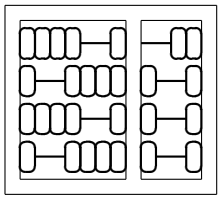
Introdução à Análise Orientada a Objetos e Projeto Arquitetural

Cecília Mary Fischer Rubira
cmrubira@ic.unicamp.br
http://www.ic.unicamp.br/∼cmrubira

Patrick Henrique da Silva Brito
pbrito@ic.unicamp.br
http://www.ic.unicamp.br/∼pbrito



Instituto de Computação - UNICAMP

2009

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Este capítulo define e discute vários conceitos fundamentais do modelo de objetos. A evolução das técnicas de programação e das abstrações nas linguagens de programação são apresentados de como uma mudança crescente na forma de desenvolver software. O paradigma de programação procedural (ou imperativo) e o paradigma orientado a objetos são comparados e as novas estruturas abstratas do modelo de objetos são apresentadas. Finalmente, o capítulo apresenta algumas das principais limitações do modelo de objetos e como a engenharia de software baseada em componentes tenta superá-las.

1.1 Estruturação de Software

Sistemas de software modernos requerem um alto grau de confiabilidade, disponibilidade e segurança. Alguns exemplos são sistemas de controle para centrais telefônicas, sistemas bancários, sistemas financeiros sistemas de e-banking, etc. Entretanto, a construção de sistemas de software complexos é uma tarefa difícil que exige o emprego de técnicas especializadas durante todo o ciclo de vida do sistema.

O modelo de objetos apresenta-se como um modelo promissor para o desenvolvimento de software confiável e robusto devido a características inerentes ao próprio modelo de objetos, tais como, abstração de dados, encapsulamento, herança e reutilização de objetos (classes e componentes). O uso de técnicas orientadas a objetos facilita o controle da complexidade do sistema, uma vez que promove uma melhor estruturação de suas partes. Além disso, o seu uso permite que elementos de software já validados sejam reutilizados, promovendo a reutilização dos casos de teste.

1.1.1 Gerenciamento da Complexidade do Software

Sistemas de software são intrinsicamente complicados e têm se tornado mais complicados ainda com os novos requisitos impostos pelas aplicações modernas: alta confiabilidade, alto desempenho, desenvolvimento rápido e barato, tudo isso associado a uma complexidade de funcionalidade crescente. É possível, e muitas vezes até desejável, ignorar a complexidade dos sistemas de software, mas isso não resolve totalmente o problema; pois ela certamente aparecerá em algum outro lugar. Aplicações de software que tratam de problemas complicados devem lidar com tal complexidade em alguma hora. Como seres humanos, nós empregamos vários mecanismos para "gerenciar" essa complexidade, tais como abstração, generalização e agregação. **Abstração** é um meio pelo qual evitamos a complexidade não desejada e é uma das ferramentas mais eficientes para lidarmos com o nosso mundo complexo.

Cientistas de computação já reconheceram há algum tempo que a chave para o sucesso no desenvolvimento de software está no controle da sua complexidade. O conceito de linguagens de programação de alto nível e seus compiladores foi um grande passo em direção a este objetivo, pois proporcionou uma abstração do funcionamento interno do computador, permitindo que o desenvolvedor de software programasse sem precisar ser um especialista no funcionamento do hardware utilizado. Depois disso, os pesquisadores e profissionais começaram a reconhecer a necessidade de melhores processos e ferramentas para gerenciar a complexidade; como conseqüência, surgiram os conceitos de programação estruturada e bibliotecas de programas.

Embora essas contribuições tenham sido valiosas, elas ainda deixam muito a desejar em termos de reutilização de software. Em outras palavras, existe uma complexidade ainda maior do que simplesmente tentar minimizar a complexidade local de cada parte de um programa. Um tipo de complexidade mais importante está relacionada com a complexidade da sua organização global: a macro complexidade da estrutura de um programa ou sistema (i.e., o grau de associação ou interdependência entre as principais partes de um programa). A estrutura dos módulos do sistema e a comunicação entre eles é representada pela arquitetura de software. Esse conceito será visto no Capítulo 7 (Seção 7.2).

Podemos dizer que a complexidade de um componente ou subsistema de software é alguma medida do esforço mental requerido para entendê-lo. De uma forma mais pragmática, ela é função dos relacionamentos entre os diversos elementos do sistema. Complexidade é um dos principais fatores que contribui para que a produção de software seja de baixa qualidade.

1.1.2 Técnicas de Estruturação de Software

Uma diferença visível entre uma organização de programa bem estruturada e uma organização mal estruturada está na complexidade. Alguns conceitos da teoria geral de sistemas podem ser aplicados para reduzir a complexidade de sistemas de software, a saber [37]: (i) particionamento de sistemas em partes que sejam muito bem delimitadas, procedimento conhecido por **modularização** do sistema; (ii) representação do sistema como uma hierarquia com separação de níveis de abstração, que proporcionam uma **visão hierárquica** do sistema; (iii) maximização da independência entre

as partes do sistema, que proporciona o **baixo acoplamento** entre os elementos do sistema; e o (iv) agrupamento das partes inter-relacionadas do sistema, o que reduz a necessidade de troca de mensagens entre as partes do sistema, nesse caso, dizemos que o sistema possui **alta coesão**.

O particionamento de um programa em componentes menores e individuais pode reduzir a sua complexidade até certo ponto. Uma justificativa plausível para essa redução é o fato do particionamento criar um número de interfaces bem definidas dentro do modelo, isto é, as interações entre os módulos seguem listas pré-definidas, com as operações de cada módulo. Essas interfaces são muito úteis para o entendimento do programa, uma vez que mostram quais elementos são relevantes e quais não são em determinados contextos, estreitando o foco de atenção. Em outras palavras, a interface "esconde" informação "irrelevante" atrás dela, sendo considerada um importante mecanismo de abstração.

O conceito de hierarquia é de vital importância tanto para o entendimento quanto para a construção de sistemas. Pelo fato da mente humana ter um limite pequeno para o número de fatos com os quais pode lidar simultaneamente, nós podemos entender melhor os sistemas que estão definidos de forma hierárquica [43]. Hierarquias permitem a estratificação de um sistema em vários níveis de detalhes. Cada camada (ou nível) abstrai os detalhes representados nas camadas inferiores. O conceito de camada facilita o entendimento do sistema, uma vez que possibilita o ocultamento de níveis de detalhes desnecessários.

Embora o particionamento e a estrutura hierárquica sejam conceitos muito importantes na estruturação de sistemas, existe um terceiro conceito relacionado que é igualmente importante: independência. Este conceito implica na maximização da independência de cada componente do sistema para que a sua complexidade seja reduzida. Portanto, o problema não é meramente particionar um programa numa hierarquia, mas determinar como particionar um programa numa estrutura hierárquica de tal forma que cada subsistema seja tão independente dos outros quanto possível. Portanto, **modularidade** pode ser definida como a propriedade de um sistema que foi decomposto num conjunto de módulos fracamente acoplados e altamente coesos.

Outra noção muito importante relacionada com as três discutidas anteriormente é o da representação explícita das inter-conexões entre os componentes do sistema [36]. Esse foco na interação entre os componentes, além de facilitar o entendimento, possibilita uma definição clara de protocolos de comunicação entre eles. Essas vantagens são percebidas mais claramente em sistemas de grande porte, onde é comum a ocorrência de efeitos colaterais decorrentes da falta de clareza desses inter-relacionamentos.

1.1.3 Crise de Software

O termo "Crise de Software"[1] foi cunhado na Conferência de Engenharia de Software da OTAN em 1968, na Europa. Naquela época concluiu-se que os métodos de produção de software não eram adequados para as necessidades crescentes das indústrias de defesa e de processamento de dados. Vários problemas foram identificados nos sistemas de software produzidos na época, por exemplo,

[1] do inglês *software crisis*

eles eram pouco predizíveis, apresentavam uma baixa qualidade, tinham alto custo de manutenção e havia muita duplicação de esforços em sua construção.

Os processos subseqüentes foram tentativas de superar essa crise de software. Entretanto, apesar da proliferação dos processos de análise e de projeto, existem várias evidências de que muitos projetos de software ainda são entregues atrasados, com o orçamento estourado e com a especificação incompleta. Nesse contexto, surge o modelo de objetos que, embora nos ofereça novas ferramentas e uma abordagem mais moderna para desenvolvimento de software, é considerado mais uma abordagem evolucionária do que revolucionária, no sentido de que o modelo engloba conceitos bem conhecidos, inovando principalmente na maneira como são estruturados e utilizados. Os benefícios advindos do emprego do modelo de objetos são vários, por exemplo, abstração de dados/encapsulamento, modularidade, reutilização, maior facilidade de extensão e manutenção.

1.2 Evolução da Abstração em Linguagens de Programação

1.2.1 Desenvolvimento de Software

Muitos pesquisadores reconhecem que existe um relacionamento muito próximo entre os pensamentos e a linguagem humana usada para expressá-los. De fato, a natureza da linguagem modela e dá forma aos nossos pensamentos e vice-versa: linguagem e pensamento modelam mutuamente um ao outro e não há uma precedência entre eles. A relação entre programas de software e a linguagem de programação usada para desenvolvê-los é semelhante [20]. Por esse motivo, as linguagens de programação são as representações abstratas mais comuns em computação.

Entretanto, é difícil transformar as abstrações do mundo diretamente nas abstrações de uma linguagem de programação sem ser necessário passar por passos intermediários, como apresentado na Figura 1.1. Neste caso, notações gráficas intermediárias são úteis para ajudar o programador na representação das suas abstrações para a linguagem. Podemos, então, definir **desenvolvimento de software** como um processo através do qual nós refinamos transformações sucessivas de uma representação de alto nível para uma representação de mais baixo nível executável num computador. Conseqüentemente, o processo inteiro é dependente das facilidades para a representação de abstrações disponíveis na linguagem de programação alvo. Se a linguagem restringe de alguma forma o modo como as abstrações podem ser definidas, ela certamente limitará a modelagem das aplicações.

A construção descendente[2] é uma técnica útil que requer que uma aplicação seja projetada considerando inicialmente sua macro-estrutura como um todo. Nos passos seguintes do desenvolvimento, esta estrutura é detalhada até que uma implementação seja produzida. O refinamento passo-a-passo(do inglês *stepwise refinement*) proporciona uma estratégia para a realização desse detalhamento [7, 39]. De fato, o processo de refinamento passo-a-passo acontece até que a solução do problema se torne um programa. Este processo considera cada subproblema como um prob-

[2]do inglês *top-down*

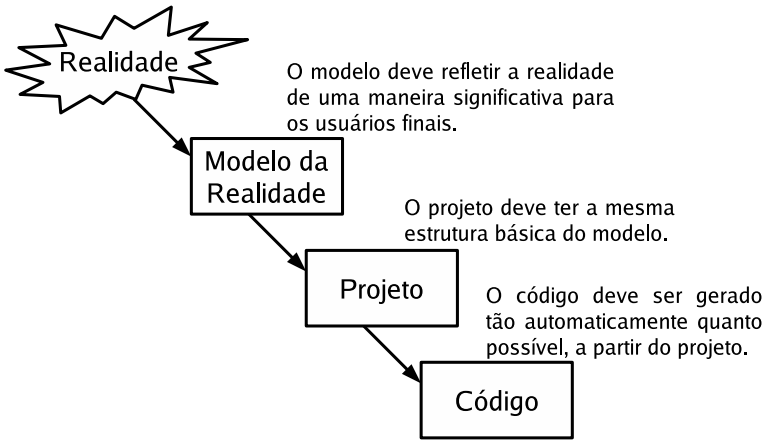


Figura 1.1: Passos Intermediários da Modelagem da Realidade

lema separado, cria uma descrição de alto nível para ele e então refina-o em termos de outros subproblemas.

A abordagem descendente é uma disciplina de Engenharia de Software madura na qual foram baseados diversos processos de projeto estruturado durante os últimos 30 anos. Como será visto na Seção 1.2, processos de análise e projeto Orientado a Objetos apresentam uma alternativa para esta abordagem convencional, especialmente quando tratamos de sistemas grandes e complexos.

No início do desenvolvimento de linguagens de programação, linguagens de montagem permitiam aos projetistas escrever programas baseados em instruções de máquina (operadores) que manipulavam os conteúdos das locações de memória (operandos). Portanto, as abstrações de dados e controle eram de muito baixo nível. Um grande passo de evolução ocorreu quando surgiram as primeiras grandes linguagens de programação imperativas (Fortran e Cobol). Fortran foi importante porque introduziu a noção de subprogramas (funções e procedimentos), enquanto Cobol introduziu a noção de descrição de dados.

Subseqüentemente, Algol-60 introduziu o conceito de estrutura de bloco, procedimento, etc. Ela influenciou fortemente numerosas linguagens de programação que foram chamadas linguagens baseadas em Algol. A filosofia de Algol-68 consistia da escolha de um conjunto adequado de mecanismos que deveriam ser combinados sistematicamente. Apesar da importância de Algol-60 para a evolução das linguagens de programação, foi Pascal que tornou-se a linguagem de programação mais popular, principalmente por causa da sua simplicidade, sistemática e eficiência de implementação. As duas linguagens possuem estruturas de controle ricas e definições de tipos e estruturas de dados, seguindo as idéias da programação estruturada criadas por Wirth, Dijkstra and Hoare [16].

1.2.2 Módulos, Pacotes e Subsistemas

Desde os anos 70, as linguagens têm se preocupado em dar mais suporte para programação em larga escala. Este tipo de programação (complementar à programação em pequena e média escala) preocupa-se com a construção de programas grandes a partir de módulos. Um **módulo** é uma unidade de programa que pode ser implementado de uma maneira mais ou menos independente dos outros módulos. Um módulo bem projetado tem um propósito bem definido, assim como suas interfaces de comunicação com outros módulos. Tal módulo é considerado **reutilizável**, quando tem grande chance de ser reutilizado, podendo ser incorporado a vários programas; e **modificável**, quando pode ser alterado sem causar grandes mudanças em outros módulos relacionados. A disciplina do ocultamento da informação, também conhecida como **encapsulamento**, foi proposta por Parnas [38], por volta de 1972 e sua idéia era encapsular variáveis globais em um módulo juntamente com o grupo de operações que tinham acesso direto a elas. Outros módulos podiam acessar essas variáveis somente indiretamente, através das operações oferecidas pelos módulos.

Somente o que o módulo faz é passado para o seu cliente; o como é implementado somente diz respeito ao implementador do módulo. Se um módulo é implementado desta forma, é dito que esse módulo encapsula seus componentes. Para a obtenção de uma interface bem definida, um módulo tipicamente faz com que apenas um número pequeno de componentes sejam visíveis para os clientes externos. Tais componentes são exportados pelo módulo, enquanto muitos outros componentes permanecem "ocultos" dentro dele. Portanto, encapsulamento sugere que à medida que uma estrutura de dados deve residir dentro de um módulo, uma interface oferece o acesso necessário a tais estruturas para módulos externos. Em resumo, o encapsulamento minimiza as inter-dependências entre módulos escritos separadamente através da definição de interfaces estritas.

Um conceito semelhante aos módulos é o que conhecemos como subsistemas. Um **subsistema** pode ser visto como um tipo particular de módulo que, por ser auto-contido, pode oferecer seus serviços sem necessitar interagir com o meio externo. Seguindo esse raciocínio, um sistema pode ser visto como um subsistema de um outro sistema maior.

Existem pelo menos três mecanismos de linguagens de programação que dão apoio à modularização: (i) módulos, como implementados em Modula-2; (ii) pacotes, como implementado em Ada, Java [3] e C# [24]; e (iii) tipo abstrato de dados, tipos definidos indiretamente através de suas operações.

Um módulo consiste de duas partes relacionadas: (i) especificação do módulo, chamada de *spec* e (ii) implementação do módulo, chamada de *body*. Enquanto a parte *spec* contém um conjunto de declarações de estruturas de dados e assinaturas de procedimentos, a parte *body* contém as implementações declaradas na *spec*. Cada entidade declarada na *spec* deve estar implementada no *body*. Entretanto, o *body* pode conter estruturas de dados e procedimentos adicionais usados para implementar as entidades visíveis. Essas entidades adicionais declaradas apenas no body devem ser privadas, sendo invisíveis aos clientes da *spec*, a fim de não interferir na interface oferecida pelo módulo.

O trecho de código a seguir apresenta um exemplo de implementado do módulo *stack* na linguagem ADA. Esse módulo representa a estrutura de uma pilha. As linhas de 1 a 6 apresentam

a especificação da pilha, que através das operações públicas `Push(...)` e `Pop(...)` (Linhas 4 e 5), se comunicam com os módulos externos. Além disso, a especificação define uma variável que representa a capacidade máxima da pilha (Linha 2). A implementação da pilha (*body*, linhas de 9 a 29) define o comportamento das operações da pilha (Linhas 14 a 28) e além disso, define um tipo `Table` (Linha 10) e duas variáveis internas (Linhas 11 e 12), invisíveis externamente.

```
1       -- *especificação*
2          Size : Positive;
3       package Stack is
4          procedure Push(E : in  Integer);
5          procedure Pop (E : out Integer);
6       end Stack;
7
8       -- *implementação*
9       package body Stack is
10         type Table is array (Positive range <>) of Integer;
11         Space : Table(1 .. Size);
12         Index : Natural := 0;
13
14         procedure Push(E : in Integer) is
15         begin
16            if Index < Size then
17               Index := Index + 1;
18               Space(Index) := E;
19            end if;
20         end Push;
21
22         procedure Pop(E : out Integer) is
23         begin
24            if Index > 0 then
25               E := Space(Index);
26               Index := Index - 1;
27            end if;
28         end Pop;
29      end Stack;
```

A maioria das linguagens de programação atuais utilizam o conceito de pacotes, discutido em maiores detalhes na Seção 4.8 do Capítulo 4, de uma maneira mais genérica que em ADA. A principal diferença entre esses dois conceitos de pacotes é que enquanto em ADA há uma separação

explícita entre especificação e implementação, em linguagens como Java e C# os pacotes determinam apenas espaço de nomes, deixando a critério dos desenvolvedores separar ou não a especificação da implementação. Os espaços de nomes são utilizados como um mecanismo para organizar elementos em grupos hierárquicos, podendo controlar inclusive quais elementos são visíveis externamente. A questão da visibilidade em orientação a objetos será abordada em detalhes na Seção 4.8.1 do Capítulo 4.

