário de manutenção de cardápio ao administrador (Figura 17.20).

Com esse formulário, o administrador precisa escolher incluir um novo sabor ou consultar um sabor já existente, como demonstra o fragmento combinado do tipo **alt**.

Na primeira alternativa, o administrador deverá informar o novo sabor e os ingredientes necessários à sua produção, o que fará o controlador disparar o método **registrarSabor**, instanciando um novo objeto da

classe **Sabor**. Embora o correto ao inserir uma nova lifeline (objeto) da classe **Sabor** fosse que este só passasse a existir a partir do momento de sua criação, optamos por aproveitar a lifeline anterior dessa classe, para não deixar o diagrama largo demais.

O método **registrarSabor**, por sua vez, inicia um loop, onde dispara o método **registrarItemIngrediente**, instanciando um novo objeto da classe **ItemIngrediente** para cada ingrediente necessário à produção do sabor. Se o retorno dos métodos for verdadeiro, o controlador mandará a interface apresentar a mensagem de **Sabor registrado com sucesso**.



*Figura 17.20 – Diagrama de Sequência Gerenciar Cardápio.*

Na segunda alternativa, o administrador deverá selecionar um dos sabores listados no formulário. Quando isso ocorre, a controladora dispara o método **consultarIngredientesSabor** em uma lifeline da classe **Sabor** para consultar os ingredientes do sabor selecionado. Para que esse método possa realizar essa tarefa, deve iniciar um laço, onde dispara o método **consultarItemIngrediente** em todas as lifelines da classe **ItemIngrediente** associadas à lifeline **Sabor** em questão. Esse método, por sua vez, chama o método **consultarIngrediente** para retornar a descrição da lifeline da classe **Ingrediente** associada a cada lifeline **ItemIngrediente**. O retorno desses métodos é enviado à controladora, que o manda apresentar na interface.

Com essas informações, o administrador poderá decidir alterar ou excluir um sabor, como demonstra o fragmento combinado do tipo **alt**.

Caso queira alterar a descrição de um sabor ou a quantidade de seus ingredientes, o administrador deverá executar as alterações sobre o formulário e confirmar. Isso fará a controladora executar o mesmo procedimento para registrar um novo sabor, disparando o método **registrarSabor** em uma lifeline da classe **Sabor** e o método **registrarItemIngrediente** em lifelines da classe **ItemIngrediente** para cada ingrediente do sabor.

Caso o administrador deseje excluir um sabor, a controladora disparará o método **excluirSabor** na lifeline da classe **Sabor** a ser destruído. Antes de finalizar, esse método executará o método **excluirItemIngrediente** para destruir todos os objetos da classe **ItemIngrediente** associados ao objeto da classe **Sabor**. Isso é necessário porque a classe **Sabor** tem uma associação de composição com a classe **ItemIngrediente**, sendo assim, quando um objeto **Sabor** for destruído, todos os objetos **ItemIngrediente** a ele associados deverão ser também destruídos.

#### *Diagrama de Sequência Emitir Produtos em Baixa no Estoque*

Quando o administrador solicita esse relatório, o controlador inicia dois laços, como demonstram os fragmentos combinados do tipo **loop** (Figura 17.21).

No primeiro laço, a controladora dispara o método

**consultarIngredientEmBaixa** em lifelines da classe **Ingrediente** para retornar todos os ingredientes com a quantidade em estoque abaixo da quantidade mínima.

No segundo laço, a controladora dispara o método **consultarBebidaEmBaixa** em lifelines da classe **Bebida** para retornar todas as bebidas com a quantidade em estoque abaixo do mínimo.

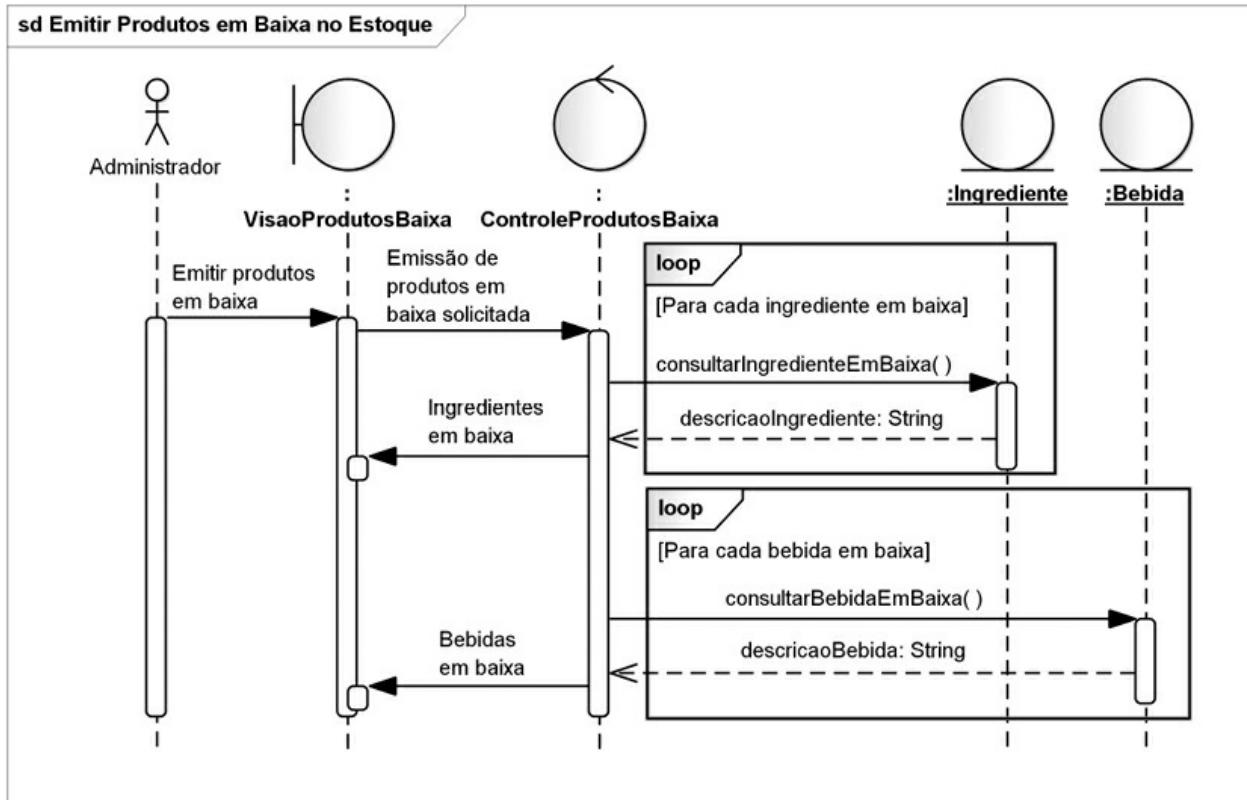


Figura 17.21 – Diagrama de Sequência *Emitir Produtos em Baixa no Estoque*.

#### *Diagrama de Sequência Emitir Compras em Aberto*

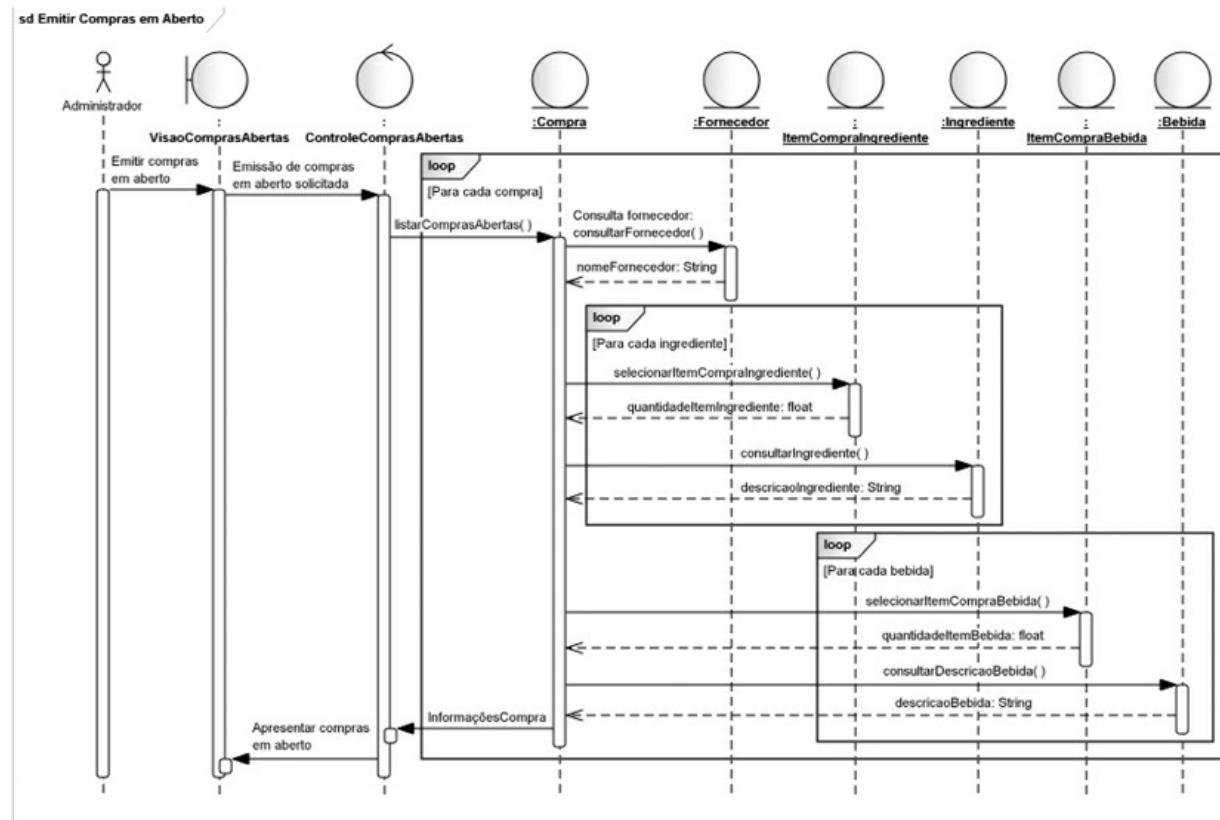
Quando esse relatório é solicitado, a controladora inicia um grande laço, representado por um fragmento combinado do tipo **loop**, onde é disparado o método **listarComprasAbertas** em cada lifeline da classe **Compra** cuja situação esteja em aberto. É necessário saber para qual fornecedor foi solicitada a compra. Para isso, o método **listarComprasAbertas** dispara o método **consultarFornecedor** em uma lifeline da classe **Fornecedor**, para retornar seus dados (Figura 17.22).

Em seguida, o método **listarComprasAbertas** inicia um novo laço, com

o objetivo de selecionar todos os ingredientes solicitados na compra. Para isso, é necessário disparar o método **selecionarItemCompraIngrediente** em cada objeto da classe **ItemCompraIngrediente** associado à compra em consulta no momento, além do método **consultarIngrediente** em uma lifeline da classe **Ingrediente** para recuperar a descrição de cada ingrediente.

Depois disso, o método **listarComprasAbertas** inicia, ainda, um último laço com o objetivo de selecionar todas as possíveis bebidas solicitadas na compra. Para isso, é chamado o método **selecionarItemCompraBebida** em lifelines da classe **ItemCompraBebida**. Após recuperar cada objeto dessa classe, é executado o método **consultarDescricaoBebida** para recuperar a descrição de cada bebida solicitada na compra.

Com base no retorno desses métodos, a controladora solicita que essas informações sejam apresentadas na interface.



*Figura 17.22 – Diagrama de Sequência Emitir Compras em Aberto.*

*Diagrama de Sequência Gerenciar Compras Fornecedores*

Quando o administrador solicita a execução do processo de gerenciamento de compras, a controladora dispara a execução do processo de emissão de compras em aberto, descrito na seção anterior, como demonstra a ocorrência de interação que se refere a esse processo. Com base nos resultados desse processo, a controladora solicita a apresentação do formulário de manutenção de compras, já carregado com as compras em aberto (Figura 17.23).

Nesse formulário, o administrador pode escolher finalizar ou registrar uma compra nova, como demonstra o fragmento combinado do tipo **alt**.

Na primeira alternativa, o administrador deve selecionar a compra que deseja finalizar e marcá-la como concluída, o que faz a controladora disparar o método **fecharCompra** em uma lifeline da classe **Compra**, modificando o estado de sua situação para 1, que significa que a compra está fechada.

Se a segunda alternativa for escolhida, a controladora, primeiramente, executará três laços, representados por fragmentos combinados do tipo **loop**, para prover informações ao formulário de gerenciamento de compras. No primeiro laço, a controladora disparará o método **consultarIngrediente** para retornar a descrição de todos os ingredientes registrados. No segundo laço, chamará o método **consultarDescricaoBebida** para retornar a descrição de cada bebida cadastrada. Finalmente, no terceiro laço, executará o método **consultarFornecedor**, para retornar o nome de todos os fornecedores da empresa.

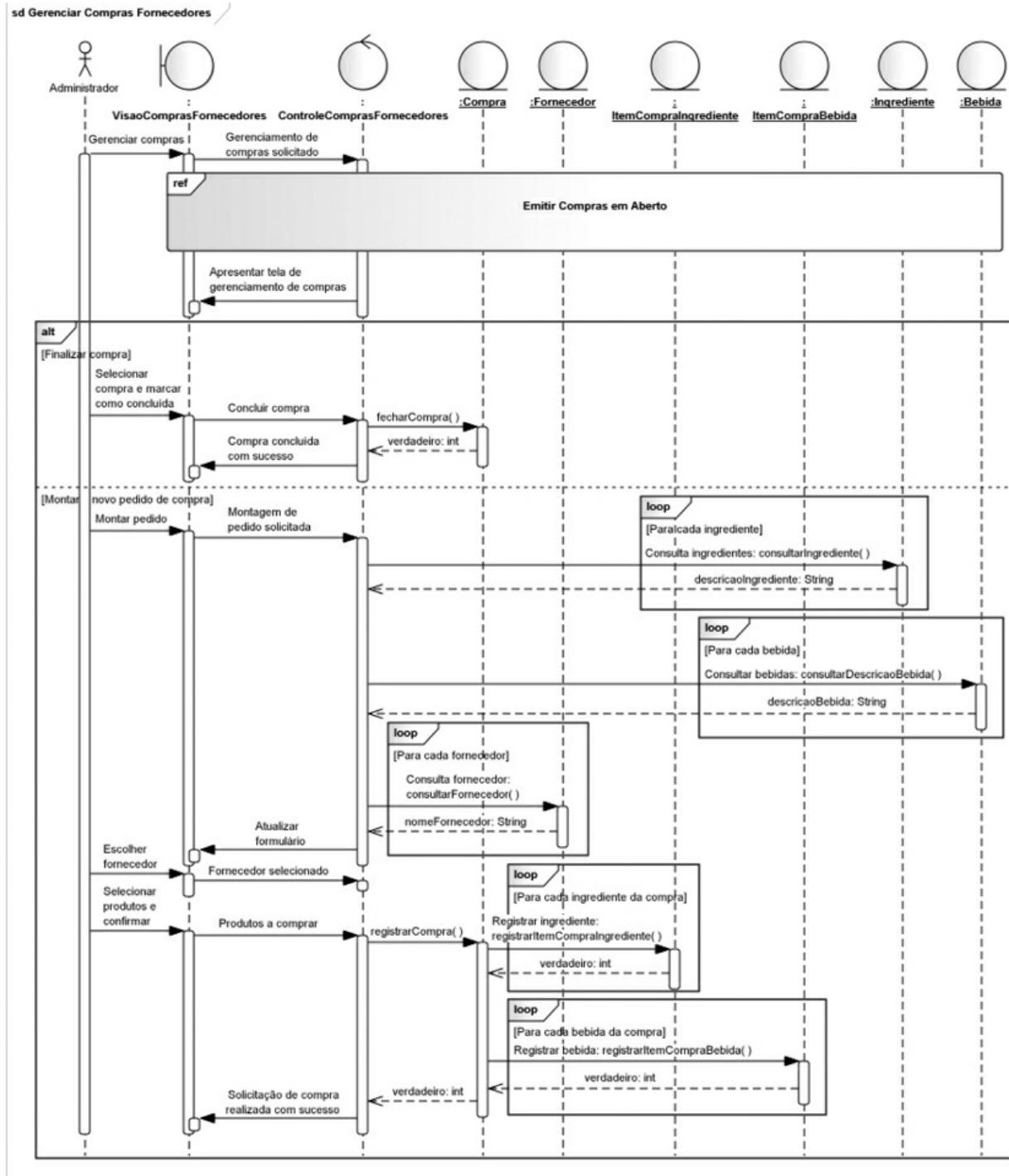


Figura 17.23 – Diagrama de Sequência Gerenciar Compras Fornecedores.

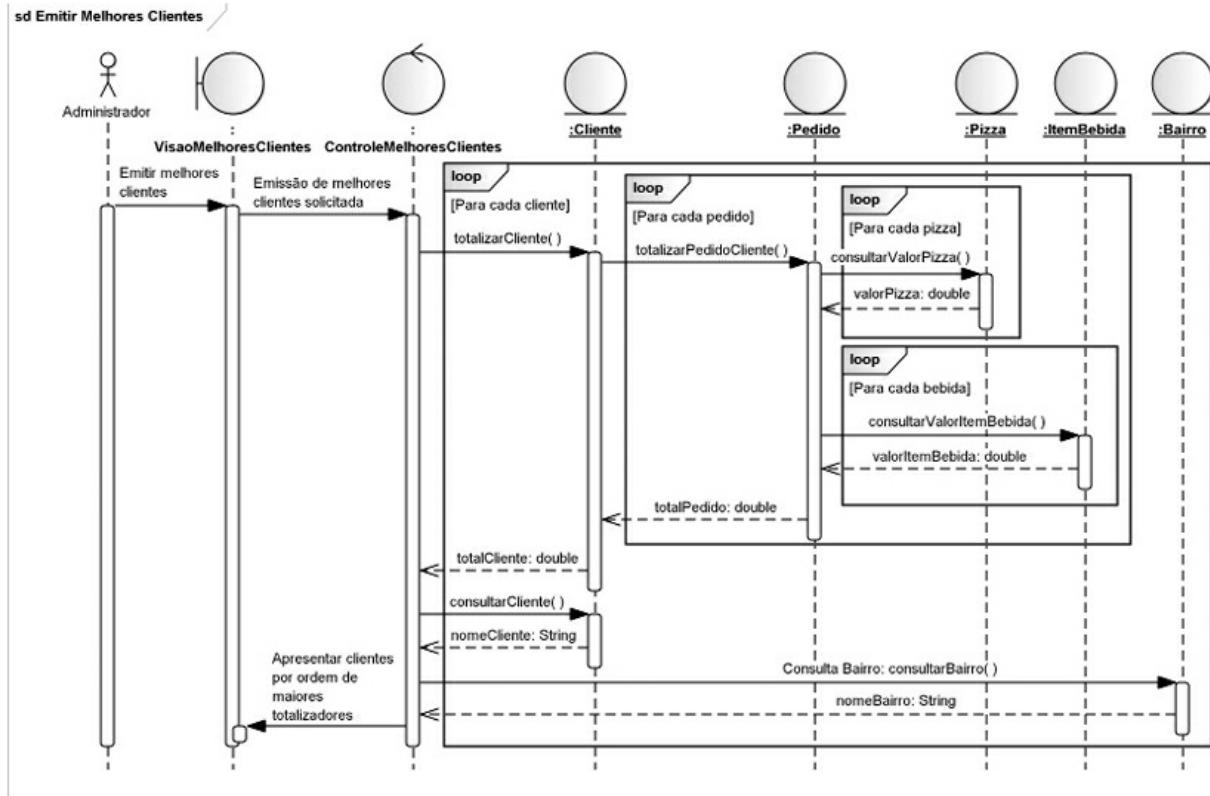
A seguir, o administrador poderá escolher o fornecedor ao qual deseja solicitar produtos e, em seguida, poderá selecionar os produtos que deseja comprar. Isso fará a controladora disparar o método `registrarCompra`, gerando um novo objeto da classe `Compra`.

Esse método, por sua vez, inicia um laço onde chamará o método **registrarItemCompraIngrediente** para instanciar um novo objeto da classe **ItemCompraIngrediente** para cada possível ingrediente que o administrador deseja solicitar. Após a finalização desse laço, é iniciado um segundo laço, onde será executado o método **registrarItemCompraBebida** para instanciar um novo objeto da classe **ItemCompraBebida** para cada possível bebida solicitada na compra.

Se o retorno desses métodos for verdadeiro, a controladora solicitará que a interface apresente uma mensagem informando que a solicitação de compra foi realizada com sucesso.

#### *Diagrama de Sequência Emitir Melhores Clientes*

Quando esse relatório é solicitado pelo administrador, a controladora inicia um grande laço, representado por um fragmento combinado do tipo **loop**, em que, para cada cliente, é disparado o método **totalizarCliente**, cujo objetivo é somar os valores de todos os pedidos já realizados por um cliente. Esse método, por sua vez, inicia um novo laço, disparando o método **totalizarPedidoCliente** para cada pedido feito por um cliente específico. Em resposta, esse método inicia dois laços: no primeiro é disparado o método **consultarValorPizza**, para retornar o valor de cada possível pizza do pedido, e no segundo laço é chamado o método **consultarValorItemBebida**, para retornar o valor de cada possível bebida do pedido. O método **totalizarPedidoCliente** totaliza os valores de cada pedido e retorna-os ao método **totalizarCliente**, que, por sua vez, totaliza todos os valores retornados relativos aos pedidos de um cliente específico e retorna-os à controladora (Figura 17.24).

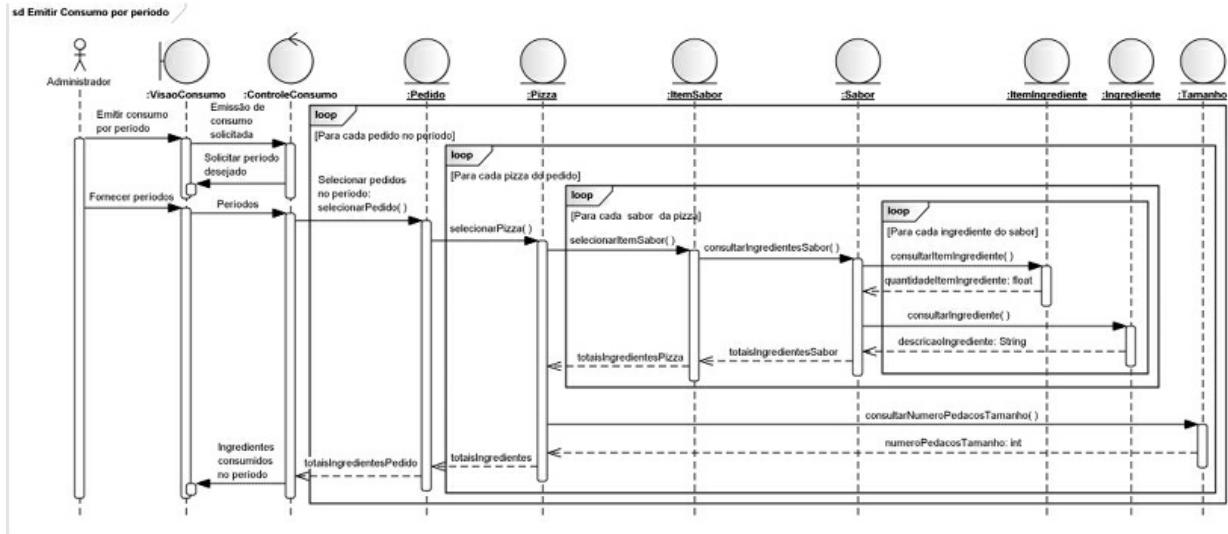


*Figura 17.24 – Diagrama de Sequência Emitir Melhores Clientes.*

Após obter o total de um cliente específico, a controladora consulta seu nome, por meio do disparo do método **consultarCliente**, e o bairro onde ele reside, mediante a execução do método **consultarBairro**, sendo assim possível saber os bairros dos melhores clientes da pizzaria. Esse processo é executado enquanto houver clientes registrados no sistema.

#### *Diagrama de Sequência Emitir Consumo por Período*

A solicitação desse relatório faz a controladora pedir à interface que solicite ao administrador os períodos iniciais e finais da listagem. O administrador deve, então, fornecer os períodos desejados e a controladora inicia um grande laço, representado por um fragmento combinado do tipo **loop**, no qual, para cada objeto da classe **Pedido**, será disparado o método **selecionarPedido**, para selecionar todos os pedidos que estejam dentro do período informado (Figura 17.25).



*Figura 17.25 – Diagrama de Sequência Emitir Consumo por Período.*

Sempre que um pedido é encontrado dentro do período informado, esse método inicia outro grande laço, para selecionar cada pizza do período, por meio da execução do método **selecionarPizza** sobre lifelines da classe **Pizza**.

Em seguida, o método **selecionarPizza** inicia um novo laço, disparando o método **selecionarItemSabor** para selecionar todos os objetos da classe **ItemSabor** associados à pizza. O método **SelecionarItemSabor**, por sua vez, dispara o método **consultarIngredientesSabor** na lifeline da classe **Sabor** associado ao objeto da classe **ItemSabor**, para retornar todos os ingredientes do sabor.

Para retornar a informação desejada, o método **ConsultarIngredientesSabor** inicia um novo laço para selecionar todos os objetos da classe **ItemIngrediente** associados ao objeto da classe **Sabor** consultado no momento. Isso é feito por meio do método **consultarItemIngrediente**. Dentro desse laço, também é disparado o método **consultarIngrediente** na lifeline da classe **Ingrediente** para retornar a descrição do ingrediente em questão.

Após receber os totais dos ingredientes consumidos por cada sabor, o método **selecionarPizza** ainda dispara um último método, **consultaNumeroPedacosTamanho**, em uma lifeline da classe **Tamanho**, para retornar o número de pedaços de cada lifeline da classe **Pizza**. Isso é necessário para calcular o número real de ingredientes gastos para preparar

cada pizza.

Os totais dos ingredientes gastos em cada pizza são retornados à controladora que os organiza e solicita sua apresentação na interface.

### 17.2.7 Diagrama de Comunicação Escolher Pizza

Nesta seção, apenas apresentaremos o diagrama de comunicação equivalente ao diagrama de sequência **Escolher Pizza**, explicado na seção 17.2.6. Optamos por apresentar somente esse processo porque esses diagramas demonstram praticamente as mesmas informações que os de sequência, mostradas de uma maneira diferente. Assim, esse diagrama tem um caráter basicamente ilustrativo de como utilizar o diagrama de comunicação para modelar o mesmo processo (Figura 17.26).

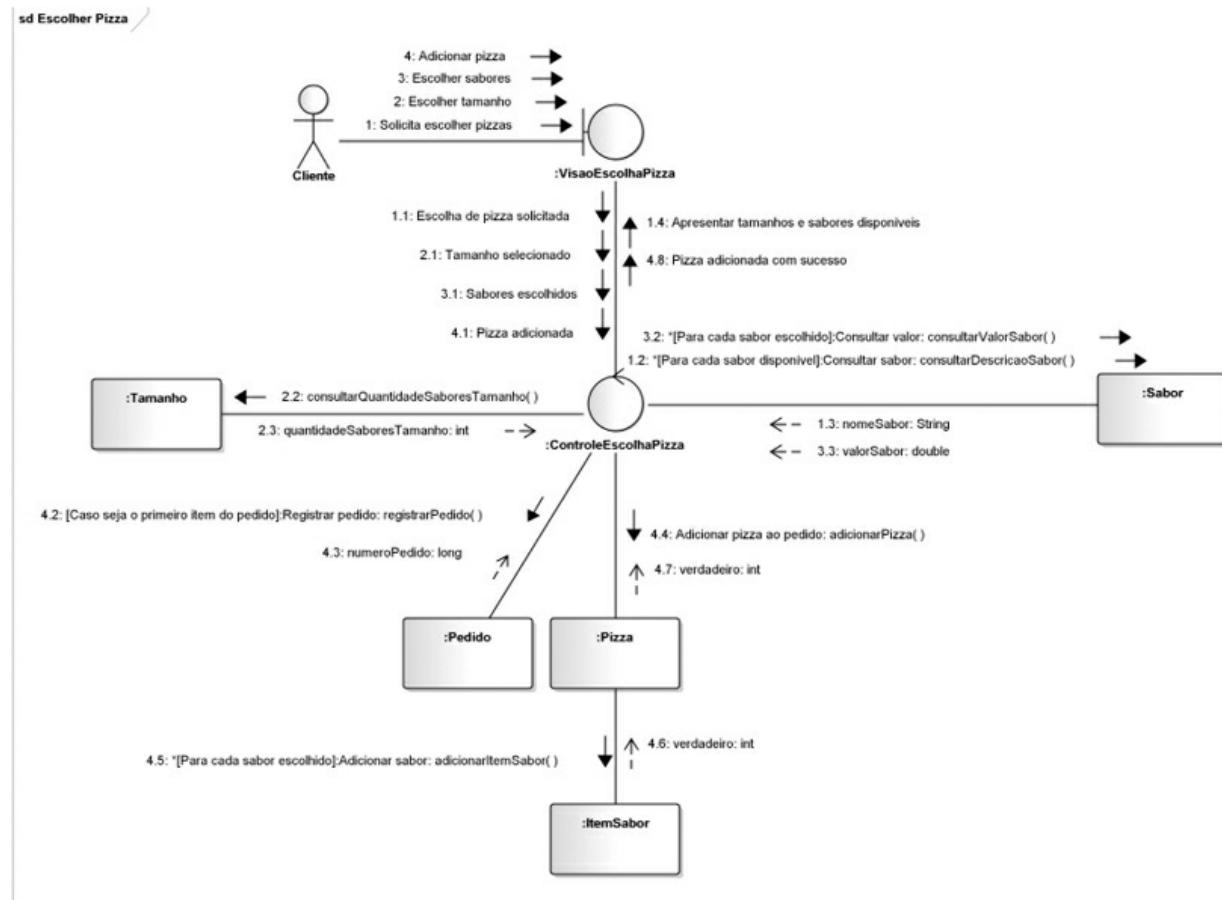


Figura 17.26 – Diagrama de Comunicação Escolher Pizza.

O leitor deve observar que os laços representados no diagrama de sequência por fragmentos combinados do tipo **loop** aqui são representados

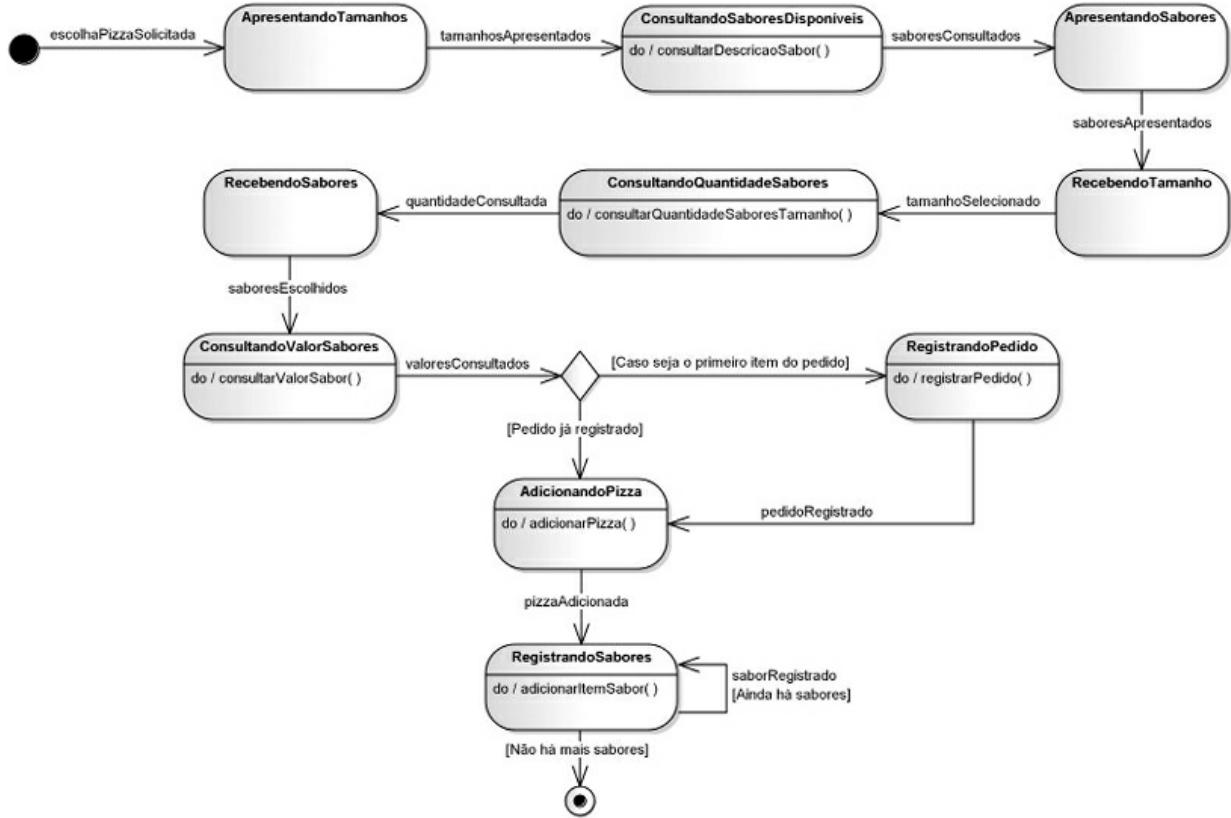
pelo operador (\*) e por condições de guarda, as quais são também usadas para estabelecer situações opcionais, representadas no diagrama de sequência por fragmentos combinados do tipo opt.