1.2.3 Tipos Abstratos de Dados

A noção de **tipos abstratos de dados** (TAD) se refere ao encapsulamento de uma estrutura de dados juntamente com as operações que manipulam essas estruturas dentro de uma região protegida. Uma linguagem dá apoio a tipos abstratos de dados quando ela possui mecanismos que permitem a sua representação diretamente. Linguagens de programação como Ada e Modula-2 dão apoio a tipos abstratos de dados, mas ainda têm certas limitações. As principais são:

(i) o sistema de tipos é unidimensional, ou seja, um programa é desenvolvido como um conjunto de tipos abstratos de dados cuja estrutura é definida no nível horizontal: as hierarquias de generalização/especialização não podem ser representadas explicitamente.

(ii) tipos abstratos de dados não são representados explicitamente em tempo de execução, isto é, embora tipos abstratos de dados sejam úteis durante as fases de análise, projeto e implementação, eles desaparecem durante o tempo de execução e o software se torna de novo um monte de linhas de código agrupadas em módulos completamente desestruturados.

No exemplo do módulo de pilha apresentado anteriormente, o TAD *Stack* seria definido pelas operações públicas `Push(...)` e `Pop(...)`, além da variável pública `Size`, declarada na especificação do módulo.

1.2.4 Programação Orientada a Objetos

O próximo passo da evolução das linguagens de programação foi introduzido com o conceito de objetos, criado por Dahl e Nygaard com a linguagem Simula-67 [17], e consolidado com a linguagem Smalltalk-76. Simula-67 introduziu os conceitos de classe, objeto e herança. O modelo clássico de objetos emprega classes para a descrição de objetos. Essas classes contém a definição da estrutura dos objetos (i.e. dados e funções). Além disso, através dos mecanismos de herança e agregação, classes já existentes podem compartilhar seu comportamento com novas classes. Essencialmente, o modelo de objetos trata dados e funções como aspectos indivisíveis no domínio do problema.

O forte relacionamento entre o modelo de objetos e a noção de tipo abstrato de dados se torna evidente, uma vez que os objetos podem ser vistos como instâncias de tipos abstrato de dados. Na verdade, na maioria das linguagens orientadas a objetos, a definição de uma classe descreve um tipo de dados associado com as operações que podem ser executadas nas instâncias desse tipo.

Fazendo um panorama histórico, destacamos seis acontecimentos principais que marcaram a criação do modelo de objetos. O primeiro deles, que já foi relatado anteriormente a introdução do conceito de objetos no ano de 1967, lançamento da linguagem Simula-67. Em seguida, em 1972 Dahl escreveu um artigo sobre ocultamento de informações. Apesar de nesta etapa da história o conceito de objetos já estar bem definido, apenas em 1976, com o lançamento da primeira versão do Smalltalk a orientação a objetos foi consolidada. A partir daí, o modelo de objetos evoluiu no sentido de oferecer novas linguagens de programação. Em 1983 foi disponibilizada a primeira versão do C++, versão orientada a objetos da disseminada linguagem C. Em 1988, foi lançada a linguagem Eiffel, a primeira linguagem considerada orientada a objetos "pura". Finalmente, já no final do século XX, mais precisamente no ano de 1995, foi lançada a primeira versão da linguagem Java, uma linguagem orientada a objetos "pura", baseada na sua antecessora C++.

Programação orientada a objetos é um modelo de programação baseado em conceitos, tais como objetos, classes, tipos, ocultamento da informação, herança, polimorfismo e parametrização. Análise e projeto orientados a objetos oferecem uma maneira de usar todos esses conceitos para a estruturação e construção de sistemas. Esses conceitos são intrinsicamente independentes de linguagens e cada uma tem sua própria maneira de implementá-los.

A essência da programação orientada a objetos é a resolução de problemas baseada na identificação de objetos do mundo real pertencentes ao domínio da aplicação e no processamento requerido por esses objetos, através de interações entre eles. Esta idéia de "programas simulando o mundo real" cresceu com o lançamento do Simula-67, que inicialmente foi projetada para o desenvolvimento de aplicações de simulação. Devido ao fato do mundo estar povoado de objetos, uma simulação de tal mundo deveria conter objetos simulados capazes de enviar e receber mensagens e reagir às mensagens recebidas.

Conseqüentemente, na programação orientada a objetos, a execução de um programa consiste de um conjunto de objetos relacionados que trocam mensagens entre si, isto é, que se comunicam através da execução das operações uns dos outros. Cada objeto tem um estado interno composto por atributos que são modificados mediante a recepção e o envio de mensagens. Programas são construídos através da definição de classes, que se relacionam e formam hierarquias de abstração (Seção 1.3.4). Este estilo de programação tem algumas características positivas, por exemplo: (i) modularidade, (ii) suporte explícito para refatorar os grupos comuns, (iii) visão unificada de dados e operações, e (iv) desenvolvimento incremental e evolucionário.

Muito do interesse no modelo de objetos é uma conseqüência da melhor estruturação do sistema e do encapsulamento entre dados e funções, o que auxilia principalmente na construção de programas complexos. Embora as especulações sobre o aumento da reutilização não tenham se mostrado reais, a estruturação do raciocínio e o menor acoplamento proporcionado pelo conceito de objetos melhoram a qualidade do produto final e reduzem os custos de manutenção do sistema.

Ao longo deste livro, diversos diagramas representando aspectos diferentes de sistemas de software orientados a objetos são apresentados. Todos estes diagramas seguem uma notação gráfica chamada UML (sigla do inglês *Unified Modeling Language*) [9], na sua versão 2.0. A linguagem UML é uma linguagem para especificar sistemas orientados a objetos que se tornou um padrão tanto na indústria quanto na academia. A linguagem UML unifica as notações propostas pelos

processos de Rumbaugh (OMT) [40], Booch [49] e Jacobson [25] e é independente do processo ou linguagem de programação adotados. Ela foi padronizada por um comitê internacional que lida com tecnologias orientadas a objetos chamado *Object Management Group* (OMG). A linguagem UML é uma linguagem de modelagem, não um processo.

1.2.5 Programação Orientada a Objetos vs. Programação Estruturada

Para a resolução de problemas, a abordagem estruturada utiliza uma técnica conhecida como **decomposição funcional**. Dessa forma, uma operação complexa é dividida em operações menores e assim sucessivamente, até atingir um nível de detalhes que possa ser implementado facilmente. Dessa forma, as funções assumem um papel central no desenvolvimento, que recebem dados de entrada e produzem dados de saída.

Apesar da estreita relação entre dados e funções, nas linguagens estruturadas esses dois conceitos são completamente disjuntos, tratando cada um deles de uma maneira particular. Por essa razão, os processos estruturados de desenvolvimento de software exigem mudanças de contexto constantes, o que dificulta o mapeamento dos artefatos de diferentes fases do desenvolvimento.

O encapsulamento de dados e funções, decorrente do conceito de objetos e classes foi um grande passo para a homogenização do raciocínio e a redução da complexidade dos sistemas. Com essa visão unificada, a idéia central passa a ser a decomposição de dados, ao invés da decomposição funcional. As funções se tornam ligadas a um modelo de dados, que juntos formam o conceito de classe. Em OO, as classes desempenham um papel semelhante aos módulos da programação estruturada, com a vantagem de garantir a coesão entre elementos agrupados (dados e funções).

A Figura 1.2 explicita a diferença estrutural existente entre a programação estruturada e a programação orientada a objetos.

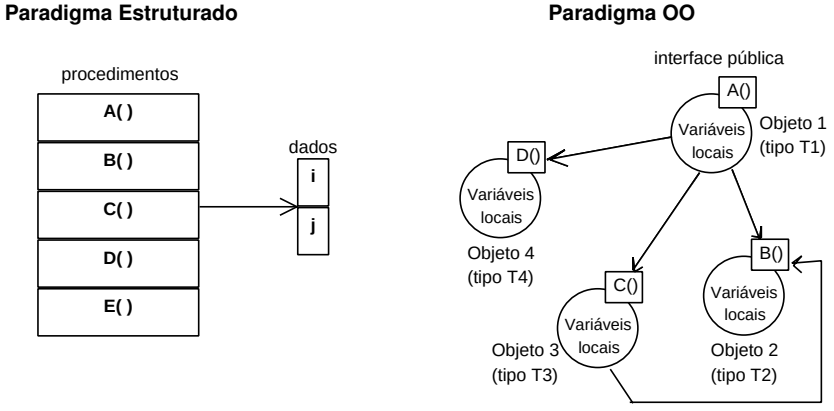


Figura 1.2: Programação Estruturada vs. Programação Orientada a Objetos

Apesar das particularidades de cada uma delas, existem algumas similaridades entre a programação estruturada e a programação orientada a objetos. A Tabela 1.1 traça um paralelo entre

essas duas abordagens.

Tabela 1.1: Comparação entre a Programação Estruturada e a Orientada a Objetos

Programação Estruturada	Programação Orientada a Objetos
tipos de dados	classes/tipos abstratos de dados
variável	objeto/instância
função/procedimento	operação/serviço
chamada de função	envio de mensagem

1.2.6 Linguagens Orientadas a Objetos

Na literatura existe uma distinção clara entre linguagens baseadas em objetos, baseadas em classes e linguagens orientadas a objetos. De acordo com a taxonomia proposta por Wegner [48] (Figura 1.3), uma linguagem é **baseada em objetos**[3] quando ela dá apoio explícito somente ao conceito de objetos. Uma linguagem é **baseda em classes**[4] quando ela dá apoio tanto para a criação de objetos, quanto para o agrupamento de objetos através da criação de novas classes, mas não dá suporte a nenhum mecanismo de herança. Uma linguagem é dita **orientada a objetos**[5] se ela proporciona suporte lingüístico para objetos, requer que esses objetos sejam instâncias de classes e além disso, oferece suporte para um mecanismo de herança. Portanto:

Linguagem Orientada a Objetos = Objetos + Classes + Herança

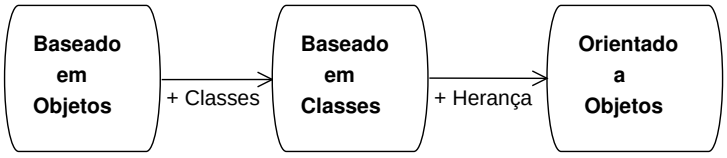


Figura 1.3: Classificação de Wegner

De acordo com essa taxonomia, linguagens como Ada85 e CLU são baseadas em objetos, enquanto C++ [45], Java, C#, Modula-3 [12] e Smalltalk-80 [21] são orientadas a objetos. Uma classificação mais abrangente é proposta por Blair *et al.* [6] que também inclui as linguagens baseadas em delegação. **Delegação** é um mecanismo parecido com o de herança, que promove o compartilhamento de comportamento entre objetos, ao invés de ser entre classes (veja a Seção 4.9). Blair classifica uma linguagem como sendo orientada a objetos se ela dá apoio à criação de objetos e a um

[3]do inglês *object-based language*

[4]do inglês *class-based language*

[5]do inglês *object-oriented language.*

mecanismo de compartilhamento de comportamento (herança ou delegação). Para propósitos deste livro, adotaremos a taxonomia proposta por Wegner, ou seja, para nós uma linguagem orientada a objetos dá suporte direto para objetos, classes e herança.

O modelo de objetos compreende um conjunto de princípios que formam a fundação do modelo de objetos. Um ponto muito importante na programação orientada a objetos é a obtenção de componentes reutilizáveis e flexíveis através de conceitos como, por exemplo, objetos, classes, tipos, herança, polimorfismo, e acoplamento dinâmico. O conceito de herança é exclusivo do modelo de objetos.

1.3 Fundamentos do Paradigma de Objetos

1.3.1 O Conceito de Objeto

O modelo de objetos representa elementos que encontramos no mundo real. Esses elementos podem ser de vários tipos, como entidades físicas (aviões, robôs, etc.) e abstratas (listas, pilhas, filas, etc.). A característica mais importante (e diferente) da abordagem orientada a objetos para desenvolvimento de software é a unificação dos dados e funções, que tradicionalmente são considerados separadamente. Essa união é materializada através do conceito de objetos.

A representação unificada desses dois elementos de programação resulta no que nós chamamos de **encapsulamento**. Além de estruturar a representação de dados e funções relacionados, o encapsulamento possibilita omitir detalhes desnecessários externamente aos objetos. A principal política para garantir esse encapsulamento é a atribuição de níveis de visibilidade: os dados devem ser definidos como privados do objeto e só podem ser consultados ou modificados através dos seus procedimentos, que são públicos. Portanto,

$$\boxed{\text{Objeto} = \text{dados(privados)} + \text{procedimentos(públicos)}}$$

A comunicação entre dois objetos de um sistema acontece através do envio de mensagens. Como um objeto não deve possuir atributos públicos, o envio de uma mensagem por um objeto deve implicar na execução de uma operação da interface pública do objeto recipiente. Como apresentado na Figura 1.4, essa restrição de comunicação visa evitar o acesso direto ao estado do objeto por parte das entidades externas a ele. Portanto, um objeto encapsula tanto o seu **estado**, isto é, seus dados, quanto o seu **comportamento**, isto é, suas operações, que são implementadas através de métodos.

No contexto de orientação a objetos, comportamento define como um objeto age e reage em termos das suas mudanças de estado e troca de mensagens [8] e é completamente definido pelas suas operações. Entretanto, na teoria de tipos abstratos de dados, o termo comportamento é usado para denotar a **interface pública** de um objeto, que é definida pelo conjunto de operações aplicáveis a ele.

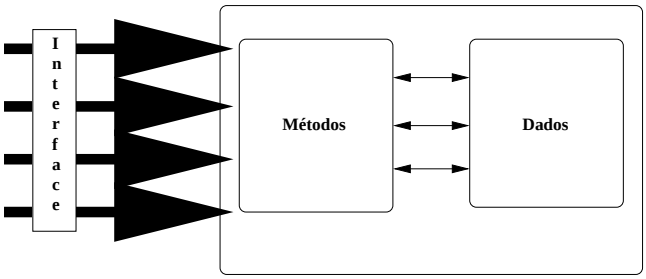


Figura 1.4: O Conceito de Objeto

Em orientação a objetos, o comportamento que um cliente pode realizar sobre um objeto é declarado através de **operações**. Essas operações são implementadas através de **métodos** e o conjunto de operações e atributos públicos constituem a interface pública de um objeto (Figura 1.5). É importante que fique clara a diferença entre uma operação e seu respectivo método: a primeira é abstrata (especificação), enquanto o segundo é concreto (implementação). C++ usa o termo função membro[6] para se referir a um método.

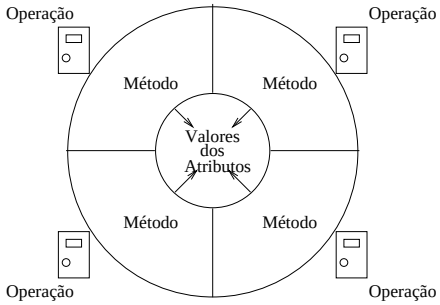


Figura 1.5: Operações, Métodos e Atributos

Em resumo, um objeto possui: (i) um estado, (ii) um comportamento bem definido, formado pelo conjunto de operações da sua interface pública, e (iii) uma identidade única.

1.3.2 Classes

Uma **classe** é a descrição de um molde que especifica as propriedades e o comportamento relativo a um conjunto de objetos similares. Todo objeto é **instância** de apenas uma classe. Toda classe tem um nome e um corpo que define o conjunto de atributos e operações possuídas pelas suas instâncias. É importante distinguirmos objetos de suas classes; o termo classe é usado para identificar um grupo de objetos e o termo objeto se refere a uma instância de uma classe. No modelo de objetos, atributos e operações são usualmente parte da definição de uma classe.

[6]do inglês *member function*

A Figura 1.6 mostra a representação em UML para classes e objetos. Uma classe pode ser representada de uma das duas formas apresentadas na Figura 1.6 (a). Um objeto, por sua vez, é representado de forma similar a uma classe, exceto pelo fato do seu nome ser sublinhado (Figura 1.6 (b)). O nome após os dois pontos representa a classe da qual o objeto foi instanciado ("joao:Usuario").

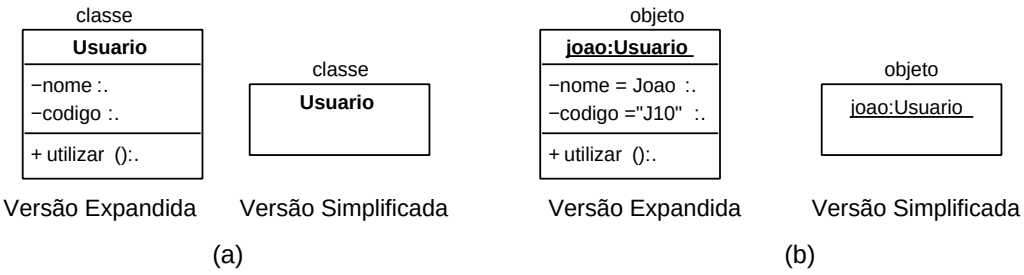


Figura 1.6: Classes e Objetos em UML

O fato dos atributos da Figura 1.6 possuírem visibilidade privada ("-") indica que eles só podem ser acessados internamente; em outras palavras, apenas as operações da própria classe poderão acessar esses atributos diretamente. Em contrapartida, os atributos e operações públicos ("+") podem ser acessados normalmente por quaisquer outros objetos do sistema. Na Seção 4.4.3 desde livro, serão apresentadas outros tipos de visibilidade.

De um modo geral, uma classe é considerada bem estruturada quando: (i) proporciona uma abstração bem definida de um elemento oriundo do vocabulário do domínio do problema ou da solução; (ii) é formada por um conjunto pequeno e bem definido de responsabilidades que devem ser cumpridas pela classe; (iii) proporciona uma separação clara entre a especificação da abstração e a sua implementação; e (iv) é simples e fácil de entender, e ao mesmo tempo tem capacidade de extensão e adaptação.

1.3.3 Classificação/Instanciação

A abstração de **classificação/instanciação** permite agrupar objetos similares em uma mesma categoria. Por exemplo, duas pessoas: João e Maria, podem ser classificadas como **instâncias** da categoria Usuario. **Classificação** pode ser descrita como uma associação *é-uma*, por exemplo, podemos dizer que João *é-um* usuário.

A Figura 1.7 apresenta a representação em UML para classificação/instanciação. Como pode ser percebido, um objeto é instância de apenas uma classe, enquanto uma classe pode ter várias instâncias associadas a ela. Neste exemplo, os objetos joao e maria são instâncias da classe Usuario. O fato dos objetos dependerem da sua classe é representada através de dependências UML (setas pontilhadas). Esse relacionamento será explicado em detalhes na Seção 1.3.8.

A linguagem UML foi elaborada de modo que extensões possam ser incorporadas a ela de maneira natural. Para tanto, ela emprega o conceito de **estereótipos**. Através de um estereótipo,

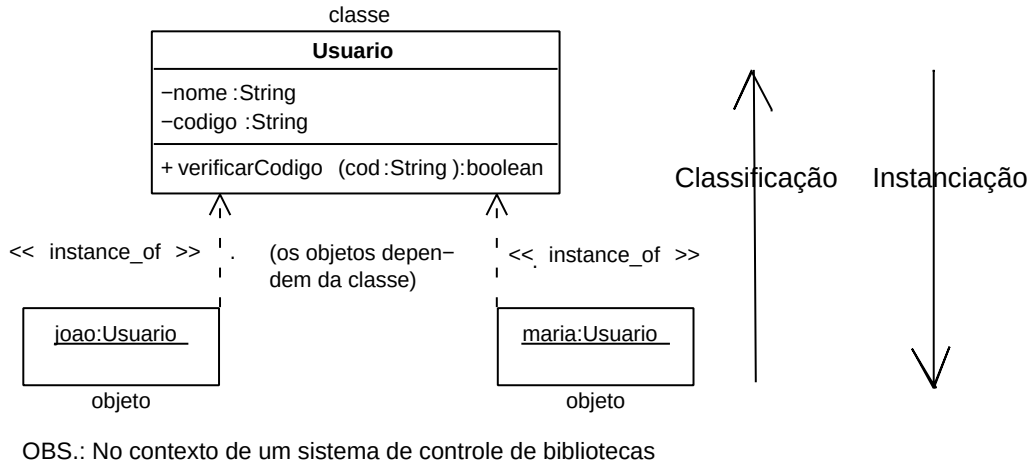


Figura 1.7: Exemplo de Classificação/Instanciação

é possível estender a UML através da definição de uma semântica diferente para um elemento já existente na linguagem (normalmente uma dependência ou uma classe). Para definir um estereótipo, basta colocar o nome do estereótipo entre os sinais '<<' e '>>' acima do nome do elemento cuja semântica se deseja modificar. Por exemplo, graças ao estereótipo $<< instance\_of >>$, o relacionamento de dependência apresentado na Figura 1.7 possui uma semântica de classificação/instanciação.

1.3.4 Abstração de Dados e Hierarquias de Abstrações

Uma das tarefas mais importantes em desenvolvimento de software é a análise do domínio do problema e a modelagem das entidades e fenômenos relevantes para a aplicação (modelagem conceitual). A modelagem conceitual envolve dois aspectos fundamentais: (i) **abstração**, relacionada com os nossos pensamentos na observação do domínio do problema e na definição dos objetos, propriedades e ações relevantes para a solução; e (ii) **representação**, que está relacionada com a notação adotada para expressar o **modelo concreto** (ou "realidade física"). Portanto, o sucesso na criação de um sistema depende da definição de uma representação e de um conjunto de operações que sejam ambas corretas e efetivas [50].

Abstração é definida como um mecanismo através do qual nós observamos o domínio de um problema e focamos nos elementos que são relevantes para uma aplicação específica em um determinado contexto de uso, ignorando todos os pontos irrelevantes. Ela remove certas distinções de tal forma que seja possível ver as coisas comuns entre diversas entidades. Em outras palavras, uma abstração descreve as características essenciais de um grupo de objetos que o distingue de todos os outros tipos existentes. Dessa forma, uma abstração proporciona limites conceituais bem definidos, com relação à perspectiva de um observador em particular.

Na área de Engenharia de Software, **abstração de dados** (ou **tipos abstratos de dados - TAD**) proporciona uma abstração sobre uma estrutura de dados em termos de uma **interface** bem definida. O princípio da abstração de dados surgiu na década de 70 como uma técnica muito importante e efetiva para o controle da complexidade de sistemas de software.

A fim de alcançar esse controle da complexidade, três noções diferentes podem ser abstraídas de uma observação: (i) categorias, grupos ou classes; (ii) ações; e (iii) propriedades. Podemos, então, definir três atividades envolvendo abstração, cada qual com sua operação inversa: classificação/instanciação, generalização/especialização e agregação/decomposição.

O agrupamento de objetos em categorias ou **classes** permite a simplificação do modelo, uma vez que torna possível o tratamento generalizado dos grupos. Dessa forma, esse tratamento se atém às características comuns, abstraindo as particularidades de cada objeto.

Quando um sistema tem muitos detalhes relevantes para serem representados através de apenas uma abstração, essa abstração pode ser decomposta hierarquicamente. Dessa forma, é possível definir abstrações relacionadas, de modo que os detalhes relevantes sejam introduzidos de maneira controlada, com vários níveis de detalhamento disponíveis. As hierarquias de classificação/instanciação foram apresentadas na Seção 1.3.3. As Seções 1.3.5 e 1.3.6 detalham as hierarquias de generalização/especialização e aagregação/decomposição, respectivamente.

1.3.5 Generalização/Especialização

As **hierarquias de generalização**, também conhecidas como hierarquias "é-um-tipo-de" ou hierarquias de herança, definem grupos de categorias. Nessas hierarquias, as características de uma categoria podem ser "herdadas" por outras, possibilitando a existência de categorias genéricas (comuns) ou específicas. Dessa forma, a generalização permite que todas as instâncias de uma categoria específica sejam também consideradas instâncias de uma categoria mais abrangente; mas o inverso não é necessariamente válido. Por ser um conceito intuitivo, a generalização é um mecanismo importante para representar abstrações do mundo real.

Em UML, um relacionamento de generalização é representado por uma linha que se inicia na classe mais específica e termina em um triângulo branco que aponta para a mais abrangente, como apresentado no exemplo da Figura 1.8. Nessa figura, as classes Aluno e Professor são representadas como tipos de Usuarios e por isso podem ser utilizadas em qualquer lugar que a classe Usuario pode ser empregada, uma vez que a operação `utilizar()` foi herdada por ambas.

1.3.6 Agregação/Decomposição

As **hierarquias de agregação**, também conhecidas como hierarquias de "parte-todo", possibilitam representar uma abstração a partir das partes que a compõem. Além disso, o fato de cada parte poder possuir seus próprios componentes, possibilita uma refatoração recursiva.

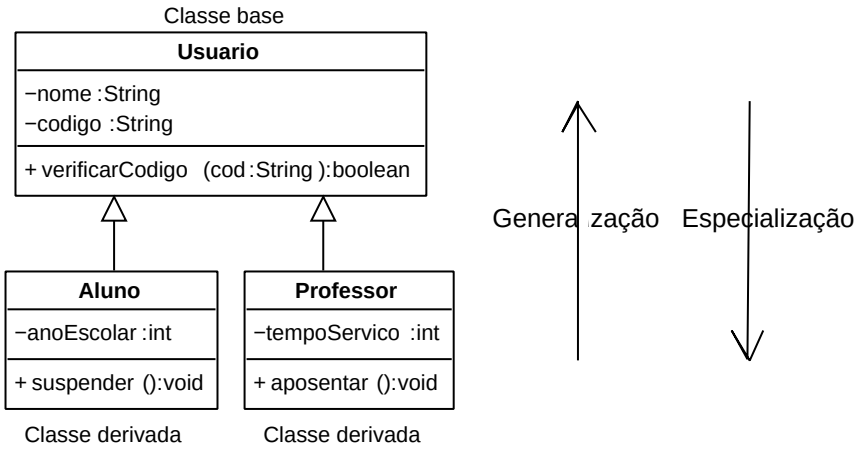


Figura 1.8: Exemplo de Generalização/Especialização

Além da representação "parte-todo", o relacionamento de agregação/decomposição adiciona um valor semântico ao modelo. No exemplo da Figura 1.9, o fato das partes estarem agregadas a Biblioteca também representam que é possível existir instâncias isoladas de Usuario e Publicacao. Para indicar a impossibilidade de existência das partes sem que necessariamente haja o todo, o analista pode utilizar uma variação da agregação, denominada **composição**.

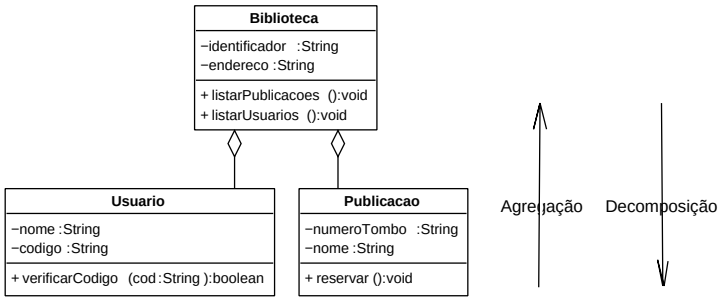


Figura 1.9: Exemplo de Agregação/Decomposição

Agregação e generalização são noções completamente diferentes e complementares. Eles são conceitos ortogonais[7] e quando empregados em conjunto formam uma ferramenta efetiva para a modelagem de sistemas. A principal diferença entre eles é que enquanto nas hierarquias de generalização/especialização, a noção de herança de tipos está implicitamente representada, nas hierarquias de agregação/decomposição isso nem sempre é verdade. Por exemplo, na hierarquia da Figura 1.8, as propriedades gerais de um Usuario, tais como `nome` e `codigo`, são implicitamente herdadas por um Aluno ou um Professor. Em contraste, na hierarquia da Figura 1.9, uma propriedade de Biblioteca, por exemplo o `endereco`, não é automaticamente herdada pelas suas partes.

[7]Duas propriedades são ortogonais quando a retirada, inclusão ou modificação de qualquer uma delas não afeta a outra.

Em UML, um relacionamento de agregação, apresentado na Figura 1.9, é representado por uma linha que se inicia nas partes e termina no todo, em um diamante branco. Para indicar a semântica de composição utiliza-se uma representação semelhante, com a diferença de que o todo é indicado por um diamante negro, ao invés de branco.

1.3.7 Associações

Assim como o relacionamento de agregação/decomposição (Seção 1.3.6), uma **associação** é um relacionamento estrutural entre classes. Porém, ao contrário do primeiro, ela é uma relação conceitual na qual a idéia de parte-todo não está presente. Uma associação representa uma visibilidade permanente entre dois tipos de objetos. Isto é, o relacionamento existe durante toda a vida dos objetos, embora as instâncias que estejam conectadas possam mudar ao longo do tempo. Sendo assim, uma referência a um parâmetro ou à criação de objetos internos aos métodos não devem ser modelados como associações, e sim como uma dependência (Seção 1.3.8). Em UML, agregação e composição são tipos especiais de associação com uma semântica adicional de *"parte/todo"*.

O relacionamento de associação entre duas classes é mostrado na Figura 1.10. Em UML, uma associação é representada por uma linha que liga as duas classes associadas. Essa linha pode ter uma seta no final, indicando que o fluxo de controle da associação flui apenas na direção para onde a seta aponta.

Na Figura 1.10 são apresentados exemplos de associações que envolvem multiplicidades. A multiplicidade de uma associação indica quantas instâncias de cada uma das classes envolvidas poderá estar associada com a outra. No primeiro exemplo, o número '1' e o símbolo '*' nas pontas da associação entre as classes Publicacao e Exemplar indica que nesse modelo, cada instância de Pubicacao pode estar associada a várias instâncias de Exemplar, enquanto que um Exemplar está associado a exatamente uma Publicacao. No segundo exemplo, cada instância de Usuario poderá estar associada a zero ou uma instância de Emprestimo, enquanto o Emprestimo não possui referências sobre os Usuarios (associação unidirecional). A quantidade "zero ou um" é indicada pelo símbolo '0..1'. Outros exemplos de convenções de UML para a especificação de multiplicidades de associações são apresentados do lado direito da figura. Na ausência de uma especificação de multiplicidade, assume-se por padrão que se trata de uma multiplicidade "1".

Para aumentar a expressividade de um diagrama de classes, associações entre duas classes podem ser nomeadas, indicando a maneira como instâncias dessas classes se relacionam. Por exemplo, analisando a Figura 1.10 (b) é possível perceber que Usuario e Emprestimo se associam devido a uma relação de realização.

Na fase de codificação do sistema, cada associação, seja ela simples, agregação ou composição, é materializada no código-fonte das classes de maneira uniforme: através de um atributo do tipo da classe agregada. Dessa forma, apesar de não herdar o comportamento das classes associadas, as suas operações públicas podem ser acessadas normalmente. No caso de relacionamentos unidirecionais, o atributo é adicionado à classe de onde a seta se origina, uma vez que apenas esta classe conhece a outra.

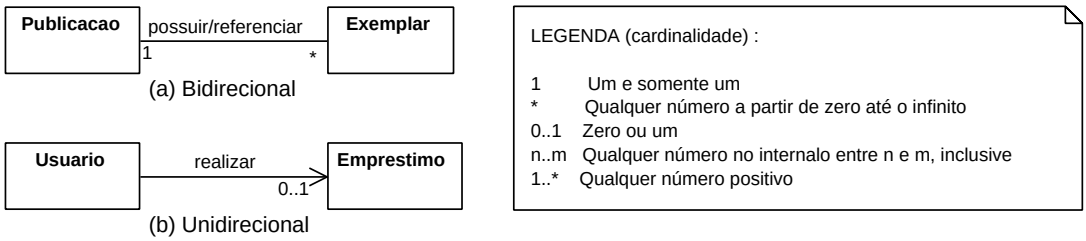


Figura 1.10: Associações em UML

1.3.8 Dependências

O relacionamento de **dependência** é usado quando se quer mostrar que um elemento usa outro. "Usar", neste contexto, pode significar "envia uma mensagem", "cria uma instância", "retornar um objeto do tipo", etc. Referências a parâmetros de um determinado tipo e criação de objetos normalmente são modeladas como dependências. Apesar de ser o tipo de relacionamento mais fraco que pode existir entre dois elementos, esse relacionamento de uso estabelece que uma mudança na especificação do elemento referenciado pode afetar os elementos que o utilizam.

Como pode ser visto na Figura 1.11, em UML uma dependência é representada graficamente por uma linha pontilhada cuja seta aponta para o elemento do qual se depende. No modelo apresentado na Figura 1.11, temos que a classe Usuario depende da Exemplar, isto é, se a classe Exemplar sofrer mudanças isso poderá afetar a classe Usuario. Essa dependência se baseou na seguinte regra de negócio hipotética: "se o exemplar for devolvido, o cliente deve ser informado". O estereóripo $<< use >>$ indica que um Usuario pode utilizar uma referência de Exemplar. O relacionamento de dependência pode ser utilizado entre classes, objetos, pacotes, e até mesmo casos de uso, que serão vistos no Capítulo 2.

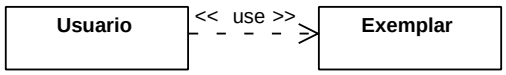


Figura 1.11: Relacionamento de Dependência em UML

1.4 Análise e Projeto Orientado a Objetos

Processos de desenvolvimento de software, independentemente dos métodos empregados, possui basicamente as quatro etapas seguintes: (i) especificação do problema; (ii) entendimento dos requisitos; (iii) planejamento da solução; e (iv) implementação de um programa numa dada linguagem de programação.

Um processo de análise e projeto auxilia na construção de um modelo do domínio de um problema e na adição de detalhes de implementação a esse modelo. A abordagem orientada a

objetos para construção de sistemas permite que um mesmo conjunto de conceitos e uma mesma notação sejam usados através de todas as fases do desenvolvimento do software: análise, projeto e implementação.

Neste capítulo, é útil definirmos alguns conceitos. No contexto deste livro, um **método** é um conjunto de atividades sistemáticas para realizar uma tarefa. Uma **técnica** é um modo de executar as atividades recomendadas por um método e um **processo** é um conjunto de métodos e técnicas com os quais um objetivo pode ser realizado. Um bom processo de análise e projeto deve proporcionar mais do que uma simples notação: ele deve fornecer orientações sobre os passos que serão executados nos diversos estágios de desenvolvimento e deve cobrir todo o ciclo de desenvolvimento de software.

1.4.1 Processos Tradicionais vs. Processos Orientados a Objetos

Em 1970 surgiu um dos primeiros modelos propostos para processos de desenvolvimento de software: o **modelo cascata**, também conhecido como modelo clássico ou abordagem descendente [44]. Com o objetivo principal de estabelecer uma ordem no desenvolvimento de grandes produtos de software, o modelo cascata define etapas seqüenciais de desenvolvimento, especificadas com base nas atividades de desenvolvimento de software praticadas na época.

Como pode ser visto na Figura 1.12, o modelo cascata classifica o desenvolvimento de software em oito etapas (ou fases) estruturadas como uma cascata. Nesse modelo, uma fase só pode ser iniciada após a conclusão da sua antecessora. Os papéis envolvidos em cada fase são definidos de acordo com a natureza das atividades desempenhadas: (i) representação das expectativas dos usuários (especificação de requisitos); (ii) modelagem do sistema (análise e projeto); (iii) implementação em uma linguagem de programação específica (codificação); (iv) execução de testes no ambiente de desenvolvimento (testes de unidade e integração e teste de sistema); (v) execução de testes no ambiente do usuário (teste de aceitação); e (vi) utilização, correção e evolução do sistema (operação e manutenção).

Além da divisão de fases bem definidas, outro motivo para o modelo cascata ter se tornado uma referência para muitos modelos posteriores, é o fato dele ser centrado em documentação. Como pode ser visto na Figura 1.12, a documentação do sistema é atualizada em todas as etapas do desenvolvimento, facilitando inclusive a comunicação entre as fases do modelo.