### 17.2.8 Diagramas de Máquinas de Estados da PizzaNet

Nesta seção, explicaremos os diagramas de máquina de estados referentes aos processos do sistema PizzaNet. Embora esse diagrama possa ser usado para modelar os estados de um único objeto, preferimos modelar os estados dos casos de uso do sistema, pois, uma vez que o sistema que estamos modelando é comercial, o número de estados assumidos por determinados objetos em certos processos é ínfimo, diversas vezes resumindo-se a um único estado. Assim, consideramos mais válido modelar os estados dos processos como um todo.

#### *Diagrama de Máquina de Estados Escolher Pizza*

Esse processo se inicia quando o cliente acessa a página da PizzaNet ou pressiona o botão **Pizzas**. Esse evento gera uma transição para o estado **ApresentandoTamanhos**, no qual são apresentados os tamanhos de pizzas oferecidos pela pizzaria. No momento em que todos os tamanhos forem apresentados, passa-se ao estado **ConsultandoSaboresDisponíveis**, em que é executado o método **consultarDescricaoSabor**, conforme demonstra a cláusula **Do**. Após a conclusão desse estado, passa-se ao estado **ApresentandoSabores**, no qual os sabores consultados são apresentados na interface. Em seguida, passa-se ao estado **RecebendoTamanho**, que é um estado estático na qual se espera que o cliente escolha o tamanho de pizza desejado. Quando o tamanho é escolhido, gera-se uma transição para o estado **ConsultandoQuantidadeSabores**, onde é executado o método **consultarQuantidadeSaboresTamanho** para verificar a quantidade máxima de sabores que a pizza pode conter (Figura 17.27).



*Figura 17.27 – Diagrama de Máquina de Estados Escolher Pizza.*

Após a consulta, é gerada uma transição para um novo estado estático, **RecebendoSabores**, em que se espera que o cliente selecione os sabores desejados para a pizza. Em seguida, quando o cliente escolhe os sabores da pizza, é gerada uma transição para o estado **ConsultandoValoresSabores**, em que é executado o método **consultarValorSabor**, para recuperar o valor de cada sabor escolhido. Após a conclusão desse estado, o processo passa a um pseudoestado de escolha, no qual é preciso verificar se o item adicionado é o primeiro item do pedido. Se isso for verdade, gera-se uma transição para o estado **RegistrandoPedido**, no qual é disparado o método **registrarPedido**, para gerar um novo objeto da classe **Pedido**.

Ao concluir o registro do pedido ou se a pizza em questão não for o primeiro item do pedido, passa-se ao estado **AdicionandoPizza**, em que é chamado o método **adicionarPizza**. Quando a pizza for adicionada, gera-se o estado **RegistrandoSabores** e é executado o método **AdicionarItemSabor**, que gera um novo objeto da classe **ItemSabor** para cada sabor escolhido para a pizza. Observe que nesse estado existe uma

autotransição que retornará a esse último estado enquanto houver sabores para registrar.

#### *Diagrama de Máquina de Estados Escolher Bebida*

Quando o cliente escolhe essa opção, é gerada uma transição para o estado **ConsultandoTiposBebidas**, no qual são pesquisados todos os tipos de bebidas oferecidas na pizzaria. Nesse estado é executado o método **consultarTipoBebida**. Após os tipos de bebida terem sido consultados, passa-se ao estado **ApresentandoTiposBebidas**, em que são apresentados os tipos consultados. Depois disso, passa-se ao estado **RecebendoTipoBebida**, no qual se espera que o cliente selecione um tipo de bebida (Figura 17.28).

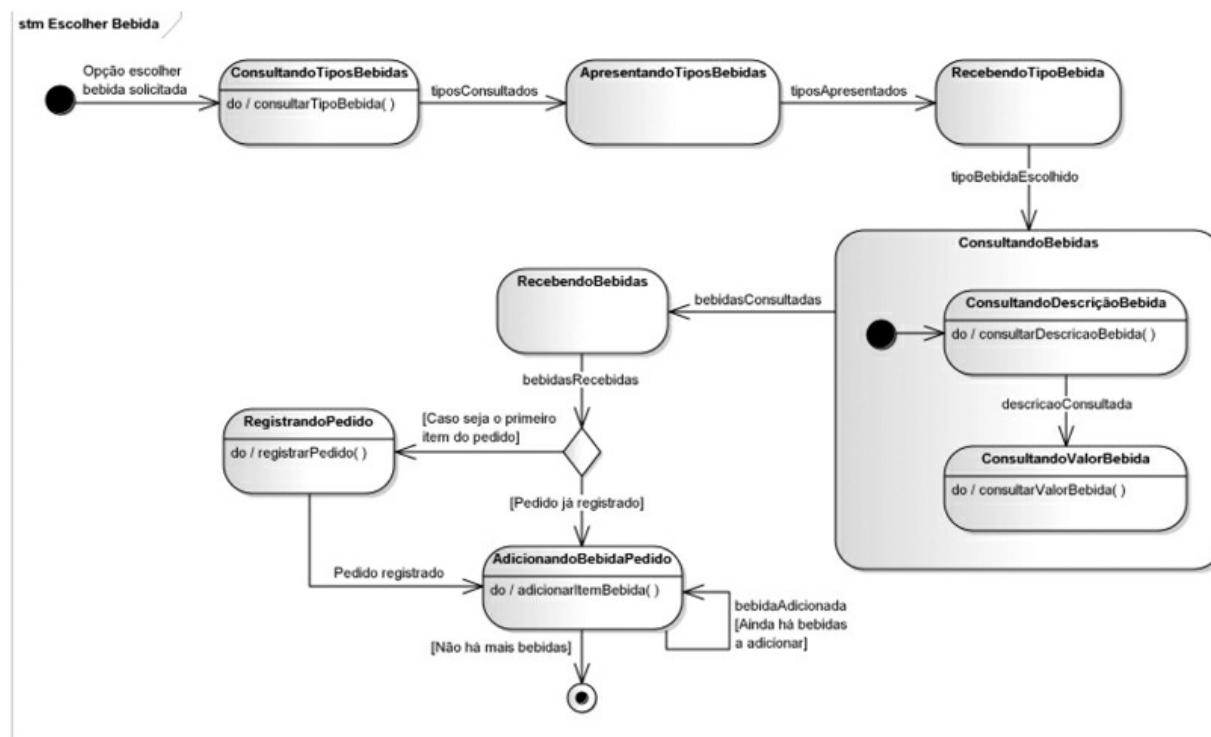


Figura 17.28 – Diagrama de Máquina de Estados Escolher Bebida.

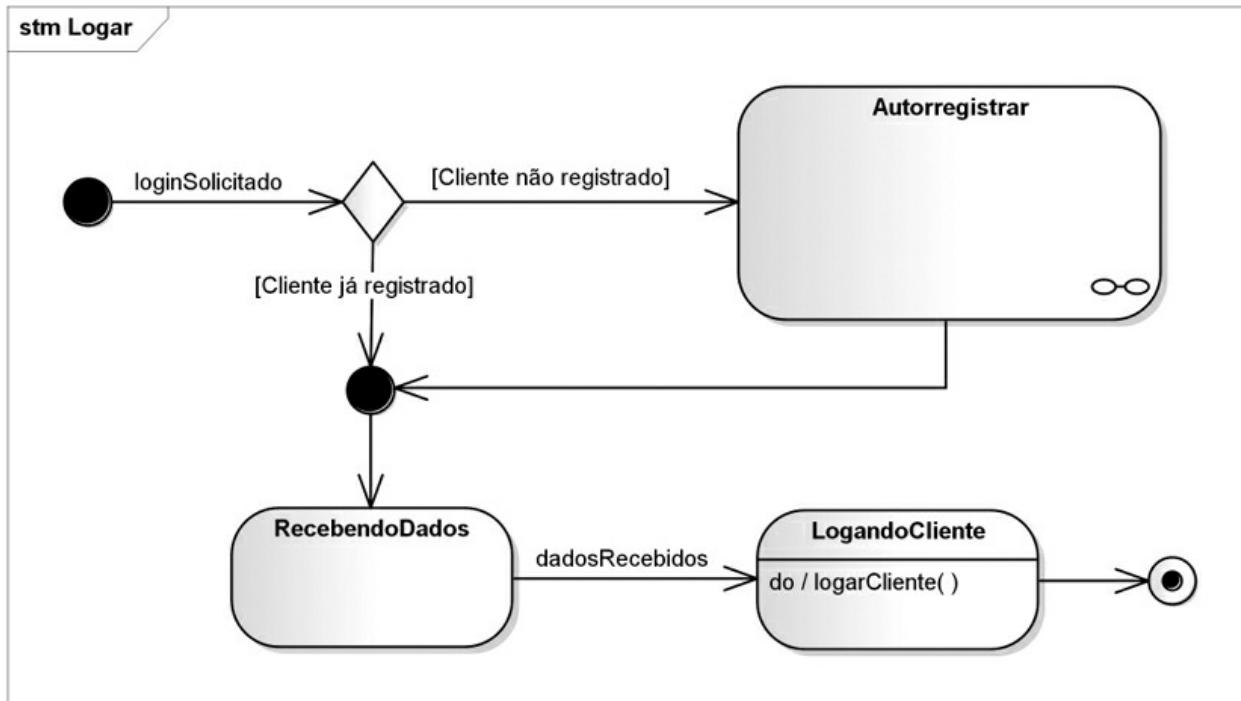
No momento em que o cliente escolher um tipo de bebida, é gerada uma transição para o estado composto **ConsultandoBebidas**, no qual são consultadas as bebidas do tipo escolhido. Nesse estado composto são gerados os subestados **ConsultandoDescriçãoBebida** e **ConsultandoValorBebida** e executados os métodos **consultarDescriçãoBebida** e **consultarValorBebida**.

Quando a consulta das bebidas for finalizada, será gerada uma transição

para um estado estático em que se aguarda até que o cliente informe as bebidas que deseja. Após o cliente escolher as bebidas e as quantidades desejadas, gera-se uma transição para um pseudoestado de escolha, em que é preciso verificar se a bebida escolhida refere-se ao primeiro item do pedido. Se esse for o caso, gera-se uma transição para o estado **RegistrandoPedido** e é executado o método `registrarPedido`. A seguir, caso a bebida escolhida não seja o primeiro item do pedido, será gerada uma transição para o estado **AdicionandoBebidaPedido**, em que é executado o método `adicionarItemBebida` para instanciar um novo objeto da classe **ItemBebida** para cada bebida escolhida pelo cliente. Observe que há uma autotransição para esse estado enquanto houver bebidas a adicionar.

#### *Diagrama de Máquina de Estados Logar*

No momento em que o cliente seleciona essa opção, uma transição para um pseudoestado de escolha é gerada. Esse pseudoestado representa a decisão do cliente em informar seu nome-login e senha, caso já possua cadastro, ou de se autorregular, caso contrário (Figura 17.29).



*Figura 17.29 – Diagrama Máquina de Estados Logar.*

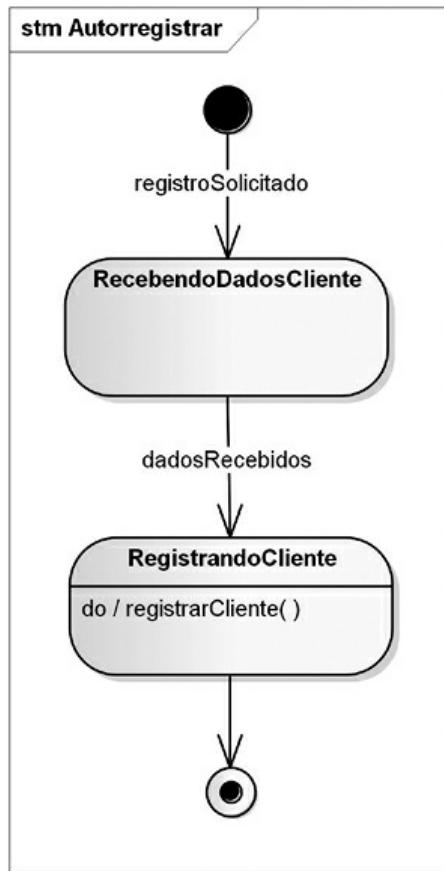
Caso o cliente não esteja registrado, será gerada uma transição para um

estado de submáquina que se refere ao processo de **Autorregisterar**, descrito em outro diagrama. Após se autorregisterar, ou caso o cliente já possua cadastro, passa-se ao estado estático **RecebendoDados**, no qual o cliente informará seu nome-login e senha. Observe que se uniu o fluxo dividido pelo pseudoestado de escolha por meio de um pseudoestado de junção.

No momento em que os dados do cliente forem recebidos, será gerada uma transição para o estado **LogandoCliente**, onde será executado o método **logarCliente**, por meio do qual o cliente será logado no sistema, caso seu nome-login e senha estejam corretos.

#### *Diagrama de Máquina de Estados Autorregisterar*

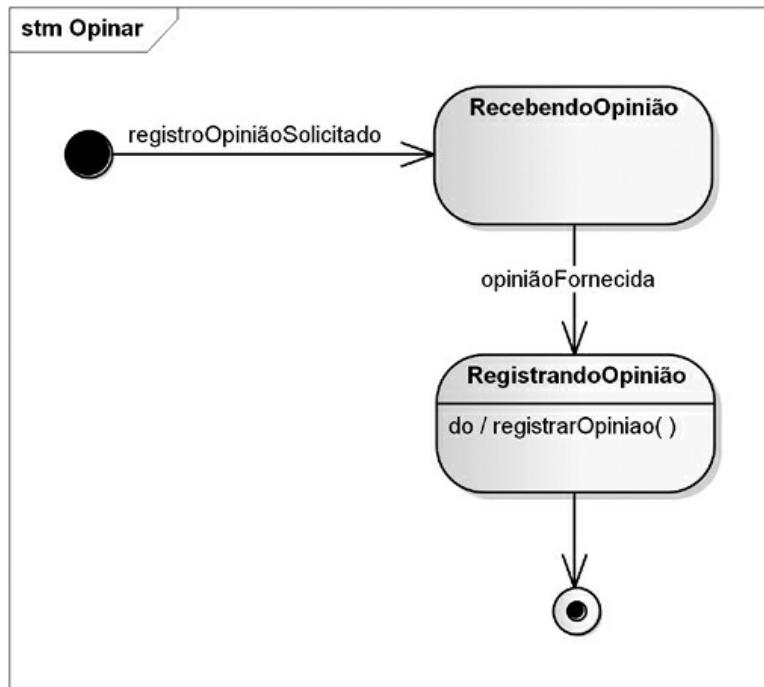
Esse processo envolve apenas dois estados, sendo o primeiro estado **RecebendoDadosCliente**, gerado pela transição causada pela solicitação de registro pelo cliente. Nesse estado, o cliente deve informar seus dados e confirmar. Quando isso ocorre, é gerada uma transição para o estado **RegistrandoCliente**, em que é executado o método **registrarCliente**, para instanciar um novo objeto da classe **Cliente** (Figura 17.30).



*Figura 17.30 – Diagrama de Máquina de Estados Autorregisterar.*

#### *Diagrama de Máquina de Estados Opinar*

Esse é um diagrama bastante simples composto de apenas dois estados. Inicia-se quando o cliente solicita a execução da funcionalidade de registrar opinião. Isso causa uma transição que gera o estado **RecebendoOpinião**, que espera que o cliente informe a sua opinião e confirme. Quando isso acontece, é gerado o estado **RegistrandoOpinião**, em que é executado o método **registrarOpiniao**, que instancia um novo objeto da classe **Opiniao** (Figura 17.31).



*Figura 17.31 – Diagrama de Máquina de Estados Opinar.*

#### *Diagrama de Máquina de Estados Visualizar Pedido*

Os estados modelados por esse diagrama são iniciados quando um cliente solicita a visualização de seu pedido. Isso produz uma transição para o estado **ConsultandoPedido**, em que é executado o método **consultarPedido**. Após o pedido ter sido consultado, é gerada uma transição para um estado composto que consultará as possíveis pizzas do pedido, denominado **ConsultandoPizzas**.

Esse estado composto contém os subestados **ConsultandoPizza**, que executa o método **consultarPizza**, **ConsultandoTamanho**, que dispara o método **consultarDescricaoTamanho**, e um subestado composto, chamado **ConsultandoSabores**, que contém, por sua vez, os subestados **ConsultandoItemSabor**, que chama o método **consultarItemSabor** para carregar todos os objetos da classe **ItemSabor** associados a uma pizza, e **ConsultandoSabor**, que executa o método **consultarDescricaoSabor** para, como o nome do método sugere, consultar o nome do sabor.

Após terem sido consultadas todas as possíveis pizzas solicitadas no pedido, passa-se a um novo estado composto, denominado **ConsultandoBebidas**, em que são pesquisadas as possíveis bebidas solicitadas no pedido. Esse estado composto contém os subestados

**ConsultandoItemBebida**, que executa o método **ConsultarItemBebida**, no qual são carregados todos os objetos da classe **ItemBebida** associados ao pedido em questão, e **ConsultandoBebida**, que chama o método **consultarDescricaoBebida**. Como o nome do método sugere, consultar o nome do sabor (Figura 17.32).

Após ter apresentado os dados das pizzas e bebidas do pedido, o processo encontra um pseudoestado de escolha, em que é oferecida a opção ao cliente de excluir qualquer item visualizado. Se o cliente optar por excluir algum item, será gerada uma transição para o estado de submáquina **ExcluirItem**, que faz referência ao processo de mesmo nome, detalhado em um diagrama separado.

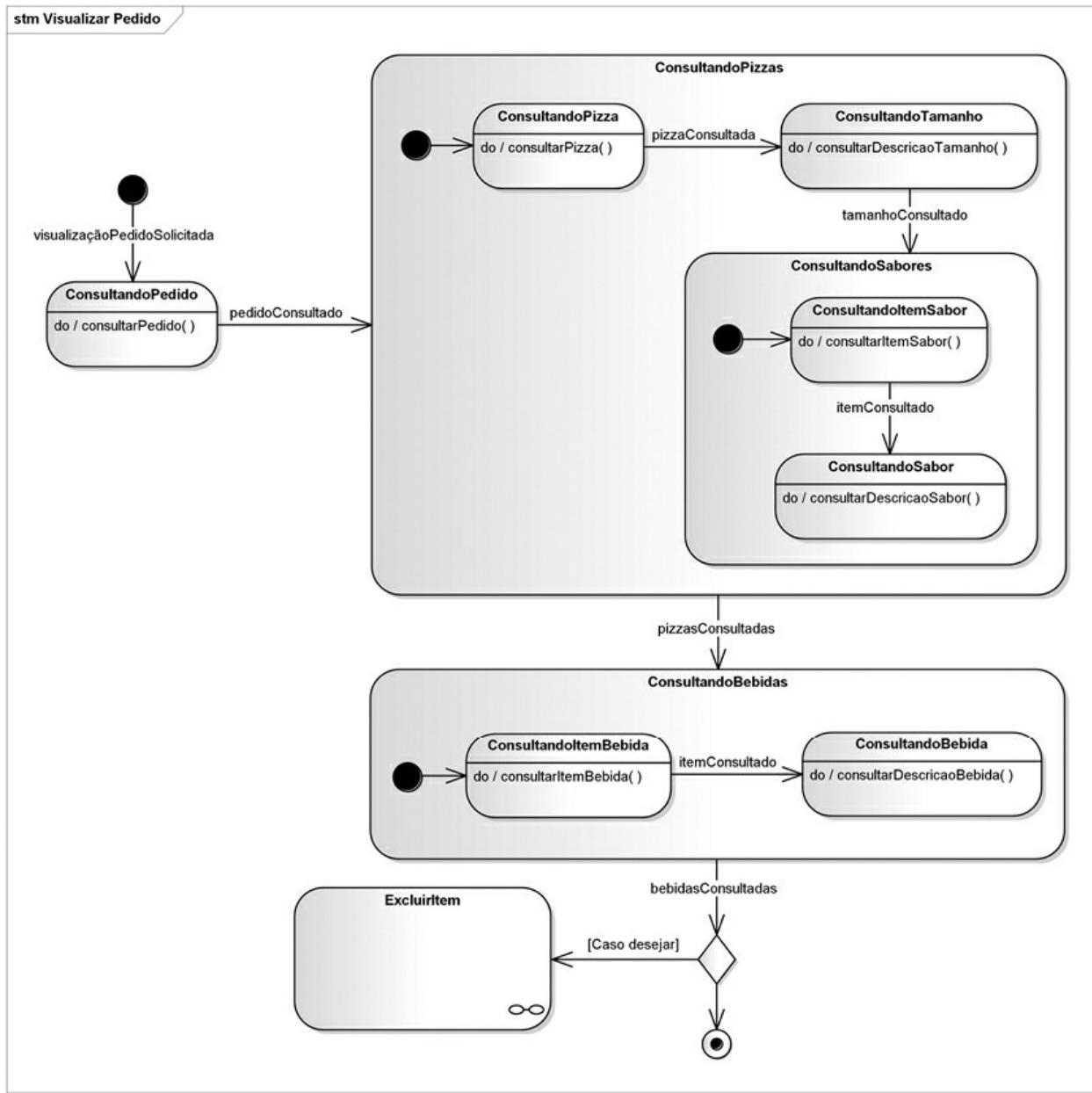


Figura 17.32 – Diagrama de Máquina de Estados Visualizar Pedido.

#### Diagrama de Máquina de Estados Excluir Item

Os estados desse processo iniciam-se quando o cliente opta por excluir um item de um pedido, mediante o processo de visualização do pedido. A solicitação de exclusão de um item gera uma transição para um pseudoestado de escolha, no qual se deve verificar se o elemento a excluir trata-se de uma pizza ou bebida (Figura 17.33).

Se o elemento a excluir for uma pizza, gera-se uma transição para o estado **ExcluindoPizza**, que exclui um objeto da classe **Pizza** por meio do

método `excluirPizza` e, após essa exclusão, gera-se uma transição para o estado `ExcluindoItemSabor`, que exclui todos os objetos da classe `ItemSabor` associados ao objeto da classe `Pizza` por meio do método `excluirItemSabor`. Observe que há uma autotransição para esse estado que será executada enquanto houver itens a ser excluídos.

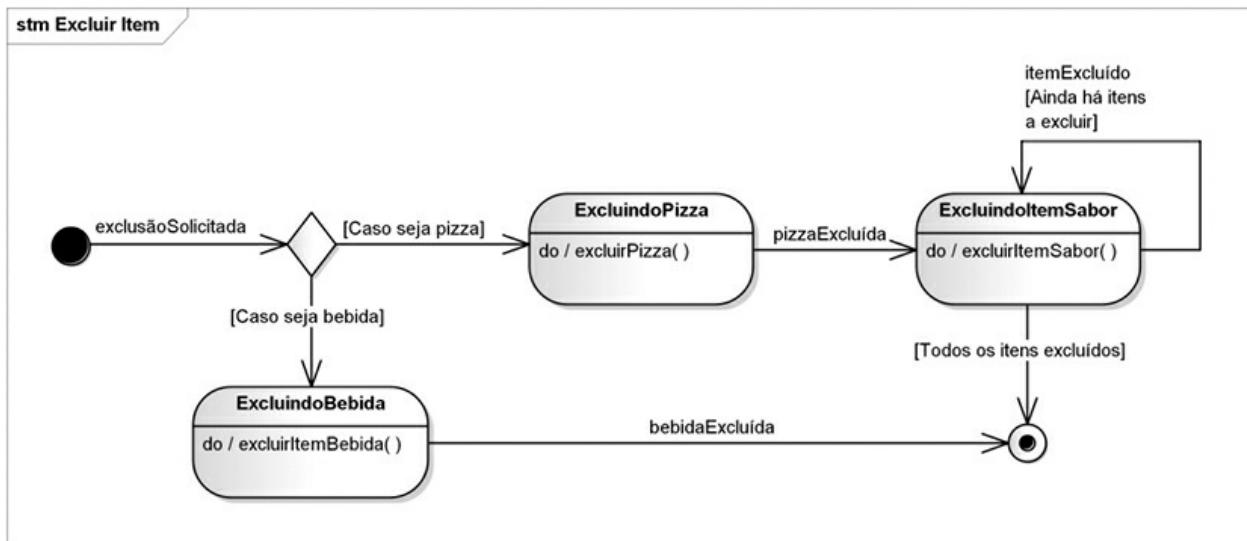


Figura 17.33 – Diagrama de Máquina de Estados Excluir Item.

Já se o elemento a excluir se referir a uma bebida, será gerada uma transição para o estado `ExcluindoBebida`, em que será executado o método `excluirItemBebida`, para excluir o objeto da classe `ItemBebida` que se refere à bebida que o cliente deseja excluir.

#### Diagrama de Máquina de Estados Visualizar Pedidos Anteriores

Nesse processo, é gerada uma transição para um estado de submáquina que se refere ao processo de `VisualizarPedido`, explicado anteriormente. Observe que existe uma autotransição para o próprio estado de submáquina, que será chamada enquanto ainda houver pedidos a apresentar (Figura 17.34).

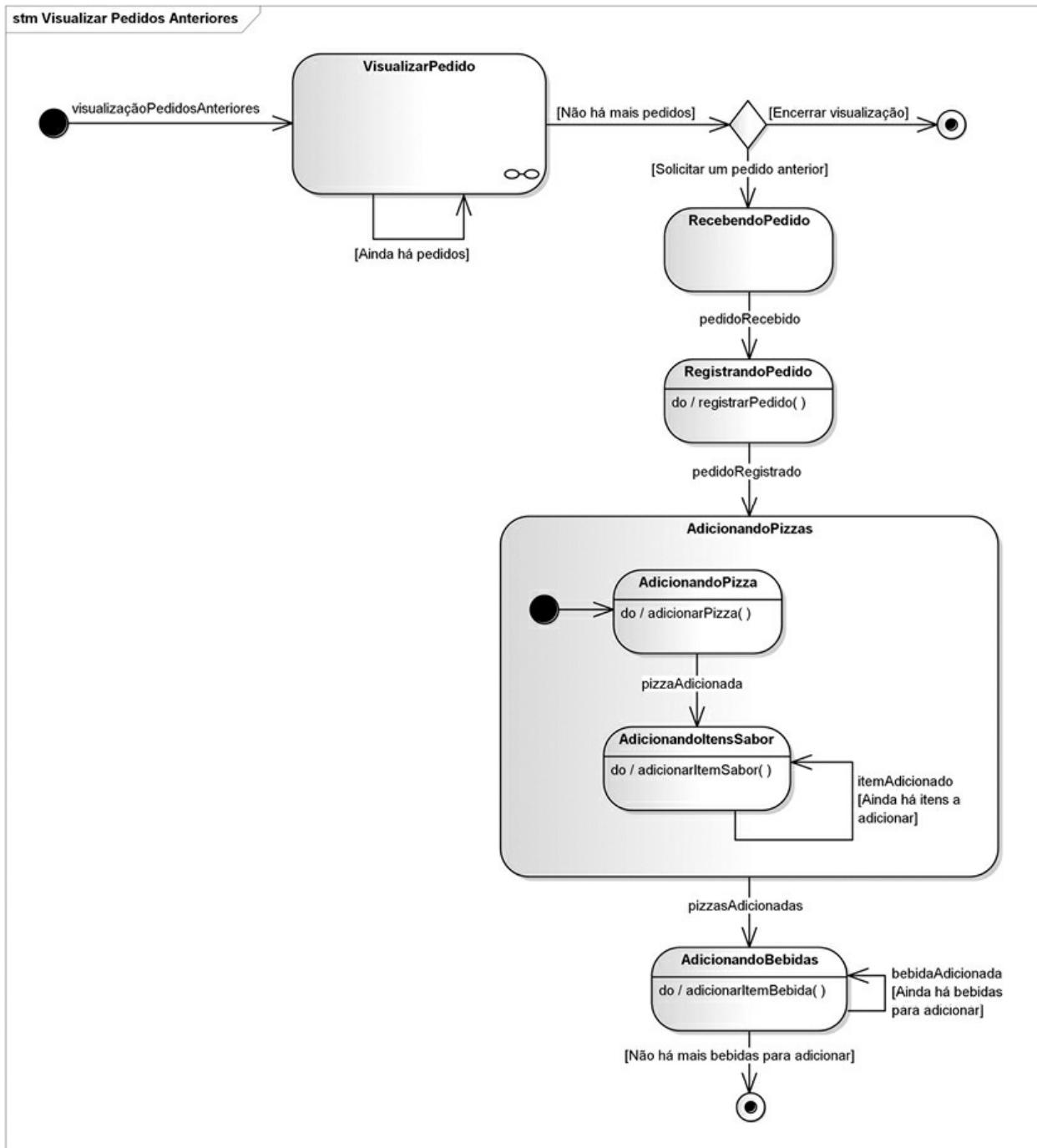
Depois que todos os pedidos anteriores tiverem sido apresentados, passe a um pseudoestado de escolha em que o cliente pode optar por encerrar a visualização nesse momento ou solicitar um pedido solicitado anteriormente.

Nessa segunda alternativa, o cliente deve escolher um pedido, o que causa uma transição para o estado `RecebendoPedido`, no qual se aguarda a

escolha do pedido. Quando o pedido é escolhido, gera-se o estado **RegistrandoPedido**, no qual é executado o método `registrarPedido`, para instanciar um novo objeto da classe **Pedido**.

Após registrar o pedido, é gerada uma transição para o estado composto **AdicionandoPizzas**, que tem dois subestados. O primeiro subestado, **AdicionandoPizza**, executa o método `adicionarPizza` para instanciar um novo objeto da classe **Pizza**. Já o segundo subestado, **AdicionandoItensSabor**, chama o método `adicionarItemSabor` para instanciar um novo objeto da classe **ItemSabor**. Uma autotransição retorna a esse estado tantas vezes quantos forem os sabores da pizza.

Quando o estado composto for finalizado, será gerada uma transição para o estado **AdicionandoBebidas**, que dispara o método `adicionarItemBebida` para gerar os objetos da classe **ItemBebida** referentes ao pedido. Novamente, uma autotransição retorna a esse estado enquanto houver bebidas para adicionar ao pedido.



*Figura 17.34 – Diagrama de Máquina de Estados Visualizar Pedidos Anteriores.*

#### *Diagrama de Máquina de Estados Visualizar Sabores Mais Pedidos*

Quando a funcionalidade de **Visualizar Sabores Mais Pedidos** é solicitada, é gerada uma transição para o estado **TotalizandoSabores**, que executa o método **totalizarSabor**, no qual são contadas as ocorrências de todos os

objetos associados a um determinado sabor (Figura 17.35).

A seguir, passa-se ao estado **ConsultandoDescriçãoSaboresTotalizados** em que a descrição de cada sabor totalizado é consultada. Finalmente, após esse estado ser concluído, é gerada uma transição para o estado **ApresentandoSaboresTotalizados**, no qual são apresentados todos os sabores com seus respectivos totais de solicitação, ordenados pelos maiores totais até os menores.

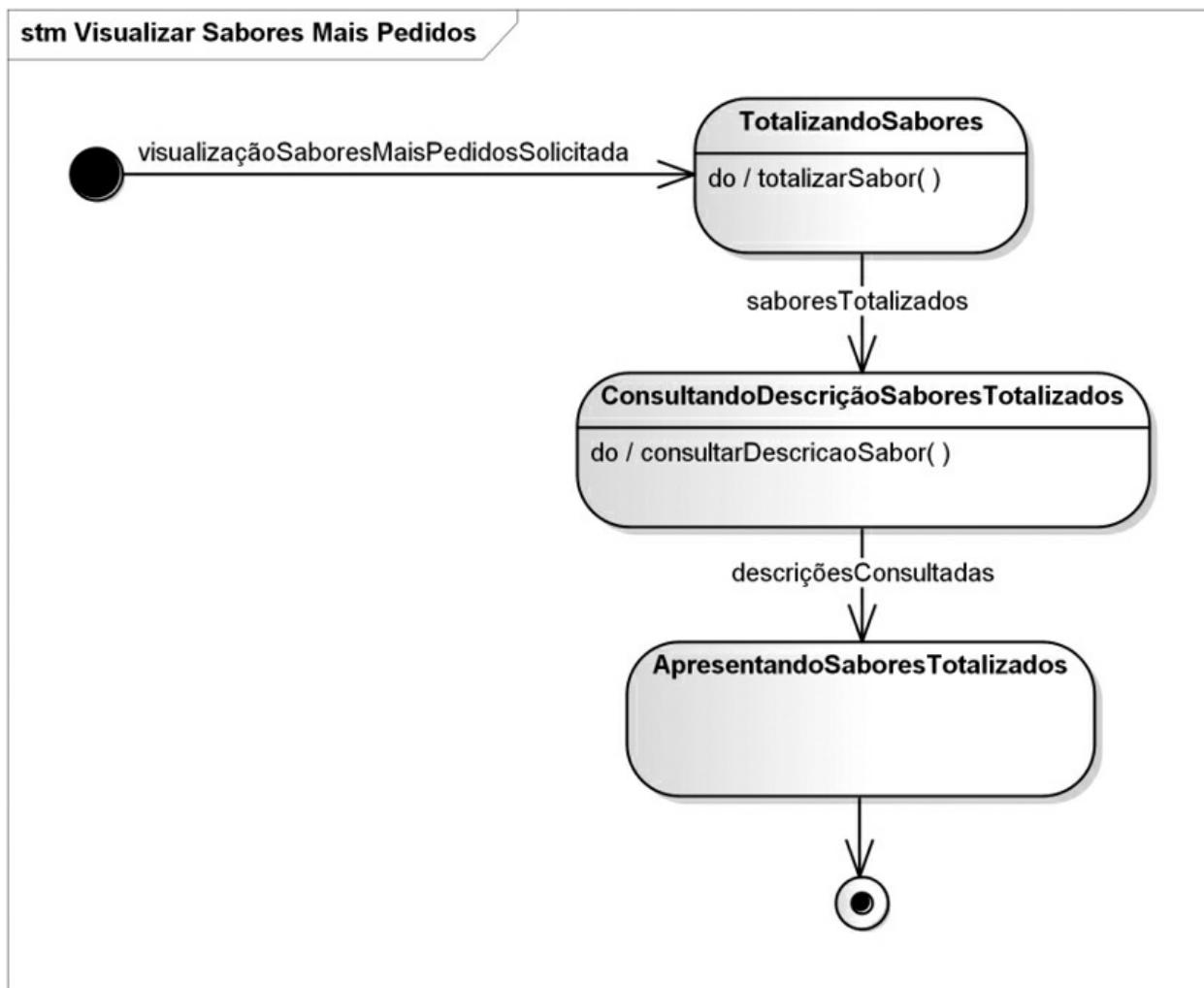


Figura 17.35 – Diagrama de Máquina de Estados Visualizar Sabores Mais Pedidos.

#### Diagrama de Máquina de Estados Concluir Pedido

Quando é solicitada essa funcionalidade, é gerada uma transição para um pseudoestado de escolha, que deve verificar se o cliente já está autenticado. Caso o cliente ainda não esteja logado, será gerada uma transição para um

estado de submáquina referente ao processo de **Logar** detalhado em outro diagrama (Figura 17.36).

Depois de autenticar o cliente ou caso já esteja logado, passa-se a um segundo estado de submáquina que se refere ao processo de **VisualizarPedido**, também já detalhado em outro diagrama. Isso é necessário porque se deve apresentar os detalhes do pedido mais uma vez antes de realmente o concluir.

A seguir, é gerada uma transição para um novo pseudoestado de escolha, que verifica se o cliente deseja alterar o endereço de entrega, caso em que é gerada uma transição para o estado **AlterandoEndereço**, em que é executado o método **registrarCliente**. Observe que o fluxo que havia sido dividido pelo pseudoestado de escolha é novamente unido por um pseudoestado de junção.

A confirmação final do pedido gera uma transição para o estado **AtualizandoEstoqueBebida**, no qual é executado o método **atualizarQuantidadeBebida**, que diminui a quantidade de bebidas solicitadas de seu estoque.

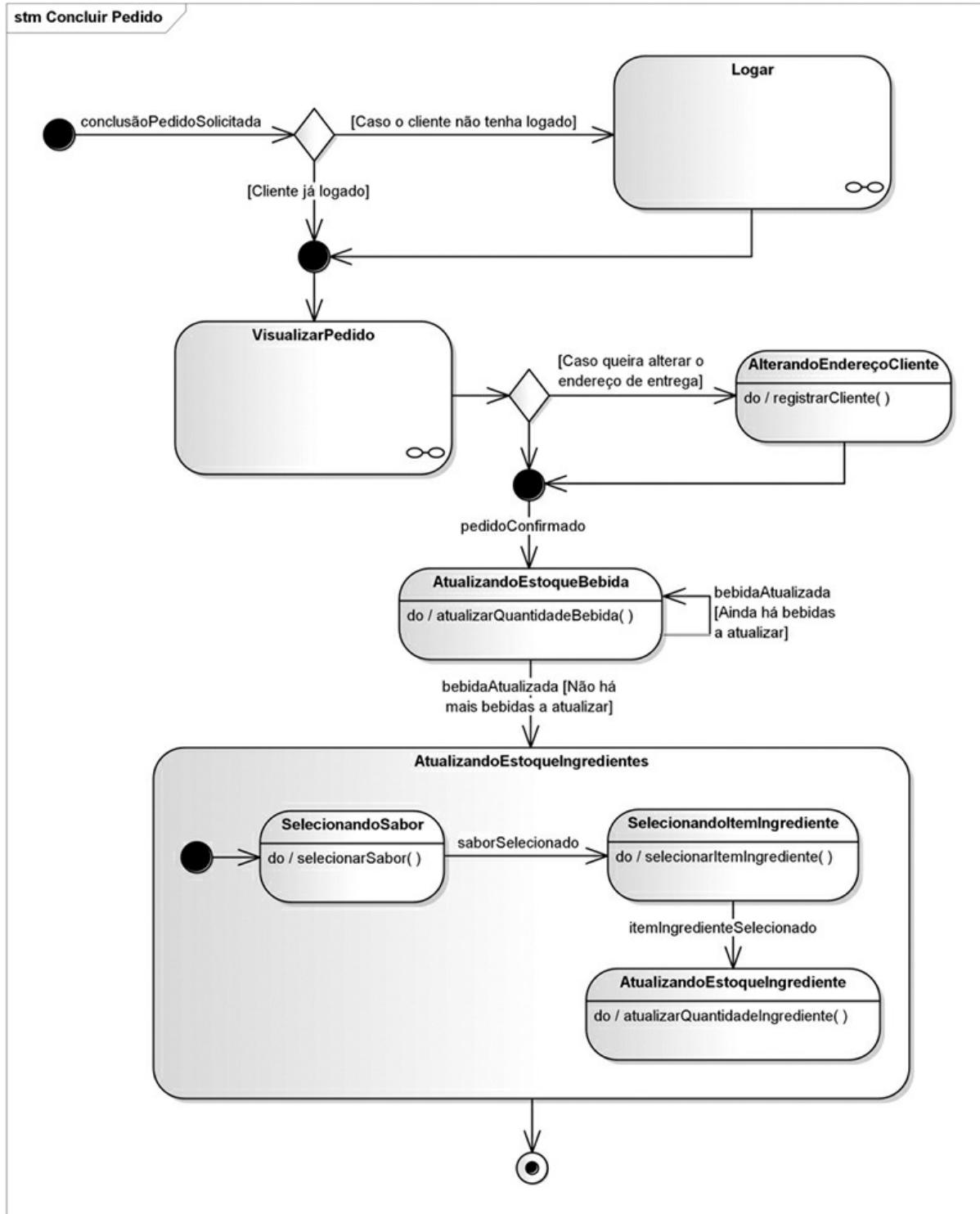


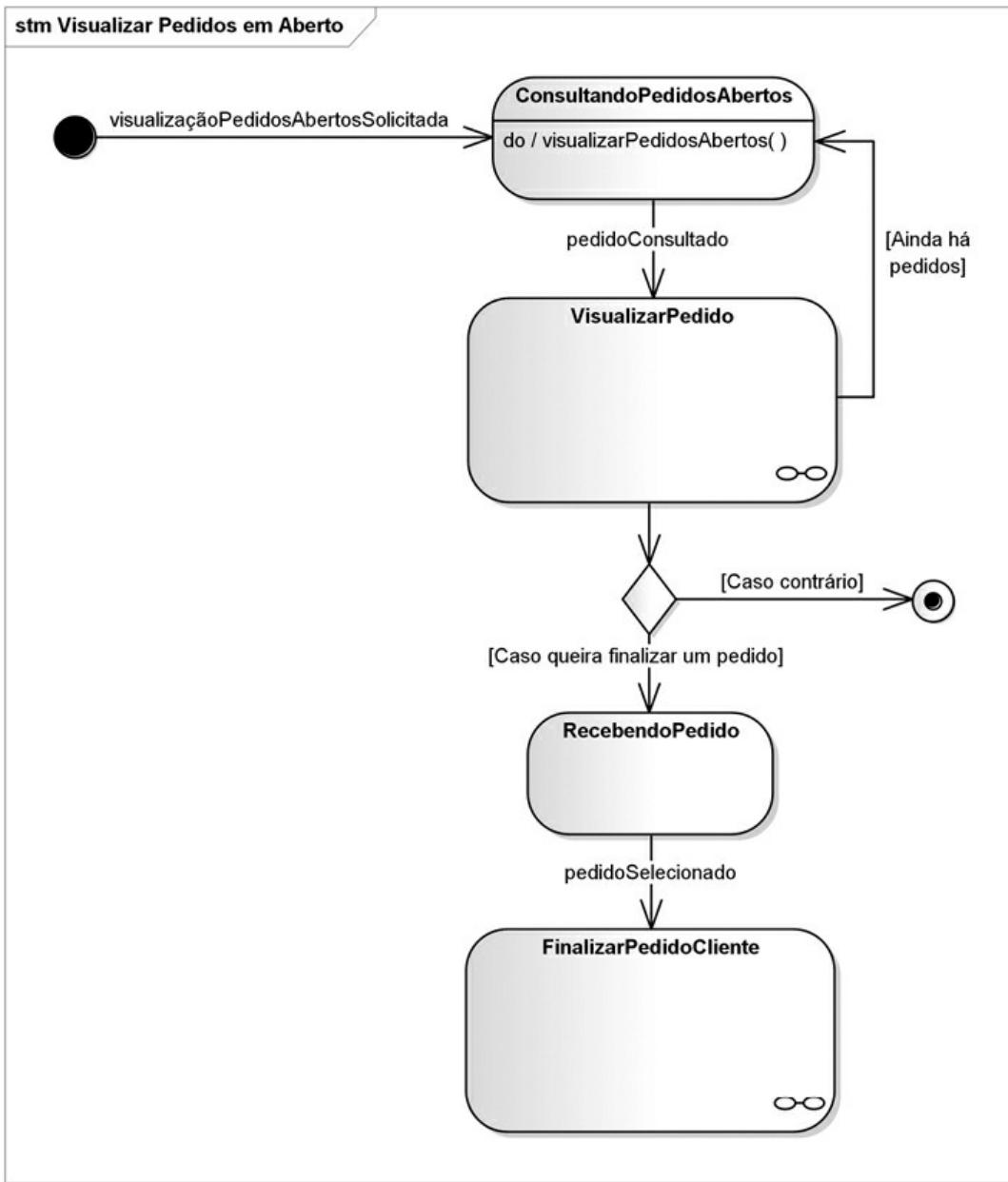
Figura 17.36 – Diagrama de Máquina de Estados Concluir Pedido.

Ao concluir esse estado, passa-se a um estado composto responsável por dar baixa nas quantidades de ingredientes necessários à produção das pizzas do pedido. Esse estado composto apresenta três subestados. No

primeiro, chamado **SelecionandoSabor**, é disparado o método **selecionarSabor** para selecionar os sabores solicitados no pedido. O segundo subestado, **SelecionandoItemIngrediente**, executa o método **selecionarItemIngrediente**, para selecionar os ingredientes de um sabor. Já o terceiro subestado, **atualizandoEstoqueIngrediente**, chama o método **atualizarQuantidadeIngrediente**, para diminuir as quantidades necessárias à produção de cada sabor da quantidade armazenada em estoque.

#### *Diagrama de Máquina de Estados Visualizar Pedidos em Aberto*

A solicitação desse serviço produz uma transição para o estado **ConsultandoPedidosAbertos**, no qual é disparado o método **visualizarPedidosAbertos**. Após selecionar um pedido em aberto, passa-se ao estado de submáquina **VisualizarPedido**, que se refere ao processo de mesmo nome, já modelado em outro diagrama. Nesse estado de submáquina são apresentados os detalhes do pedido selecionado. Observe que esse processo é chamado para cada pedido em aberto e, após consultar um pedido específico, há uma transição que retorna para o estado **ConsultandoPedidosAbertos**, enquanto houver pedidos em aberto para apresentar (Figura 17.37).



*Figura 17.37 – Diagrama de Máquina de Estados Visualizar Pedidos em Aberto.*

Quando todos os pedidos em aberto tiverem sido visualizados, passa-se a um pseudoestado de escolha, em que o funcionário deve escolher entre encerrar o processo de visualização ou finalizar um pedido. Caso o funcionário escolha a segunda alternativa, será gerada uma transição para um estado de espera, denominado **RecebendoPedido**, no qual se aguarda que o pedido a ser finalizado seja selecionado. O evento de seleção de pedido gera uma transição para outro estado de submáquina que se refere

ao processo de **FinalizarPedido**, no qual, como se pode inferir pelo nome do processo, o pedido selecionado será finalizado.

#### *Diagrama de Máquina de Estados Finalizar Pedido Cliente*

No momento em que esse processo é solicitado, é gerada uma transição para o estado **ConsultandoFuncionários**, no qual é executado o método **consultarFuncionario** para consultar cada um dos funcionários registrados no sistema. A finalização desse estado gera uma transição para o estado **ApresentandoFuncionários**, em que os funcionários consultados serão apresentados no formulário de finalização de pedido (Figura 17.38).

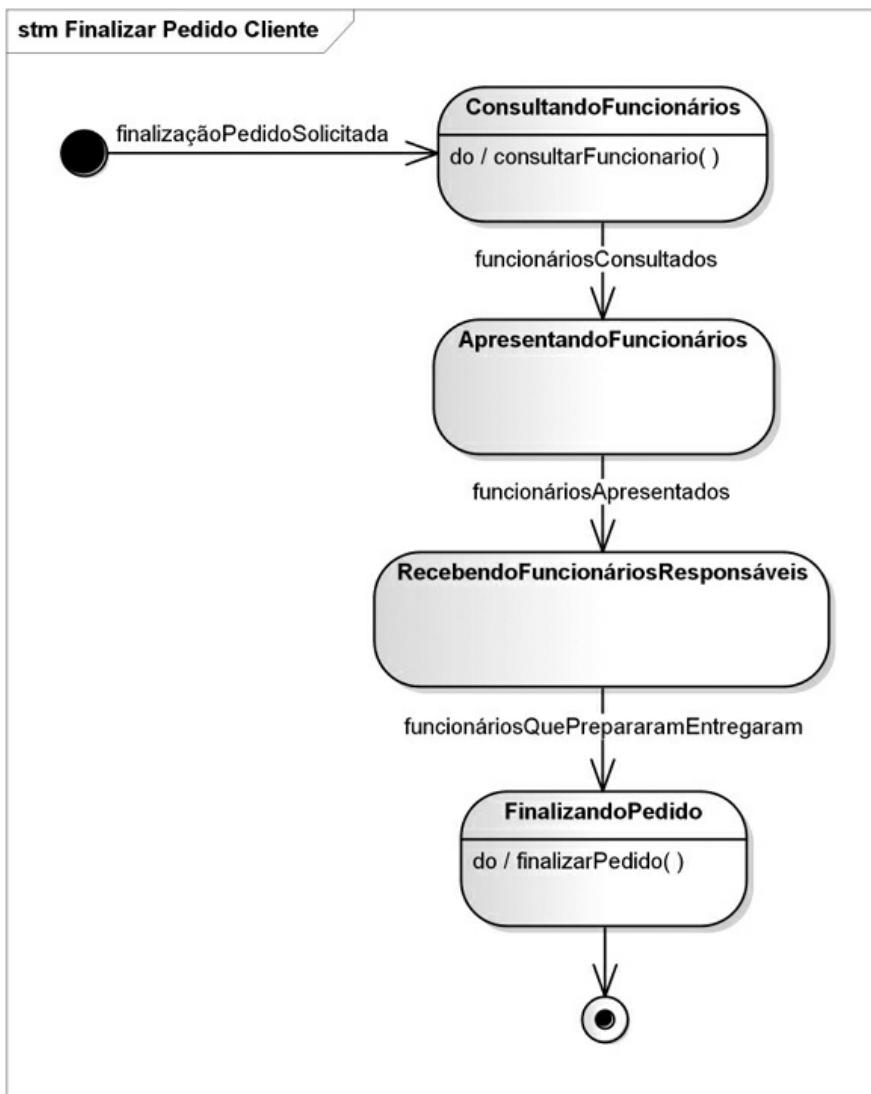


Figura 17.38 – Diagrama de Máquina de Estados Finalizar Pedido Cliente.  
A seguir, passa-se ao estado **RecebendoFuncionáriosResponsáveis** em

que se aguarda que sejam selecionados os funcionários que prepararam e entregaram o pedido. A seleção dos funcionários gera o estado **FinalizandoPedido**, no qual é chamado o método **finalizaPedido**. A execução desse método, além de registrar os funcionários que prepararam e entregaram o pedido, mudará a situação do pedido para 2, significando que foi atendido, ou seja, finalizado.

#### *Diagrama de Máquina de Estados Gerenciar Cardápio*

Os estados desse processo iniciam-se com o estado **ConsultandoSabores**, no qual são consultados todos os sabores registrados no sistema, por meio do método **consultarDescricaoSabor**. A conclusão desse estado gera uma transição para o estado **ApresentandoSabores**, que apresenta todos os sabores retornados na interface do software.

Depois que todos os sabores tiverem sido apresentados, será gerada uma transição para o estado **ConsultandoIngredientes**, em que serão consultados todos os ingredientes registrados no sistema. Isso é realizado por meio do método **consultarIngrediente**. A finalização desse estado gera uma transição para o estado **ApresentandoIngredientes**, em que todos os ingredientes consultados são apresentados na interface (Figura 17.39).

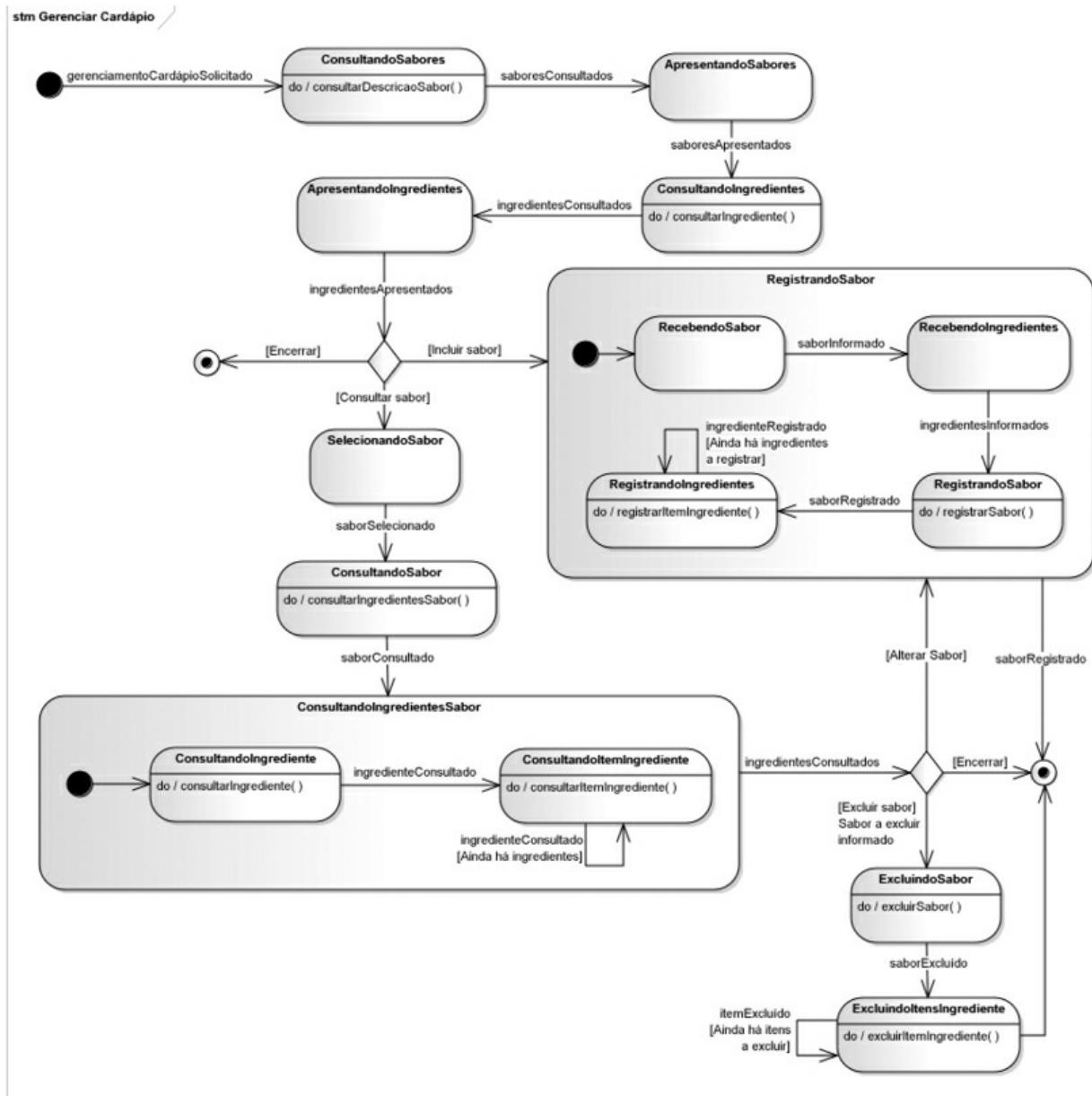


Figura 17.39 – Diagrama de Máquina de Estados Gerenciar Cardápio.

A seguir, gera-se uma transição para um pseudoestado de escolha que representa a decisão de o administrador registrar um novo sabor, consultar um sabor já registrado ou encerrar o processo de gerenciamento de cardápio.

Caso o administrador queira inserir um novo sabor, será gerada uma transição para o estado composto **RegistrandoSabor**, no interior do qual serão executados quatro subestados. Os dois primeiros são denominados **RecebendoSabor** e **RecebendoIngredientes**. Como o leitor pode intuir, esses dois subestados são subestados estáticos nos quais se aguarda que o

administrador informe o sabor e os ingredientes que o compõem. Na verdade, isso poderia ser representado por um único subestado, mas preferimos dividir em dois para deixar o diagrama mais detalhado.

No momento em que os ingredientes do novo sabor forem informados, será gerada uma transição para o subestado **RegistrandoSabor**, no qual será executado o método **registrarSabor**, e **RegistrandoIngredientes**, em que será executado o método **registrarItemIngrediente**, para instanciar novos objetos da classe **ItemIngrediente** relacionados ao novo sabor. Observe que existe uma autotransição para esse estado enquanto houver ingredientes a registrar. Após a conclusão desse estado composto, o processo de inclusão de um novo sabor é encerrado.

Caso o administrador deseje consultar um sabor existente, deve selecioná-lo no formulário, o que gera uma transição para o estado **SelecionandoSabor**. Após o sabor ser selecionado, passa-se ao estado **ConsultandoSabor**, em que é disparado o método **consultarIngredientesSabor** para consultar os ingredientes do sabor que está sob consulta. Como a classe **Sabor** não contém todas as informações necessárias, é necessário gerar uma transição para um estado composto denominado **ConsultandoIngredientesSabor**.

Esse estado composto contém dois subestados. No primeiro, denominado **ConsultandoIngrediente**, é executado o método **consultarIngrediente** para recuperar a descrição de cada ingrediente do sabor consultado. O segundo subestado, chamado **ConsultandoItemIngrediente**, chama o método **consultarItemIngrediente** para recuperar as informações dos objetos da classe **ItemIngrediente** associados ao sabor.

Após a conclusão dessa consulta, o processo atinge um novo pseudoestado de escolha, em que se deve optar por alterar um sabor, excluir um sabor ou encerrar o processo.

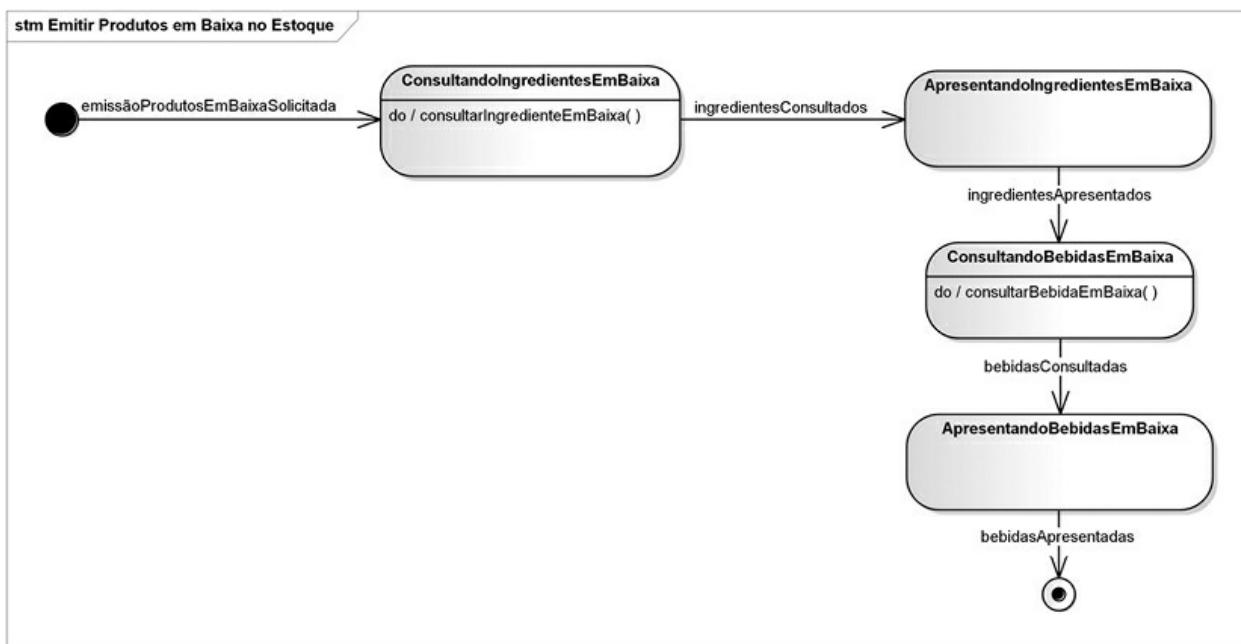
Caso o administrador escolha alterar um sabor, ele deve alterar as informações deste, bem como os ingredientes necessários ao sabor. Isso gera uma transição para um comportamento idêntico ao representado no estado composto **RegistrandoSabor** e, por esse motivo, uma transição o atinge novamente.

Porém, se o administrador quiser excluir um sabor, deverá selecioná-lo e solicitar sua exclusão no formulário, fazendo que uma transição atinja o

estado **ExcluindoSabor**, no qual é executado o método **excluirSabor** para excluir o objeto da classe **Sabor** selecionado. Após o sabor ser excluído, gera-se uma transição para o estado **ExcluindoItensIngrediente**, que executa o método **excluirItemIngrediente**, o qual destrói todos os objetos da classe **ItemIngrediente** associados ao objeto da classe **Sabor** em questão. Isso é necessário porque há uma associação de composição entre a classe **Sabor** e a classe **ItemIngrediente**. Assim, quando um objeto da classe **Sabor** for excluído, todos os seus objetos-parte deverão ser igualmente excluídos. Há uma autotransição para esse último estado que será executada enquanto houver objetos **ItemIngrediente** para excluir.

#### *Diagrama de Máquina de Estados Emitir Produtos em Baixa no Estoque*

Esse processo possui apenas quatro estados. O primeiro, denominado **ConsultandoIngredientesEmBaixa**, dispara o método **consultarIngredientesEmBaixa**, para retornar todos os ingredientes cuja quantidade em estoque esteja abaixo da mínima. A conclusão desse estado gera uma transição para o estado **ApresentandoIngredientesEmBaixa**, no qual os ingredientes consultados são carregados na interface do software (Figura 17.40).



*Figura 17.40 – Diagrama de Máquina de Estados Emitir Produtos em Baixa no Estoque.*

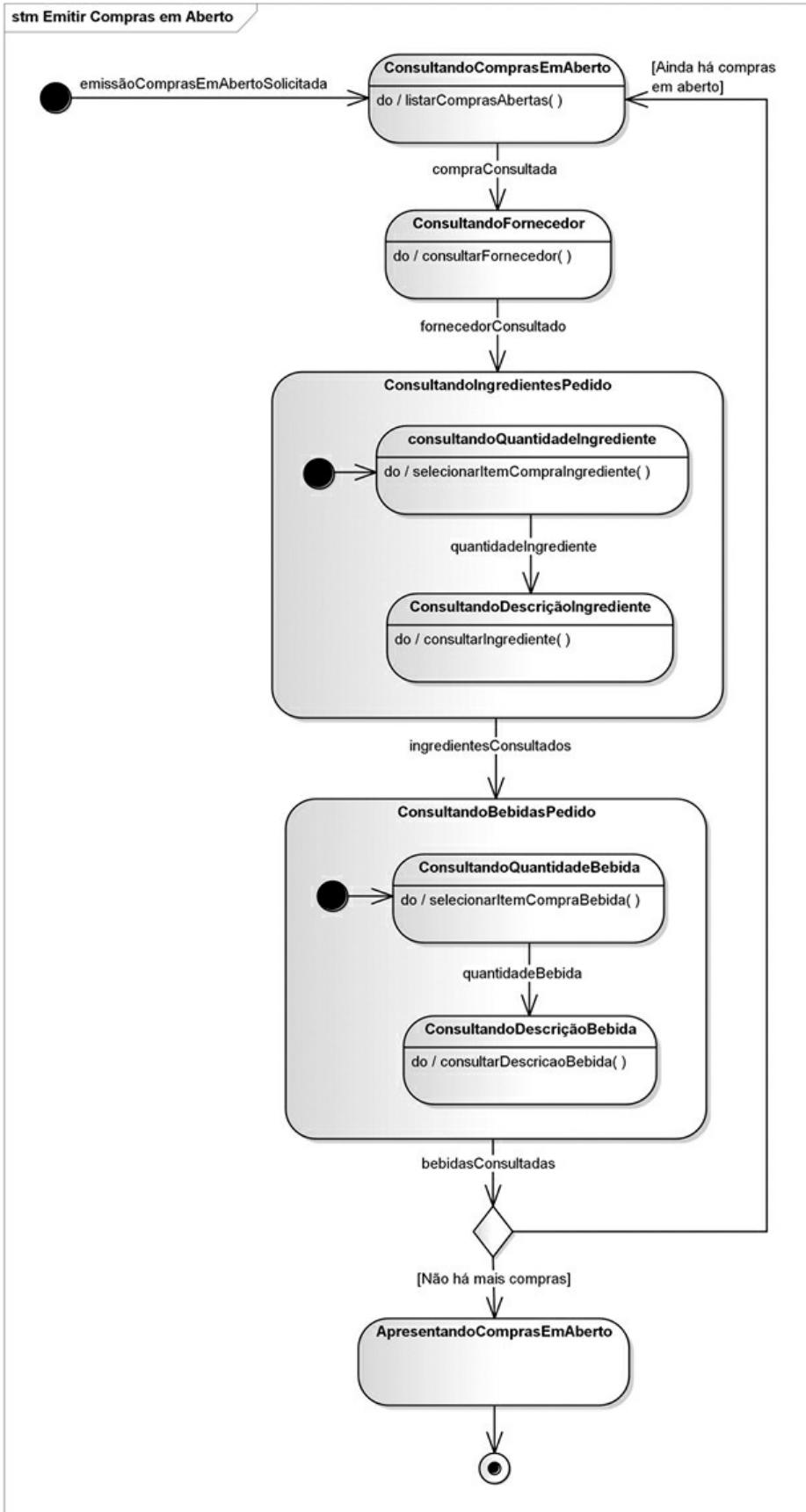
Depois que os ingredientes tiverem sido apresentados, passa-se ao estado **ConsultandoBebidasEmBaixa**, no qual é chamado o método **consultarBebidaEmBaixa**, para retornar as bebidas cujas quantidades estão em baixa no estoque. A finalização desse estado gera uma transição para o estado **ApresentandoBebidasEmBaixa**, que simplesmente apresenta na interface do sistema as bebidas em baixa consultadas.

#### *Diagrama de Máquina de Estados Emitir Compras em Aberto*

O disparo desse processo gera uma transição para o estado **ConsultandoComprasEmAberto**, que executará o método **listarComprasAbertas**. Em seguida, passa-se ao estado **ConsultandoFornecedor**, que executa o método **consultarFornecedor** para retornar os dados do fornecedor a que cada compra se refere.

A seguir, passa-se ao estado composto **ConsultandoIngredientesPedido**, cujo primeiro subestado **consultandoQuantidadeIngrediente** executa o método **selecionarItemCompraIngrediente** para retornar a quantidade dos objetos da classe **ItemCompraIngrediente** associados ao objeto da classe **Compra**. Já o segundo subestado **ConsultandoDescriçãoIngrediente** chama o método **consultarIngrediente** para recuperar a descrição de cada ingrediente (Figura 17.41).

O estado composto seguinte, chamado **ConsultandoBebidasPedido**, é semelhante ao anterior, contendo o subestado **ConsultandoQuantidadeBebida**, que dispara o método **selecionarItemCompraBebida**, para selecionar todos os objetos da classe **ItemCompraBebida** associados à compra; esse estado composto contém também o subestado **ConsultandoDescriçãoBebida**, que chama o método **consultarDescriçãoBebida**, cuja função é retornar a descrição da bebida relacionada ao objeto da classe **ItemCompraBebida**.