Apesar do seu impacto para a engenharia de software, o modelo cascata ainda é considerado muito rígido, com pouca interação com o usuário e com pouca ênfase na gerência do desenvolvimento. A rigidez do modelo é uma conseqüência da impossibilidade de voltar para uma fase anterior do desenvolvimento. Dessa forma, o modelo cascata só é adequada quando o sistema não é muito grande, e pertence a um domínio estável e bem conhecido da equipe de desenvolvimento. Por esse motivo, surgiram algumas evoluções do modelo cascata que tentam solucionar algumas das suas deficiências. Alguns exemplos das variações do modelo cascata são: (i) modelo cascata relaxado, que possibilita que o fluxo de desenvolvimento volte para atividades anteriores, tornando o modelo mais próximo da realidade atual; (ii) modelo cascata com prototipação, que propõe o desenvolvimento de

protótipos a serem validados pelos usuários, antes do início do desenvolvimento, e (iii) modelo em V, que ressalta a relação entre as atividades de teste e as de desenvolvimento.

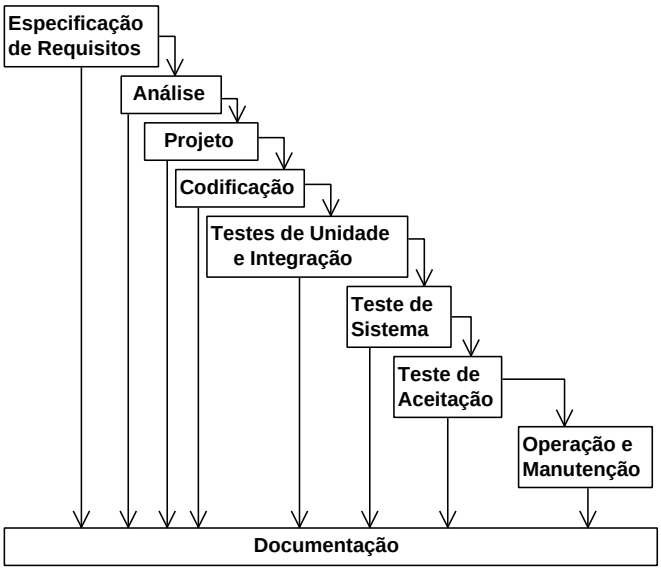


Figura 1.12: Modelo Cascata

Uma evolução mais estrutural do modelo cascata é o **modelo espiral**, originalmente proposto por Boehm em 1988. Além das fases tradicionais do modelo cascata, este modelo adiciona o conceito de análise de risco, realizada constantemente durante o desenvolvimento evolutivo incremental.

Como pode ser visto na Figura 1.13, o modelo espiral é flexível, possibilitando ajustes em qualquer fase do desenvolvimento, inclusive na especificação de requisitos. Essa característica o torna mais adequado ao desenvolvimento de software atual, reduzindo as chances de insucesso no desenvolvimento.

A dimensão radial no espiral da Figura 1.13 representa o custo acumulado do projeto, enquanto a dimensão angular representa o progresso através da espiral. O modelo espiral é composto de quatro grandes fases de desenvolviment, representadas pelos quatro quadrantes da espiral da Figura 1.13. Cada quadrante possui atividades, que são executadas durante o percurso da espiral.

Um ciclo (ou iteração) de desenvolvimento se inicia no primeiro quadrante ("determinação de objetivos, alternativas e restrições"), onde ocorre o comprometimento dos envolvidos e o estabelecimento de uma estratégia para alcançar os objetivos. Em seguida, no segundo quadrante (avaliação de alternativas, identificação e solução de riscos) é feita a análise e avaliação dos risco previstos e dos respectivos planos de contenção. O modelo recomenda a utilização de protótipos para auxiliar a análise dos riscos. O projeto só continua se os riscos forem considerados aceitáveis. No terceiro quadrante do desenvolvimento, ocorre o desenvolvimento propriamente dito. Neste momento do desenvolvimento, o modelo espiral sugere a execução do modelo cascata, visto anteriormente. Finalmente, no último quadrante de cada iteração, o produto construído é avaliado e além disso, a próxima iteração do desenvolvimento é planejada, para iniciar o ciclo seguinte a partir do primeiro

quadrante.

Por ser uma abordagem evolucionária incremental, o desenvolvimento orientado a objetos segue o modelo espiral de desenvolvimento. No decorrer do desenvolvimento orientado a objetos, as fases de análise e projeto não são fases distintas e monolíticas; ao contrário, elas são apenas passos ao longo do caminho do desenvolvimento incremental e iterativo do software.

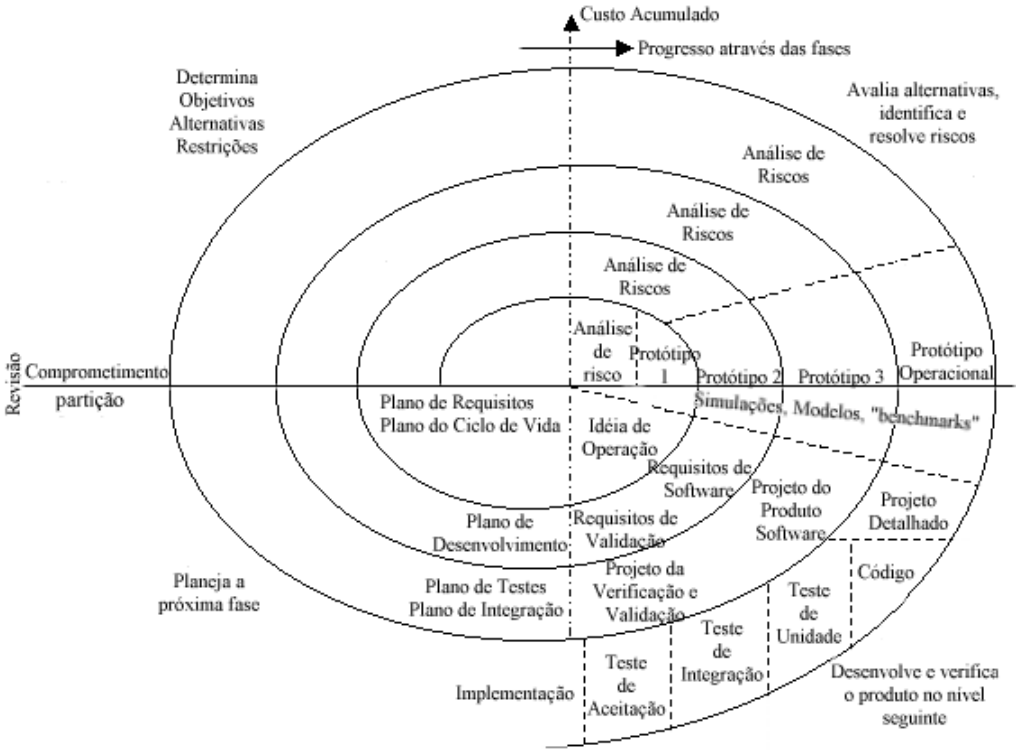


Figura 1.13: Modelo Espiral

Metodologias de projeto orientadas a objetos promovem uma melhoria substancial sobre os modelos tradicionais. A idéia chave da análise e projeto orientados a objetos é o foco em objetos e classes, ao invés de funções (ou procedimentos). O projetista não começa pela identificação das diferentes funcionalidades do sistemas, mas pela identificação dos objetos que o modelam. Uma motivação para essa abordagem é que mudanças na especificação dos requisitos tendem a afetar menos os objetos do que as funções. Conseqüentemente, o emprego de objetos produz sistemas que são menos vulneráveis a mudanças. Além disso, componentes de um software orientado a objetos têm uma probabilidade maior de serem reutilizados, pois encapsulam representação de dados e implementação. Outra vantagem importante do modelo de objetos é o fato dele oferecer uma melhor integração das diferentes fases do desenvolvimento, já que todas elas manipulam as mesmas abstrações: objetos e classes.

1.4.2 Limitações do Modelo de Objetos

Apesar do amadurecimento relativamente alto das linguagens, processos e ferramentas de apoio ao desenvolvimento orientado a objetos, existe muito trabalho ainda a ser feito para o oferecimento de suporte para projetos de grande porte usando processos orientados a objetos. A dificuldade se deve à dificuldade de se gerenciar a complexidade crescente dos sistemas utilizando unidades de abstração pequenas,como as classes. Um grande passo para superar essa limitação é a adoção de representações abstratas do sistema, como por exemplo o conceito de arquitetura de software, que será apresentado do Capítulo 7.

Outra limitação do modelo de objetos é a baixa granularidade de reutilização de software. No desenvolvimento orientado a objetos não é tão simples obter elementos reutilizáveis entre diversas aplicações, especialmente usando-se herança. Existe a necessidade de técnicas mais efetivas para a obtenção de reutilização de software. Com o uso de herança, os objetos tornarem-se muito dependentes da aplicação para a qual foram projetados. Para promover um maior desacoplamento e uma reutilização de código em alta granularidade, cada vez mais está se utilizando o conceito de engenharia de software baseada em componentes, que será apresentado na Seção 1.5 e é complementar ao desenvolvimento orientado a objetos.

1.5 Engenharia de Software Baseada em Componentes

1.5.1 Componentes de Software

Apesar de todas as vantagens atribuídas à programação modular do modelo de objetos (classes e pacotes), os desenvolvedores de software identificaram novas necessidades de aperfeiçoamento da modularização. A partir do conceito de módulo, apresentado na Seção 1.2, foram definidas novas restrições com o intuito de reduzir o acoplamento entre os módulos relacionados e conseqüentemente reduzir as dependências do sistema. Dessa forma surgiu o conceito de componente de software.

A idéia de utilizar o conceito de componentes na produção de software data de 1976 [32]. Apesar desse tempo relativamente longo, o interesse pela engenharia de software baseada em componentes só foi intensificado vinte anos depois, após a realização do Primeiro Workshop Internacional em Programação Orientada a Componentes, o WCOP'96.

Apesar de ser um conceito muito utilizado no universo do *hardware*, não existe um consenso geral sobre a definição do que seja um componente de software. Porém, um aspecto muito importante é sempre ressaltado na literatura: **componente de software** é um módulo que encapsula dentro de si seu projeto e implementação, além de todo acesso ao meio externo (outros componentes) ser sempre mediado através de interfaces pré-definidas. O baixo acoplamento decorrente dessa política proporciona muitas flexibilidades, tais como: (i) facilidade de montagem de um sistema a partir de componentes já existentes (reutilização em larga escala); e (ii) facilidade de substituição de componentes que implementam interfaces equivalentes.

Uma definição complementar de componentes, adotada na maioria dos trabalhos publicados atualmente, foi proposta em 1998 [46]. Segundo Szyperski, um componente de software é uma unidade de composição com interfaces especificadas através de contratos e dependências de contexto explícitas. Sendo assim, além de explicitar as interfaces com os serviços oferecidos (**interfaces providas**), um componente de software deve declarar explicitamente as dependências necessárias para o seu funcionamento, através das **interfaces requeridas**. A Figura 1.14 mostra a representação UML de dois componentes de software conectados. O componente SistemaDeEmprestimos oferece os serviços da interface IEmprestimo e necessita dos serviços da interface ICadastro. Já o componente SistemaDeCadastro oferece os serviços da interface ICadastro e não necessita de nenhum serviço externo. Perceba que os componentes se conectam e o único conhecimento que um tem sobre o outro são os serviços das suas interfaces.

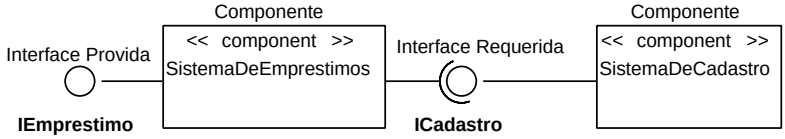


Figura 1.14: Representação de um componente UML

Além dessa distinção clara entre interfaces providas e requeridas, os componentes seguem três princípios fundamentais, que apesar de serem comuns ao modelo de objetos, representam um aumento de granularidade da abstração. São eles [14]:

1. **Unificação de dados e funções:** Um componente deve encapsular o seu estado (dados relevantes) e a implementação das suas operações oferecidas, que acessam esses dados. Assim como acontece em OO, essa ligação estreita entre os dados e as operações ajudam a aumentar a coesão do sistema;

2. **Encapsulamento:** Os clientes que utilizam um componente devem depender somente da sua especificação, nunca da sua implementação. Essa separação de interesse[8] reduz o acoplamento entre os módulos do sistema e melhora a manutenção do mesmo.

3. **Identidade:** Assim como os objetos, cada instância de componente possui um identificador único que o difere das demais.

1.5.2 Reusabilidade, Modularidade e Extensibilidade

Como discutido na Seção 1.1.3, a produtividade é uma preocupação constante da indústria de software. Durante a evolução das tecnologias e processos de desenvolvimento, constatou-se que uma das razões principais para o problema da baixa produção de software é a dificuldade para reutilizar o código gerado pelos modelos tradicionais. O modelo de objetos representa uma evolução nesse sentido, já que promove a modularidade e o encapsulamento; hierarquias de classes e objetos são,

[8] do inglês *separation of concerns*

em geral, componentes portáveis entre aplicações. Se bem projetados, esses componentes podem ser reutilizados em várias aplicações. Porém, como visto na Seção 1.4.2, apesar das suas qualidades, o modelo de objetos sozinho não é suficiente para maximizar a reutilização de software em larga escala. Para isso, ele necessita de evoluções relativas principalmente à granularidade de reutilização e à redução do acoplamento entre os módulos do sistema. Essas limitações podem ser compensadas utilizando-se o conceito de componente de software.

Meyer, o criador da linguagem orientada a objetos Eiffel [33], aponta cinco critérios para a obtenção de modularidade. Esses critérios são gerais e podem ser aplicados a qualquer tipo de modelo, embora seja mais difícil segui-los usando-se modelos tradicionais. Os critérios são os seguintes: (i) **decomponibilidade:** facilidade para particionar o sistema em unidades manejáveis; (ii) **componibilidade:** módulos podem ser livremente combinados em outros sistemas; (iii) **entendimento:** a compreensão de uma parte contribui para o entendimento do todo; (iv) **continuidade:** pequenas mudanças no sistema implicam em pequenas mudanças no comportamento; e (v) **proteção de condições excepcionais:** condições excepcionais ou errôneas são confinadas aos subsistemas nas quais elas ocorrem ou afetam apenas as partes diretamente a elas relacionadas.

Todos esses critérios são mais facilmente alcançados quanto utilizamos paradigmas de desenvolvimento que visam o baixo acoplamento do desenvolvimento, abstraindo-se as particularidades da implementação. Além disso, o aumento de granularidade das unidades de desenvolvimento favorece ainda mais a modularidade e reutilização.

Além do baixo acoplamento que deve haver entre os componentes, existe um outro fator que influencia diretamente o grau de reuso dos elementos de um sistema: o contexto de uso. Em outras palavras, estudos atuais mostram que um componente de software tem muito mais chance de ser reutilizado quando conhecemos os domínios específicos onde ele pode ser utilizado.

1.5.3 Desenvolvimento Baseado em Componentes

O Desenvolvimento Baseado em Componentes (DBC) pode ser visto como um paradigma de desenvolvimento de software onde a unidade básica é o componente. A alta granularidade de desenvolvimento do DBC facilita a construção de sistemas complexos, além de viabilizar a reutilização sistemática em larga escala. Sempre pensando em componentes, o DBC possui duas vertentes complementares [44]: (i) **desenvolvimento para reuso**, que consiste no desenvolvimento de componentes elementares que podem ser utilizados em sistemas maiores; e (ii) **desenvolvimento com reuso**, que consiste na composição de sistemas, a partir de componentes já existentes.

Apesar de ser um novo paradigma de programação, o DBC é na verdade complementar ao desenvolvimento orientado a objetos. Pela inexistência de linguagens orientadas a componentes e pela orientação a objetos proporcionar uma representação compatível com a realidade, a maioria dos autores recomenda o desenvolvimento para reuso seja realizado com linguagens orientadas a objetos. A única restrição recomendada é que essas linguagens possuam o conceito de módulo (por exemplo, pacotes) e interface, como por exemplo Java e C#.

A popularização do DBC está sendo motivada principalmente pelas pressões sofridas na

indústria de software por prazos mais curtos e produtos de maior qualidade. O aumento de produtividade é decorrente da reutilização de componentes existentes na construção de novos sistemas. Já o aumento da qualidade é uma conseqüência do fato dos componentes utilizados já terem sido empregados e testados em outros contextos. Porém, vale a pena ressaltar que apesar desses testes prévios serem benéficos, a reutilização de componentes não dispensa a execução dos testes no novo contexto onde o componente está sendo reutilizado.

1.6 Resumo

As dificuldades enfrentadas na época da crise do software evidenciaram a necessidade de se utilizar técnicas sistemáticas de estruturação dos sistemas. Com o objetivo de facilitar a construção de sistemas complexos, as linguagens de programação foram incorporando novos conceitos, tais como instruções de alto nível, módulos, objetos e classes. A adição desses conceitos proporcionou tanto um aumento da granularidade do desenvolvimento, quanto o encapsulamento crescente dos dados e funções relacionados entre si.

Com o advento da Programação Orientada a Objetos, conceitos como objetos, classes e hierarquias de abstração ganharam evidência. Além de encapsular seus dados e suas funções, um objeto se comunica com outro através da troca de mensagens, que consiste na execução de operações do objeto alvo. Todo objeto é instância de exatamente uma classe. Sendo assim, uma classe é a descrição de um molde que especifica as propriedades e o comportamento relativo a um conjunto de objetos similares. Classes podem ser estruturadas de forma hierárquica, através do conceito de hierarquias de abstração.

As duas principais hierarquias de abstração existentes no paradigma de objetos são: generalização/especialização e agregação/decomposição. As hierarquias de generalização/especialização representam a idéia de herança de tipo, isto é, a subclasse "é-um-tipo-da" superclasse. Já as hierarquias de agregação/decomposição são utilizadas para decompor a classe através das partes que a compõe. Essas hierarquias são complementares e juntas, oferecem o arcabolso básico para modelar sistemas orientados a objetos.

Assim como as linguagens de programação, os processos de desenvolvimento de software também evoluiram. O modelo de desenvolvimento tradicional, conhecido como modelo cascata, representa um marco no desenvolvimento de software, estruturando-o em fases bem delimitadas, que deveriam ser executadas sequencialmente. O caráter interativo do desenvolvimento de software motivou várias evoluções do modelo cascata e uma delas é o modelo espiral. Além de representar o processo de desenvolvimento de maneira iterativa, esse modelo adicionou uma fase importante para a construção de software: a análise de risco. Nesta abordagem o software entregue aos poucos, o que aumenta a modularidade do sistema construído. O desenvolvimento de software orientado a objetos adota o modelo espiral.

Apesar de todas as vantagens da orientação a objetos, ele apresenta algumas limitações que incentivam a pesquisa por novos paradigmas e tecnologias complementares. O aumento do tamanho e complexidade do software, por exemplo, desperta a necessidade de se ter unidades de desen-

volvimento com granularidade superior às classes. Além disso, a exigência po tempo e costo de desenvolvimento cada vez menores evidencia a necessidade de técnicas de reutilização em grande escala. Seguindo essa direção, o conceito de desenvolvimento baseado em componentes vem se apresentando como uma tendência para suprir as deficiências do paradigma de objetos. Vale a pena reforçar o caráter complementar das duas tecnologias, uma vez que cada componente de software pode ser incarado como um sistema orientado a objetos.

1.7 Exercícios

1. Explique e contraste cada um dos seguintes pares de termos: (i) instância e classe; (ii) comportamento e estado; (iii) agregação e herança; (iv) herança e delegação.

2. Dê um exemplo de duas hierarquias: uma de generalização/especialização e outra de agregação/decomposição. Descreva as diferenças e similaridades entre as duas hierarquias.

3. Construa uma hierarquia que represente os diversos tipos de contas disponíveis para os clientes do seu banco. Lembre-se de representar a classe mais genérica, denominada `ContaBancaria`.

4. Defina atributos e operações para o tipo abstrato de dados `CarteiraDeDinheiro`. Pense a respeito do comportamento de uma carteira e sobre os atributos relacionados com esse comportamento.

5. Defina uma classe representando uma pessoa com os seguintes atributos: nome, ano de nascimento e altura (em metros). Defina operações para a iniciação desses atributos. Adicione uma operação que retorna a idade aproximada de uma pessoa de acordo com um determinado ano de entrada. Adicione outra operação que retorna sua altura em centímetros.

6. Defina uma classe para um tipo abstrato de dados `ItemDeEstoque`. Ela deve conter os atributos do nível de estoque e preço unitário. Defina métodos para consultar os valores desses atributos e também para iniciá-los usando parâmetros. Adicione mais dois métodos para permitir a atualização do nível de estoque de forma apropriada (isto é, para baixa e reposição de estoque).

7. Suponha que você esteja desenvolvendo uma classe que represente um baralho de cartas. Quais operações você deveria oferecer na interface pública da classe? Faria sentido você modelar essa abstração usando duas classes separadas: uma para modelar o baralho e outra para representar as cartas? Por quê?

8. Para os diagramas de classe da Figura 1.15, discuta os "erros" de modelagem cometidos (caso eles existam) e proponha soluções corretivas.

9. Crie uma hierarquia de generalização/especialização que modele os alimentos encontrados nas prateleiras de um supermercado. Identifique atributos associados a cada uma das classes.

10. Crie um modelo de objetos que represente um carro que possa ser colocado e retirado de uma garagem. Quantas abstrações você acha que são necessárias para que o modelo represente

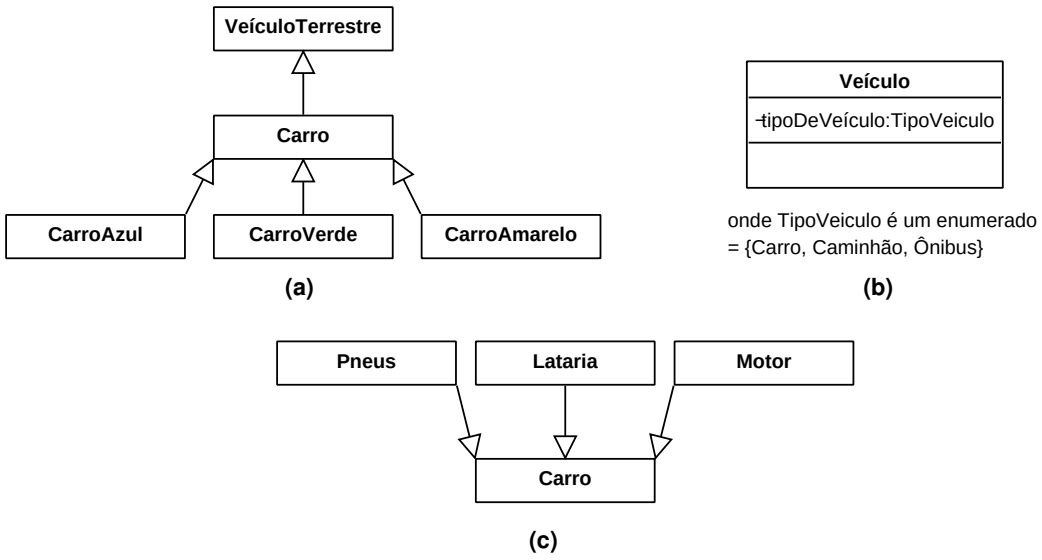


Figura 1.15: Diagramas de classes com Possíveis Erros de Modelagem

o mais fielmente possível a realidade? Quais operações você poderia oferecer na interface pública das classes usadas?

11. Defina uma hierarquia de classes que represente uma coleção de números. Uma coleção é um grupo de objetos. Um conjunto é um tipo de coleção onde não existem duplicatas e uma lista é uma coleção onde seus elementos estão ordenados. Identifique o comportamento de cada tipo abstrato de dados envolvido na sua modelagem.

12. Modele uma hierarquia de classes parcial para um sistema de controle de reservas e ocupações de um hotel. O hotel é constituído por um conjunto de quartos (simples e duplos), auditórios para conferências e salões de festas. O sistema controla a ocupação e desocupação dos seus cômodos, bem como, permite que o usuário verifique o estado atual de cada cômodo, isto é, se ele está ocupado ou desocupado.

13. Classifique e crie uma hierarquia de generalização/especialização contendo os diferentes tipos de itens encontrados na biblioteca de uma universidade. Sugestão: livros, periódicos, teses, fitas de vídeo, DVD etc. Identifique os atributos e as operações de cada tipo abstrato de dados criado.

Capítulo 2

Modelagem de Casos de Uso

2.1 Introdução

No contexto do software, a fase de especificação de requisitos tem o objetivo de representar a idéia do usuário a respeito do sistema que será desenvolvido. Essa representação é feita através do documento de requisitos, que deve compreender a necessidade real dos usuários e suas expectativas, dimensionar a abrangência do sistema e delimitar o seu escopo.

Em geral, os requisitos podem ser vistos como condições ou capacidades necessárias para resolver um problema ou alcançar um objetivo. Para a elicitação dessas capacidades, os processos normalmente estruturam a fase de especificação de requisitos nas quatro etapas seguintes: (i) identificação do domínio do problema; (ii) identificação das funcionalidades esperadas pelo sistema (objetivos); (iii) definição de restrições existentes para o desenvolvimento; (iv) identificação dos atributos de qualidade desejados.

As funcionalidades especificadas para o sistema e os seus atributos de qualidade recebem um tratamento diferenciado no documento de especificação de requisitos. Por essa razão, os requisitos são classificados em dois tipos [44]: (i) **requisitos funcionais**, composto pelas funcionalidades especificadas; e (ii) **requisitos não funcionais**, que materializam os atributos de qualidade do sistema. Apesar de não representarem funcionalidades diretamente, os requisitos não-funcionais podem interferir na maneira como o sistema deve executá-las.

De acordo com a norma ABNT/ISO 9126, os requisitos não-funcionais de um software podem ser classificados em seis grupos, de acordo com a sua característica principal: (i) **completude** (do inglês *functionality*), que quantifica a grau de satisfação das necessidades do cliente; (ii) **confiabilidade**, que identifica a capacidade do software em manter a sua integridade após a ocorrência de falhas não controladas; (iii) **usabilidade**, que identifica a facilidade de se compreender o funcionamento e operação do software; (iv) **eficiência**, que identifica a capacidade do software em

desempenhar as suas atividades de forma adequada em relação ao tempo e aos recursos alocados; (v) **manutenibilidade**, que identifica a capacidade do software em sofrer modificações; e (vi) **portabilidade**, que identifica a capacidade de adaptação do software quando transferido para outros ambientes e/ou plataformas.

Em UML, a notação utilizada para a representação dos requisitos do sistema o diagrama de casos de uso. Pelo fato da UML ser uma linguagem unificada, os processos que a utilizam evidenciam a importância dos casos de uso. Esses artefatos, que representam o objetivo do sistema, é a base para todo o processo de desenvolvimento, sendo o ponto de ligação entre o cliente e toda a equipe de desenvolvimento.

A modelagem de casos de uso é uma técnica que auxilia o entendimento dos requisitos de um sistema computacional através da criação de descrições narrativas dos processos de negócio, além de delimitar o contexto do sistema computacional [29]. Um **caso de uso** é uma narrativa que descreve uma seqüência de eventos de um agente externo (ator) usando o sistema para realizar uma tarefa. Casos de uso são utilizados para descrever os requisitos funcionais, que indicam o que o sistema deverá ser capaz de fazer, sem entrar em detalhes sobre como essas funcionalidades serão materializadas. Como pode ser visto na Figura 2.1, o modelo de casos de uso é um dos principais artefatos oriundos da fase de especificação de requisitos e serve de entrada para a fase de análise. Casos de uso bem estruturados denotam somente o comportamento essencial do sistema ou subsistema e não podem ser nem muito gerais, a ponto de prejudicar o seu entendimento, nem muito específicos, a ponto de detalhar como as tarefas serão implementadas.

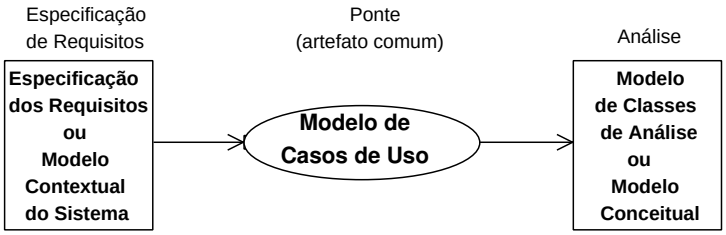


Figura 2.1: Casos de Uso na Especificação dos Requisitos

Além de capturar o conhecimento de clientes e futuros usuários do sistema sobre seus requisitos, casos de uso também são úteis para definir cronogramas e auxiliam na elaboração dos casos de teste do sistema.

A fim de fixar os conceitos apresentados neste capítulo, será utilizado o exemplo de um sistema de controle de bibliotecas, descrito a seguir.

Enunciado: Sistema de Controle de Bibliotecas

Um sistema de controle de bibliotecas é um sistema computacional usado para controlar o empréstimo e a devolução de exemplares de uma biblioteca. O usuário pode fazer um empréstimo de um exemplar durante um certo período e, ao final desse tempo, o exemplar deve ser devolvido, ou o empréstimo deve ser renovado.

O atendente é um funcionário que interage com os usuários e com o sistema de controle da biblioteca através de um terminal. As principais características do sistema são listadas a seguir:

1. Um usuário do sistema, que pode ser um aluno, um professor ou um outro funcionário da universidade, pode reservar publicações e também cancelar reservas previamente agendadas.
2. Um usuário deve estar devidamente cadastrado no sistema para usar os seus serviços. O sistema é operado pelo atendente da biblioteca, que também é um funcionário da universidade.
3. Um usuário pode emprestar exemplares previamente reservados ou não. Se foi feita uma reserva, ela deve ser cancelada no momento do seu empréstimo.
4. No caso da devolução de um exemplar em atraso, existe uma multa que deve ser paga. Essa multa é calculada com base no número de dias em atraso. Além disso, se o exemplar estiver atrasado por mais de 30 dias e se o usuário não for um professor, além de pagar a multa, o usuário é suspenso por um período de 2 meses.
5. Um exemplar da biblioteca pode ser bloqueado/desbloqueado por um professor por um período de tempo. Nesse caso, o exemplar fica disponível numa estante, podendo ser consultado por usuários da biblioteca, mas não pode ser emprestado.
6. O período de empréstimo é variável, dependendo do tipo de usuário (aluno, funcionário ou professor).
7. A manutenção dos dados do acervo da biblioteca é feita pela bibliotecária, que também é funcionária da universidade. Essa funcionária é responsável pela inclusão de novos exemplares, exclusão de exemplares antigos e pela atualização dos dados dos exemplares cadastrados.

Os exemplares podem ser livros, periódicos, manuais e teses. As publicações são identificadas pelo seu número do tombo, além de outras características como o título, a editora e o número da edição correspondente.

A biblioteca só empresta suas obras para usuários cadastrados. Um usuário é identificado através de seu número de registro. Outras informações relevantes são seu nome, instituto/faculdade a que pertence e seu tipo (aluno/funcionário/professor).

2.2 Casos de Usos

Um **caso de uso** é uma descrição de um processo de negócio relativamente longo com um começo, meio e fim. Ele representa as principais funcionalidades que o sistema deve oferecer de maneira observável por seus interessados externos, isto é, os seus **atores**. Os casos de uso, na verdade, representam funções no nível do sistema e podem ser utilizados para visualizar, especificar, construir e documentar o comportamento esperado do sistema durante o levantamento e análise de seus requisitos. A definição formal de caso de uso, segundo Larman [29], é: “um conjunto de seqüências

de ações que um sistema desempenha para produzir um resultado observável, de valor para um ator específico".

Por se preocupar unicamente com o problema e utilizar conceitos do domínio, a utilização de casos de uso também facilita a comunicação entre os usuários finais, os especialistas no domínio e os desenvolvedores do sistema.

Um exemplo de caso de uso do sistema de bibliotecas seria o Emprestar Exemplar. A Figura 2.2 apresenta a representação gráfica desse caso de uso, de acordo com a linguagem UML. O caso de uso Emprestar Exemplar pode ser descrito da seguinte forma:

> **Caso de Uso**: Emprestar Exemplar
> **Atores**: Usuário, Atendente, Sistema de Cadastro.
> **Descrição**: Este caso de uso representa o processo de empréstimo de um ou vários exemplares da biblioteca. O empréstimo se inicia com a solicitação feita pelo usuario ao atendente. Em seguida, através de um terminal, o atendente solicita ao sistema o empréstimo dos respectivos exemplares.

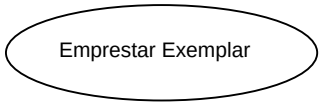


Figura 2.2: Caso de Uso Emprestar Exemplar em UML

2.3 Atores e Papéis

Um **ator** é uma entidade externa ao sistema computacional que participa de um ou mais casos de uso. Essa participação normalmente é baseada na geração de eventos de entrada ou no recebimento de alguma resposta do sistema. De uma maneira geral, atores representam entidades interessadas, que podem interagir diretamente com o sistema.

A representação dos atores se dá a partir do papel que eles representam, como por exemplo, usuario, atendente, aluno, etc. Em geral, atores podem ser: (i) papéis que pessoas representam nos casos de uso, (ii) dispositivos de hardware mecânicos ou elétricos, ou até mesmo (iii) outros sistemas computacionais.

A interação entre um ator e o sistema acontece através de troca de mensagens e em UML, é representada por um relacionamento de associação (uma linha contínua, como visto na Seção 1.3.7). Em relação aos atores, A Figura 2.3 apresenta as duas maneiras de representá-los: (i) na forma de um boneco; ou (ii) como uma classe com o estereótipo $<< actor >>$.

No exemplo do sistema de bibliotecas, podemos identificar inicialmente dois atores: o atendente

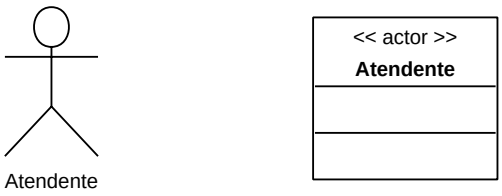


Figura 2.3: Representação de Atores em UML

e o usuário. Na Figura 2.4 podem ser vistas as associações que indicam a participação desses atores no caso de uso Emprestar Exemplar.

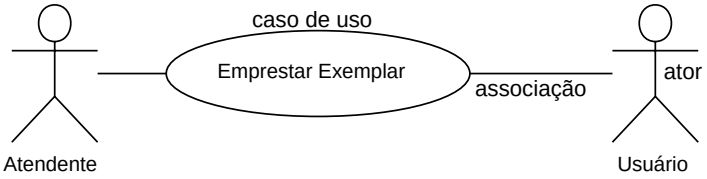


Figura 2.4: Caso de Uso Emprestar Exemplar com seus atores

Como visto anteriormente, os atores são usados para representar qualquer entidade externa com a qual o sistema interage. Dessa forma, a interferência do tempo em um sistema, por exemplo, pode ser representada através de um ator chamado tempo, associado aos seus respectivos casos de uso. De modo particular no sistema de bilbiotecas, o cancento de uma reserva após um período caracteriza casos de uso associados ao ator tempo. Outra solução é não representar o tempo explicitamente, o considerando como um componente interno do sistema. Nesse caso, o caso de uso inicia a si próprio numa determinada hora e por isso é associado apenas aos demais atores interessados no seu funcionamento. Por exemplo, os sistemas e interessados que recebem a saída gerada por ele.

2.4 Fluxo de Eventos

O **fluxo de eventos** de um caso de uso é uma seqüência de comandos declarativos que descreve as etapas de sua execução, podendo conter desvios de caminhos e iterações. Esse fluxo deve especificar o comportamento de um caso de uso de uma maneira suficientemente clara para que alguém de fora possa compreendê-lo facilmente. Em outras palavras, ele deve permanecer focado no domínio do problema e não em sua solução. Além disso, esse fluxo deverá incluir como e quando o caso de uso inicia e termina, quais são os atores interessados no sistema e quando e como o caso de uso interage com eles.

O fluxo de eventos de um caso de uso é composto por: um (i) fluxo básico e (ii) zero ou mais fluxos alternativos. Enquanto o **fluxo básico** descreve a seqüência de passos mais esperada para a execução da funcionalidade do caso de uso, os **fluxos alternativos** descrevem desvios do fluxo básico. Esses fluxos podem ser especificados de diversas maneiras, incluindo descrição textual

informal, texto semi-formal (através de assertivas lógicas), pseudocódigo, ou ainda uma combinação dessas maneiras.