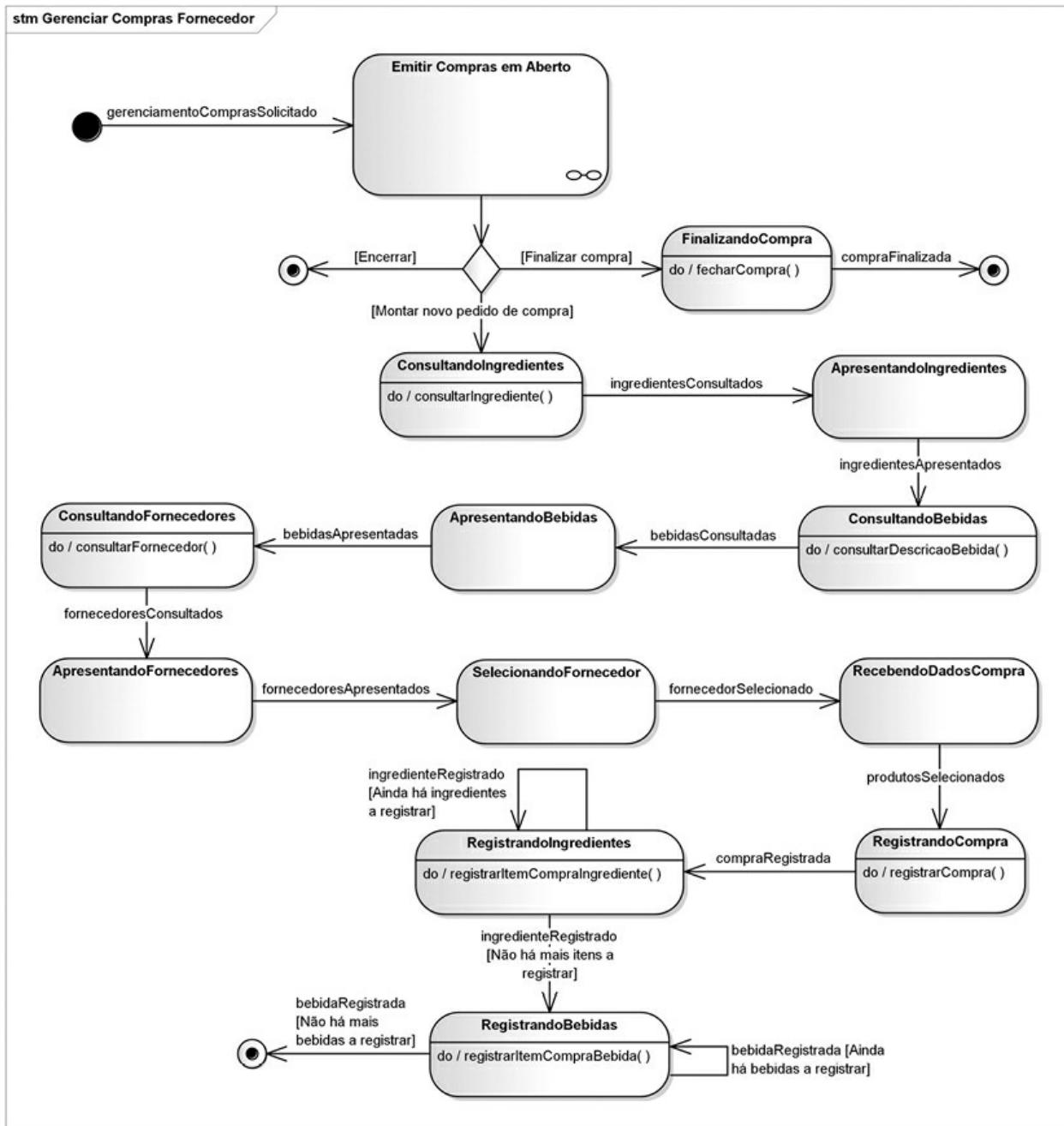
*Figura 17.41 – Diagrama de Máquina de Estados Emitir Compras em Aberto.*

Ao ser finalizado esse último estado composto, é gerada uma transição para um pseudoestado de escolha, em que se verifica se ainda há compras em aberto para consultar, caso em que todo o processo anterior descrito se repete. Caso contrário, são apresentadas as informações das compras em aberto consultadas.

*Diagrama de Máquina de Estados Gerenciar Compras Fornecedores*

Esse processo se inicia com uma transição para um estado de submáquina que se refere ao processo **Emitir Compras em Aberto**, explicado na subseção anterior.

A partir da listagem apresentada por esse estado de submáquina, o administrador deve tomar uma decisão, aqui representada por um pseudoestado de escolha, em que deve optar por finalizar uma compra, montar um novo pedido de compra ou encerrar o processo (Figura 17.42).



*Figura 17.42 – Diagrama de Máquina de Estados Gerenciar Compras Fornecedores.*

Se o administrador desejar finalizar uma compra, deverá selecionar a compra em questão e marcá-la como finalizada, o que gerará uma transição para o estado **FinalizandoCompra**, no qual será executado o método **fecharCompra**, que mudará a situação da compra para 1, significando que se encontra finalizada.

Porém, se o administrador quiser montar um novo pedido de compra,

primeiramente será gerada uma transição para o estado **ConsultandoIngredientes**, em que será executado o método **consultarIngrediente** para recuperar a descrição de todos os ingredientes utilizados pela pizzaria. A conclusão desse estado gera uma transição para o estado **ApresentandoIngredientes**, que carrega na interface os ingredientes consultados no estado anterior.

A seguir, passa-se ao estado chamado **ConsultandoBebidas**, no qual é executado o método **consultarDescricaoBebida** para retornar a descrição de todas as bebidas comerciadas pela PizzaNet. Depois de concluir esse estado, passa-se ao estado **ApresentandoBebidas**, em que são apresentadas as bebidas consultadas.

A seguir, gera-se um novo estado, denominado **ConsultandoFornecedores**, em que é chamado o método **consultarFornecedor** para retornar os dados de todos os fornecedores que trabalham com a empresa. Após a finalização desse estado, como de praxe se passa ao estado em que os fornecedores consultados são apresentados. Após os fornecedores terem sido carregados na interface, assume-se um estado estático, denominado **SelecionandoFornecedor**, no qual se aguarda até que o administrador escolha um fornecedor para solicitar produtos.

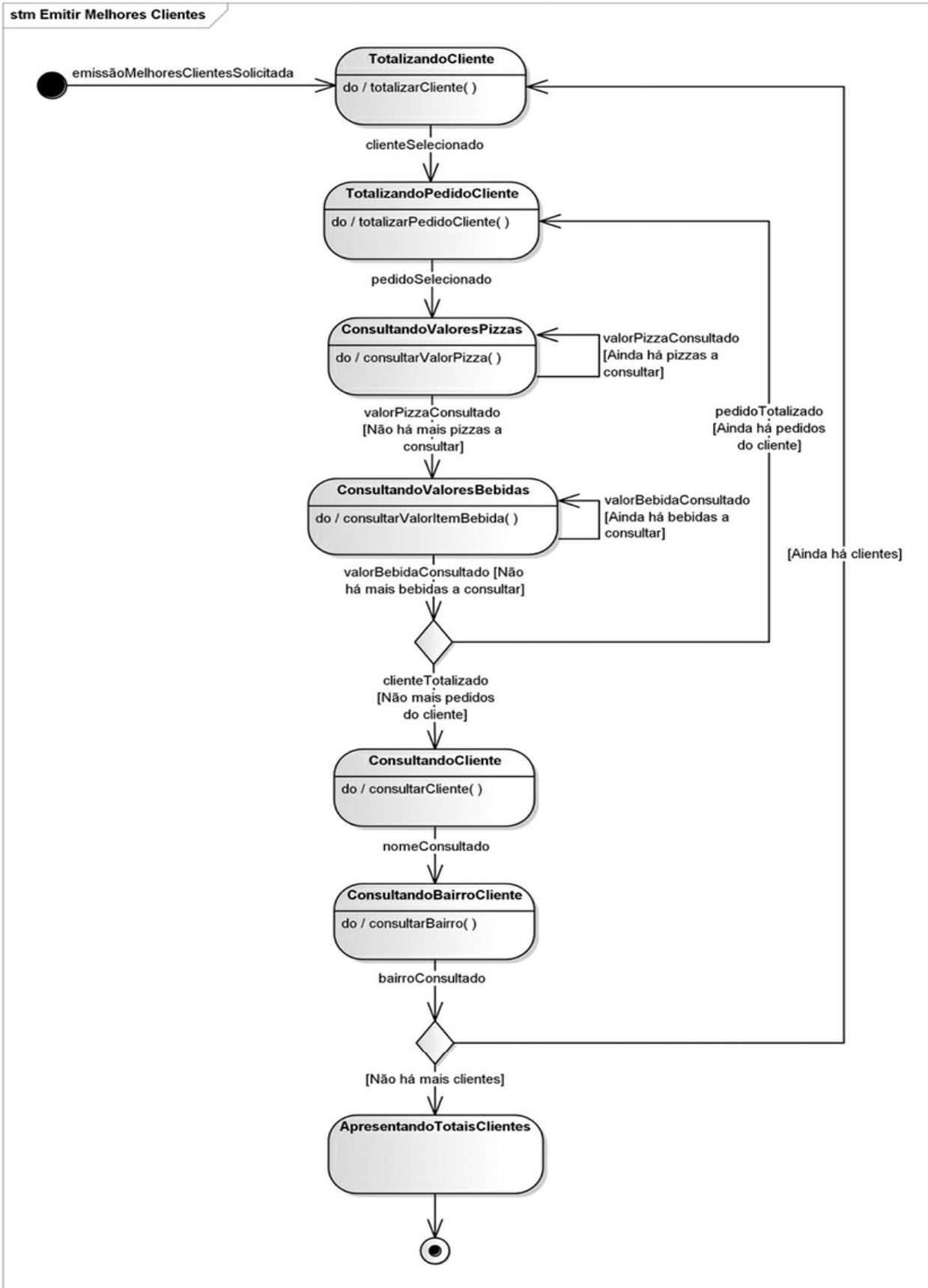
A partir daí, passa-se ao estado **RecebendoDadosCompra**, no qual o estado ficará aguardando que o administrador informe os dados da compra, isto é, selecione os possíveis ingredientes e bebidas desejados. No momento em que os dados são confirmados, é gerada uma transição para o estado **RegistrandoCompra**, em que é disparado o método **registrarCompra**, para instanciar um novo objeto da classe **Compra**. Em seguida, é gerada uma transição para o estado **RegistrandoIngredientes**, no qual é chamado o método **registrarItemCompraIngrediente**, para instanciar os objetos da classe **ItemCompraIngrediente** associados à compra. Posto que pode haver muitos objetos a instanciar, é gerada uma autotransição para esse estado para cada ingrediente selecionado pelo administrador.

Finalmente, após o registro dos ingredientes, gera-se uma transição para o estado **RegistrandoBebidas**, no qual é executado o método **registrarItemCompraBebida**, que faz o mesmo com relação aos objetos

da classe `ItemCompraBebida`, ou seja, gera um novo objeto dessa classe para cada bebida solicitada pelo administrador ao fornecedor. Como muitas bebidas podem ser solicitadas, há uma autotransição para esse estado até que as bebidas solicitadas se esgotem.

*Diagrama de Máquina de Estados Emitir Melhores Clientes*

O primeiro estado desse processo, denominado `TotalizandoCliente`, executa o método `totalizarCliente`, para selecionar cada cliente registrado na empresa e iniciar a totalização de seus pedidos (Figura 17.43).



*Figura 17.43 – Diagrama de Máquina de Estados Emitir Melhores Clientes.*

Após selecionar um cliente, passa-se ao estado **TotalizandoPedidoCliente**, no qual executa o método **totalizarPedidoCliente**, que seleciona cada pedido de um cliente e inicia sua totalização. Como os objetos da classe **Pedido** não possuem todas as informações necessárias para essa totalização, é necessária a execução de métodos em objetos de outras classes.

Assim, no momento em que é selecionado um pedido de um cliente em totalização, é gerada uma transição para o estado **ConsultandoValoresPizza**, que chama o método **consultarValorPizza**, para retornar o valor de cada pizza solicitada no pedido. Como demonstra a autotransição, observe que esse estado se repete tantas vezes quantas forem as pizzas contidas no pedido.

Quando esse estado se conclui, é gerada uma transição para o estado **ConsultandoValoresBebidas**, no qual é executado o método **consultarValorItemBebida**, para retornar o valor de cada bebida solicitada no pedido. Aqui, novamente há uma autotransição que retorna ao estado enquanto houver bebidas relacionadas ao pedido.

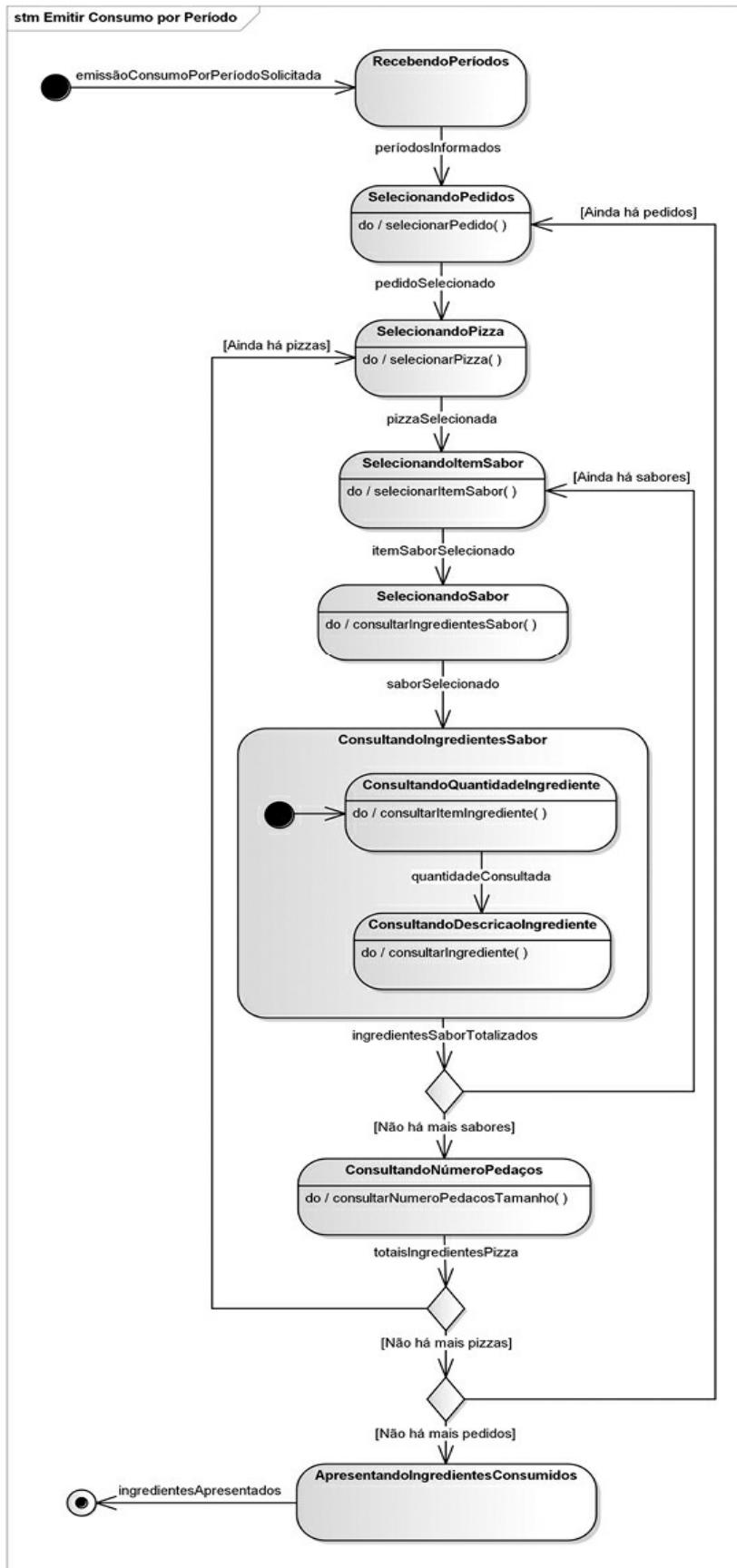
Quando esse estado finaliza, o pedido é totalizado e passa-se a um pseudoestado de escolha, no qual é verificado se ainda há pedidos do cliente selecionado. Em caso positivo, volta-se ao estado **TotalizandoPedidoCliente**, que selecionará o pedido seguinte. Caso contrário, os valores gastos pelo cliente atualmente selecionado são totalizados e passa-se ao estado **ConsultandoCliente**, que executará o método **consultarCliente** e retornará o nome do cliente em questão. Após esse estado ser concluído, passa-se ao estado **ConsultandoBairroCliente** e executa-se o método **consultarBairro** para retornar o bairro do cliente totalizado.

Ao retornar o bairro do cliente, atinge-se um novo pseudoestado de escolha, no qual é verificado se ainda há clientes a totalizar. Em caso positivo, retorna-se ao estado **TotalizandoCliente** e repete-se toda a operação. Caso contrário, passa-se ao estado **ApresentandoTotaisClientes**, em que os clientes e os totais de seus pedidos serão apresentados na interface por ordem de mais consumo.

### *Diagrama de Máquina de Estados Emitir Consumo por Período*

Os estados desse processo se iniciam quando o administrador solicita a listagem de consumo por período. Quando isso ocorre, é gerada uma transição para o estado **RecebendoPeríodos**, que ficará aguardando que o administrador informe as datas abrangidas no relatório. Quando estas forem fornecidas, será gerada uma transição para o estado **SelecionandoPedidos**, no qual é executado o método **selecionarPedido** para retornar cada pedido no período (Figura 17.44).

A seguir, passa-se ao estado **SelecionandoPizza**, no qual se executa o método **selecionarPizza**, para selecionar cada pizza de um pedido específico. No momento em que uma pizza for selecionada, passa-se ao estado seguinte **SelecionandoItemSabor**, no qual é executado o método **selecionarItemSabor**, para selecionar os objetos da classe **ItemSabor** associados ao objeto da classe **Pizza**.



*Figura 17.44 – Diagrama de Máquina de Estados Emitir Consumo por Período.*

Quando um objeto da classe **ItemSabor** é selecionado, passa-se para o estado composto **ConsultandoIngredientesSabor**, que contém dois subestados. O primeiro deles, **ConsultandoQuantidadeIngrediente**, executa o método **consultarItemIngrediente**, que retorna a quantidade de um objeto da classe **ItemIngrediente** associado ao objeto **Sabor**. O segundo subestado, denominado **ConsultandoDescricaoIngrediente**, dispara o método **consultarIngrediente**, para retornar a descrição de cada ingrediente.

A conclusão desse estado composto conclui a totalização dos ingredientes do sabor selecionado e gera uma transição para um pseudoestado de escolha em que se verifica se ainda há sabores a totalizar. Em caso positivo, volta-se ao estado **SelecionandoItemSabor** para selecionar o objeto **ItemSabor** seguinte. Caso contrário, passa-se ao estado **ConsultandoNúmeroPedaços**, em que é chamado o método **consultarNúmeroPedacosTamanho**, para retornar o número de pedaços da pizza selecionada, que será utilizado para totalizar os ingredientes da pizza.

Depois, atinge-se um novo pseudoestado de escolha, no qual se determina se ainda há pizzas no pedido. Em caso positivo, retorna-se ao estado **SelecionandoPizza**, para selecionar a próxima pizza do pedido atualmente sob totalização. Caso contrário, atinge-se um último pseudoestado de escolha que verifica se ainda há pedidos a totalizar.

Caso ainda haja pedidos, volta-se ao estado **SelecionandoPedidos** e repete-se todo o processo. Caso contrário, gera-se uma transição para o estado **ApresentandoIngredientesConsumidos**, no qual são apresentados os totais dos ingredientes consultados e encerra-se o processo.

### **17.2.9 Diagramas de Atividade da PizzaNet**

Nesta seção, serão modeladas as atividades dos processos apresentados anteriormente, por meio do diagrama de atividade. O leitor notará que muitos dos diagramas apresentados nesta seção estão bastante detalhados, pois o diagrama de atividade é um dos diagramas mais detalhados da UML. Em alguns casos, os diagramas poderiam até mesmo ser

simplificados. No entanto, as instruções representadas pelos diagramas desta seção são gerais, uma vez que estão dissociadas de uma linguagem de programação específica.

#### *Diagrama de Atividade Escolher Pizza*

A primeira ação modelada nesse diagrama representa a seleção de todos os sabores registrados no sistema. Tais informações são recuperadas de objetos da classe **Sabor**, como demonstra a direção do fluxo de objetos. Na próxima ação, a atividade posiciona-se sobre o primeiro objeto da classe **Sabor** encontrado e a ação seguinte retorna a descrição do sabor. Na verdade, essa última ação, a rigor, também deveria ter um fluxo de objetos com um objeto da classe **Sabor**, mas optamos por apresentar somente as operações mais importantes nesses diagramas.

A seguir, a atividade encontra um nó de decisão que deve verificar se ainda existem sabores a apresentar. Se isso for verdadeiro, a atividade posiciona-se sobre o próximo sabor e volta a retornar sua descrição, ficando nesse laço enquanto houver sabores a apresentar.

Quando o laço se encerrar, passa-se à ação em que a atividade ficará aguardando que o usuário escolha o tamanho da pizza que deseja. Quando este for escolhido, a atividade passa à ação que consulta a quantidade de sabores permitidos para o tamanho. A seguir, passa-se à ação na qual a atividade recebe os sabores desejados pelo cliente (Figura 17.45).

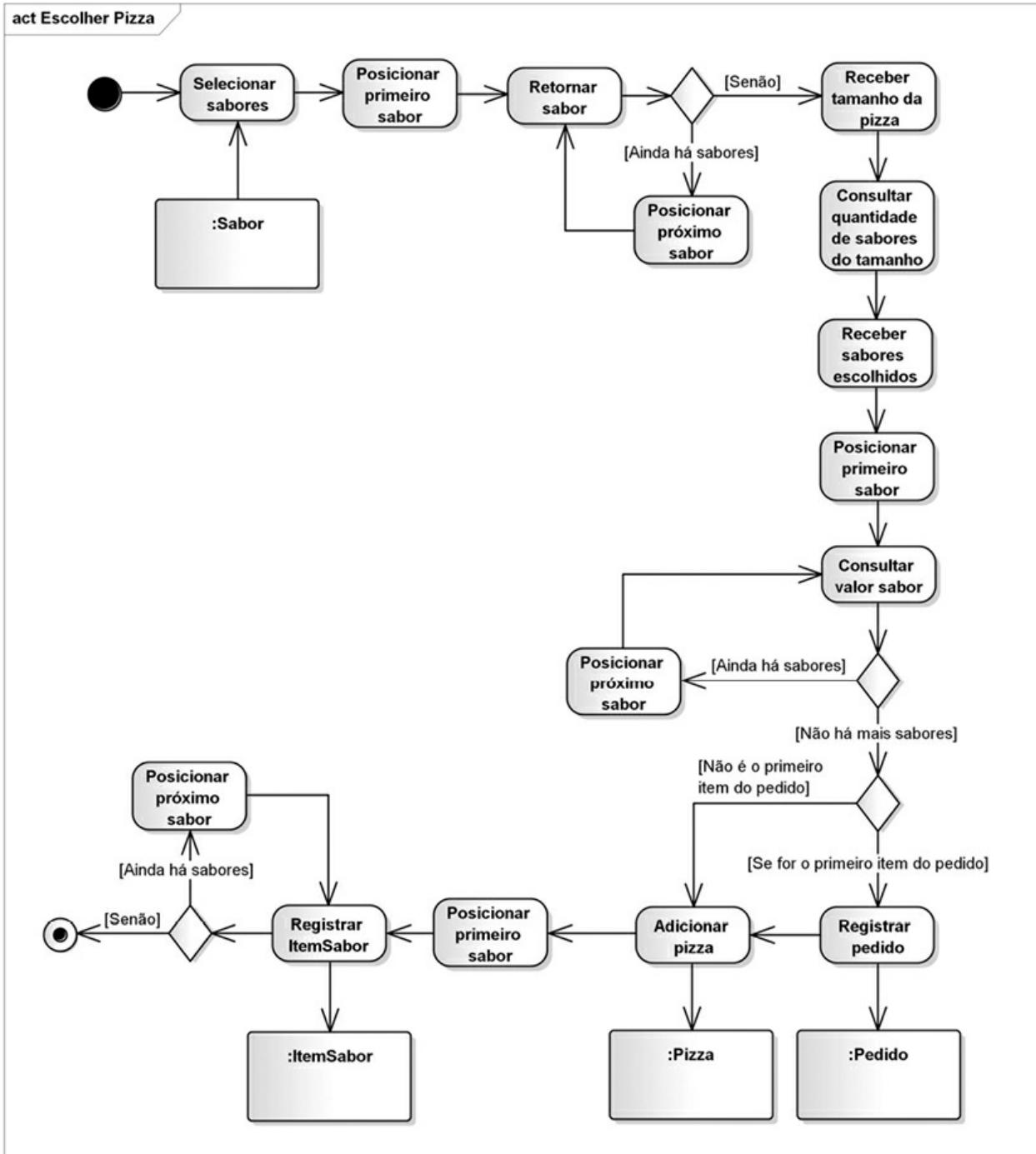


Figura 17.45 – Diagrama de Atividade Escolher Pizza.

No momento em que os sabores são escolhidos, a atividade executa a ação em que se posiciona sobre o primeiro sabor escolhido e, em seguida, consulta seu valor, passando para um nó de decisão que determina se ainda há sabores a consultar, caso em que a atividade se posiciona sobre o próximo sabor e repete a operação.

Caso não haja mais sabores, a atividade atinge outro nó de decisão, que deve verificar se a pizza escolhida é o primeiro item do pedido. Se isso for verdadeiro, será executada a ação **Registrar pedido**, que instancia um novo objeto da classe **Pedido**, conforme demonstra o fluxo de objetos.

A seguir, se a pizza não se constituir no primeiro item do pedido, a atividade passará à ação em que a pizza é adicionada ao pedido, gerando-se um novo objeto da classe **Pizza**. Em seguida, a atividade se posiciona sobre o primeiro sabor escolhido e inicia um laço onde o sabor é registrado, gerando-se um novo objeto da classe **ItemSabor**, passando-se a um nó de decisão que verifica se ainda há sabores a registrar. Em caso positivo, a atividade se posiciona sobre o próximo sabor, registrando-o a seguir. Esse laço é executado enquanto houver sabores; no momento em que não houver mais sabores, a atividade será encerrada.

#### *Diagrama de Atividade Escolher Bebida*

A primeira ação modelada nesse diagrama refere-se à consulta dos tipos de bebida oferecidos pela pizzaria. Essa informação é recuperada de objetos da classe **TipoBebida**, conforme demonstra o fluxo de objetos. Após essa consulta, a atividade se posiciona no primeiro tipo de bebida recuperado e retorna sua descrição. Em seguida, a atividade atinge um nó de decisão em que se determina se ainda há tipos de bebida a apresentar. Em caso positivo, passa-se à ação em que a atividade se posiciona no tipo de bebida seguinte e volta a retornar sua descrição.

Caso não haja mais tipos de bebida, passa-se à ação em que se aguarda que o cliente escolha um tipo de bebida. A ação seguinte consulta o tipo de bebida escolhido e passa a ação em que serão selecionadas todas as bebidas do tipo selecionado. Observe que existe um fluxo de objetos nessa ação e que a seta desse fluxo indica que as informações são transmitidas de objetos da classe **Bebida** para a ação (Figura 17.46).

A seguir, a atividade posiciona-se sobre o primeiro objeto da classe **Bebida** encontrado, passando para a ação em que é retornada a descrição da bebida e, em seguida, à ação em que é retornado o valor da bebida em questão. A seguir, a atividade atinge um nó de decisão, em que se deve determinar se ainda há bebidas a pesquisar, caso em que a atividade posiciona-se sobre a próxima bebida e repete a operação, ficando nesse

laço enquanto houver bebidas a consultar.

Quando não houver mais bebidas a apresentar, a atividade passará a ação em que serão recebidas as bebidas e as quantidades desejadas pelo cliente, passando-se para um nó de decisão que verificará se a bebida selecionada constitui-se no primeiro item do pedido. Nesse caso, a atividade executará a ação **Registrar pedido**, em que é instanciado um novo objeto da classe **Pedido**, como demonstra o fluxo de objetos.

Depois disso, se a bebida não for o primeiro item do pedido, a atividade passará à ação em que é instanciado um novo objeto da classe **ItemBebida**, referente à bebida selecionada.

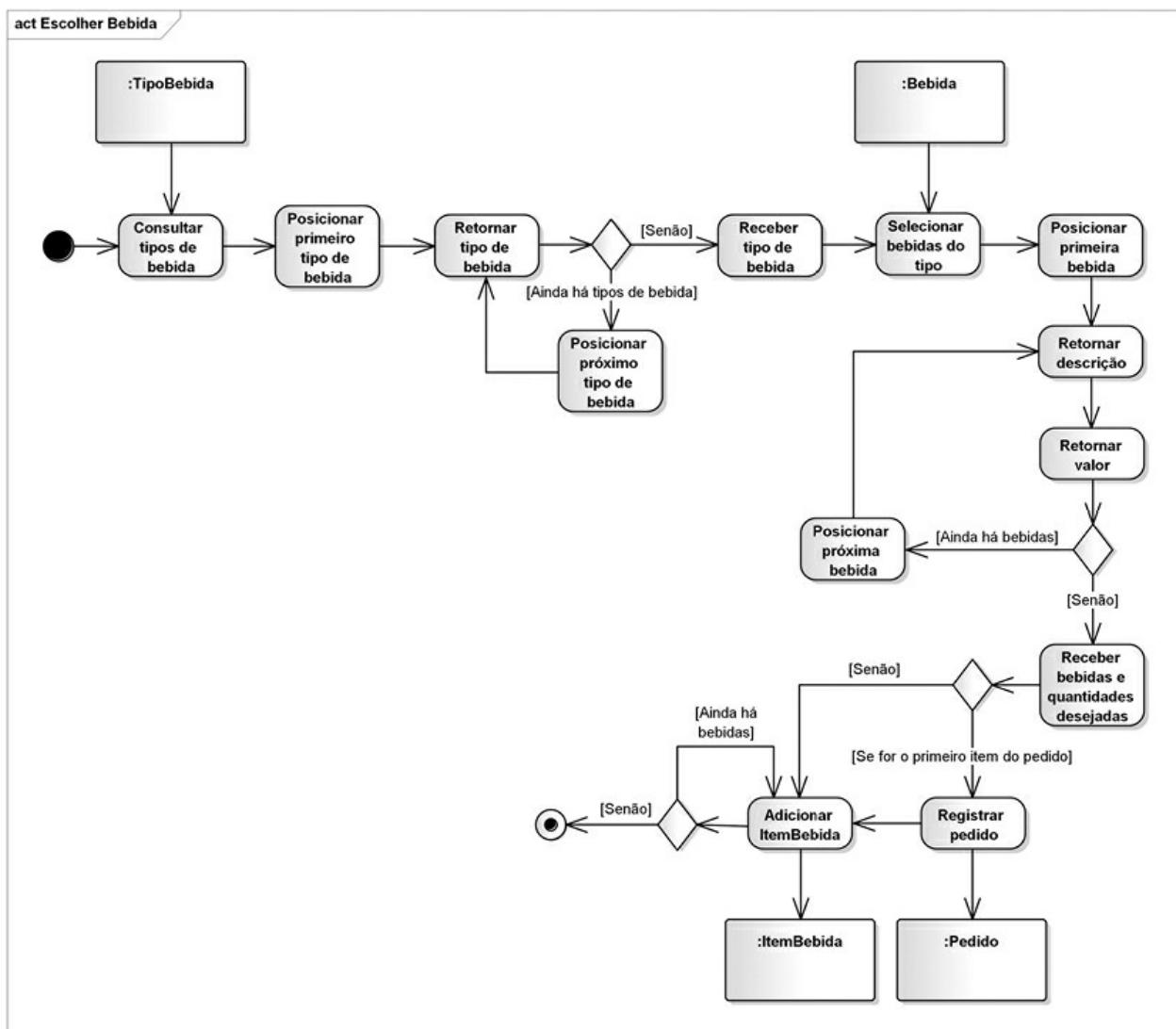


Figura 17.46 – Diagrama de Atividade Escolher Bebida.

### *Diagrama de Atividade Logar*

Esse diagrama inicia-se com um nó de decisão cuja função é determinar se o cliente já se encontra registrado. Se o cliente não estiver registrado, o fluxo atingirá uma ação de chamada de comportamento que invocará a atividade **Autorregisterar**, por meio da qual o usuário poderá realizar o próprio cadastro no sistema (Figura 17.47).

Caso o cliente já se encontre cadastrado, serão executadas as ações relativas à atividade **Logar** propriamente dita. Nessa atividade, primeiramente serão recebidos o **nome-login** e a **senha** do cliente. A seguir, passa-se à ação **validar login**, para determinar se esse é um **nome-login** válido, o que leva a um nó de decisão que deve realizar essa verificação.

Se o **nome-login** for considerado inválido, a atividade será encerrada, caso contrário, passa-se à ação em que a senha é validada, o que leva a um segundo nó de decisão, em que é verificado se a senha fornecida é válida. Caso a senha não seja válida, a atividade será encerrada; caso contrário, executa-se a ação **Logar**, que autentica o cliente no sistema. Depois, encerra-se a atividade.

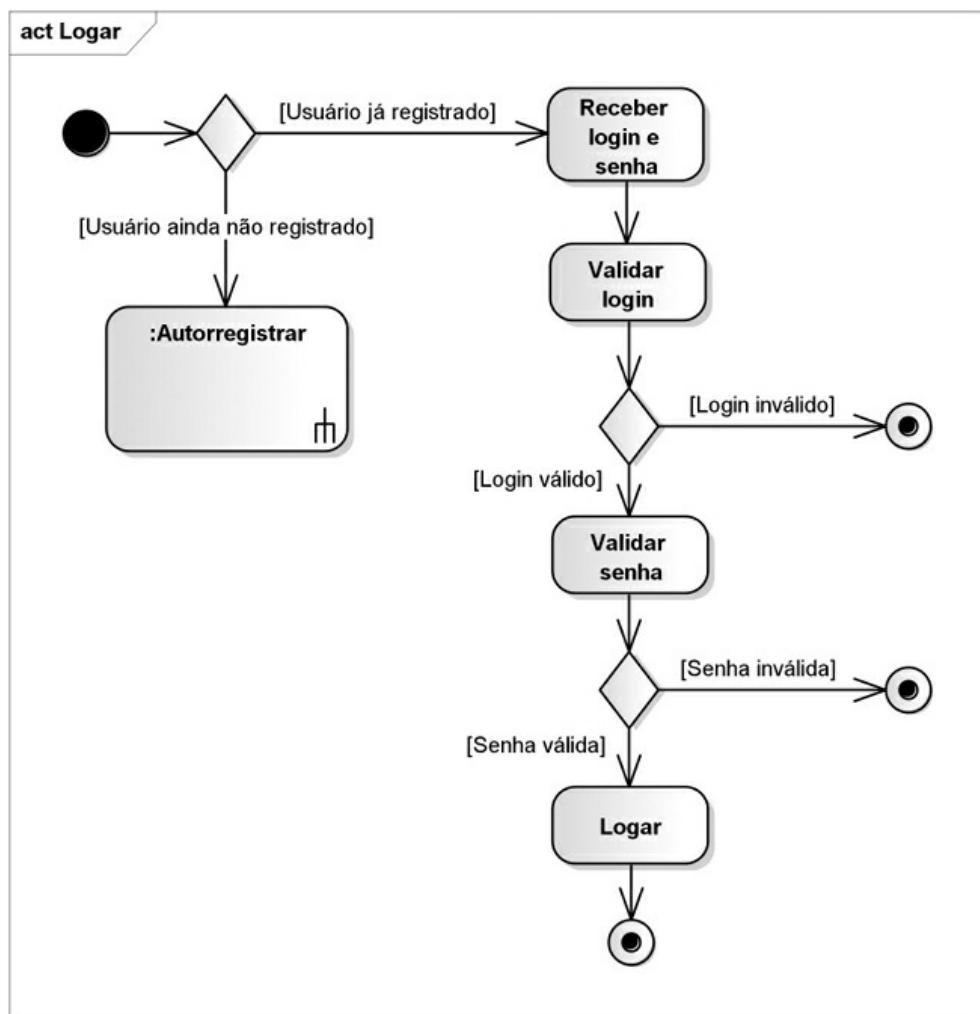
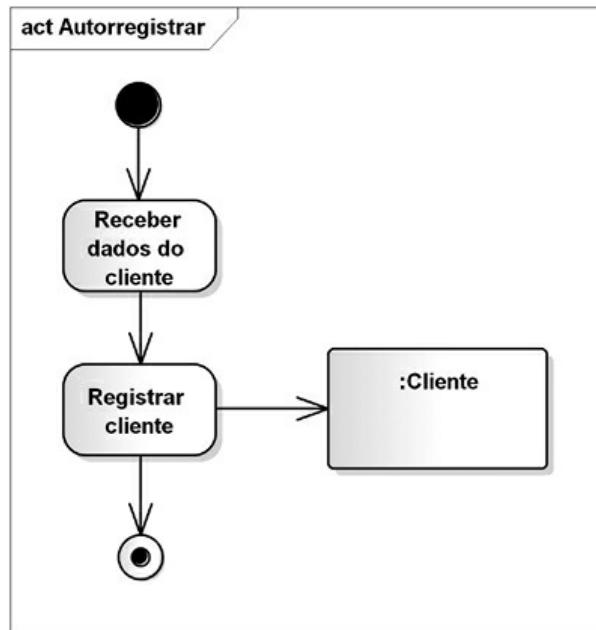


Figura 17.47 – Diagrama de Atividade Logar.

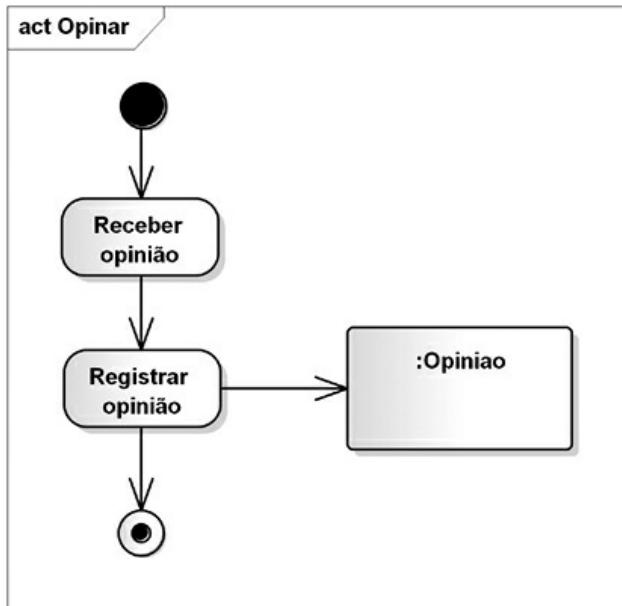
Diagrama de Atividade Autorregisterar



*Figura 17.48 – Diagrama de Atividade Autorregisterar.*

Esta é uma atividade muito simples composta de apenas dois nós de ação. No primeiro, são recebidos os dados do cliente e, no segundo, este é registrado, gerando-se uma nova instância da classe **Cliente**, conforme demonstra o fluxo de objetos entre essa ação e o objeto da classe **Cliente**.

#### *Diagrama de Atividade Opinar*



*Figura 17.49 – Diagrama de Atividade Opinar.*

Essa atividade é bastante semelhante à anterior, composta igualmente de

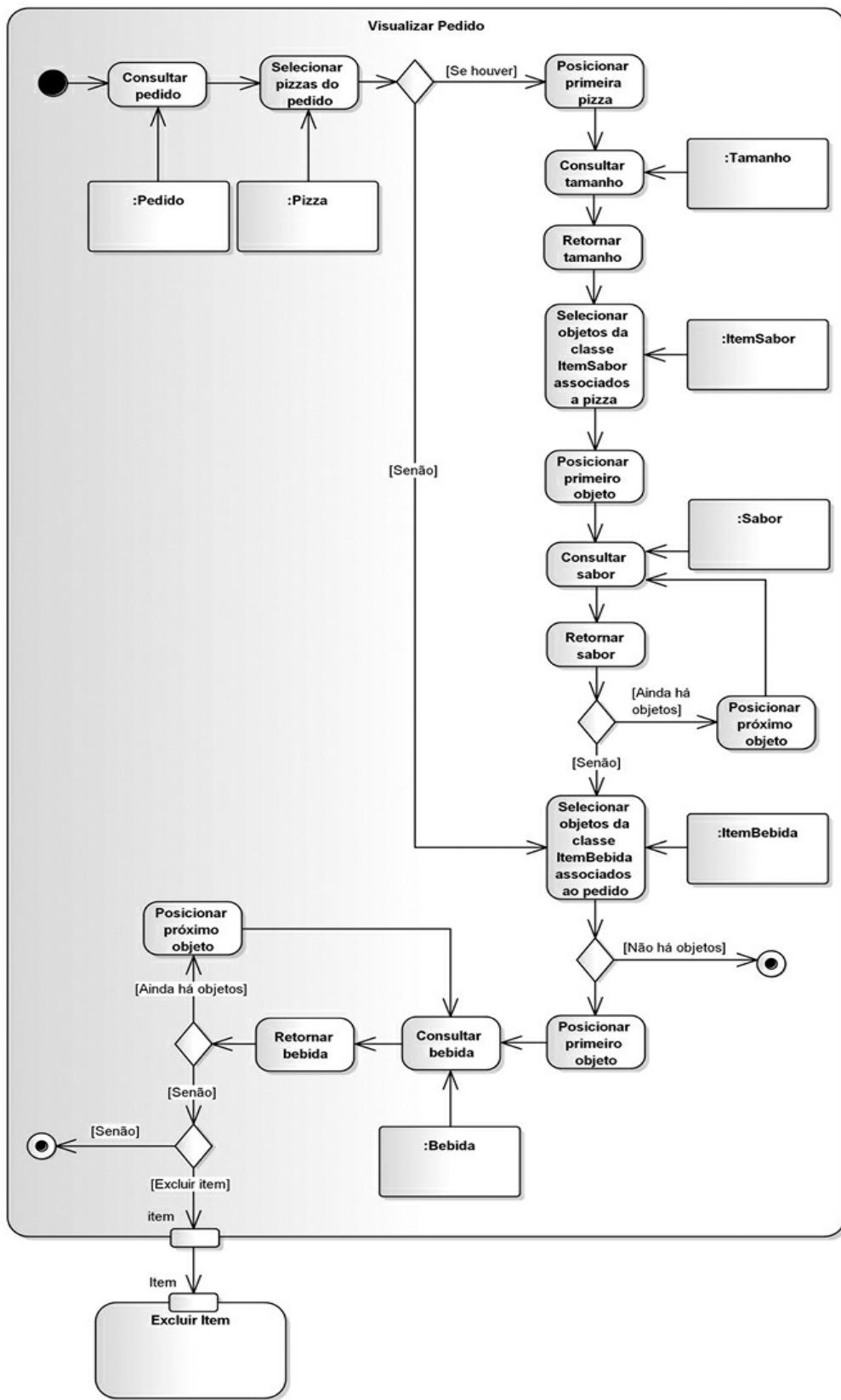
dois nós de ação. No primeiro nó de ação, é recebida a opinião do cliente e, no segundo, esta é registrada, gerando um novo objeto da classe **Opiniao**.

#### *Diagrama de Atividade Visualizar Pedido*

Esse diagrama contém duas atividades, sendo a segunda apenas referenciada, sem detalhar seus nós de ação. A atividade principal, denominada **Visualizar Pedido**, inicia-se pelo nó de ação que consulta o pedido a ser visualizado. O leitor perceberá que existe um fluxo de objeto entre essa ação e um objeto da classe **Pedido**, sobre o qual é realizada essa operação.

Aqui, cabe um comentário: temos evitado nos diagramas de atividade inserir muitos fluxos de objeto para não os deixar carregados demais, porém, como essa atividade envolve muitos objetos de diferentes classes, preferimos, nesse caso, detalhar todos os fluxos de objeto desse diagrama, o que também permite ilustrar um desses casos ao leitor. Em outros diagramas, no entanto, tentaremos modelar somente os fluxos de objeto mais importantes, detalhando todos os fluxos de objeto somente quando isso for realmente necessário para compreender melhor o diagrama, como é a situação desse exemplo (Figura 17.50).

act Visualizar Pedido



*Figura 17.50 – Diagrama de Atividade Visualizar Pedido.*

Após a consulta do pedido, passa-se a ação que seleciona as pizzas do pedido consultado. Note que essa informação é recuperada por meio de um fluxo de objetos, a partir de objetos da classe **Pizza**. Logo depois, a atividade encontra um nó de decisão, em que é verificado se o pedido inclui pizzas. Em caso afirmativo, é executada a ação em que a atividade posiciona-se sobre o primeiro objeto da classe **Pizza**. Na ação seguinte, é consultado o tamanho da pizza, demonstrado por um fluxo de objetos, e passa-se à ação na qual esse tamanho é retornado.

A seguir, são selecionados todos os objetos da classe **ItemSabor** associados ao objeto da classe **Pizza**. Observe que há um fluxo de objetos entre essa ação e um objeto da classe **ItemSabor** para representar essa seleção. Em seguida, a atividade posiciona-se sobre o primeiro objeto da classe **ItemSabor** e, na ação seguinte, é consultada a descrição desse **Sabor**, buscada de um objeto da classe **Sabor**, como demonstra o fluxo de objetos. A ação seguinte retorna a descrição do sabor consultado.

Em seguida, a atividade encontra um nó de decisão que verifica se ainda há objetos da classe **ItemSabor** para apresentar. Em caso positivo, a atividade posiciona-se sobre o próximo objeto e repete a operação descrita anteriormente, ficando nesse laço enquanto houver objetos a apresentar.

Quando não houver mais objetos da classe **ItemSabor** a apresentar (ou se não existirem pizzas no pedido), será executada a ação **Selecionar objetos da classe ItemBebida associados ao pedido**, que, por meio de um fluxo de objetos, seleciona todos os objetos da classe **ItemBebida** associados ao objeto **Pedido**, consultado no início da atividade.

Se não forem encontrados objetos dessa classe associados ao pedido, o processo será encerrado. Caso contrário, a atividade posiciona-se sobre o primeiro objeto da classe **ItemBebida** e, na próxima ação, consulta a descrição desse item por meio de um fluxo de objetos com um objeto da classe **Bebida**. A descrição da bebida é retornada pela ação seguinte.

A atividade passa, então, a um nó de decisão, que verifica se ainda há objetos da classe **ItemBebida** a apresentar. Se isso for verdadeiro, a atividade posiciona-se sobre o próximo objeto e volta a consultar o objeto **Bebida** a ele associado, repetindo essa operação enquanto houver objetos a

apresentar.

Se não houver mais objetos a apresentar, a atividade encontra outro nó de decisão, em que o cliente pode escolher entre terminar o processo ou excluir algum item apresentado. Nesse último caso, o fluxo atinge um nó de parâmetro de atividade que receberá o item a excluir. Esse valor será passado como parâmetro por meio de um fluxo de objetos para a atividade **Excluir Item**, que o receberá através de outro nó de parâmetro de atividade. Essa atividade é responsável por excluir um item do pedido e será explicada na próxima subseção.

#### *Diagrama de Atividade Excluir Item*

Esse diagrama inicia-se com um nó de decisão em que é preciso determinar se o item a excluir trata-se de uma pizza ou de uma bebida. No primeiro caso, executa-se uma ação para excluir um objeto da classe **Pizza**, conforme demonstra o fluxo de objetos. Depois disso, a atividade posiciona-se sobre o primeiro objeto da classe **ItemSabor** associado ao objeto da classe **Pizza**. Em seguida, a atividade executa uma ação para excluir o objeto da classe **ItemSabor**. Observe que há um fluxo de objetos entre o nó de ação e o objeto da classe **ItemSabor**, que representa a exclusão do objeto (Figura 17.51).

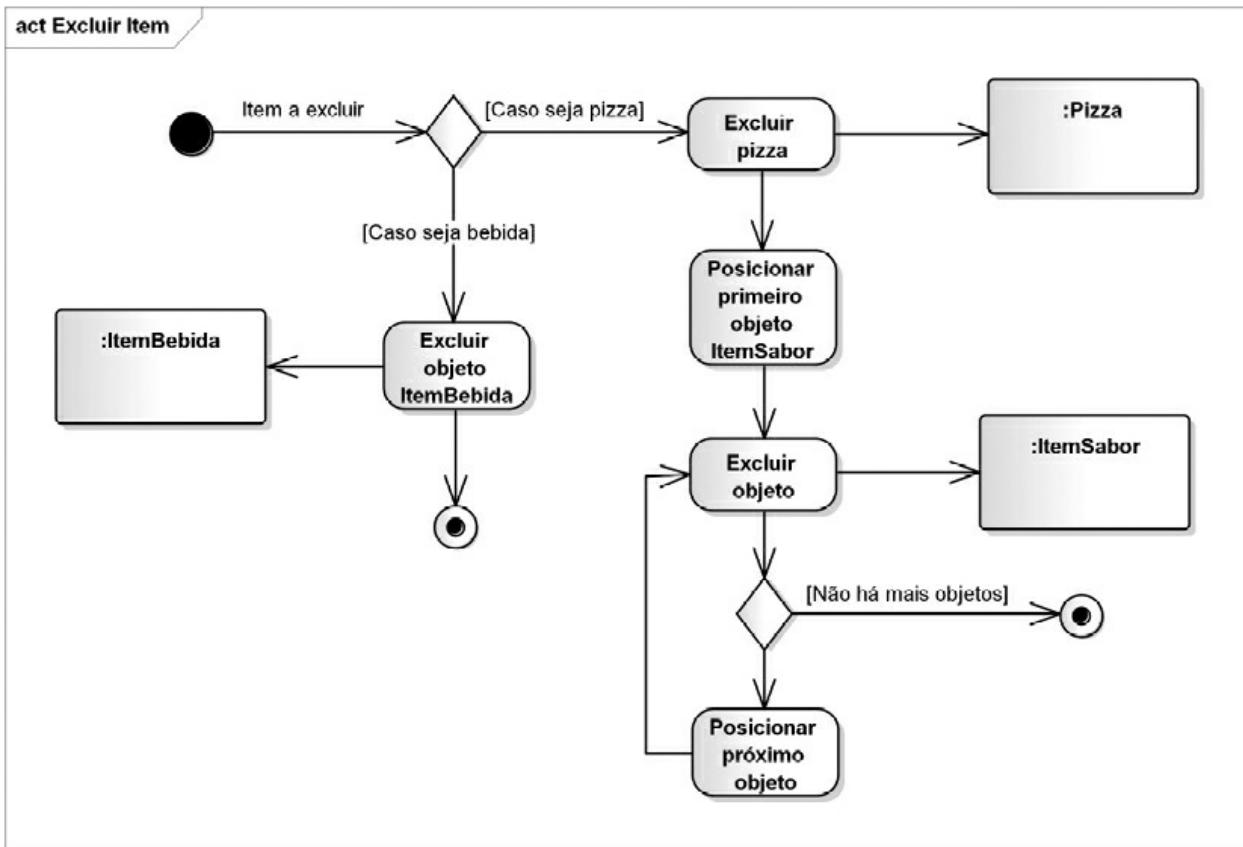


Figura 17.51 – Diagrama de Atividade Excluir Item.

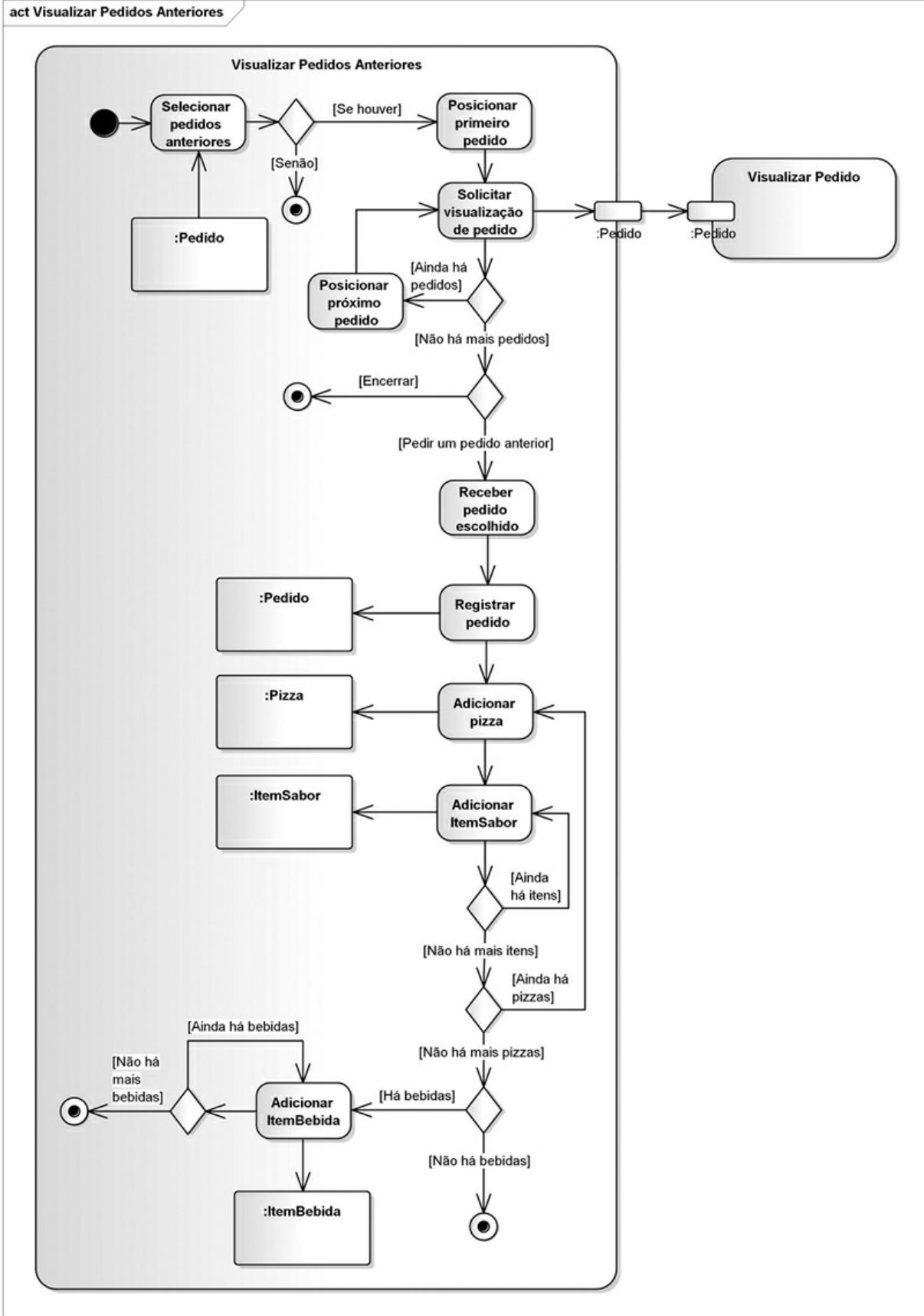
A seguir, a atividade encontra um nó de decisão que verifica se ainda há objetos. Se não houver mais objetos, a atividade será encerrada, caso contrário será executada uma ação que se posiciona no próximo objeto e em seguida o exclui. Esse laço se repete enquanto houver objetos da classe **ItemSabor** para ser excluídos. Já se o item a excluir se tratar de uma bebida, a atividade simplesmente executará uma ação para destruir o objeto da classe **ItemBebida**.

#### Diagrama de Atividade Visualizar Pedidos Anteriores

Esse diagrama é composto de duas atividades: a principal atividade, chamada **Visualizar Pedidos Anteriores**, detalha seus nós de ação, enquanto a atividade **Visualizar Pedido** é apenas referenciada (Figura 17.52).

Na atividade principal, primeiramente é executada uma ação para selecionar todos os pedidos anteriores do cliente. Observe que essa pesquisa é feita por meio de um fluxo de objetos, representando a consulta a objetos da classe **Pedido**.

A seguir, a atividade passa a um nó de decisão, que deve verificar se foram encontrados pedidos anteriores do cliente. Em caso negativo, a atividade é encerrada. Caso contrário, a atividade posiciona-se sobre o primeiro objeto da classe **Pedido** encontrado e, na ação seguinte, solicita a visualização desse pedido. Como o processo de visualização de um pedido foi modelado anteriormente em outro diagrama, passa-se o número do pedido por meio de um nó de parâmetro de atividade para a atividade **Visualizar Pedido**, que se encarregará de detalhá-lo.



### *Figura 17.52 – Diagrama de Atividade Visualizar Pedidos Anteriores.*

A seguir, a atividade encontra um nó de decisão, que testa se ainda há pedidos a apresentar. Em caso positivo, a atividade posiciona-se sobre o próximo objeto da classe **Pedido** e volta a solicitar sua visualização, permanecendo nesse laço enquanto houver pedidos a apresentar.

Quando esse laço se encerra, passa-se a um novo nó de decisão, que representa a escolha do cliente, que deve decidir entre encerrar o processo ou repetir a solicitação de um pedido anterior. Caso o cliente escolha a segunda alternativa, a atividade receberá o pedido escolhido pelo cliente, passando a seguir para a ação que regista o novo pedido, instanciando um novo objeto da classe **Pedido**, como demonstra o fluxo de objetos.

Depois disso, a atividade registrará a primeira pizza do pedido, instanciando um novo objeto da classe **Pizza**, como demonstra o fluxo de objetos e, em seguida, registrará o primeiro sabor da pizza, instanciando um novo objeto da classe **ItemSabor**. A seguir, um nó de decisão verifica se ainda há sabores associados à pizza, caso em que se repete a última ação. Se não houver mais sabores, um novo nó de decisão testará se ainda há pizzas associadas ao pedido, situação em que o fluxo volta à ação **Adicionar pizza** e repete o comportamento já descrito, permanecendo nesse laço enquanto houver pizzas para adicionar.

Quando não houver mais pizzas, a atividade encontrará um novo nó de decisão que testa se há bebidas associadas ao pedido. Caso não haja, a atividade será encerrada. Caso contrário, será instanciado um novo objeto da classe **ItemBebida** para cada bebida solicitada no pedido, como demonstra o nó de decisão que verifica se ainda há bebidas. Quando não houver mais bebidas, a atividade será encerrada.

### *Diagrama de Atividade Visualizar Sabores Mais Pedidos*

Para desenvolver essa atividade, primeiramente é executada a ação que seleciona todos os objetos da classe **ItemSabor**, conforme demonstra o fluxo de objetos. É importante lembrar que cada objeto dessa classe armazena um sabor de uma pizza específica contida em um pedido. Após a seleção, a atividade posiciona-se sobre o primeiro objeto encontrado, e, na ação seguinte, a ocorrência da solicitação do sabor em questão é somada a um totalizador. Haverá um totalizador para cada sabor (Figura 17.53).

Em seguida, a atividade encontra um nó de decisão que verifica se ainda há objetos da classe **ItemSabor**. Em caso positivo, a atividade se posiciona sobre o próximo objeto e volta à ação que totaliza os sabores.

Quando não houver mais objetos, a atividade será posicionada sobre o primeiro totalizador, passando à ação que consulta a descrição do sabor em um objeto da classe **Sabor**, como demonstra o fluxo de objetos. A ação seguinte retorna a descrição e o total do sabor totalizado e passa a um nó de decisão, em que é verificado se ainda há totalizadores, caso em que a atividade se posicionará sobre o próximo totalizador e repetirá a consulta da descrição do sabor, conforme já explicado. Quando não houver mais totalizadores, a atividade será encerrada.

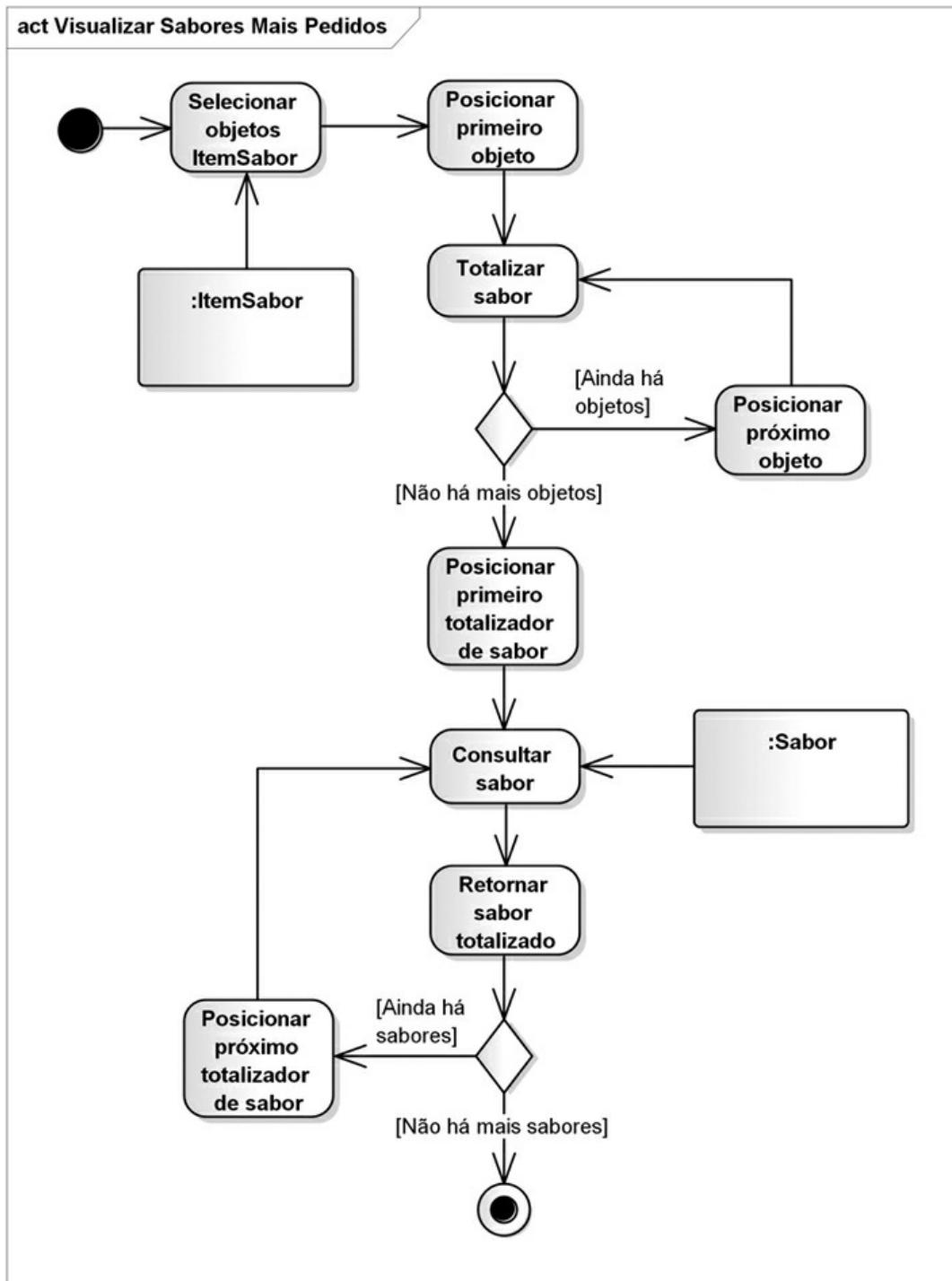


Figura 17.53 – Diagrama de Atividade Visualizar Sabores Mais Pedidos.

#### Diagrama de Atividade Concluir Pedido

O diagrama inicia-se com um nó de decisão, que deve verificar se o cliente ainda não se autenticou, caso em que será necessário invocar o comportamento de **Logar**, já descrito em outro diagrama de atividade. A

seguir, caso o cliente já esteja autenticado, passa-se à ação que invoca o comportamento de **Visualizar Pedido**, também descrito em outro diagrama de atividade, no qual serão apresentados mais uma vez os itens contidos no pedido do cliente, esperando uma última confirmação (Figura 17.54).

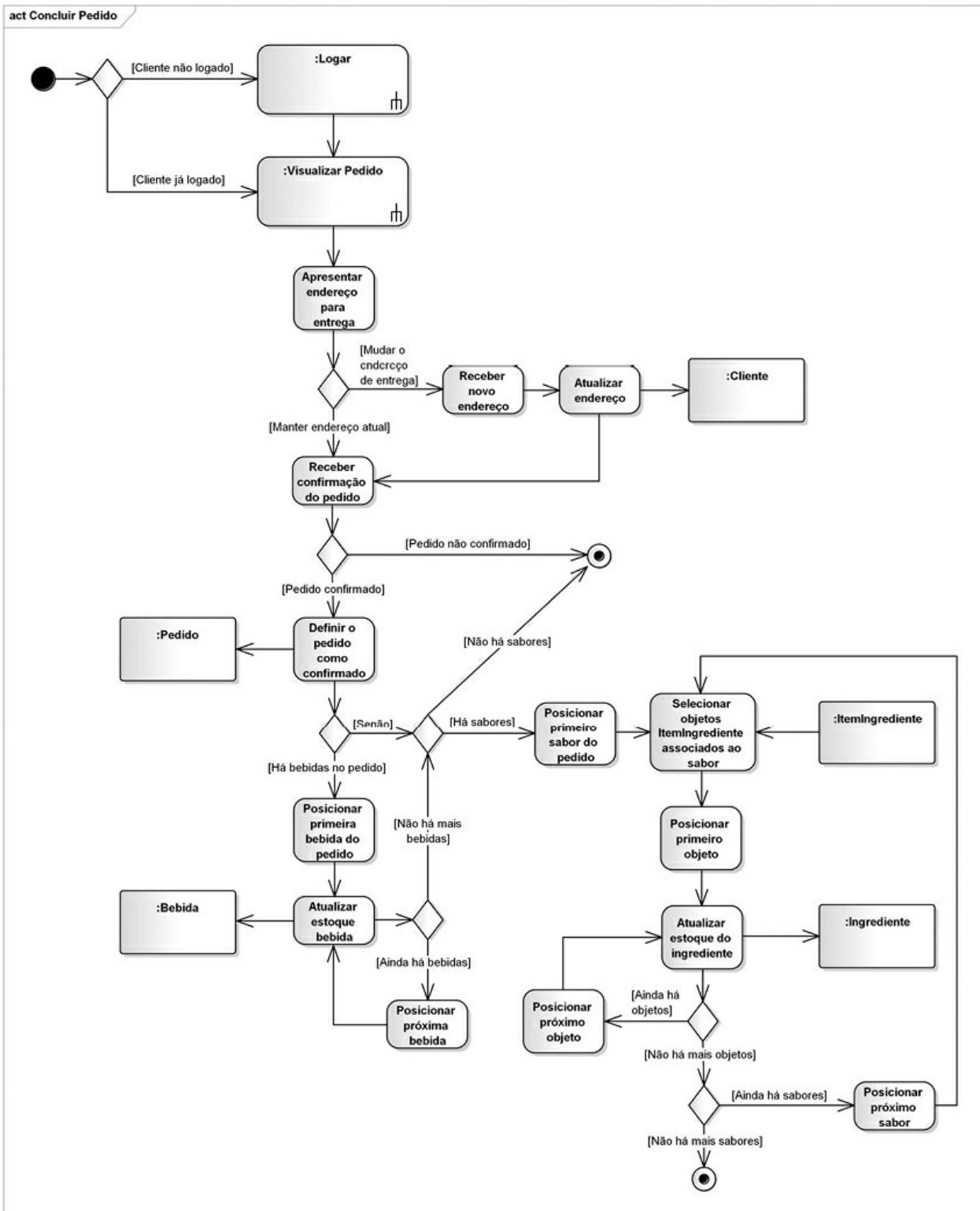


Figura 17.54 – Diagrama de Atividade Concluir Pedido.

A seguir, passa-se a apresentar o endereço para entrega do pedido e, em seguida, a atividade encontra um nó de decisão que verifica se o cliente

deseja mudar o endereço de entrega. Se isso for verdadeiro, passa-se à ação que recebe o novo endereço e, a seguir, à ação que atualiza o endereço do cliente. O fluxo de objetos demonstra que a atualização é feita sobre um objeto da classe **Cliente**.