A seguir, é mostrado o fluxo de eventos do caso de uso Emprestar Exemplar. O fluxo básico é descrito através de pseudocódigo, enquanto os fluxos alternativos são descritos informalmente, por meio de descrição textual. Para facilitar a leitura do caso de uso, será adotada a seguinte convenção: cada frase que descreve uma ação deve iniciar com o nome "sistema", ou o nome do ator que a executa. Em seguida, deve-se indicar o verbo da ação e finalmente o complemento da frase. Quando apropriado, o complemento da frase deve referenciar o paciente da ação. Sendo assim, o formato sugerido para cada frase é o seguinte:
$< sistema/ator\_agente > + < verbo > + < complemento\_frase > + < sistema/ator\_paciente >$.

Fluxo Básico de Eventos:

1. O usuário solicita empréstimo de um ou mais exemplares de publicações (livro, periódico, tese ou manual), fornecendo o seu código e os exemplares desejados;
2. O atendente solicita o empréstimo ao sistema, fornecendo o código do usuário;
3. Para cada exemplar a emprestar:
   - 3.1 O atendente fornece o número de registro do exemplar.
   - 3.2 O sistema valida o usuário e verifica o seu *status* ("Normal" ou "Suspenso") através de seu número de registro.
   - 3.3 O sistema verifica se o exemplar pode ser emprestado pelo usuário em questão;
   - 3.4 Se o *status* do usuário for "Normal" e o exemplar estiver disponível:
     - 3.4.1. O sistema verifica se a publicação do exemplar está reservada. Se estiver reservada:
       - A. O sistema cancela a reserva, passando o número de tombo da publicação
     - 3.4.2. O sistema calcula o período do empréstimo, que depende do tipo de usuário - 7 dias para alunos e 15 para professores
     - 3.4.3. O sistema registra o empréstimo do exemplar;
     - 3.4.4. O sistema atualiza seu banco de dados com a informação de que o exemplar não irá se encontrar na biblioteca até completar o período.

Fluxos Alternativos

Fluxo Alternativo 1:

No Passo 3.4, se o usuário estiver suspenso, este é informado de sua proibição de retirar exemplares e o empréstimo não é realizado.

Fluxo Alternativo 2:

No Passo 3.4, se todas as cópias da publicação estiverem emprestadas ou reservadas, o sistema informa ao atendente que não será possível realizar o empréstimo daquele exemplar. Se tiver outros exemplares para emprestar, vá para o Passo 3.1 do fluxo básico.

Como pôde ser visto neste exemplo, os fluxos alternativos são executados quando condições especiais pré-definidas são detectadas durante a execução do fluxo básico. Neste caso, a execução do fluxo básico é interrompida e o fluxo alternativo correspondente à condição é executado. No exemplo acima, se o usuário não estiver suspenso e o exemplar não estiver disponível, (Passo 3.4 do fluxo básico), o segundo fluxo alternativo é executado. Se for bem sucedido, ao final deste, a execução é retomada a partir do Passo 3.1 do fluxo básico (próximo exemplar). Também é possível que um fluxo alternativo leve ao encerramento precoce do fluxo de eventos, como é o caso do primeiro fluxo alternativo do exemplo acima.

2.5 Cenários

Um **cenário** é uma seqüência de comandos/ações simples, que representa uma execução específica do fluxo de eventos, com todas as decisões e iterações já conhecidas de antemão. Um cenário representa uma interação ou execução de uma instância de um caso de uso. O fluxo de eventos de um caso de uso produz um **cenário primário**, que representa a funcionalidade básica do caso de uso, e zero ou mais **cenários secundários**, que descrevem desvios do cenário primário.

O cenário primário de um caso de uso é escrito supondo que tudo dá certo e ilustra uma situação típica de sucesso. Normalmente o cenário primário de um caso de uso corresponde à execução dos passos de seu fluxo básico quando nenhum desvio é tomado. Cenários secundários representam situações menos comuns, incluindo aquelas em que um erro ocorre. Cenários secundários que representam situações de erro também são conhecidos como **cenários excepcionais**.

Tipicamente, para cada cenário derivado de um fluxo de eventos, deve ser possível responder as quatro questões seguintes: (i) como o cenário começa? (ii) o que causa o término do cenário? (iii) quais respostas são produzidas pelo cenário? (iv) existem desvios condicionais no fluxo de eventos?

Seguindo o nosso exemplo, o cenário primário do caso de uso Emprestar Exemplar pode ser descrito da seguinte forma:

1. O usuário José chega à biblioteca para tomar emprestado um exemplar do livro *São Bernardo*. José apresenta o exemplar e informa o seu código pessoal, que é "A55";
2. O atendente solicita o empréstimo, fornecendo o código "A55" para José;
3. O atendente olha o exemplar entregue e fornece o número do registro ao sistema;
4. O sistema verifica que José está com *status* "Normal";
5. O sistema verifica que o exemplar está disponível para locação;
6. O sistema calcula que o exemplar pode ficar emprestado a José por 7 dias;
7. O sistema registra o empréstimo;
8. O sistema atualiza o registro, finalizando o empréstimo;

9. O atendente entrega o exemplar a José;
10. José vai embora.

Cenários secundários são descritos de maneira similar, mas levando em consideração desvios do cenário primário. Por exemplo, no Passo 3 do cenário primário, o código do produto pode ser inválido e, nesse caso, o sistema deve indicar um erro. De forma análoga, no Passo 5 o exemplar pode não estar disponível e então a transação de empréstimo do exemplar em questão deve ser cancelada. O primeiro cenário alternativo poderia ser descrito da seguinte forma:

1. O usuário José chega à biblioteca para tomar emprestado um exemplar do livro *São Bernardo*. José apresenta o exemplar e informa o seu código pessoal, que é "A55";
2. O atendente solicita o empréstimo, fornecendo o código "A55" para José;
3. O atendente olha o exemplar entregue e fornece o número do registro ao sistema;
4. O sistema verifica que José está com *status* "Suspenso";
5. O sistema avisa ao atendente que José não pode realizar o empréstimo e cancela a operação;
6. O atendente informa a José;
7. José vai embora.

2.6 Formato e Convenções para Casos de Uso

Formato de Caso de Uso

Além do fluxo de eventos, a descrição de um caso de uso pode incluir informações que aumentem a rastreabilidade dos requisitos e facilitem a comunicação entre clientes e a equipe de desenvolvimento do sistema [15]. Não existem padrões na indústria ou na literatura sobre quais informações devem ser incluídas na descrição de um caso de uso; autores normalmente preferem indicar uma lista de ítens que podem ser relevantes e sugerem que sejam escolhidos de acordo com a utilidade de cada um para um determinado projeto de desenvolvimento [18]. O livro de Cockburn [15] apresenta diversos formatos com diferentes informações e estilos de descrição.

Neste livro adotaremos o seguinte formato para a descrição de casos de uso, baseado no RUP [26]:

1. **Nome do Caso de Uso**
   (a) **Breve Descrição**
   ...texto...
   (b) **Atores**
   ...texto...

   (c) **Pré-condições**:
   ...texto...
   (d) **Pós-condições**:
   ...texto...
   (e) **Requisitos Especiais (requisitos não-funcionais)**:
   ...texto...
   (f) **Referência Cruzada (requisitos mapeados)**:
   ...lista de requisitos...
2. **Fluxo de Eventos**
   (a) **Fluxo Básico**
   ...passos do cenário...
   (b) **Fluxo Alternativo 1**
   ...passos do cenário...
   (c) **Fluxo Alternativo 2**
   ...passos do cenário...
   ...
   (d) **Fluxo Alternativo N**
   ...passos do cenário...

No modelo acima, um caso de uso é identificado por seu nome. Este é seguido por (i) uma descrição sucinta, similar às que foram apresentadas na Seção 2.2; (ii) a lista dos atores que participam desse caso de uso; (iii) um conjunto de **pré-condições**, predicados que devem ser verdadeiros para que o cenário seja executado; (iv) um conjunto de **pós-condições**, predicados que devem ser verdadeiros ao final da execução do cenário; (v) uma lista de requisitos especiais, requisitos não-funcionais que o sistema deve apresentar durante a execução do caso de uso, como desempenho e confiabilidade (Seção 2.1); e (vi) uma referência ao requisito (ou conjunto de requisitos) satisfeito(s) pelo caso de uso. A especificação dos fluxos de eventos segue o formato apresentado na Seção 2.4.

2.7 Diagramas de Casos de Uso

Um **diagrama de casos de uso** é uma representação gráfica que mostra um conjunto de casos de uso, atores e seus relacionamentos para um determinado sistema. Ele é um diagrama que contextualiza o sistema no ambiente no qual ele se insere e possui quatro elementos básicos: (i) atores; (ii) casos de uso; (iii) interações; e (iv) fronteira do sistema. A Figura 2.5 apresenta o diagrama de casos de uso para o sistema da biblioteca. Esse diagrama inclui o caso de uso Emprestar Exemplar apresentado na Figura 2.4, além de três casos de uso novos: Devolver Exemplar, Reservar Publicação e Cancelar Reserva.

Além do seu papel central para a modelagem dos requisitos funcionais de um sistema, os diagramas de casos de uso também são muito utilizados para facilitar a compreensão de sistemas legados, por meio de engenharia reversa [9].

Como pode ser visto na Figura 2.5, a fronteira do sistema é representada graficamente através de uma linha ao redor dos casos de uso. Essa delimitação explícita do contexto representa a forma como o sistema interage com as entidades externas associadas a ele, que são representadas pelos atores.

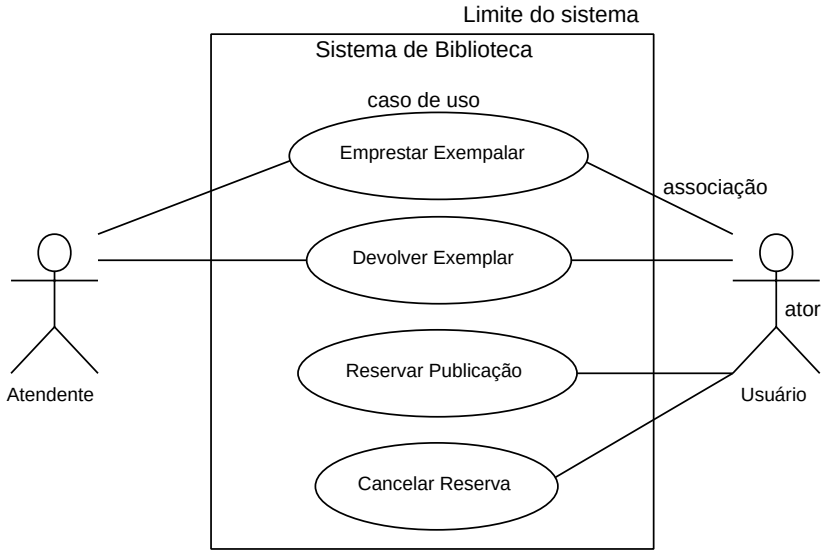


Figura 2.5: Diagrama de Casos de Uso para o Sistema de Biblioteca

2.8 Relacionamentos entre Casos de Uso

Além das associações entre os casos de uso e seus atores, em um diagrama de casos de uso é possível definir outros relacionamentos, tanto entre os casos de uso, quanto entre os atores. A linguagem UML define três tipos de relacionamentos, que aumentam o valor semântico do diagrama. São eles: (i) generalização (herança); (ii) inclusão ($<< include >>$); e (iii) extensão ($<< extend >>$). Esses relacionamentos são utilizados com a finalidade principal de fatorar tanto o comportamento comum entre os casos de uso, quanto as variações desses comportamentos.

A seguir, as Seções 2.8.1, 2.8.2 e 2.8.3 apresentam cada um desses relacionamentos em maiores detalhes.

2.8.1 Generalização

O relacionamento de generalização entre casos de uso é similar à generalização entre classes, vista na Seção 1.3.5, isto é, o caso de uso derivado herda tanto o significado do caso de uso base, quanto o seu comportamento, que normalmente é estendido e especializado. Por exemplo, na Figura 2.6 (a), o caso de uso Emprestar Exemplar é responsável por efetuar o empréstimo de um exemplar. Podemos

definir dois casos de uso derivados, Emprestar sem Renovação e Renovar Empréstimo, de tal forma que ambos se comportem igual ao caso de uso Emprestar Exemplar, exceto pelo fato de explicitarem o tipo específico de empréstimo que é feito. Os dois casos de uso derivados podem então ser usados em qualquer lugar onde o caso de uso Emprestar Exemplar é esperado.

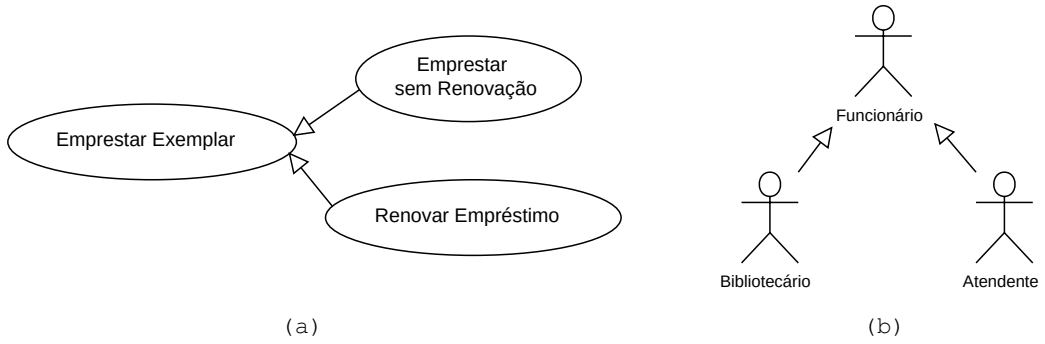


Figura 2.6: Exemplo de Generalização no Diagrama de Casos de Uso

Num diagrama de casos de uso, o relacionamento de generalização também pode ser usado entre atores, significando que um ator desempenha os mesmos papéis de um outro ator, podendo ainda desempenhar papéis extras. A Figura 2.6 (b) mostra um exemplo de generalização entre atores, onde os atores Bibliotecário e Atendente são derivados de um ator base chamado Funcionário. Isso significa que todos os papéis desempenhados pelo ator Funcionário podem ser desempenhados por qualquer ator derivado, seja ele um Bibliotecário ou um Atendente.

2.8.2 Inclusão

Um relacionamento de inclusão (<< *include* >>) entre casos de uso significa que o caso de uso base sempre incorpora explicitamente o comportamento de outro caso de uso em um ponto específico. Sendo assim, mesmo que o caso de uso incluído não esteja associado a nenhum ator, ele é executado como parte do caso de uso que o inclui. Costuma-se usar um relacionamento de inclusão para evitar a descrição de um mesmo conjunto de fluxo de eventos, através da fatoração do comportamento comum.

Um relacionamento de inclusão é representado em UML como uma dependência (ver Seção 1.3.8) cuja seta aponta para o caso de uso incluído, isto é, o caso de uso base "depende" do caso de uso incluído, significando que se o caso de uso incluído for modificado, o caso de uso base também deve ser revisto. Essa dependência recebe o estereótipo << *include* >>, como mostrado na Figura 2.7. No exemplo da Figura 2.7, o caso de uso Emprestar Exemplar explicita o fato da validação do usuário fazer parte da sua lógica de negócio. Dessa forma, sempre que o caso de uso Emprestar Exemplar é executado, o caso de uso Validar Usuário é executado.

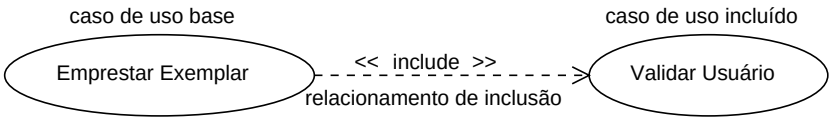


Figura 2.7: Exemplo de Inclusão entre Casos de Uso

2.8.3 Extensão

Um relacionamento de extensão (<< *extend* >>) entre casos de uso significa que o caso de uso base incorpora implicitamente o comportamento de outro caso de uso num ponto específico. No relacionamento de extensão, diferentemente do que ocorre no relacionamento de inclusão, o caso de uso extensor só tem seu comportamento incorporado pelo caso de uso base em algumas circunstâncias específicas. As condições necessárias para que o caso de uso base possa ser estendido devem ser especificadas explicitamente através de pontos pré-estabelecidos na especificação. Esses pontos são chamados de **pontos de extensão**.

Um relacionamento de extensão é representado em UML como uma dependência cuja seta aponta para o caso de uso base (não para o caso de uso extensor), isto é, o caso de uso extensor "depende" do caso de uso base. Essa dependência possui o estereótipo << *extend* >>, como mostrado no exemplo da Figura 2.8.

O relacionamento de extensão é utilizado para modelar a parte do caso de uso que o usuário considera como sendo um comportamento condicional do sistema. Desta forma, é possível, por exemplo, separar o comportamento opcional do comportamento obrigatório, ou até mesmo expressar um subfluxo separado que é executado apenas em circunstâncias específicas. A Figura 2.8 apresenta o caso de uso Emprestar Exemplar, que é estendido pelo caso de uso Cancelar Reserva.

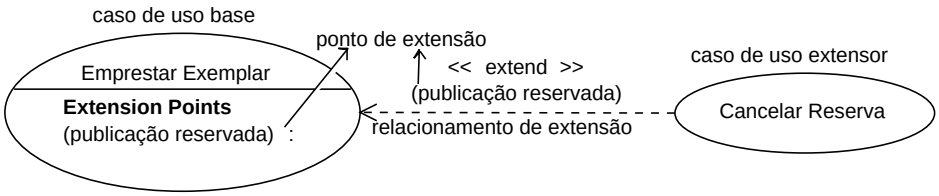


Figura 2.8: Exemplo de Extensão entre Casos de Uso

Agora que já conhecemos como casos de uso podem se relacionar entre si, o fluxo de eventos do caso de uso Emprestar Exemplar, que já foi definido na Seção 2.4, poderia explicitar o local onde outros casos de uso são executados. Para isso, serão utilizados os relacionamentos de << *include* >> e << *extend* >> A versão refinada desse fluxo básico é apresentada a seguir:

Fluxo Básico de Eventos:

1. O usuário solicita empréstimo de um ou mais exemplares de publicações (livro, periódico, tese ou manual), fornecendo o seu código e os exemplares desejados;

2. O atendente solicita o empréstimo ao sistema, fornecendo o código do usuário;
3. Para cada exemplar a emprestar:
   3.1 O atendente fornece o número de registro do exemplar.
   3.2 O sistema valida o usuário e verifica o seu *status* ("Normal" ou "Suspenso") através de seu número de registro. (<< *include* >> Validar Usuário);
   3.3 O sistema verifica se o exemplar pode ser emprestado pelo usuário em questão;
   3.4 Se o *status* do usuário for "Normal" e o exemplar estiver disponível:
      3.4.1. O sistema verifica se a publicação do exemplar está reservada. Se estiver reservada (publicação reservada):
         A. O sistema cancela a reserva, passando o número de tombo da publicação (<< *extend* >> Cancelar Reserva)
      3.4.2. O sistema calcula o período do empréstimo, que depende do tipo de usuário - 7 dias para alunos e 15 para professores (<< *include* >> Calcular Tempo de Empréstimo)
      3.4.3. O sistema registra o empréstimo do exemplar;
      3.4.4. O sistema atualiza seu banco de dados com a informação de que o exemplar não irá se encontrar na biblioteca até completar o período.