A seguir, ou caso o cliente não queira modificar seu endereço, passa-se à ação **Receber confirmação do pedido**, em que se aguarda que o cliente confirme o pedido ou não. Após o cliente informar sua decisão, passa-se a um nó de decisão que julgará a resposta do cliente. Caso o cliente não tenha confirmado o pedido, a atividade será encerrada. Caso contrário, passa-se à ação **Definir o pedido como confirmado**, que alterará a situação do pedido para 1, que, por convenção, determina que o pedido foi confirmado pelo cliente. Depois disso, passa-se a um nó de decisão, no qual se deve verificar se o pedido contém bebidas. Em caso positivo, a atividade se posicionará sobre a primeira bebida do pedido e, em seguida, executará a ação **Atualizar estoque bebida**, que diminuirá a quantidade solicitada do estoque da bebida, como demonstra o fluxo de objetos.

Em seguida, passa-se a outro nó de decisão que testa se ainda há bebidas no pedido, caso em que a atividade será posicionada sobre a próxima bebida e repetirá a operação, permanecendo nesse laço até não haver mais bebidas a atualizar no estoque.

Quando isso ocorrer, ou caso não haja bebidas no pedido, passa-se a um novo nó de decisão em que se deve verificar se há sabores associados ao pedido. Se não houver sabores, a atividade será encerrada nesse ponto, caso contrário, será posicionada sobre o primeiro sabor, passando a seguir para a ação que seleciona os objetos da classe **ItemIngrediente** associados ao sabor, conforme demonstra o fluxo de objetos entre essa ação e um objeto da classe **ItemIngrediente**.

A seguir, a atividade posiciona-se sobre o primeiro desses objetos, passando a seguir para a ação que atualiza o estoque do ingrediente, diminuindo a quantidade necessária para produzir o sabor de sua quantidade em estoque. Isso é demonstrado por meio de um fluxo de objetos.

Logo a seguir, a atividade encontra um nó de decisão no qual se deve verificar se ainda há objetos da classe **ItemIngrediente**, caso em que a atividade deverá se posicionar sobre o objeto seguinte e repetir a

atualização do estoque.

Quando não houver mais objetos da classe **ItemIngrediente**, a atividade verificará se ainda há sabores a atualizar, caso em seré posicionada sobre o próximo sabor e voltará a selecionar os objetos da classe **ItemIngrediente**, repetindo-se a atualização do estoque.

Finalmente, quando não houver mais sabores associados ao pedido confirmado, a atividade será encerrada.

#### *Diagrama de Atividade Visualizar Pedidos em Aberto*

Esse diagrama representa três atividades: somente a atividade principal detalha seus nós de ação, visto que as outras atividades são apenas referenciadas.

A atividade principal, que se refere ao processo de visualização de pedidos em aberto propriamente dito, inicia-se com a ação **Selecionar pedidos em aberto**, que busca entre os objetos da classe **Pedido** todos os pedidos cuja situação seja 1, ou seja, que ainda não foram atendidos. O nó de decisão, imediatamente seguinte a esse nó de ação, deve determinar se foram encontrados pedidos em aberto nessa pesquisa. Caso não tenha sido encontrado nenhum pedido nessa situação, a atividade será encerrada (Figura 17.55).

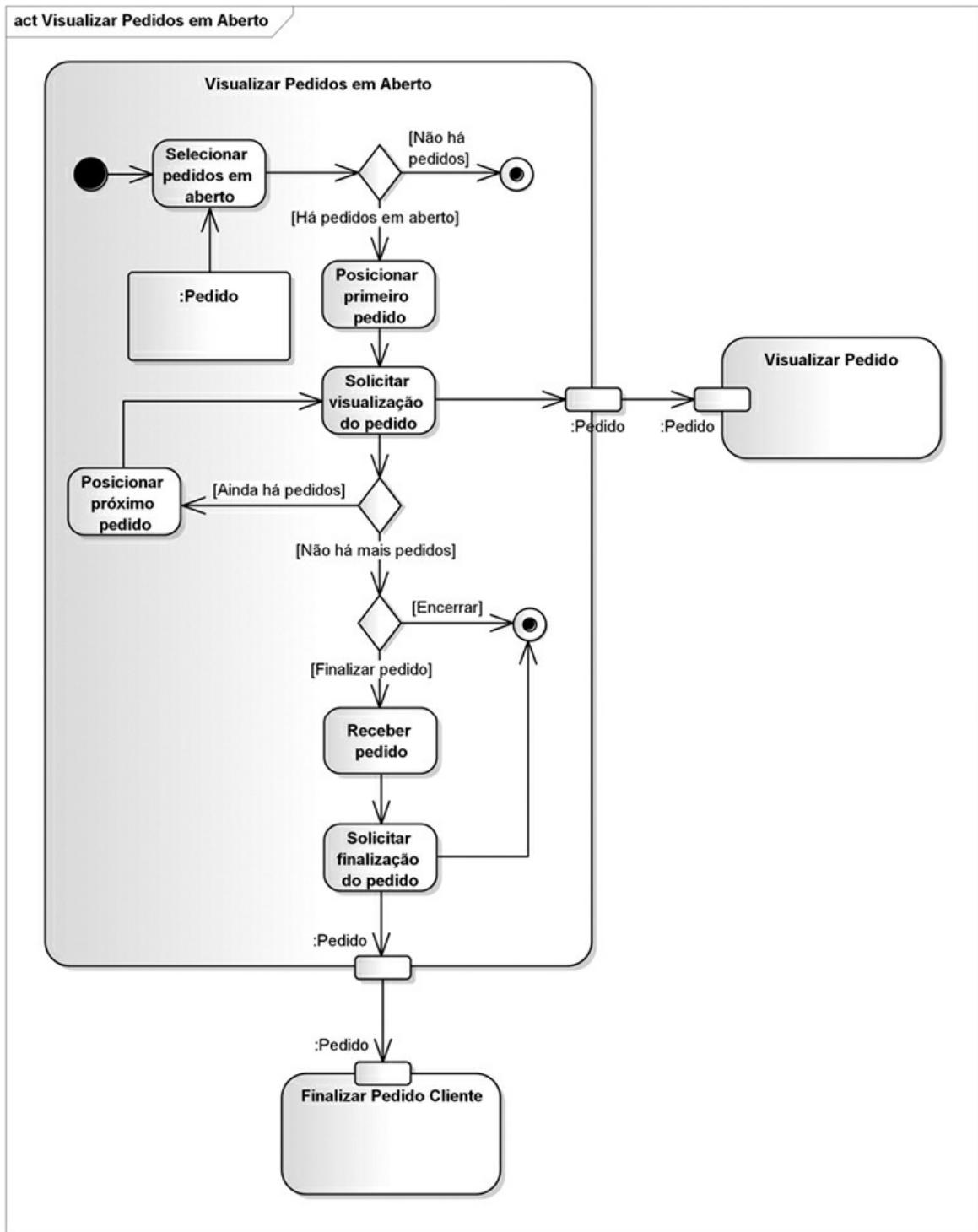


Figura 17.55 – Diagrama de Atividade Visualizar Pedidos em Aberto.

Se houver pedidos em aberto, a atividade será posicionada sobre o primeiro objeto encontrado, passando, na ação seguinte, a solicitar sua visualização. Observe que essa solicitação é repassada para a atividade **Visualizar Pedido**, que recebe o número do pedido consultado por meio de nós de

parâmetro de atividade.

Em seguida, chega-se a outro nó de decisão que verifica se ainda há pedidos a apresentar. Em caso positivo, a atividade será posicionada sobre o objeto seguinte e repetirá a solicitação de visualização. Quando não houver mais pedidos, a atividade encontrará um novo nó de decisão, que representa a escolha do funcionário em encerrar o processo ou finalizar um pedido.

Caso o funcionário queira finalizar um pedido, passa-se à ação em que o pedido escolhido pelo funcionário é recebido e, em seguida, à ação que solicita a finalização do pedido. Essa ação envia o número do pedido a finalizar, por meio de nós de parâmetro de atividade, para a atividade **Finalizar Pedido Cliente**, que se encarregará de finalizá-lo. A seguir, a atividade é encerrada.

#### *Diagrama de Atividade Finalizar Pedido Cliente*

Essa atividade inicia-se pela ação que seleciona todos os funcionários registrados na empresa. Essa consulta é feita sobre objetos da classe **Funcionario**, como demonstra o fluxo de objetos. Em seguida, há um teste, representado por um nó de decisão, que verifica se foram encontrados funcionários. Caso não tenham sido encontrados funcionários, a atividade será encerrada (Figura 17.56).

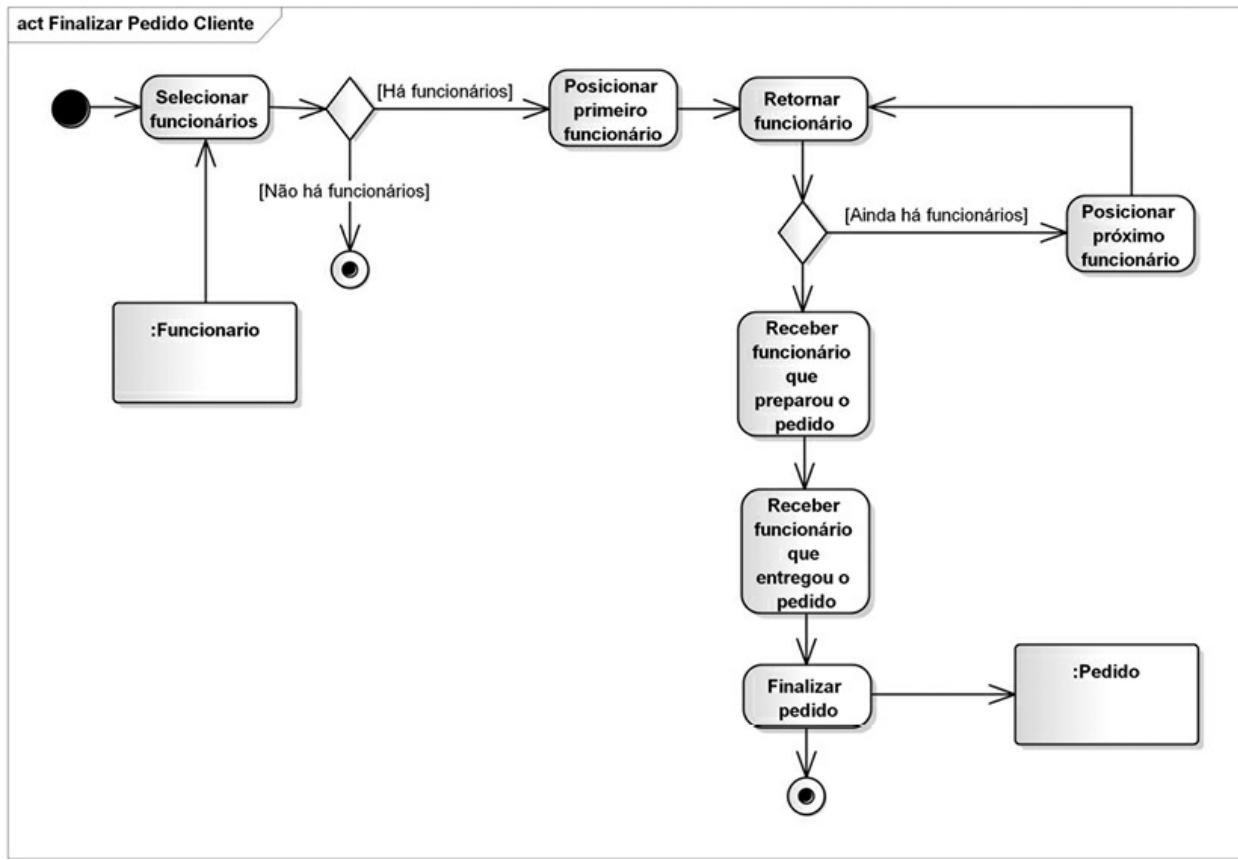


Figura 17.56 – Diagrama de Atividade Finalizar Pedido Cliente.

Se a pesquisa encontrou funcionários, a atividade posiciona-se sobre o primeiro objeto encontrado e, na ação seguinte, retorna seu nome para a interface do sistema. Em seguida, um nó de decisão verifica se ainda há funcionários a carregar. Em caso positivo, a atividade posiciona-se sobre o objeto seguinte da classe **Funcionario** e volta a retornar seu nome, repetindo essa operação enquanto houver funcionários.

A seguir, a atividade recebe o funcionário que preparou e o funcionário que entregou o pedido, em dois nós de ação diferentes, passando, em seguida, para o último nó de ação no qual é finalizado o pedido, ou seja, sua situação passa a armazenar o valor 2, que, por convenção, determina que o pedido foi atendido. Observe que o objeto da classe **Pedido** é atualizado por meio de um fluxo de objetos.

#### Diagrama de Atividade Gerenciar Cardápio

Esse diagrama se inicia com a ação que seleciona os sabores registrados no sistema, como demonstra o fluxo de objetos entre essa ação e o objeto da classe **Sabor**. Essa consulta seleciona todos os objetos dessa classe. Em

seguida, a atividade atinge um nó de decisão que verifica se foi encontrado algum objeto da classe **Sabor**.

Em caso positivo, a atividade posiciona-se sobre o primeiro objeto da classe **Sabor** e, na ação seguinte, retorna a descrição do sabor para a interface do sistema. A seguir, a atividade encontra outro nó de decisão, responsável por determinar se ainda há sabores a carregar, caso em que a atividade se posicionará no objeto seguinte e retornará a descrição do novo sabor, repetindo-se esse laço enquanto houver sabores a apresentar (Figura 17.57).

Quando não houver mais sabores a apresentar ou caso não tenha sido encontrado nenhum sabor, passa-se a ação **Selecionar ingredientes**, que seleciona todos os objetos existentes da classe **Ingrediente**, como mostra o fluxo de objetos. Se nenhum ingrediente for encontrado, a atividade será encerrada, como demonstra o nó de decisão seguinte. Caso contrário, a atividade será posicionada sobre o primeiro ingrediente e, na ação seguinte, o retornará à interface. O próximo nó de decisão informa que essa operação será repetida enquanto houver ingredientes, com a atividade posicionando-se no ingrediente seguinte, até estes se encarem.

A partir daí, a atividade encontra um novo nó de decisão, que representa a escolha do administrador em encerrar o processo, incluir um novo sabor ou consultar um sabor já existente.

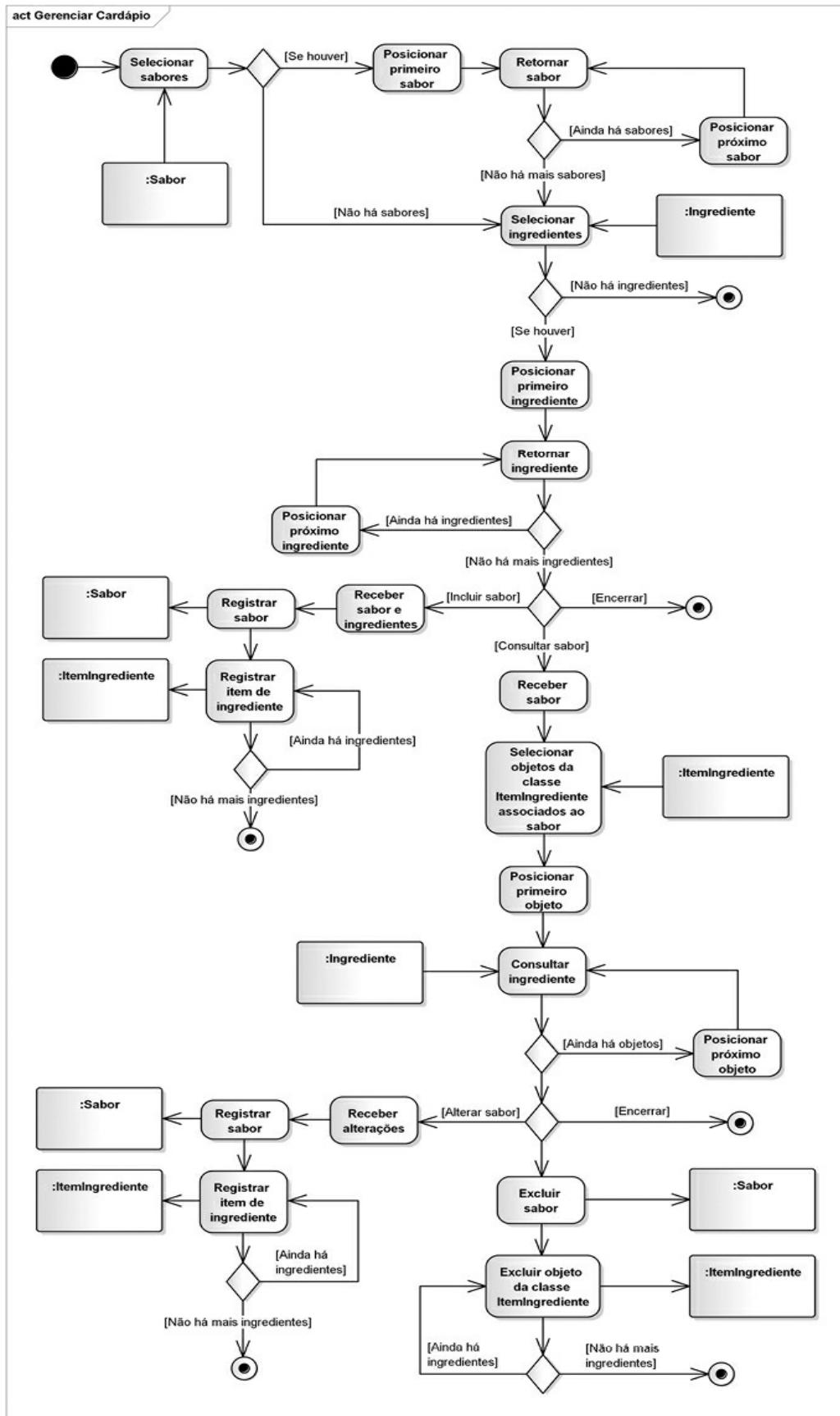
Se o administrador quiser inserir um novo sabor, a atividade passa a ação em que recebe o novo sabor e seus ingredientes, informados pelo administrador. Em seguida passa-se à ação **Registrar sabor**, onde, por meio de um fluxo de objetos, essa ação instancia um novo objeto da classe **Sabor**. A ação seguinte instancia um novo objeto da classe **ItemIngrediente**, enquanto houver ingredientes para registrar, como demonstra o nó de decisão. Quando não houver mais ingredientes a registrar, a atividade é encerrada.

Caso o administrador queira consultar um sabor já existente, a atividade passa a ação em que recebe o sabor escolhido pelo administrador e, em seguida, executa a ação **Selecionar objetos da classe ItemIngrediente associados ao sabor**, posicionando-se, em seguida, no primeiro objeto selecionado. A seguir, a ação seguinte retorna a descrição do objeto **ItemIngrediente**, por meio da consulta ao objeto da classe **Ingrediente** a

ele associado. Essa situação se repete enquanto houver objetos da classe **ItemIngrediente** a apresentar, como demonstra o próximo nó de decisão.

A seguir, a atividade encontra um novo nó de decisão, o qual representa a escolha de o administrador encerrar o processo, alterar um sabor ou excluir um sabor.

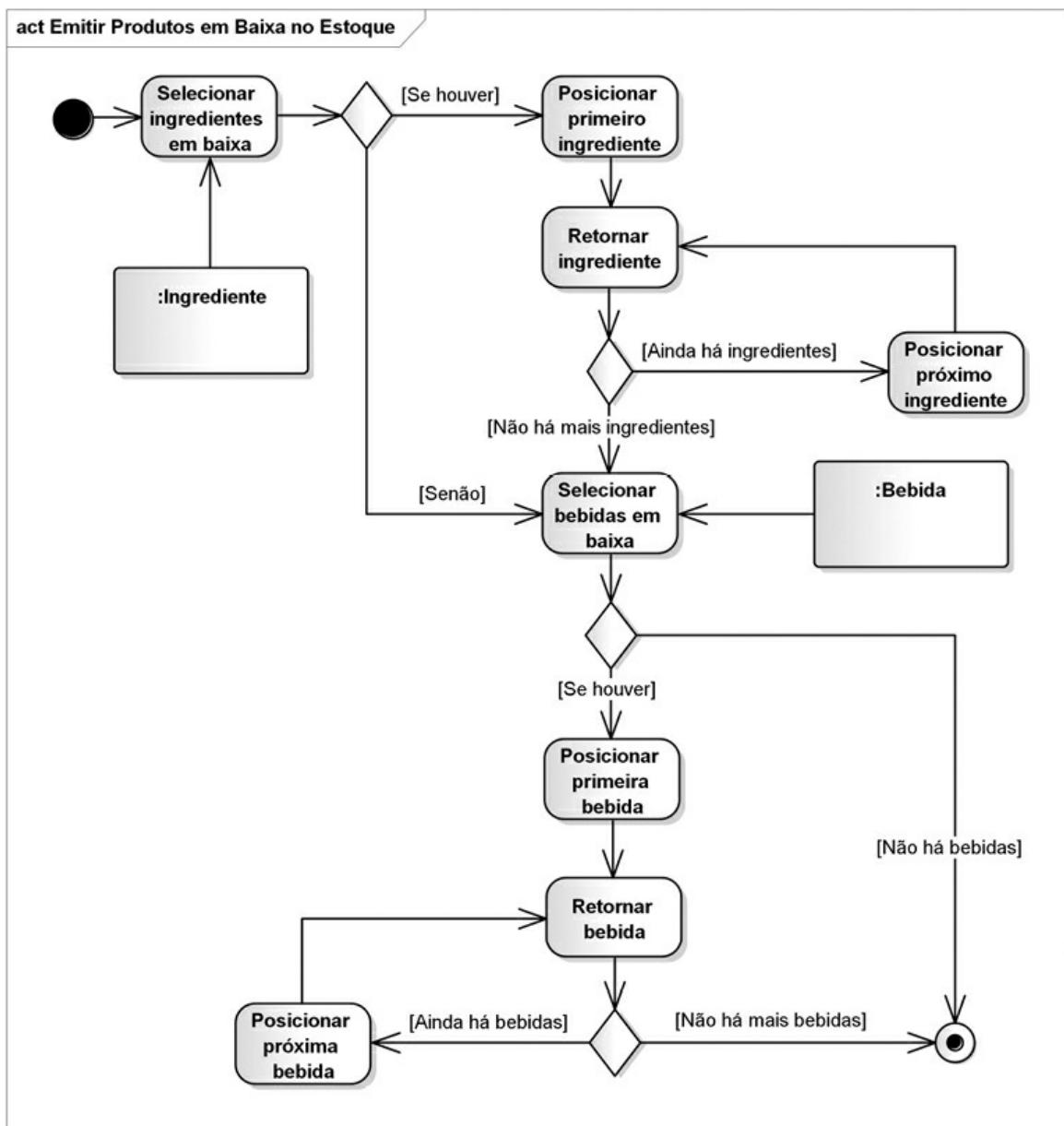
Caso o administrador queira alterar um sabor, procede-se de maneira semelhante à rotina de inserção de um novo sabor. Já se o administrador quiser excluir um sabor, será executada a ação **Excluir sabor**, que destrói o objeto da classe **Sabor** em questão e, em seguida, a atividade inicia um laço em que serão destruídos todos os objetos da classe **ItemIngrediente** associados ao sabor. Quando não houver mais objetos **ItemIngrediente** a excluir, a atividade será encerrada.



*Figura 17.57 – Diagrama de Atividade Gerenciar Cardápio.*

#### *Diagrama de Atividade Emitir Produtos em Baixa no Estoque*

A primeira ação dessa atividade seleciona todos os objetos da classe **Ingrediente** cujas quantidades em estoque estejam abaixo da quantidade mínima. O nó de decisão seguinte a essa ação testa se foi encontrado algum ingrediente em baixa (Figura 17.58).



*Figura 17.58 – Diagrama de Atividade Emitir Produtos em Baixa no Estoque.*

Em caso positivo, a atividade posiciona-se sobre o primeiro objeto encontrado e, na ação seguinte, apresenta esse ingrediente. Essa operação será repetida enquanto houver ingredientes em baixa.

Depois disso, caso não tenha sido encontrado nenhum ingrediente em baixa, são selecionadas todas as bebidas em baixa, isto é, todos os objetos da classe **Bebida** cuja quantidade em estoque esteja abaixo da quantidade mínima. Se não for encontrado nenhum objeto dessa classe com o estoque abaixo do mínimo, a atividade será encerrada nesse momento.

Caso contrário, a atividade posiciona-se no primeiro objeto da classe **Bebida** e, na ação seguinte, retorna-o, repetindo essa operação enquanto houver objetos da classe **Bebida** para apresentar. Quando estes se encerrarem, a atividade também será terminada.

#### *Diagrama de Atividade Emitir Compras em Aberto*

Essa atividade inicia-se com a seleção de todas as compras em aberto, em que é feita uma pesquisa sobre todas as compras cuja situação esteja definida como em aberto, ou seja, ainda não foram entregues. A seguir, é feito um teste para determinar se alguma compra em aberto foi encontrada; em caso negativo, a atividade será encerrada (Figura 17.59).

Caso tenham sido encontradas compras em aberto, a atividade posiciona-se sobre a primeira delas, passando em seguida a consultar o fornecedor associado ao objeto da classe **Compra** em questão. Depois disso, é executada a ação **Selecionar objetos da classe ItemCompraIngrediente**, que seleciona todos os objetos dessa classe associados à compra.

Em seguida, um nó de decisão verifica se foram encontrados objetos dessa classe. Em caso positivo, a atividade posiciona-se sobre o primeiro desses objetos, retorna a quantidade solicitada, consulta a descrição do ingrediente associado ao objeto e retorna essa descrição. Esse comportamento será repetido enquanto houver objetos da classe **ItemCompraIngrediente** a apresentar.

A seguir, caso não tenham sido solicitados ingredientes na compra, a atividade selecionará todos os objetos da classe **ItemCompraBebida** associados à compra. Caso existam objetos dessa classe associados à compra, a atividade será posicionada sobre o primeiro desses objetos, retornará a quantidade solicitada, consultará a descrição da bebida

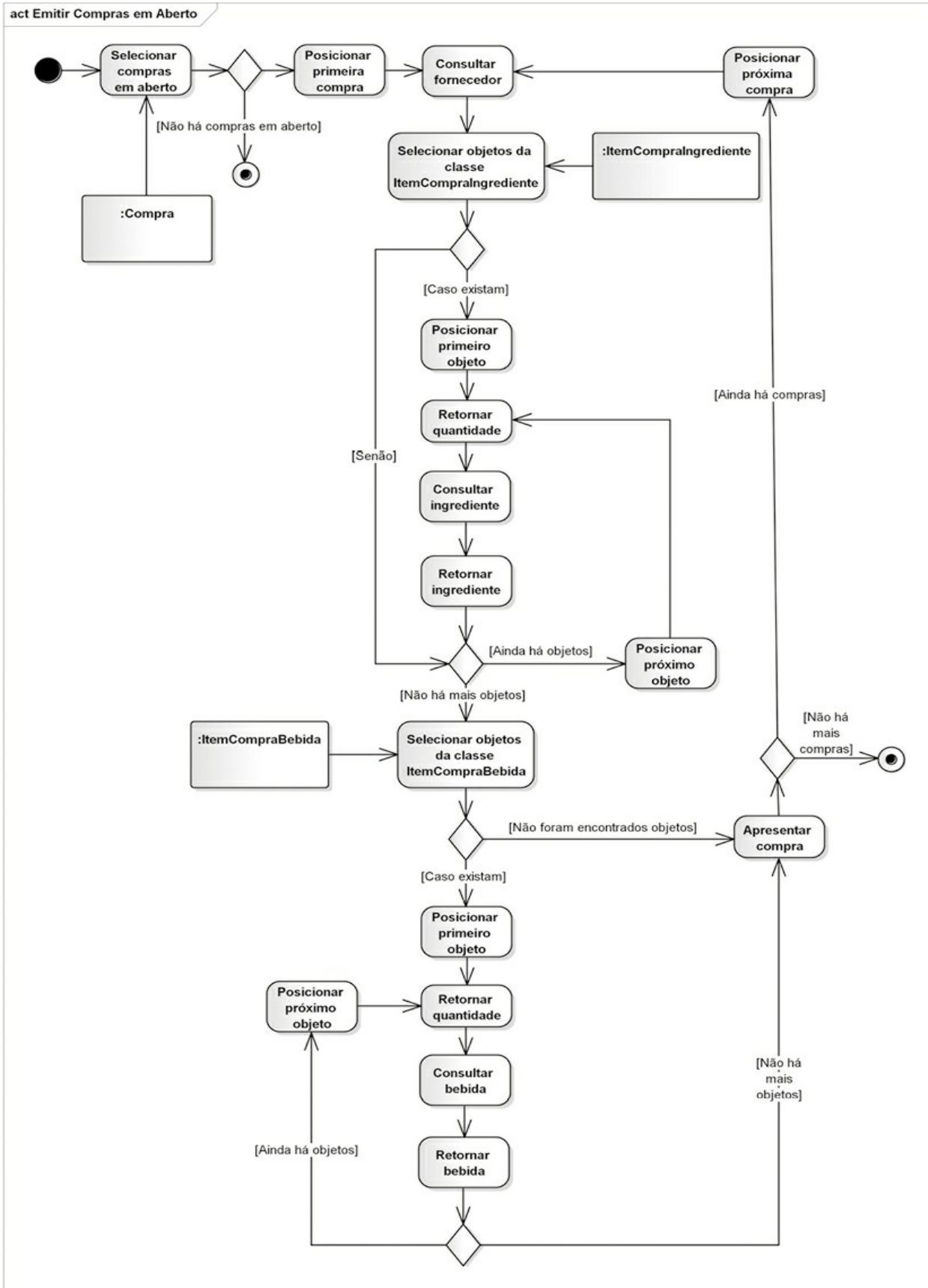
associada ao objeto e retornará essa descrição. Esse comportamento se repetirá enquanto houver bebidas associadas à compra.

Quando se esgotarem os objetos da classe **ItemCompraBebida** ou caso não tenham sido solicitadas bebidas nessa compra, a atividade apresentará as informações da compra em questão e, em seguida, encontrará um novo nó de decisão, cuja função será verificar se ainda existem compras em aberto a apresentar. Caso existam, a atividade será posicionada sobre o próximo objeto da classe **Compra** e repetirá toda a operação já descrita, caso contrário será encerrada.

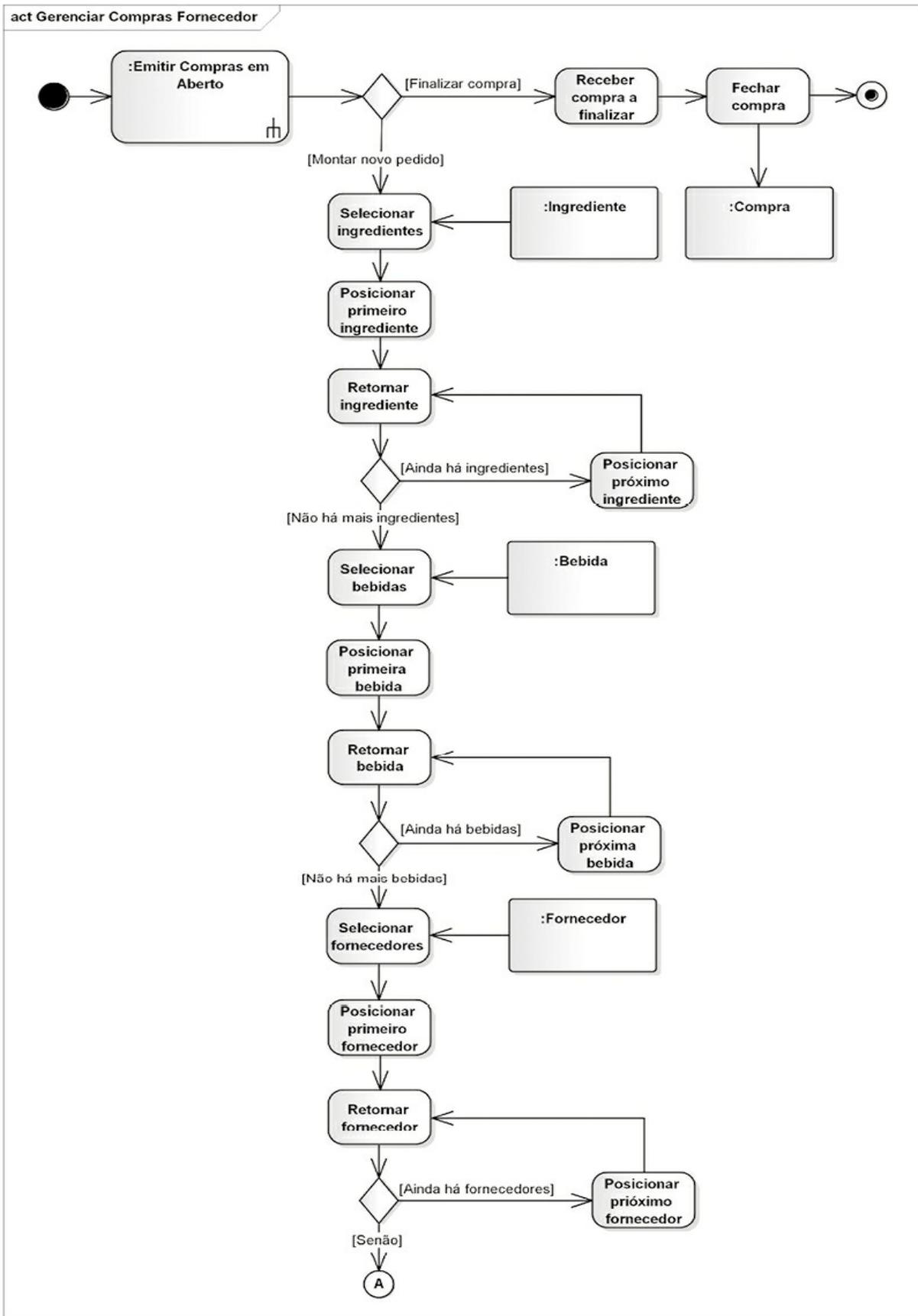
#### *Diagrama de Atividade Gerenciar Compras Fornecedores*

Esse diagrama inicia-se com a invocação do processo de **Emitir Compras em Aberto**, descrito na subseção anterior. A partir da listagem apresentada na atividade de emissão de compras em aberto, o administrador pode escolher finalizar uma compra ou montar um novo pedido, como demonstra o nó de decisão (Figura 17.60).

Se o administrador quiser finalizar uma compra, a atividade receberá a compra a finalizar e, na próxima ação, fechará a compra, definindo a situação do objeto da classe **Compra** como finalizada (definindo o valor do atributo **situacaoCompra** como 1). Terminado isso, a atividade é concluída.



*Figura 17.59 – Diagrama de Atividade Emitir Compras em Aberto.*



*Figura 17.60 – Diagrama de Atividade Gerenciar Compras Fornecedores.*

No entanto, caso o administrador queira montar uma nova compra, a atividade em resposta selecionará todos os objetos da classe **Ingrediente** registrados, posicionando-se a seguir no primeiro objeto selecionado, e retornará a descrição do ingrediente à interface do software na ação seguinte. Esse comportamento se repetirá enquanto houver ingredientes a carregar na interface.

Quando não houver mais ingredientes, a atividade selecionará todas as bebidas cadastradas no sistema, como demonstra o fluxo de objetos. A seguir, a atividade será posicionada sobre o primeiro objeto selecionado e, na ação seguinte, retornará a descrição da bebida para a interface. Esse laço será repetido enquanto houver bebidas a apresentar.

Depois disso, a atividade selecionará todos os fornecedores registrados no sistema, posicionando-se em seguida sobre o primeiro objeto da classe **Fornecedor** encontrado, retornando seu nome à interface na ação seguinte. Novamente, a atividade permanecerá executando esse comportamento enquanto houver fornecedores a carregar na interface.

Observe que quando não houver mais fornecedores, a atividade atingirá um conector, que representa um atalho para a continuação desse diagrama, apresentando na figura seguinte. Isso foi necessário porque esse diagrama é muito extenso e tivemos, portanto, que o dividir em duas figuras (Figura 17.61).

Na continuação dessa atividade, a ação seguinte recebe o fornecedor escolhido pelo administrador e a próxima ação recebe os produtos a comprar. Em seguida, a compra é registrada, gerando um novo objeto da classe **Compra**, como demonstra o fluxo de objetos.

Em seguida, a atividade encontra um nó de decisão, que deve verificar se o administrador escolheu ingredientes para comprar. Nesse caso, a atividade posiciona-se no primeiro ingrediente, registra-o, gerando um novo objeto da classe **ItemCompraIngrediente**, e permanece nessa operação enquanto houver ingredientes a registrar.

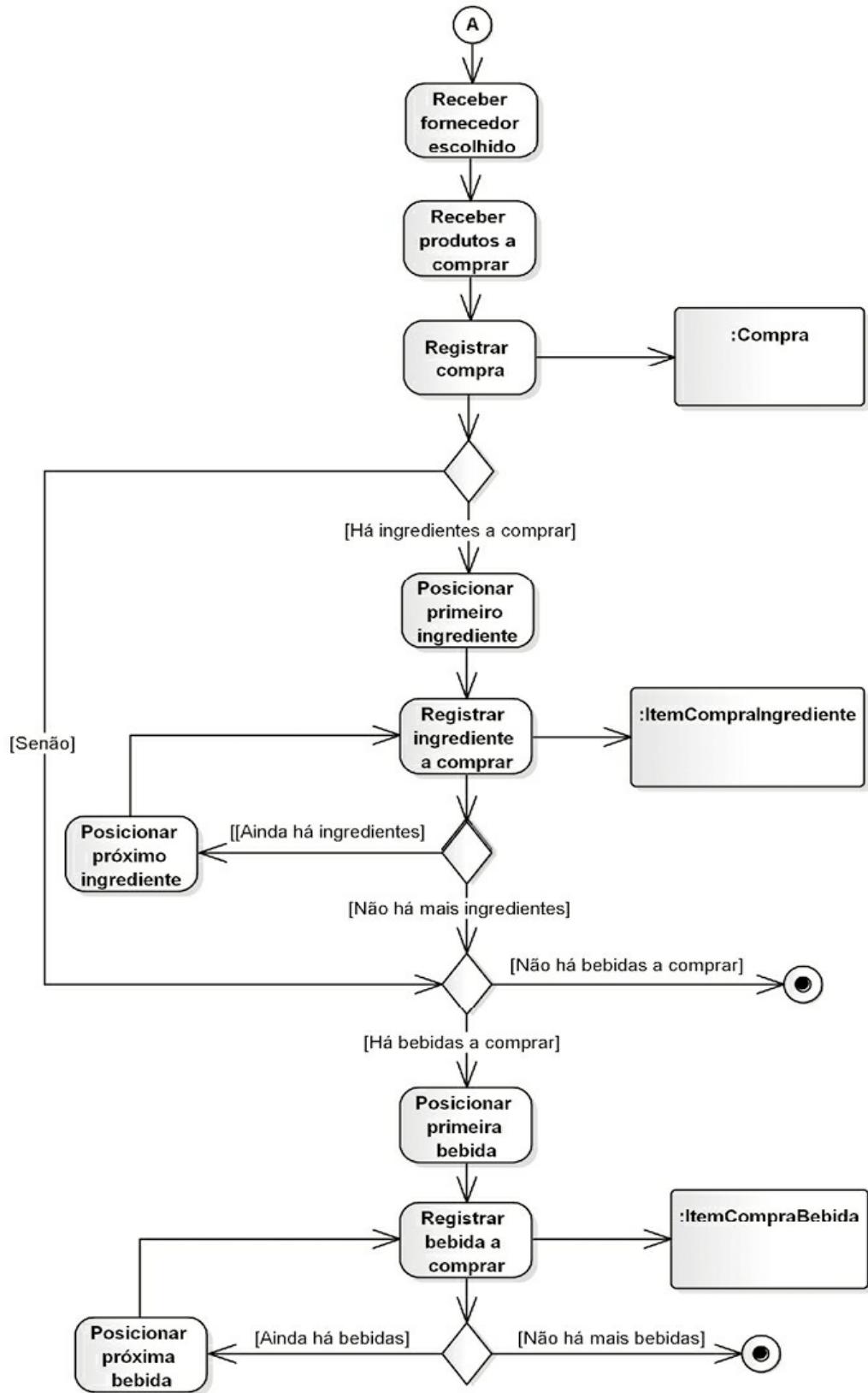
A seguir, caso não tenham sido solicitados ingredientes na compra, a atividade verifica, por meio de um nó de decisão, se foram solicitadas bebidas. Em caso negativo, a atividade encerra-se nesse momento, e, caso

contrário, a atividade posiciona-se sobre a primeira bebida solicitada, registra-a, gerando um novo objeto da classe **ItemCompraBebida**, e permanece nesse laço enquanto houver bebidas a registrar. Quando estes acabarem, a atividade será encerrada.

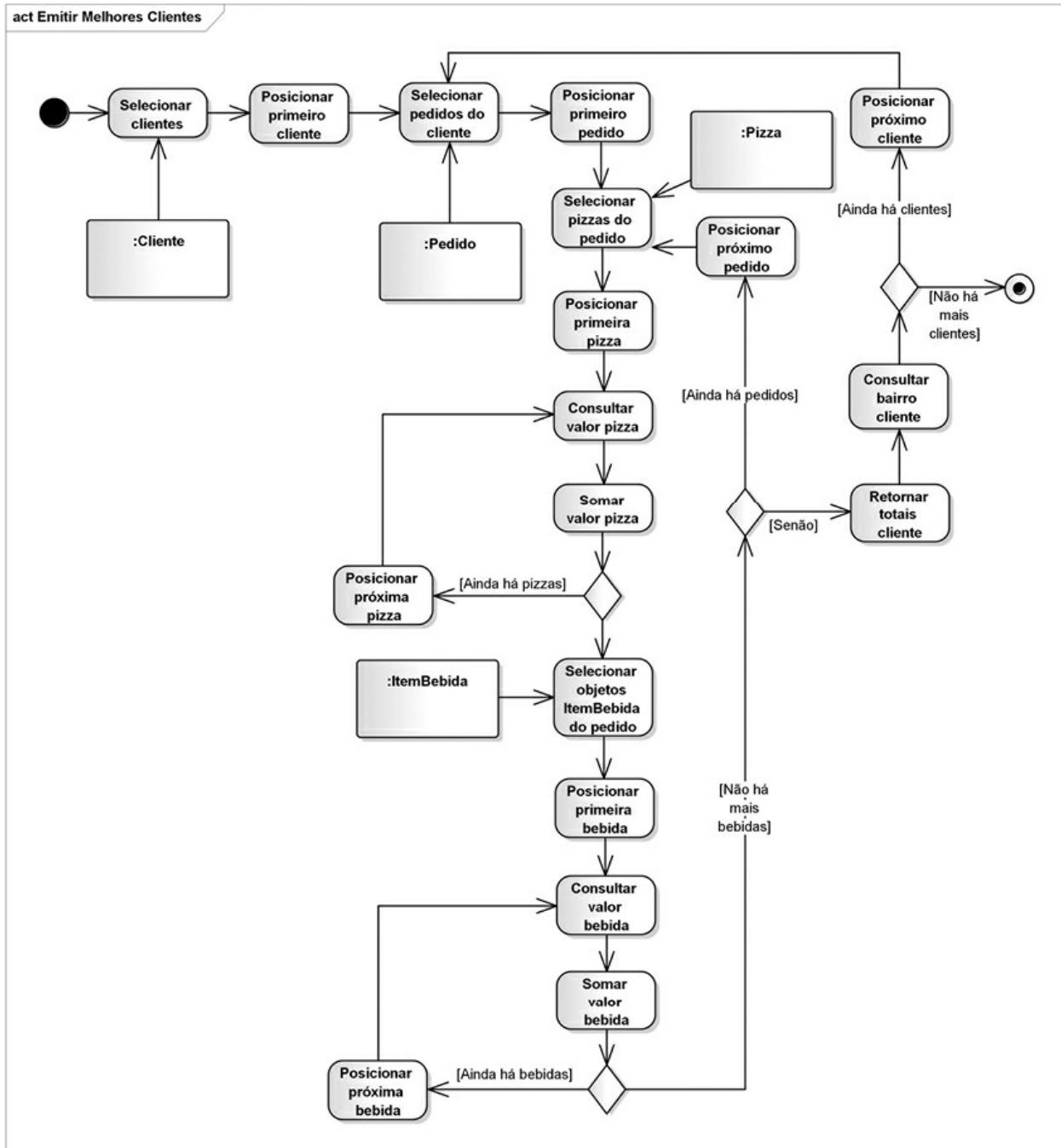
#### *Diagrama de Atividade Emitir Melhores Clientes*

A solicitação dessa listagem obriga a atividade a selecionar todos os clientes do sistema, posicionando-se sobre o primeiro e selecionando todos os pedidos já solicitados por este. Em seguida, a atividade será posicionada no primeiro pedido do cliente e selecionará todas as pizzas solicitadas nele, posicionando-se na primeira pizza, consultando seu valor e somando este a um totalizador dos gastos do cliente. A seguir, se ainda houver pizzas, a atividade será posicionada na pizza seguinte e repetirá a operação anterior (Figura 17.62).

act Gerenciar Compras Fornecedor - Continuação



*Figura 17.61 – Diagrama de Atividade Gerenciar Compras Fornecedor – Continuação.*



*Figura 17.62 – Diagrama de Atividade Emitir Melhores Clientes.*

Quando não houver mais pizzas, a atividade selecionará todos os objetos **ItemBebida** associados ao pedido, posicionando-se no primeiro, consultando seu valor e somando-o ao totalizador do cliente. Esse

comportamento será repetido enquanto houver bebidas a totalizar.

No momento em que não houver mais bebidas, a atividade verificará se ainda existem pedidos do cliente atual. Se ainda existirem pedidos, a atividade será posicionada sobre o próximo pedido e repetirá a rotina já descrita.

Se não existirem mais pedidos, a atividade retornará os totais do cliente, consultará seu bairro e tentará se posicionar no próximo cliente, se houver. Se ainda existirem clientes, toda a operação descrita será repetida. Caso contrário, o sistema apresentará a listagem de clientes em ordem decrescente de seus totalizadores e o processo será encerrado.

#### *Diagrama de Atividade Emitir Consumo por Período*

A primeira ação dessa atividade constitui-se em receber os períodos desejados para a listagem. A seguir, a atividade seleciona todos os objetos da classe **Pedido** que se encontrem dentro do período informado. Se não for encontrado nenhum pedido nesse período, a atividade será encerrada.

Caso contrário, a atividade será posicionada sobre o primeiro objeto **Pedido** encontrado, selecionará todos os objetos da classe **Pizza** a ele associados e será posicionada sobre o primeiro objeto **Pizza**.

Depois disso, serão selecionados todos os objetos da classe **ItemSabor** associados ao objeto **Pizza**, em seguida a atividade será posicionada sobre o primeiro objeto e consultará o objeto da classe **Sabor** a ele associado.

Na sequência, a atividade precisará selecionar todos os objetos da classe **ItemIngrediente** associados ao objeto **Sabor**, posicionarse no primeiro objeto encontrado e somar a quantidade necessária do ingrediente para incrementar o totalizador do sabor. A seguir, é consultada a descrição do objeto da classe **Ingrediente** associado ao objeto **ItemIngrediente**, atingindo-se, em seguida, um nó de decisão que verifica se ainda há objetos da classe **ItemIngrediente**. Se houver, a atividade posiciona-se no próximo objeto e repete a operação.

Caso não haja mais objetos da classe **ItemIngrediente**, a atividade testará se ainda há objetos da classe **ItemSabor** associados ao objeto **Pizza**. Em caso positivo, a atividade será posicionada sobre o próximo objeto **ItemSabor** e repetirá a rotina de consultar o objeto **Sabor** associado ao objeto **ItemSabor** e todo o restante da rotina (Figura 17.63).

Se não houver mais objetos **ItemSabor**, a atividade consultará o número de pedaços da pizza, por meio de uma consulta a um objeto da classe **Tamanho** associado ao objeto da classe **Pizza**. Isso é necessário para finalizar a totalização dos ingredientes da pizza, realizada na ação seguinte.

Depois, a atividade verifica se ainda há objetos **Pizza** associados ao objeto **Pedido**. Em caso positivo, a atividade posiciona-se sobre o próximo objeto **Pizza** e repete-se toda a operação descrita a partir daí.

Se não houver mais objetos **Pizza**, a atividade deverá determinar se ainda há objetos da classe **Pedido**. Se ainda houver objetos **Pedido**, a atividade será posicionada sobre o próximo objeto e selecionará os objetos da classe **Pizza** associados ao objeto **Pedido**, repetindo todo o algoritmo descrito a partir desse momento. Caso não haja mais pedidos, a atividade será encerrada.

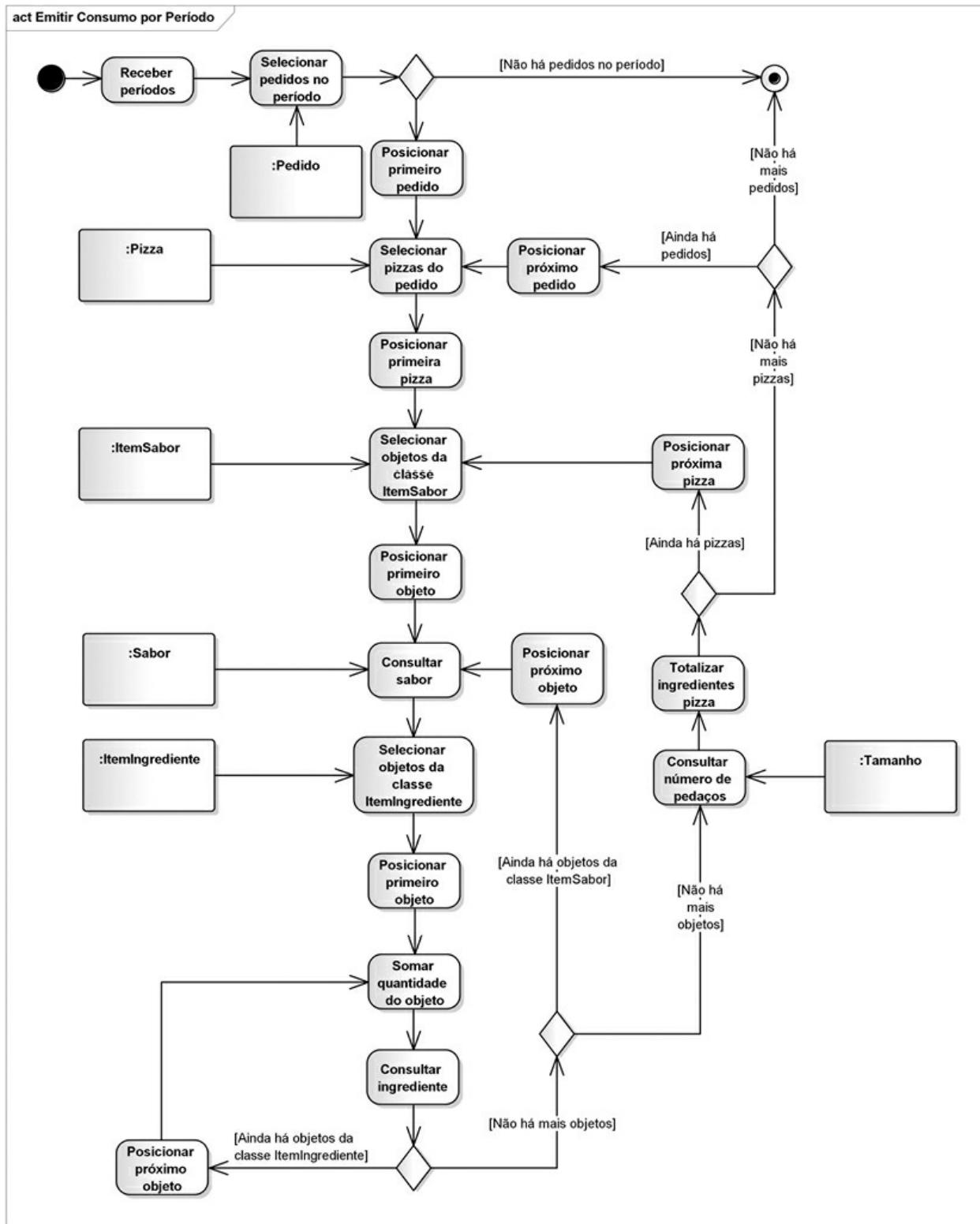
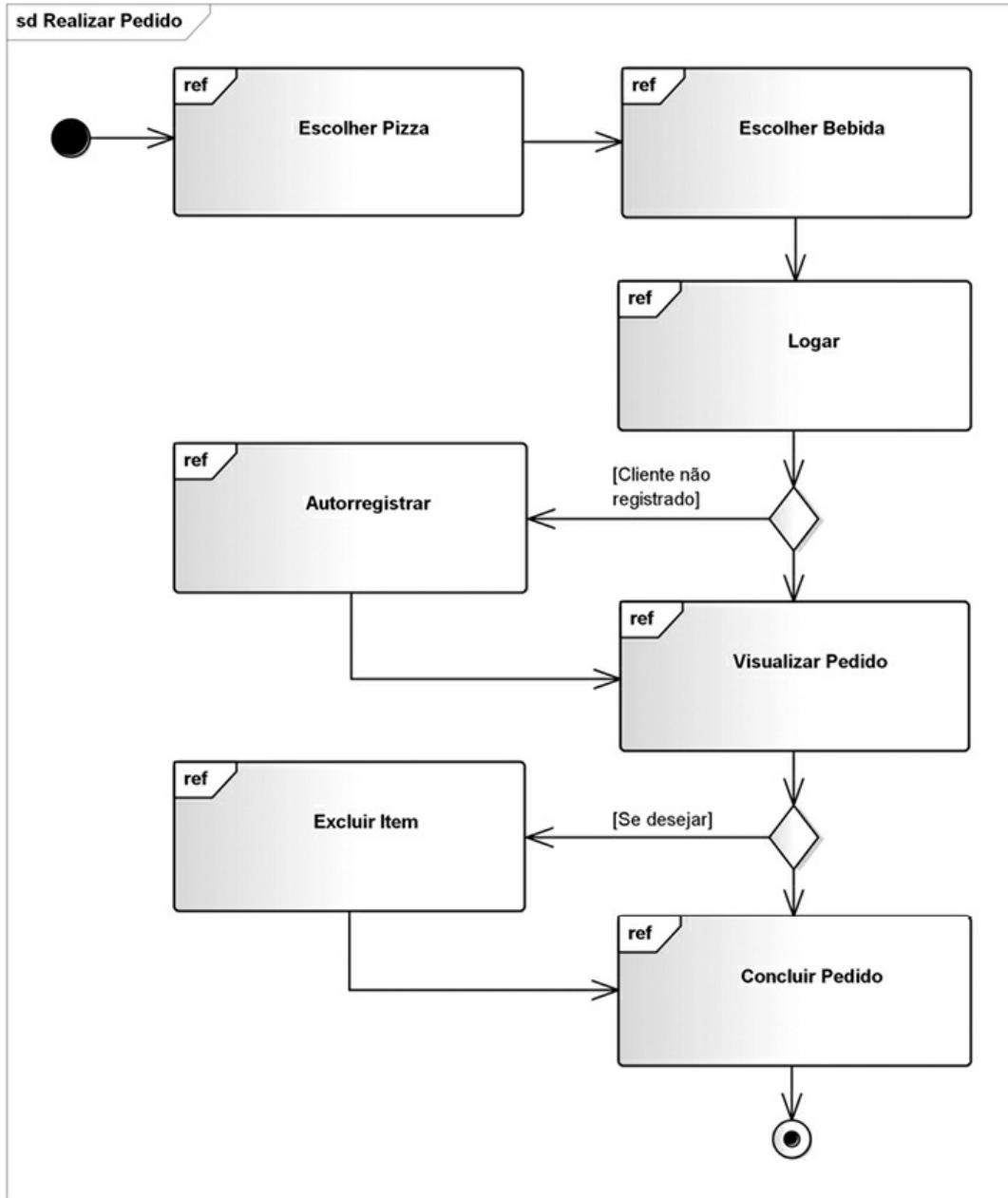


Figura 17.63 – Diagrama de Atividade Emitir Consumo por Período.

### **17.2.10 Diagrama de Visão Geral de Interação – Realizar Pedido**

Nesta seção, modelaremos o processo geral para que um cliente realize um pedido por meio de um diagrama de visão geral de interação, apresentado na figura 17.64. O leitor notará que utilizamos somente quadros de ocorrência de interação, pois não havia espaço para inserir quadros de interação completos.



*Figura 17.64 – Diagrama de Visão Geral de Interação – Realizar Pedido.*

Esse diagrama é fácil de interpretar, representando todos os processos necessários para que um cliente realize um pedido. Primeiramente, é

executado o processo de **Escolher Pizza**, passando para o processo de **Escolher Bebida**. Depois disso, o cliente necessita se autenticar no sistema, por meio do processo **Logar** e, caso não esteja registrado no sistema, deverá se autorregular, como demonstra o nó de decisão seguinte ao processo **Logar**.

Após estar autenticado, é necessário visualizar o pedido (o cliente pode fazer isso a qualquer momento, no entanto, somente para concluir o pedido isso é obrigatório). Durante a visualização do pedido, o cliente pode, se assim o desejar, excluir algum item. Isso é representado pelo nó de decisão seguinte ao processo de **Visualizar Pedido**, definindo que, se o cliente quiser, será executado o processo de **Excluir Item**.

Finalmente, é executado o processo de **Concluir Pedido**, em que o pedido será definido como concluído e será dada baixa nos itens necessários para atendê-lo.

### 17.2.11 Diagrama de Componentes da PizzaNet

Nesta seção, modelaremos o diagrama de componentes da PizzaNet. Nesse diagrama, os componentes modelados referem-se aos módulos executáveis do sistema, bem como os módulos necessários a seu funcionamento (Figura 17.65).

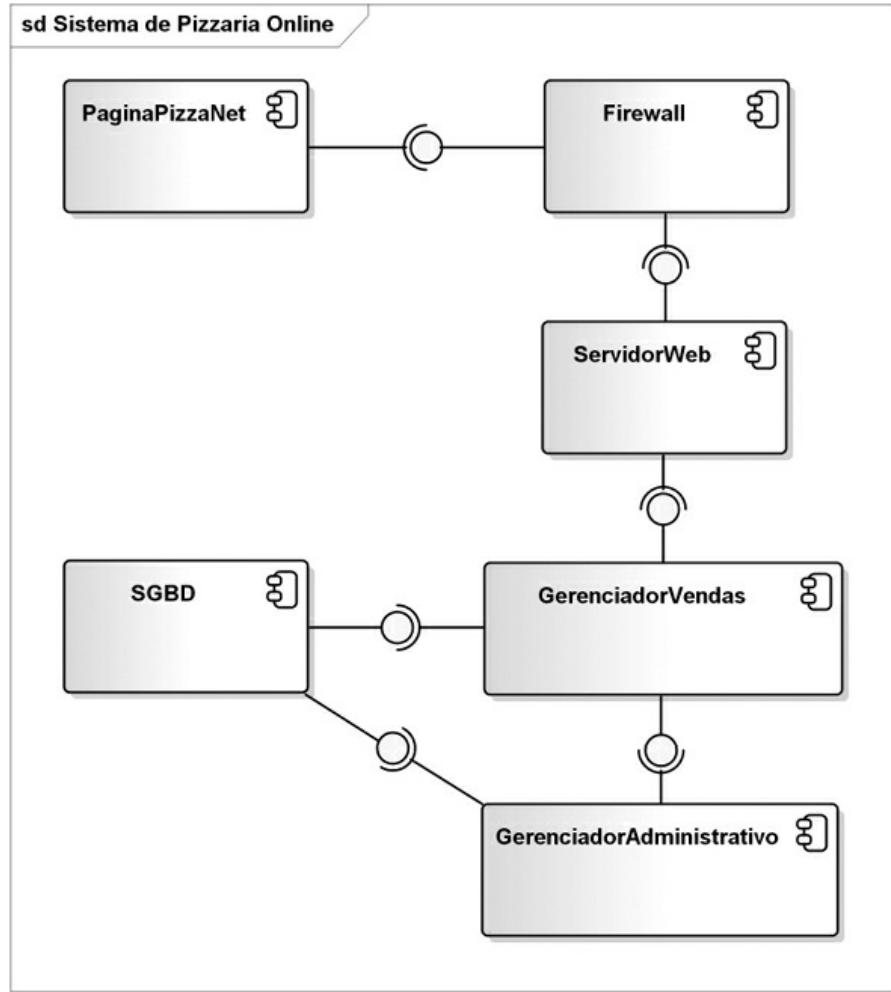


Figura 17.65 – Diagrama de Componentes da PizzaNet.

Como o leitor pode observar, esse diagrama é formado pelos seguintes componentes:

- **PaginaPizzaNet** – Representa a página da pizzaria, que é carregada e atualizada na máquina do cliente e por meio da qual este solicita pedidos à PizzaNet.
- **Firewall** – Representa um software de firewall que recebe o tráfego de informações externas ao sistema, incluindo os eventos que ocorrem nas páginas dos clientes da pizzaria. Sua função é impedir a entrada de usuários e softwares não autorizados na rede interna da empresa ou a saída de informações não autorizadas do sistema. Esse componente não faz realmente parte do sistema, mas é indispensável para seu bom funcionamento.
- **ServidorWeb** – Representa um software responsável, entre outras

coisas, por gerenciar os múltiplos pedidos de conexão solicitados pelos clientes da PizzaNet. Da mesma forma que o componente anterior, não faz realmente parte do sistema, mas é necessário a ele.

- **GerenciadorVendas** – Representa o módulo que atende às solicitações dos clientes, como escolha de pizzas, de bebidas, visualização do pedido, conclusão do pedido etc. Em suma, este é o componente que implementa todas as funcionalidades oferecidas aos clientes da pizzaria, representando o subsistema de vendas definido no diagrama de pacotes deste capítulo. Poderia ser dividido em diversos módulos, mas acreditamos que um único componente pode desempenhar essa função.
- **GerenciadorAdministrativo** – Engloba todos os serviços oferecidos aos funcionários da empresa, como finalizar um pedido, solicitar compras de produtos ou emitir os melhores clientes. Representa o subsistema administrativo. Consideramos que apenas um componente seria capaz de gerenciar todos essas funcionalidades, embora nada impedissem que estas fossem divididas em mais de um componente.
- **SGBD** – Finalmente, esse componente representa um Sistema Gerenciador de Banco de Dados responsável por armazenar e recuperar as informações necessárias ao sistema.

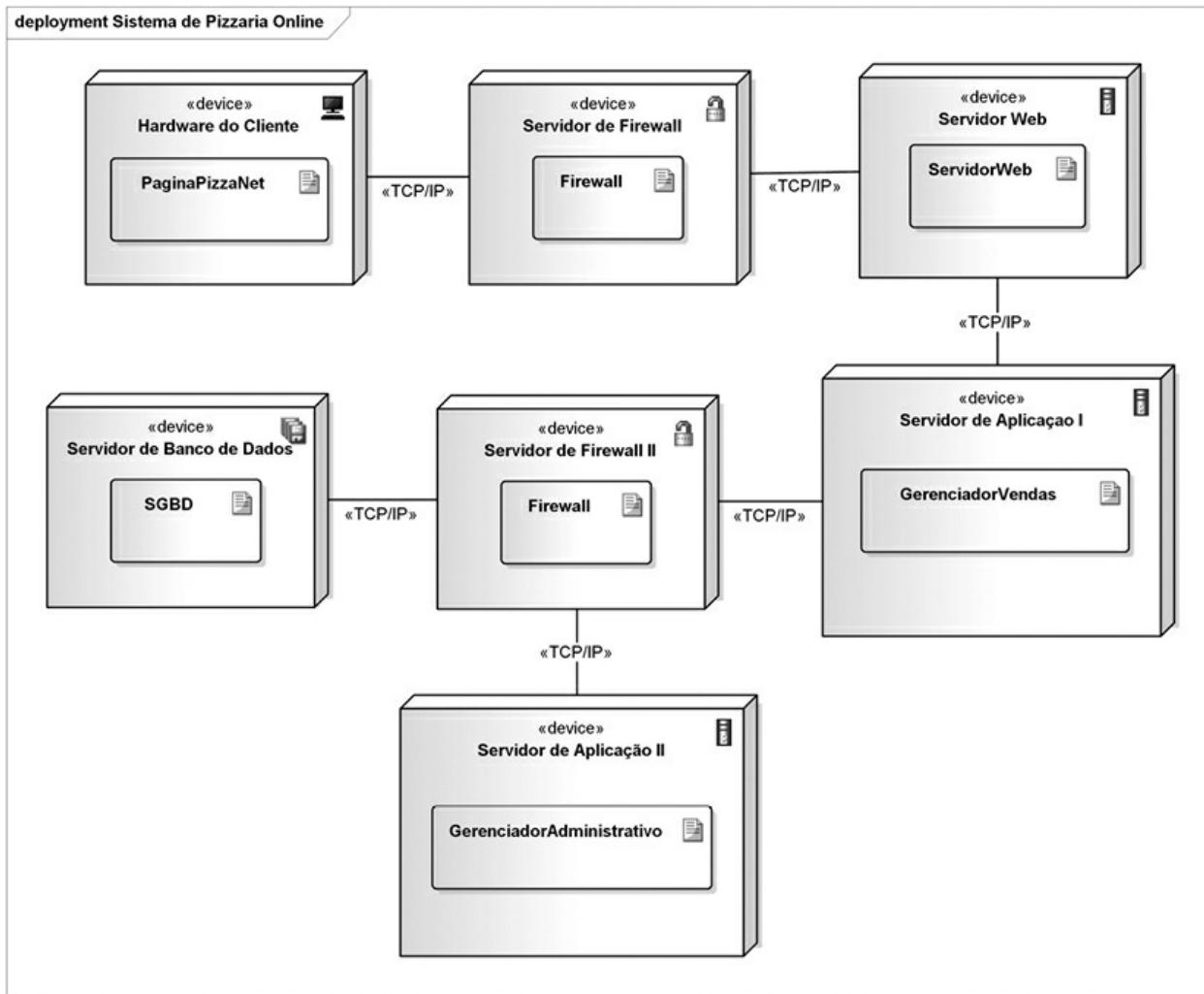
### 17.2.12 Diagrama de Implantação da PizzaNet

Nesta seção, concluiríremos a modelagem do sistema de pizzaria online, por meio do diagrama de implantação da PizzaNet, em que demonstraremos as necessidades físicas para que o sistema possa ser implantado. Nesse diagrama, os componentes modelados na seção anterior são manifestados por artefatos, contidos pelos nós que os suportarão (Figura 17.66).

A seguir, descreveremos cada um dos nós que compõem esse diagrama:

- **Hardware do Cliente** – Representa as máquinas do cliente, sobre o qual será carregada a página da PizzaNet, como demonstra o artefato **PaginaPizzaNet**, que é uma manifestação do componente de mesmo nome.
- **Servidor de Firewall** – Representa a máquina em que rodará um software de firewall, que procurará impedir invasões de pessoas não autorizadas à rede interna da pizzaria.