Neste exemplo, `(publicação reservada)` é um ponto de extensão. Um caso de uso pode ter vários pontos de extensão, que podem inclusive aparecer mais do que uma vez. Em condições normais, o caso de uso Emprestar Exemplar é executado independentemente da satisfação dessas condições. Pporém, se o usuário solicitar o empréstimo de um exemplar que está reservado por ele, então o ponto de extensão `(publicação reservada)` é ativado e o comportamento do caso de uso Cancelar Reserva é então executado. Em seguida, o fluxo de controle continua do lugar de onde foi interrompido (Passo 3.4.2).

2.9 Método para a Modelagem de Casos de Uso

Quando a modelagem de casos de uso é utilizada para melhorar o entendimento dos requisitos funcionais do sistema computacional, é importante que haja a participação ativa dos clientes/usuários do sistema. O usuário entende como o sistema será usado e esse conhecimento é capturado na forma de casos de uso. Esses casos de uso são carregados através das fases de análise, projeto, implementação, testes e manutenção do sistema, servindo como guia e base para o seu entendimento. A Figura 2.9 mostra como os casos de uso são realizados através das fases de análise e projeto. O modelo conceitual da análise refina os casos de uso e as classes do modelo de projeto, por sua vez, refinam o modelo conceitual da análise. Em geral, durante essas transformações o número de classes aumenta.

O estereótipo << *trace* >> entre a fase de análise e a fase de especificação de requisitos, mostrado na Figura 2.9, indica qual conjunto de elementos da análise corresponde à especificação do caso de uso Emprestar Exemplar. Similarmente para o estereótipo << *trace* >> entre as fases

de análise e projeto. Além de possibilitar o rastreamento entre os artefatos das fases do desenvolvimento do software, o relacionamento de dependência traz informações importantes que auxiliam o gerenciamento da evolução do sistema. Como visto na Seção 1.3.8, o fato do modelo de análise depender do modelo de casos de uso, implica que se o caso de uso Emprestar Exemplar for alterado, o modelo de análise correspondente deve ser revisto. A mesma semântica também é válida entre os modelos de projeto e de análise.

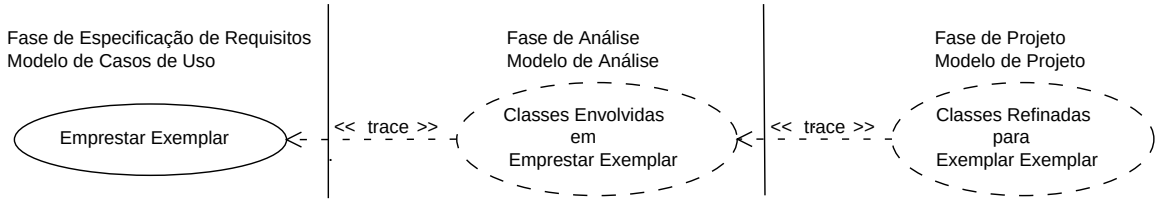


Figura 2.9: Realizações dos Casos de Uso em Modelos Diferentes

Nesta seção são apresentados três métodos para a construção do modelo de casos de uso, partindo de uma especificação inicial informal dos requisitos do sistema, chegando até a descrição gráfica de seus fluxos de eventos. Apesar de suas características particulares, os métodos expostos aqui são complementares e podem ser utilizados em conjunto, a fim de maximizar a eficácia da elicitação e representação dos requisitos. As abordagens mostradas são:

1. **Identificação de Casos de Uso Baseada em Atores.** Essa técnica se baseia na identificação das funcionalidades requeridas por cada um dos interessados no sistema. O ponto forte dessa abordagem é o seu caráter intuitivo, que motiva a sua popularização na literatura.

2. **Identificação de Casos de Uso Baseada em Atributos.** Essa abordagem enfatiza a necessidade de analisar as informações relevantes para cada entidade conceitual do sistema, isto é, os atributos identificados nas descrições textuais. A partir daí, são identificadas as funcionalidades relativas ao gerenciamento e atualização dessas informações, tais como cadastro e manutenção de dados.

3. **Identificação de Casos de Uso Baseada em Análise de Domínio.** O objetivo principal dessa abordagem é a identificação dos requisitos inerentes ao domínio do sistema. Por serem normalmente comuns a outros sistemas do mesmo domínio, esses requisitos representam as funcionalidades mais propícias à reutilização.

Cada um desses métodos é composto por uma seqüência de passos que estruturam a modelagem dos requisitos de um sistema através de casos de uso. Além da utilização desses métodos, é bom ter sempre em mente as seguintes dicas que orientam a especificação de casos de uso:

1. Um caso de uso não diz nada sobre o funcionamento interno do sistema, isto é, o sistema é visto como uma caixa preta;
2. Casos de uso são parte do domínio do problema e não da solução;

3. Um caso de uso diz como atores interagem com o sistema e como o sistema responde;
4. Um caso de uso é sempre iniciado ou por um ator, ou por outro caso de uso do qual faça parte;
5. Um caso de uso oferece um resultado observável, sob o ponto de vista do ator;
6. Um caso de uso é completo, isto é, ele possui um começo, um meio e um fim;
7. O fim de um caso de uso é indicado quando o seu resultado observável é obtido pelo ator;
8. Podem ocorrer várias interações entre os atores e os casos de uso, durante a execução dos fluxos de eventos.

O sistema de controle de bibliotecas, descrito no início do capítulo, terá seus casos de uso identificados e especificados em maiores detalhes. Para isso, serão utilizadas as três abordagens de modelagem de casos de uso apresentados anteriormente.

2.9.1 Identificação de Casos de Uso Baseada em Atores

Nesse método, o primeiro passo para a identificação dos casos de uso é identificar os atores que irão interagir com o sistema. Como visto na Seção 2.3, esses atores podem ser pessoas ou outros sistemas externos com os quais o sistema especificado interage. Para ajudar na tarefa de descoberta desses atores, existem algumas perguntas a serem respondidas [42]. A seguir, são apresentadas as sete principais questões, respondidas de acordo com o sistema da biblioteca.

1. Quem opera o sistema?

   Resp.: O sistema pode ser operado pelo atendente ou pelo bibliotecário.
2. Quem é responsável pela sua administração?

   Resp.: A administração do sistema fica por conta do bibliotecário.
3. Quem é responsável pela manutenção dos seus dados?

   Resp.: A manutenção dos dados é feita pelo atendente (dados de usuários) e pelo bibliotecário (dados de usuários e do acervo).
4. Quem necessita das suas informações?

   Resp.: As informações são úteis para o usuário (alunos, professores e funcionários), para o atendente e para o bibliotecário.
5. Quem oferece informações para o sistema?

   Resp.: As informações podem ser oferecidas pelo usuário (informações pessoais), pelo atendente (informações pessoais) e pelo bibliotecário (informações sobre o acervo).
6. Os outros sistemas utilizam algum dado/processamento do sistema especificado?

Resp.: O sistema contábil necessita de informações sobre o valor de todas as multas pagas pelos usuários.

7. Acontece algo automaticamente/periodicamente no sistema?

   Resp.: Sim. Uma reserva pode ser cancelada automaticamente, caso o empréstimo não tenha sido efetuado no período estipulado.

Os atores identificados com as respostas dadas foram os seguintes: (i) atendente; (ii) usuário; (iii) aluno; (iv) professor; (v) funcionário; (vi) bibliotecário; (vii) sistema contábil; e (viii) tempo.

Numa biblioteca pequena, é possível que vários desses papéis sejam desempenhados pela mesma pessoa. Em cada papel, esta pessoa age de forma diferente e também espera respostas diferentes do sistema. Dado que os atores já foram descobertos, é possível iniciar a especificação dos casos de uso, a partir do que cada um dos atores espera do sistema. Para isso, para cada um dos atores identificados, devemos considerar seis pontos importantes, que serão apresentados a seguir. As respostas elaboradas se referem ao ator usuário do exemplo da biblioteca.

1. Quais tarefas o ator deseja que o sistema realize?

   Resp.: O usuário deseja "emprestar um exemplar", "devolver um exemplar", "reservar um exemplar", e "cancelar reserva".

2. Quais informações o ator deve fornecer para o sistema?

   Resp.: O usuário pode fornecer as seguintes informações: nome, endereço, título da publicação, número de tombo da publicação, número de registro do exemplar.

3. Existem eventos que o ator deve comunicar ao sistema?

   Resp.: O usuário pode comunicar uma possível mudança de endereço.

4. O ator precisa ser informado de alguma coisa importante pelo sistema?

   Resp.: O usuário deve ser informado quando uma publicação reservada por ele possuir algum exemplar disponível para ser emprestado.

5. O ator é responsável por iniciar ou terminar a execução do sistema?

   Resp.: Sim, nas funcionalidades: 'consultar uma publicação", "reservar um exemplar", e "cancelar reserva".

6. O sistema armazena informações? O ator necessita manipulá-las, isto é, ler, atualizar ou apagar?

   Resp.: Sim. O usuário pode desejar saber os últimos exemplares alugados por ele, ou ainda atualizar seus dados pessoais.

Com base nessas respostas, podemos identificar os seguintes casos de usos: (i) Manter Dados Usuário. O usuário deve ser cadastrado na biblioteca, fornecendo informações sobre seu nome, endereço, e tipo de vínculo com a universidade; (ii) Emprestar Exemplar. O usuário pode emprestar exemplares da biblioteca; (iii) Devolver Exemplar. O usuário pode devolver exemplares emprestados a ele; (iv) Reservar Publicação. O usuário pode reservar um número de exemplares de uma

determinada publicação; (v) Cancelar Reserva. O usuário pode cancelar uma reserva feita por ele; (vi) Contactar Usuário. O usuário pode ser contactado quando uma publicação reservada por ele estiver disponível; e (vii) Consultar Histórico Usuário. O usuário pode consultar os últimos exemplares emprestados a ele.

Considerando o atendente, o bibliotecário e o professor, podemos identificar outros casos de uso: (i) Manter Dados Publicação. O bibliotecário pode gerenciar os dados das publicações do sistema; (ii) Manter Dados Exemplar. O bibliotecário pode gerenciar os dados dos exemplares de cada pubicação; (iii) Consultar Histórico Biblioteca. Antes de comprar novos exemplares, uma informação útil ao bibliotecário é saber quais publicações são mais populares; (iv) Bloquear Exemplar. Um professor pode bloquear o empréstimo de exemplares específicos; e (v) Desbloquear Exemplar. Os exemplares bloqueados para empréstimo podem voltar a ser emprestados (desbloqueados);

2.9.2 Identificação de Casos de Uso Baseada em Atributos

Uma outra forma de identificar os casos de uso é considerar os possíveis atributos das entidades do sistema alvo. Por exemplo, uma publicação pode ter associados a ela: um título, o gênero da publicação (técnica, não-técnica), a data de lançamento, etc. Pensando nesses atributos, é possível identificar casos de uso relacionados com funcionalidades de consulta e manutenção. Por exemplo: (i) Consultar Publicação., que representa a consulta da disponibilidade dos exemplares de uma determinada publicação.

De forma análoga, analisando a entidade usuário seria possível identificar o caso de uso Consultar Usuário, uma vez que o atendente pode encontrar o registro de um determinado usuário, informando um dos seus dados pessoais.

2.9.3 Identificação de Casos de Uso Baseada em Análise de Domínio

Analisando o domínio do problema, é possível identificar entidades e funcionalidades mais propícias a serem reutilizadas posteriormente.

A seguir, para cada uma das quatro etapas da análise de domínio, relatamos o produto final obtido para o sistema da biblioteca.

1. **Estudo da viabilidade do domínio.** Esta etapa consistiu na identificação e seleção dos domínio relacionados para o sistema em questão. Foi identificada uma grande semelhança entre sistemas de bibliotecas e sistemas de locação. Além disso, também julgou-se que o domínio do sistema analisado possui características de sistemas de livraria. Baseado nesses domínios, constituiu-se o domínio dos "sistemas de empréstimo de publicações", que agrega as características dos domínios de: (i) sistemas de locação; e de (ii) sistemas de livraria.

2. **Planejamento do domínio.** Em relação à análise de risco relativa ao nosso exemplo, não

foram identificados riscos sérios de projeto. Essa decisão se baseou no fato de se tratar de um domínio amplamente conhecido e sem a necessidade de se utilizar tecnologias imaturas.

3. **Contextualização do domínio.** Nesta etapa foi avaliada a contextualização do domínio do ponto de vista do sistema em particular. Com o intuito de acompanhar a tendência especificada no domínio do sistema, é desejável existir um módulo de consulta e reserva pela Internet.

4. **Aquisição do conhecimento do domínio.** Esta fase é responsável pela elicitação final e representação das informações e requisitos relacionados ao domínio. Os artefatos produzidos nesta etapa são descrições textuais semelhantes aos fluxos de eventos dos casos de uso, apresentados na Seção 2.4. Por esse motivo, essa atividade deve ser desempenhada normalmente em conjunto, tanto pelo especialista no domínio, quanto pelo engenheiro de requisitos e os artefatos produzidos aqui servirão de base para a especificação dos casos de uso do sistema.

   No contexto do nosso exemplo, foram identificadas algumas características inerentes aos sistemas de bibliotecas. O conhecimento dessas características foi decorrente da análise dos domínios identificados anteriormente. As principais características encontradas são enumeradas a seguir:

   (a) Uma biblioteca normalmente disponibiliza vários ítens distintos para serem emprestados, não apenas material impresso;

   (b) No caso da devolução ser atrasada, normalmente é cobrada uma multa proporcional ao tempo de atraso (Cobrar Multa) e o usuário pode ser suspenso temporária ou permanentemente ou (Suspender Usuário e Cancelar Suspensão);

   (c) Durante o cadastro de usuários, pode ser conveniente consultar instituições de proteção ao crédito (ator Sistema de Crédito);

Após identificar os casos de uso do sistema utilizando os métodos apresentados anteriormente, o próximo passo é detalhar as suas especificações. Sendo assim, as atividades e conceitos mostrados a partir da Seção 2.9.4 se baseiam apenas nas particularidades do negócio e nos casos de uso identificados, independentemente da abordagem (ou combinação de abordagens) utilizada para tal.

2.9.4 Construção de um Glossário e Termos

Um glossário contém a definição de todos os conceitos utilizados na especificação e modelagem do sistema, que possam comprometer o seu entendimento [29]. Sendo assim, a definição de um glossário busca tanto definir termos desconhecidos, quanto esclarecer conceitos aparentemente similares. O principal benefício desse esclarecimento da terminologia adotada é a melhoria da comunicação, reduzindo os riscos de desentendimento entre os interessados do sistema.

Um entendimento consistente dos termos utilizados no domínio do desenvolvimento é extremamente importante durante as fases de desenvolvimento [29]. Apesar de ser criado originalmente na

fase de elicitação de requisitos e planejamento do projeto, esse glossário deve ser refinado continuamente em cada ciclo iterativo. Dessa forma, no decorrer do desenvolvimento, novos termos podem ser adicionados e as definições anteriores podem ser atualizadas.

Uma simplificação do modelo de definição de glossários proposto por Larman [29] é mostrado na Tabela 2.1. O campo `termo` representa o conceito cujo significado é definido no campo `comentário`. Além dessas informações, esse modelo possibilita a especificação de `informações adicionais` referentes ao campo. Essas informações adicionais podem ser utilizadas, por exemplo, para contextualizar o termo descrito em relação à fase do desenvolvimento que ele foi identificado.

Tabela 2.1: Modelo Sugerido para a definição do Glossário

Termo	Comentário	Inf. Adicionais
Entidade a ser definida	Explicação descritiva	Informações adicionais

No contexto do exemplo da biblioteca, é interessante que seja estabelecida uma distinção entre alguns termos, a fim de evitar ambigüidade nas descrições dos casos de uso. Esse glossário é mostrado na Tabela 2.2, que define os termos (Publicação e Exemplar).

Tabela 2.2: Glossário dos Termos da Biblioteca

Termo	Comentário	Inf. Adicionais
Publicação	nome coletivo para todos os exemplares de uma determinada publicação. Essa abstração de tipo é utilizada para realizar as operações de consulta e reserva.	(i) fase de requisitos; (ii) exigência do cliente.
Exemplar	cópia individual de uma publicação que pode ser emprestada pelo usuário. Essa é a abstração de tipo que representa o objeto físico.	(i) fase de requisitos; (ii) exigência do cliente.

2.9.5 Levantamento Inicial dos Casos de Uso

Uma vez que os casos de uso fundamentais foram identificados, podemos refiná-los descrevendo-os com maiores detalhes. Por exemplo, o caso de uso Devolver Exemplar envolve as seguintes questões que devem ser considerados: (i) a devolução está atrasada? (ii) o exemplar está danificado?

Todas essas questões sugerem uma revisão dos casos de uso identificados até agora, gerando a seguinte lista parcial de casos de uso:

[**Caso #1**] Reservar Publicação. O usuário pode reservar um número de exemplares de uma determinada publicação.

[**Caso #2**] Cancelar Reserva. O usuário pode cancelar uma reserva feita por ele.

[**Caso #3**] Contactar Usuário. O usuário pode ser contactado quando uma publicação reservada por ele estiver disponível.

[**Caso #4**] Emprestar Exemplar. O usuário pode emprestar exemplares da biblioteca.

[**Caso #5**] Devolver Exemplar. O usuário pode devolver exemplares emprestados a ele.