- **Servidor Web** – Representa o hardware necessário para suportar um software que deverá gerenciar os múltiplos pedidos de conexão solicitados pelos clientes da PizzaNet.
- **Servidor de Aplicação I** – Representa o servidor que suportará o subsistema de vendas da PizzaNet, representado pelo artefato **GerenciadorVendas**.
- **Servidor de Firewall II** – Este é um servidor que deve suportar um segundo software de firewall, fornecendo segurança extra aos dois servidores mais internos da rede da PizzaNet.
- **Servidor de Banco de Dados** – Representa um servidor que deverá executar um sistema gerenciador de banco de dados, que deverá armazenar e recuperar as informações necessárias ao sistema.
- **Servidor de Aplicação II** – Representa o último servidor da rede, onde é executado o subsistema administrativo da PizzaNet, representado pelo artefato **GerenciadorAdministrativo**.



*Figura 17.66 – Diagrama de Implantação da PizzaNet.*

Obviamente, em um sistema simples como este, dificilmente seriam necessários tantos servidores. Muitos dos módulos aqui representados poderiam estar contidos em uma única máquina. Na verdade, a rigor, o sistema todo poderia ser executado em um único servidor. Criamos esse diagrama mais complexo para ilustrar como um sistema poderia ser distribuído entre vários servidores.

# CAPÍTULO 18

## A UML 2.5

Em suas versões anteriores, a especificação da linguagem UML era dividida em dois documentos que descreviam sua infraestrutura e sua superestrutura respectivamente. A partir da versão 2.5, com o objetivo de simplificar a linguagem, a documentação da UML passou a englobar um único documento formal. Essa decisão foi tomada para tornar a especificação mais fácil de ler e eliminar redundâncias.

Ao longo deste último capítulo, falaremos sobre algumas questões referentes à estrutura da linguagem. O conteúdo aqui apresentado foi retirado, em sua maioria, da documentação formal da UML 2.5 disponível em [www.uml.org](http://www.uml.org).

### 18.1 Áreas Semânticas

A semântica da UML trata do significado-padrão das declarações feitas em um modelo UML sobre o sistema sendo modelado. As construções de modelagem da UML são divididas em duas categorias semânticas:

- Semântica estrutural, que define o significado de elementos de modelo estruturais sobre indivíduos no domínio sendo modelado, que podem ser verdadeiros em um ponto específico no tempo.
- Semântica comportamental, que define o significado de elementos de modelo comportamentais que fazem declarações sobre como indivíduos no domínio sendo modelo mudam ao longo do tempo.

A semântica estrutural fornece a base para a semântica comportamental. Construções de modelagem estrutural são estruturadas sobre um base comum de conceitos fundamentais, como tipo, espaço de nome (namespace), relacionamento e dependência. Construções de modelagem específica incluem um certo número de diferentes tipos de classificadores: tipos de dados, classes, sinais, interfaces e componentes, construções

necessárias para modelar valores e instâncias e construções para pacotes e perfis.

A base da semântica comportamental estrutura-se sobre a fundação estrutural para fornecer um arcabouço (framework) básico para a execução de comportamentos. Essa semântica comportamental comum também enfoca a comunicação que pode ocorrer entre objetos estruturais com comportamento associado. Ações são as unidades de comportamento fundamental usadas para definir comportamentos de granularidade fina.

Há também construções de modelagem suplementar que possuem aspectos tanto estruturais como comportamentais, como casos de uso, implantações e fluxos de informação.

## 18.2 Conceitos Básicos: Modelos, Metamodelos e Metaclasses

Aqui, forneceremos algumas definições relativas a termos que serão amplamente usados nas seções seguintes.

Um modelo captura uma visão de um sistema físico, ou seja, é uma abstração do sistema com certo propósito, como descrever aspectos estruturais ou comportamentais do software. Esse propósito determina o que deve ser incluído no modelo e o que é irrelevante. Assim, o modelo descreve completamente aqueles aspectos do sistema físico relevantes ao propósito do modelo, no nível apropriado de detalhes.

Um metamodelo define uma linguagem para expressar modelos. O papel de um metamodelo é definir a semântica para modelar elementos dentro de um modelo sendo instanciado. Dessa forma, um modelo é uma instância de um metamodelo.

Uma metaclass é uma classe, em um metamodelo, cujas instâncias são elementos concretos da UML, como classes, casos de uso ou atores. Metaclasses são usadas para construir metamodelos.

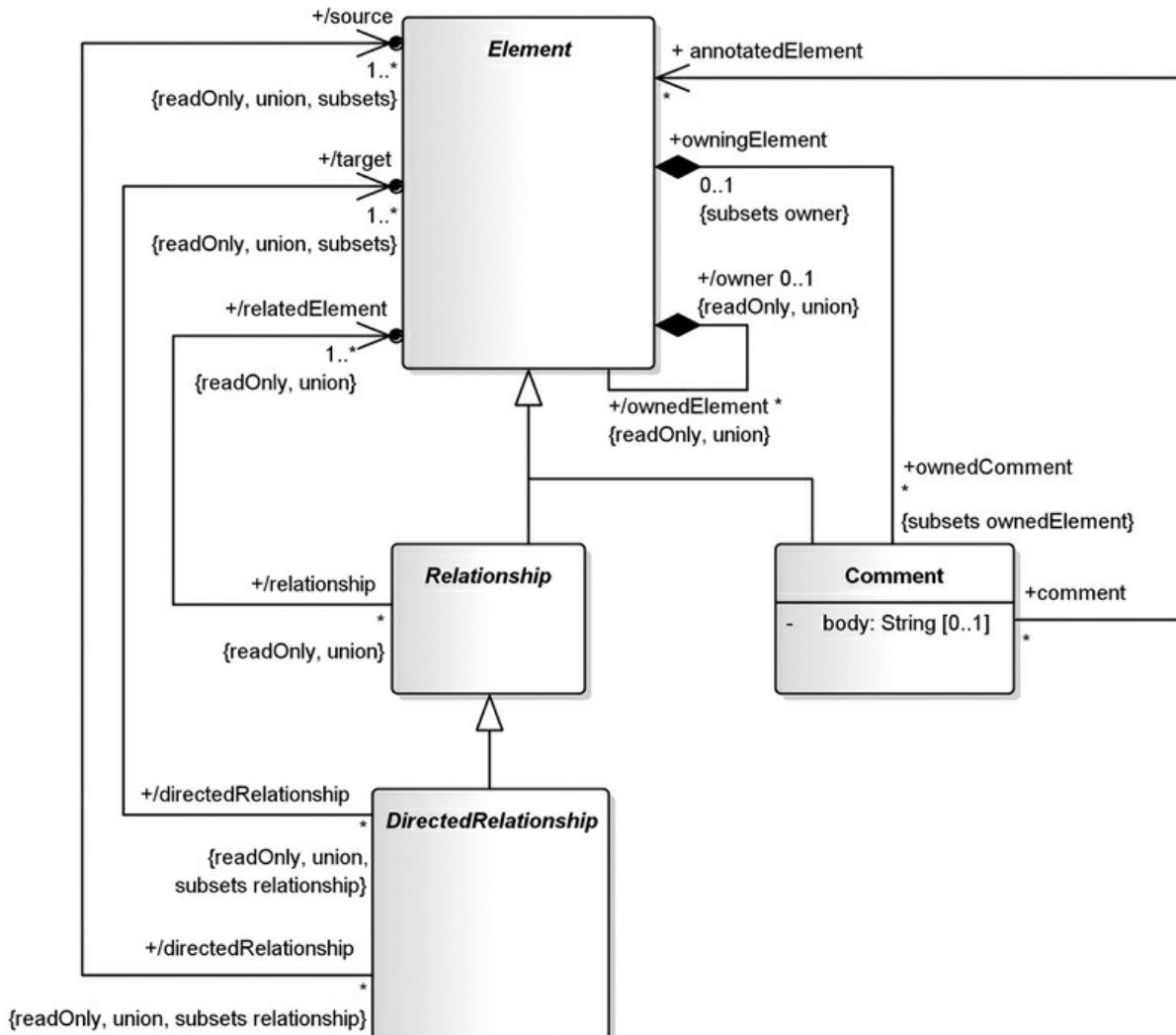
## 18.3 Estrutura Comum

A estrutura comum define os conceitos de modelagem básica em que se baseia toda a modelagem estrutural da UML. Muitas de suas metaclasses são abstratas e a partir destas são especializadas classes concretas.

### 18.3.1 Raiz (Root)

Os conceitos raiz de Element (Elemento) e Relationship (Relacionamento) fornecem a base para todos os outros conceitos de modelagem da UML. A figura 18.1 apresenta as metaclasses da raiz da estrutura comum da UML.

Nas subseções seguintes, descreveremos as metaclasses que compõem esse metamodelo.



*Figura 18.1 – Metaclasses da raiz da estrutura comum da UML.*

### 18.3.1.1 Metaclasses Element

Um **Element** (elemento) é um constituinte de um modelo. Descendentes de **Element** (metaclasses derivadas de **Element**) fornecem a semântica apropriada ao conceito que representam. Todo elemento tem a capacidade de possuir outros elementos. Quando um **Element** é removido de um

modelo, todos os seus elementos possuídos (`ownedElements`) são também removidos do modelo. A sintaxe abstrata para cada tipo de `Element` especifica quais outros tipos de `Elements` ele pode possuir.

### 18.3.1.2 Metaclasses Comment

Todo `Element` pode possuir comentários (`Comments`). Os comentários possuídos (`ownedComments`) por um `Element` não adicionam semântica, mas podem representar informação útil ao leitor do modelo.

### 18.3.1.3 Metaclasses Relationship

Um `Relationship` (relacionamento) é um `Element` que especifica algum tipo de relacionamento entre outros `Elements`. Descendentes de `Relationship` fornecem a semântica apropriada ao conceito que representam. Um `DirectedRelationship` representa um `Relationship` entre uma coleção de elementos de um modelo fonte e uma coleção de elementos de um modelo de destino. Um `DirectedRelationship` é direcionado a partir dos elementos fonte para os elementos de destino.

## 18.4 Metaclasses Utilizadas para a Modelagem de Classes

Para ilustrar melhor os conceitos de metaclasses e metamodelos, apresentamos, na figura 18.2, o metamodelo contendo as metaclasses que representam os conceitos utilizados para a modelagem de classes, atualizado de acordo com a última versão da UML. Um outro metamodelo utilizado pela estrutura da UML é apresentado no capítulo 16, demonstrando as metaclasses que representam os conceitos de modelagem de casos de uso.

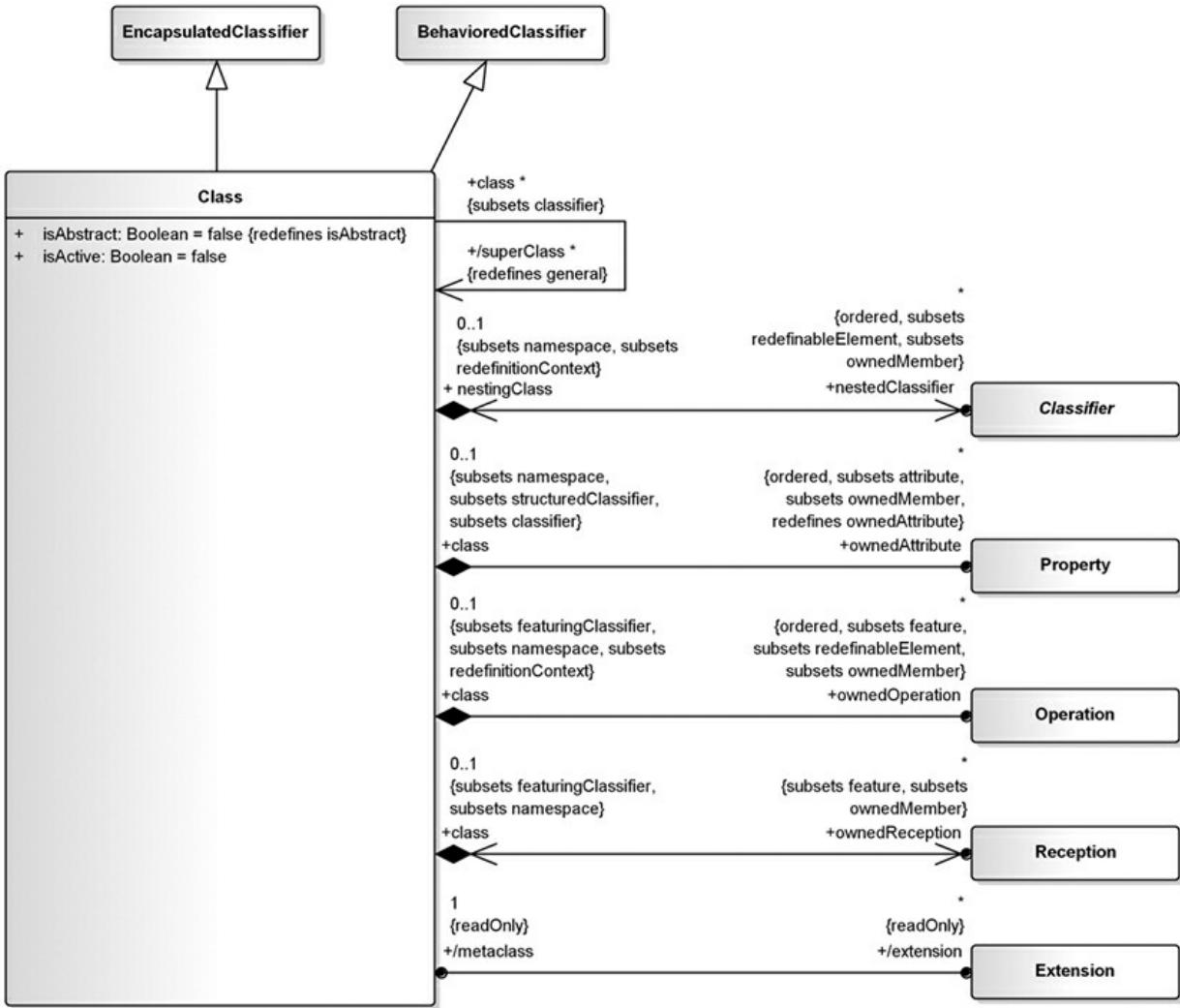


Figura 18.2 – Metamodelo contendo as Metaclasses Utilizadas para a Modelagem de Classes.

Ao observarmos essa figura, percebemos que a metaclasses **Class** representa o conceito de Classe explicado no capítulo 4. Dessa forma, uma classe representada em um modelo nada mais é que uma instância da metaclasses **Class**.

Ao visualizarmos essa figura, também percebemos que a metaclasses **Class** é uma especialização tanto da metaclasses **EncapsulateClassifier** como da **BehavioredClassifier**. Observa-se ainda que essa metaclasses possui associação com cinco metaclasses, **classifier**, **Property**, **Operation**, **Reception** e **Extension**, além de se associar consigo mesma, denotando que uma classe pode ser uma superclasse em relação a outra.

Nas próximas seções, explicaremos sucintamente as metaclasses contidas

nesse metamodelo.

#### **18.4.1 Metaclasses Classifier**

Um classificador (classifier) é uma metaclasses abstrata que representa uma classificação de instâncias. Descreve um conjunto de instâncias com características em comum. É um espaço de nome cujos membros podem incluir características. Um classificador é um tipo e pode ter generalizações, tornando possível definir relacionamentos de generalização para outros classificadores. Essa metaclasses pode especificar uma hierarquia de generalização por meio de referência aos seus classificadores gerais. É um elemento redefinível, o que significa que é possível redefinir classificadores aninhados.

Um classificador é um mecanismo que descreve características comportamentais e estruturais, porém as instâncias de um classificador não são objetos como as instâncias de uma classe, mas elementos UML concretos (que incluem classes). Não pode ser utilizado em um modelo, somente suas instâncias o podem sê-lo. Classificadores são utilizados apenas em metamodelos.

#### **18.4.2 Metaclasses StructuredClassifier**

Esta é uma metaclasses abstrata. Classificadores estruturados podem possuir uma estrutura interna abrangendo um rede de papéis conectados, que podem eles mesmos ser instâncias de classificadores estruturados, e uma estrutura externa consistindo em uma ou mais **Portas** (Ports).

#### **18.4.3 Metaclasses EncapsulatedClassifier**

Esta metaclasses estende a metaclasses **StructuredClassifier**, permitindo que um classificador encapsulado seja isolado de seu ambiente, por meio da capacidade de este possuir **Portas**.

#### **18.4.4 Metaclasses BehavioredClassifier**

A metaclasses **BehavioredClassifier** é um classificador com especificações de comportamento definidas em seu espaço de nomes (namespace). A metaclasses **NameSpace** foi explicada no capítulo 16.

## **18.4.5 Metaclasses StructuralFeature**

Esta é uma metaclasses abstrata que representa uma característica tipada de um classificador que especifica a estrutura de instâncias do classificador.

## **18.4.6 Metaclasses BehavioralFeature**

Uma **BehavioralFeature** (característica comportamental) é uma característica de um classificador que especifica um aspecto do comportamento de suas instâncias. Uma característica comportamental é implementada (realizada) por um comportamento. Uma característica comportamental especifica que um classificador responderá a uma requisição pela invocação de seu método implementado.

## **18.4.7 Metaclasses Property**

Uma **Property** (propriedade) é uma **StructuralFeature**. Uma propriedade relacionada pelo elemento possuído (**ownedElement**) a um classificador, que não seja uma associação, representa um atributo e pode também representar um final de associação. Relaciona uma instância do classificador a um valor ou conjunto de valores do tipo do atributo.

## **18.4.8 Metaclasses Operation**

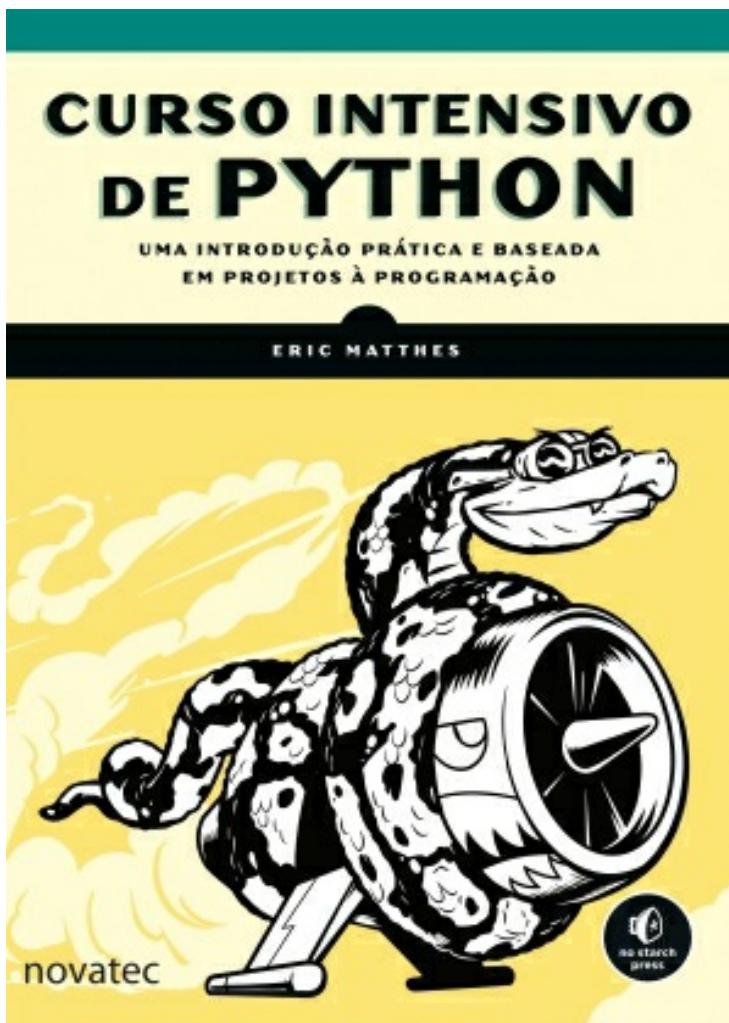
Uma operação é uma característica comportamental de um classificador que especifica o nome, o tipo, os parâmetros e as restrições para invocar um comportamento associado.

## **18.4.9 Metaclasses Reception**

Uma recepção é uma declaração de que um classificador está preparado para receber um sinal.

## **18.4.10 Metaclasses Extension**

Uma extensão é usada para indicar que as propriedades de uma metaclasses são estendidas por meio de um estereótipo e fornece a habilidade de adicionar ou remover estereótipos em uma classe.



## Curso Intensivo de Python

Matthes, Eric

9788575226025

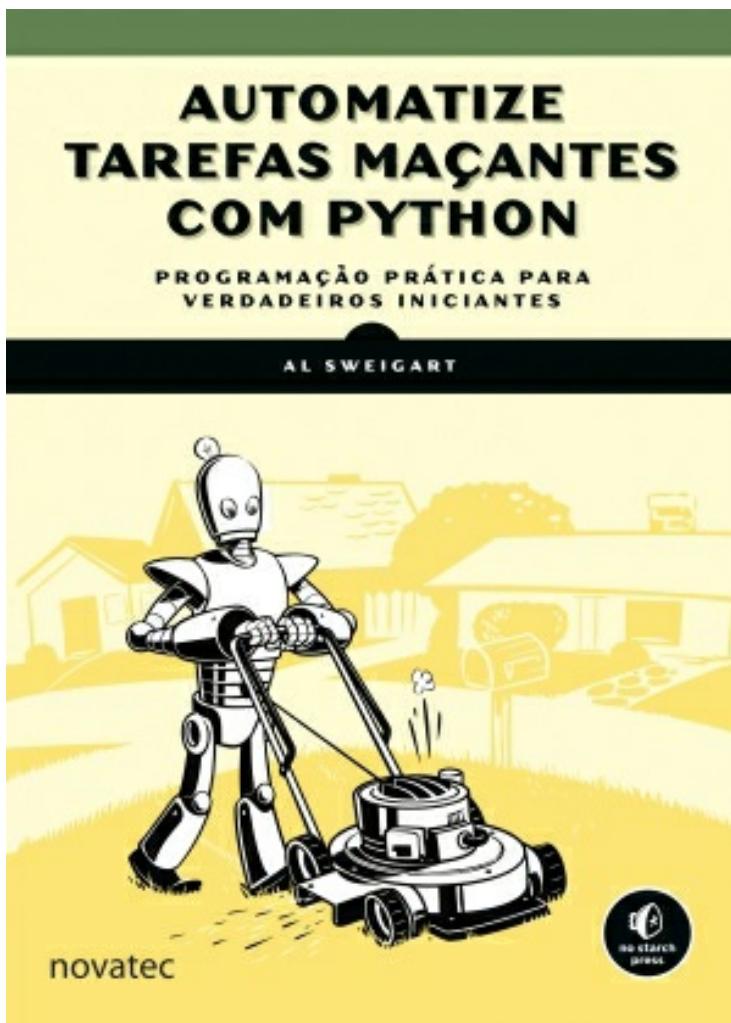
656 páginas

[Compre agora e leia](#)

Aprenda Python – rapidamente! Curso Intensivo de Python é uma introdução completa e em ritmo acelerado à linguagem Python, que

fará você escrever programas, resolver problemas e criar soluções que funcionarão em um piscar de olhos. Na primeira metade do livro você conhecerá os conceitos básicos de programação, como listas, dicionários, classes e laços, e praticará a escrita de códigos limpos e legíveis, com exercícios para cada assunto. Você também aprenderá a deixar seus programas interativos e a testar seu código de modo seguro antes de adicioná-lo a um projeto. Na segunda metade do livro você colocará seu novo conhecimento em prática com três projetos substanciais: um jogo de arcade, inspirado no Space Invaders, visualizações de dados com as bibliotecas extremamente práticas de Python, e uma aplicação web simples que poderá ser implantada online. À medida que avançar no Curso Intensivo de Python, você aprenderá a: usar bibliotecas Python e ferramentas eficazes, incluindo matplotlib, NumPy e Pygal; criar jogos 2D que respondam a pressionamentos de teclas e a cliques de mouse, com aumento no nível de dificuldade à medida que o jogo prosseguir; trabalhar com dados para gerar visualizações interativas; criar e personalizar aplicações web e implantá-las de modo seguro online; lidar com equívocos e erros para que você possa resolver seus próprios problemas de programação. Se você está pensando seriamente em explorar a programação, Curso Intensivo de Python deixará você pronto num instante para escrever programas de verdade. Por que esperar mais? Ligue seus motores e comece a programar! Utiliza Python 2 e 3

[Compre agora e leia](#)



## Automatize tarefas maçantes com Python

Sweigart, Al

9788575226087

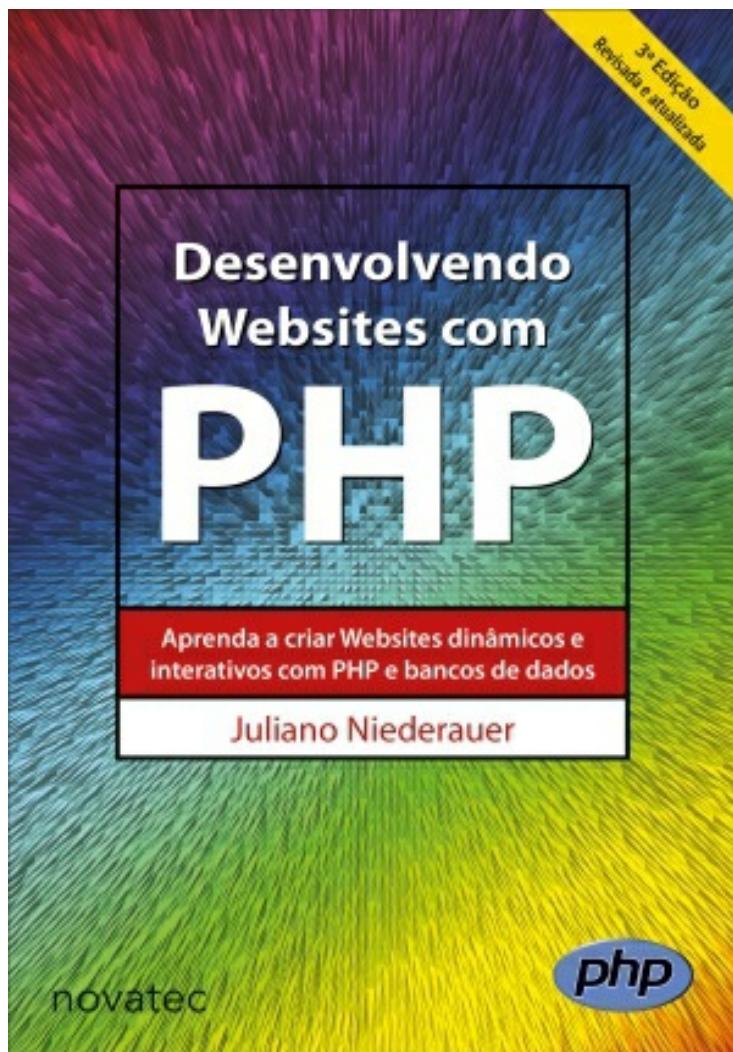
568 páginas

[Compre agora e leia](#)

APRENDA PYTHON. FAÇA O QUE TEM DE SER FEITO. Se você já passou horas renomeando arquivos ou atualizando centenas de

células de planilhas, sabe quão maçantes podem ser esses tipos de tarefa. Que tal se você pudesse fazer o seu computador executá-las para você? Com o livro Automatize tarefas maçantes com Python, você aprenderá a usar o Python para criar programas que farão em minutos o que exigiria horas para ser feito manualmente – sem que seja necessário ter qualquer experiência anterior com programação. Após ter dominado o básico sobre programação, você criará programas Python que realizarão proezas úteis e impressionantes de automação sem nenhum esforço: Pesquisar texto em um arquivo ou em vários arquivos. Criar, atualizar, mover e renomear arquivos e pastas. Pesquisar na Web e fazer download de conteúdos online. Atualizar e formatar dados em planilhas Excel de qualquer tamanho. Separar, combinar, adicionar marcas-d'água e criptografar PDFs. Enviar emails para lembretes e notificações textuais. Preencher formulários online. Instruções passo a passo descreverão cada programa e projetos práticos no final de cada capítulo desafiarão você a aperfeiçoar esses programas e a usar suas habilidades recém-adquiridas para automatizar tarefas semelhantes. Não gaste seu tempo executando tarefas que um macaquinho bem treinado poderia fazer. Mesmo que não tenha jamais escrito uma linha de código, você poderá fazer o seu computador realizar o trabalho pesado. Saiba como em Automatize tarefas maçantes com Python.

[Compre agora e leia](#)



## Desenvolvendo Websites com PHP

Niederauer, Juliano

9788575226179

320 páginas

[Compre agora e leia](#)

Desenvolvendo Websites com PHP apresenta técnicas de programação fundamentais para o desenvolvimento de sites

dinâmicos e interativos. Você aprenderá a desenvolver sites com uma linguagem utilizada em milhões de sites no mundo inteiro. O livro abrange desde noções básicas de programação até a criação e manutenção de bancos de dados, mostrando como são feitas inclusões, exclusões, alterações e consultas a tabelas de uma base de dados. O autor apresenta diversos exemplos de programas para facilitar a compreensão da linguagem. Nesta obra, você irá encontrar os seguintes tópicos: O que é PHP e quais são suas características; Conceitos básicos e avançados de programação em PHP; Como manipular diversos tipos de dados com o PHP; Criação de programas orientados a objetos (OOP); Comandos PHP em conjunto com tags HTML; Utilização de includes para aumentar o dinamismo de seu site; Como tratar os dados enviados por um formulário HTML; Utilidade das variáveis de ambiente no PHP; Criação de banco de dados em MySQL, PostgreSQL ou SQLite; Comandos SQL para acessar o banco de dados via PHP; Como criar um sistema de username/password para seu site; Utilização de cookies e sessões; Leitura e gravação de dados em arquivos-texto; Como enviar e-mails pelo PHP.

[Compre agora e leia](#)

# **JOVEM E BEM-SUCEDIDO**

Um guia para a realização  
profissional e financeira



novatec

Juliano Niederauer

## Jovem e Bem-sucedido

Niederauer, Juliano

9788575225325

192 páginas

[Compre agora e leia](#)

Jovem e Bem-sucedido é um verdadeiro guia para quem deseja

alcançar a realização profissional e a financeira o mais rápido possível. Repleto de dicas e histórias interessantes vivenciadas pelo autor, o livro desmistifica uma série de crenças relativas aos estudos, ao trabalho e ao dinheiro. Tem como objetivo orientar o leitor a planejar sua vida desde cedo, possibilitando que se torne bem-sucedido em pouco tempo e consiga manter essa realização no decorrer dos anos. As três perspectivas abordadas são: ESTUDOS: mostra que os estudos vão muito além da escola ou faculdade. Aborda as melhores práticas de estudo e a aquisição dos conhecimentos ideais e nos momentos certos. TRABALHO: explica como você pode se tornar um profissional moderno, identificando oportunidades e aumentando cada vez mais suas fontes de renda. Fornece ainda dicas valiosas para desenvolver as habilidades mais valorizadas no mercado de trabalho. DINHEIRO: explica como assumir o controle de suas finanças, para, então, começar a investir e multiplicar seu patrimônio. Apresenta estratégias de investimentos de acordo com o momento de vida de cada um, abordando as vantagens e desvantagens de cada tipo de investimento. Jovem e Bem-sucedido apresenta ideias que o acompanharão a vida toda, realizando importantes mudanças no modo como você planeja estudar, trabalhar e lidar com o dinheiro.

[Compre agora e leia](#)



# WIRESHARK

PARA PROFISSIONAIS DE SEGURANÇA

Usando Wireshark e o Metasploit Framework

WILEY  
novatec

Jessey Bullock  
Jeff T. Parker

## Wireshark para profissionais de segurança

Bullock, Jessey

9788575225998

320 páginas

[Compre agora e leia](#)

Um guia essencial para segurança de rede e para o Wireshark – um

conjunto de ferramentas repleto de recursos O analisador de protocolos de código aberto Wireshark é uma ferramenta de uso consagrado em várias áreas, incluindo o campo da segurança. O Wireshark disponibiliza um conjunto eficaz de recursos que permite inspecionar a sua rede em um nível microscópico. Os diversos recursos e o suporte a vários protocolos fazem do Wireshark uma ferramenta de segurança de valor inestimável, mas também o tornam difícil ou intimidador para os iniciantes que queiram conhecê-lo.

Wireshark para profissionais de segurança é a resposta: ele ajudará você a tirar proveito do Wireshark e de ferramentas relacionadas a ele, por exemplo, a aplicação de linha de comando TShark, de modo rápido e eficiente. O conteúdo inclui uma introdução completa ao Metasploit, que é uma ferramenta de ataque eficaz, assim como da linguagem popular de scripting Lua. Este guia extremamente prático oferece o insight necessário para você aplicar o resultado de seu aprendizado na vida real com sucesso. Os exemplos mostram como o Wireshark é usado em uma rede de verdade, com o ambiente virtual Docker disponibilizado; além disso, princípios básicos de rede e de segurança são explicados em detalhes para ajudar você a entender o porquê, juntamente com o como. Ao usar a distribuição Kali Linux para testes de invasão, em conjunto com o laboratório virtual e as capturas de rede disponibilizadas, você poderá acompanhar os diversos exemplos ou até mesmo começar a pôr em prática imediatamente o seu conhecimento em um ambiente de rede seguro. A experiência prática torna-se mais valiosa ainda pela ênfase em uma aplicação coesa, ajudando você a explorar vulnerabilidades e a expandir todas as funcionalidades do Wireshark, estendendo-as ou integrando-as com outras ferramentas de segurança.

[Compre agora e leia](#)