[**Caso #6**] Cobrar Multa. Multa cobrada nos casos onde há devoluções em atraso.

[**Caso #7**] Bloquear Exemplar. Um professor pode bloquear o empréstimo de exemplares específicos.

[**Caso #8**] Desbloquear Exemplar. Os exemplares bloqueados para empréstimo podem voltar a ser emprestados (desbloqueados).

[**Caso #9**] Suspender Usuário. Usuários podem ser suspensos para locação.

[**Caso #10**] Cancelar Suspensão. Usuários suspensos provisoriamente podem voltar a realizar empréstimos.

[**Caso #11**] Manter Dados Usuário. O usuário deve ser cadastrado na biblioteca, fornecendo informações sobre seu nome, endereço, e tipo de vínculo com a universidade.

[**Caso #12**] Manter Dados Publicação. O bibliotecário pode gerenciar os dados das publicações do sistema.

[**Caso #13**] Manter Dados Exemplar. O bibliotecário pode gerenciar os dados dos exemplares de cada pubicação.

[**Caso #14**] Consultar Usuário. O atendente pode encontrar o registro de um determinado usuário informando um dos seus dados pessoais.

[**Caso #15**] Consultar Publicação. O usuário pode consultar a disponibilidade dos exemplares de uma determinada publicação.

[**Caso #16**] Consultar Histórico Usuário. O usuário pode consultar os últimos exemplares emprestados a ele.

[**Caso #17**] Consultar Histórico Biblioteca. Antes de comprar novos exemplares, uma informação util ao bibliotecário é saber quais publicações são mais populares.

Refinamento de Casos de Usos Relacionados

Uma vez que uma lista de casos de uso tenha sido obtida, você pode unir e refinar aqueles que são similares e definir várias versões, cenários e variantes para cada um deles. Por exemplo, os casos de uso #14 e #15 são parte dos casos de uso #11 e #12, respectivamente. As diversas atividades de manutenção: consulta, cadastro, alteração e exclusão podem ser descritas como fluxos do mesmo caso de uso. Outra possibilidade é definí-los como extensões do caso de uso mais geral, utilizando

o relacionamento de $<< extend >>$, como mostrado na Figura 2.10 para o caso de uso Manter Dados Publicação. Apesar das várias possibilidades de especificação de um diagrama de casos de uso, existe uma relação de compromisso (do inglês *tradeoff*) entre a complexidade dos casos de uso e o número excessivo de casos de uso em um sistema.

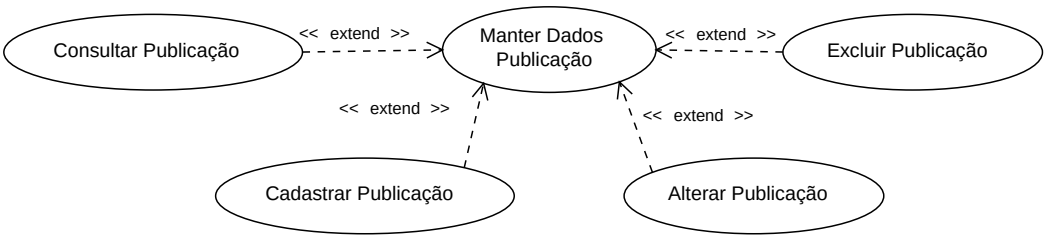


Figura 2.10: Modelagem Alternativa do Caso de Uso Manter Dados Publicação

2.9.6 Descrição de Casos de Usos

Cada caso de uso da lista preliminar deve receber um nome único e, em seguida, deve ser atribuído a ele um comentário expandido que proporcione um entendimento mais detalhado. Por exemplo, o caso de uso #5 (Devolver Exemplar) pode expandir a sua descrição inicial, de modo a envolver os atores interessados na sua execução (usuário e atendente). Uma descrição mais completa do caso de uso Devolver Exemplar é apresentada a seguir:

> **Caso de Uso:** Devolver Exemplar.
> **Atores:** Usuário e Atendente.
> **Descrição**: Este caso de uso representa o processo de devolução de empréstimo de um ou vários exemplares da biblioteca. A devolução se inicia com a solicitação feita pelo usuario ao atendente. Em seguida, através de um terminal, o atendente solicita ao sistema a devolução dos respectivos exemplares. Se a devolução estiver em atraso e o usuário não for um professor, é cobrada uma multa e o usuário pode ser suspenso.

2.9.7 Gerenciamento de Casos de Uso Complexos

Para promover o reuso e tornar mais claras as especificações dos casos de uso do sistema, é útil usar os relacionamentos entre casos de uso descritos na Seção 2.7. Por exemplo, o caso de uso Cancelar Reserva, identificado na Seção 2.9.5, descreve em detalhes como um usuário cancela uma reserva feita para uma determinada publicação. Por outro lado, o caso de uso Emprestar Exemplar pode envolver o cancelamento de uma reserva como parte do processo de empréstimo. Neste caso, ao invés de copiarmos o caso de uso Cancelar Reserva para o caso de uso Emprestar Exemplar, podemos dizer que Cancelar Reserva estende Emprestar Exemplar através do relacionamento $<< extend >>$.

É possível usar também o relacionamento de inclusão, visto na Seção 2.8.3. Por exemplo, sempre

que o empréstimo é realizado ou quando um exemplar é devolvido ou bloqueado, o usuário deve ser validado para verificar a sua possibilidade de executar o serviço solicitado (caso de uso Validar Usuário). Com o intuito de explicitar a reutilização desde o modelo de casos de uso, pode-se dizer que o caso de uso Validar Usuário é incluído pelos casos de uso Emprestar Exemplar, Devolver Exemplar e Bloquear Exemplar.

Pacotes

Para controlar a complexidade do modelo, à medida que o número de casos de usos cresce, se faz necessário utilizar o conceito de módulos. Conforme descrito na Seção 1.2.2, a modularização é implementada em UML através do conceito de pacotes. Um **pacote** agrupa um conjunto de entidades UML relacionadas. Vários critérios diferentes podem ser utilizados na hora de definir como os casos de uso serão empacotados. Por exemplo, você pode agrupar casos de uso que interagem com o mesmo ator, ou aqueles que estabelecem relacionamentos de inclusão, extensão ou generalização (casos de uso mais acoplados). Como exemplo de uma abordagem mista, a Figura 2.11 apresenta a visão modularizada dos casos de uso do sistema de bibliotecas. Enquanto os Pacotes 1 e 2 agrupam os casos de uso baseado nos atores, os Pacotes 3 e 4 agrupam pela semelhança das suas funcionalidades. A Figura 2.11 também representam uma hierarquia de casos de uso, agrupando os casos de uso principais juntos e colocando os menos importantes em pacotes secundários (sub-pacotes). Dessa forma, o pacote de nível mais alto, chamado de *top-level*, representa o modelo completo do sistema.

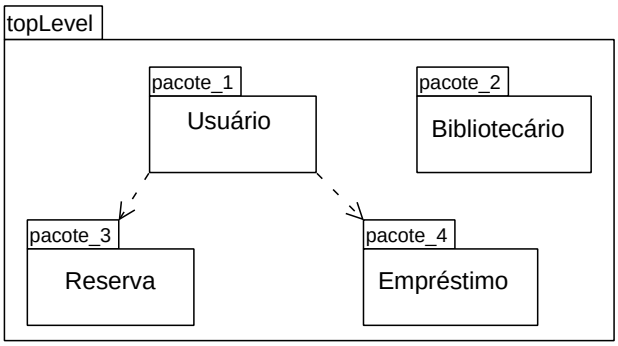


Figura 2.11: Pacotes de Casos de Uso

2.9.8 Descrições Formais de Casos de Usos

A definição de assertivas é uma maneira muito utilizada para a formalização das especificações dos casos de uso. Esse princípio se baseia na técnica de projeto por contrato(do inglês *design by contract*), definido por Bertrand Meyer [34]. Sua filosofia principal é a definição de restrições que devem ser satisfeitas antes (pré-condições) ou após (pós-condições) a execução da funcionalidade especificada. As pré- e pós-condições são adicionadas à especificação dos casos de uso, conforme o

modelo mostrado na Seção 2.6.

Uma **pré-condição** de um caso de uso descreve restrições no sistema antes de um caso de uso iniciar. Por exemplo, antes do caso de uso Devolver Exemplar começar, o usuário deve estar cadastrado na biblioteca e deve ter emprestado o exemplar que está sendo devolvido. Além disso, a biblioteca deve estar aberta. Um caso de uso só tem o compromisso de executar corretamente nos cenários onde as suas pré-condições são satisfeitas. Cada um desses cenários podem ter uma versão mais detalhada das suas pré-condições.

As **pós-condições** de um caso de uso descrevem o estado do sistema e possivelmente dos atores, depois que o caso de uso foi completado. Elas são verdadeiras para todos os cenários do caso de uso, embora cada cenário possa descrever suas próprias pós-condições mais detalhadamente. Por exemplo, ao final do caso de uso Devolver Exemplar, a situação do usuário deve estar regularizada. Além disso, uma multa pode ou não ter sido cobrada, dependendo do tipo de usuário e das condições da devolução.

2.9.9 Diagrama de Casos de Uso do Sistema da Biblioteca

Utilizando os conceitos apresentados na Seção 2.7, a Figura 2.12 mostra o diagrama de casos de uso do sistema de gerenciamento de bibliotecas. Lembrando que dependendo da decisão do analista, poderiam ser produzidas outras versões corretas do diagrama.

2.9.10 Diagrama de Atividades para Fluxo de Eventos

Diagramas de atividades possuem elementos adicionais que tornam possível a representação de desvios condicionais e execução concorrente de atividades. Dessa forma, todos os fluxos de eventos de um caso de uso (básico e alternativos) podem ser representados graficamente através de um único diagrama. Um diagrama de atividades é bastante similar a uma máquina de estados na qual os estados correspondem a atividades e as transições são, em sua maioria, vazias.

A Figura 2.13 apresenta um diagrama de atividades contendo os principais elementos que podem ser encontrados em um diagrama de atividades. O fluxo de eventos descrito pelo diagrama começa no **estado inicial**, indicado pelo círculo preto na parte superior da figura, e termina no **estado final**, indicado pelos dois círculos concêntricos, o menor preto e o maior branco, na parte inferior da figura 2.13.

Um **desvio** ou **ramificação** (do inglês *decision*) é uma estrutura utilizada para especificar caminhos alternativos de execução. Como pode ser visto na Figura 2.13, ele é representado por um losango que possui uma transição de entrada e várias saídas distintas. A seleção da saída correta se dá a partir da resolução de expressões booleanas, definidas em cada uma das saídas. Um desvio é similar a um comando condicional de uma linguagem de programação. Na Figura 2.13, o único desvio existente escolhe a Atividade B se a condição [cond1] for verdadeira e a Atividade C se a condição [cond2] for verdadeira (relembrando: as condições devem ser disjuntas). Uma

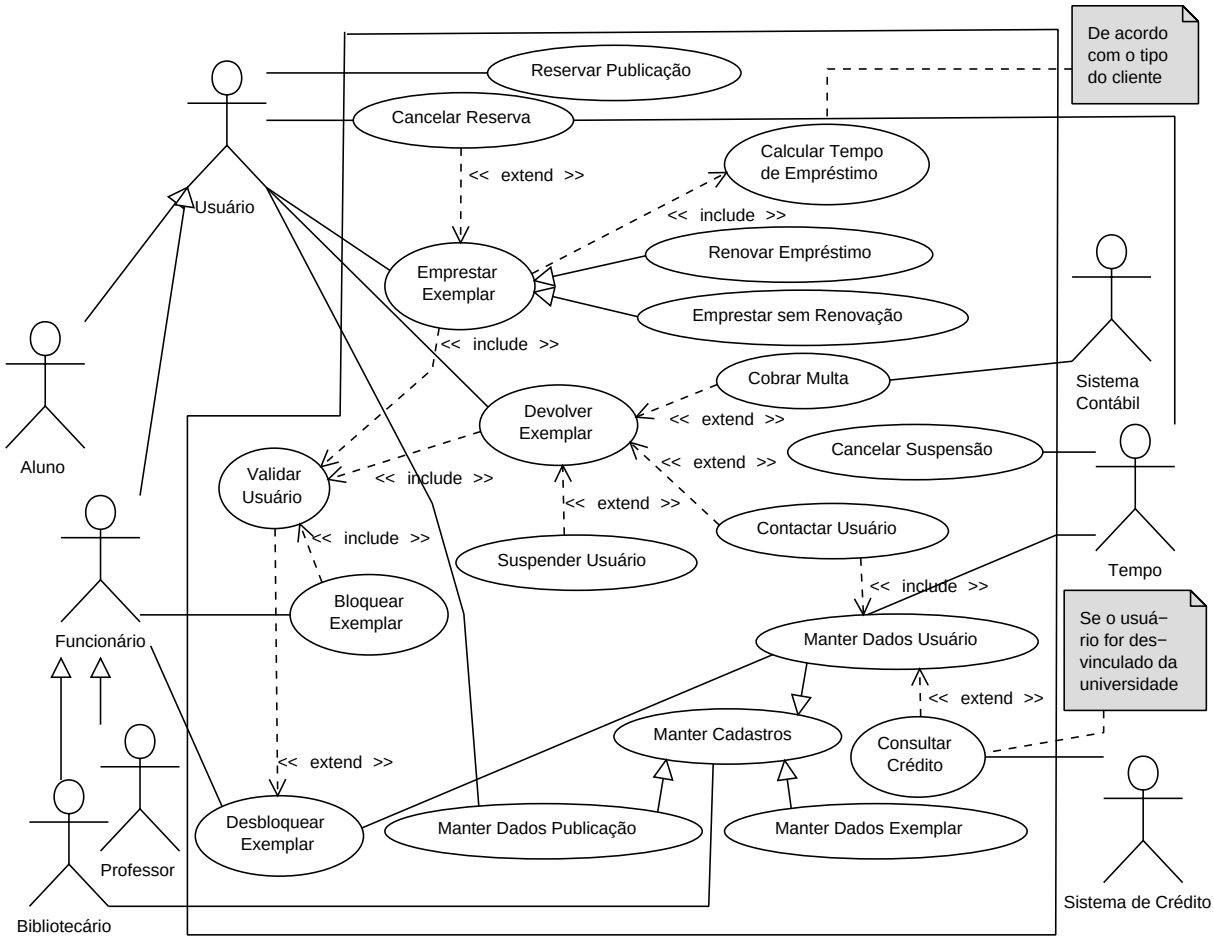


Figura 2.12: Diagrama de Casos de Uso do Sistema da Biblioteca

intercalação (do inglês *merge*), que é opcional, indica o final da execução de um bloco condicional iniciado por um desvio. Apesar de também ser representado por um losango, a intercalação é complementar ao desvio, possuindo múltiplas transições de entrada e apenas uma de saída. Ao término do bloco condicional, independentemente de qual atividade tenha sido executada (B ou C), a Atividade D é executada.

Uma das características mais importantes de um diagrama de atividades é a facilidade de se representar atividades que são executadas de forma concorrente. Para indicar que dois subfluxos do fluxo de eventos devem executar concorrentemente, usa-se o símbolo de **separação** (do inglês *fork*), que é representado por uma barra sólida (horizontal ou vertical) com exatamente um fluxo de entrada e vários fluxos de saída. Essas saídas indicam subfluxos que são executados em paralelo quando a transição é acionada.

Sempre que iniciamos uma execução concorrente, é necessário indicar o ponto da execução onde a concorrência termina. Nesse local acontece o que chamamos de sincronização, permanecendo assim até que todos os caminhos de execução finalizem. Para representar o fim da concorrência

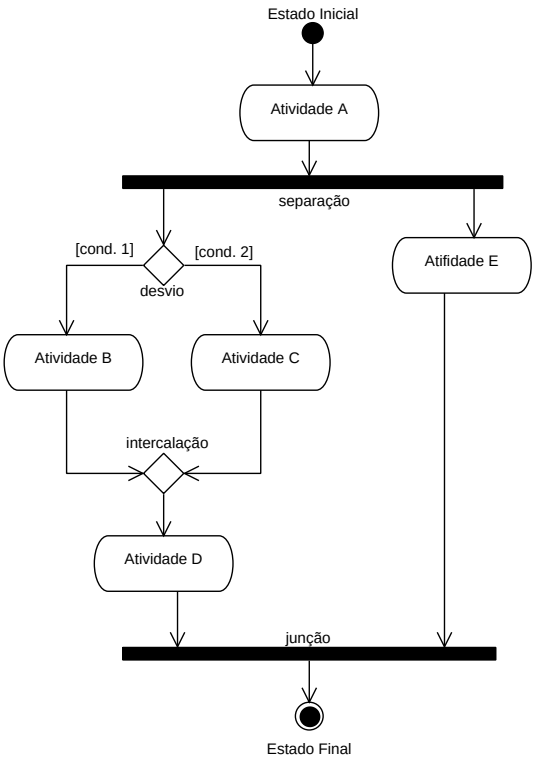


Figura 2.13: Um Exemplo Simples de Diagrama de Atividades

em um diagrama de atividades, utiliza-se o símbolo de **junção** (do inglês *join*), também conhecido como união. Sua representação também se dá a partir de uma barra sólida, porém, com duas ou mais entradas e uma única saída. Um detalhe importante é o balanceamento entre separações e junções. Isso significa que o número de fluxos que saem de uma separação deve ser necessariamente o mesmo número de fluxos que entram na junção correspondente.

Na Figura 2.13, logo depois que a Atividade A termina, dois subfluxos concorrentes são executados; um contendo apenas a Atividade E e o outro contendo as atividades B, C e D. Esses subfluxos se juntam imediatamente antes do estado final ser alcançado.

A Figura 2.14 apresenta um diagrama de atividades correspondente ao fluxo de eventos do caso de uso Emprestar Exemplar. Neste diagrama são representados dois fluxos alternativos. O primeiro é iniciado quando o usuário está suspenso para empréstimo. O segundo por sua vez, é ativado quando apesar do cliente estar apto a realizar empréstimos, o exemplar desejado encontra-se indisponível para empréstimo. Essa indisponibilidade pode ser conseqüência de um bloqueio por parte de um professor.

Além de oferecer uma maneira gráfica de representar o fluxo de eventos de um caso de uso, diagramas de atividade também são úteis para descrever algoritmos seqüenciais complicados e especificar o comportamento de aplicações paralelas e concorrentes.

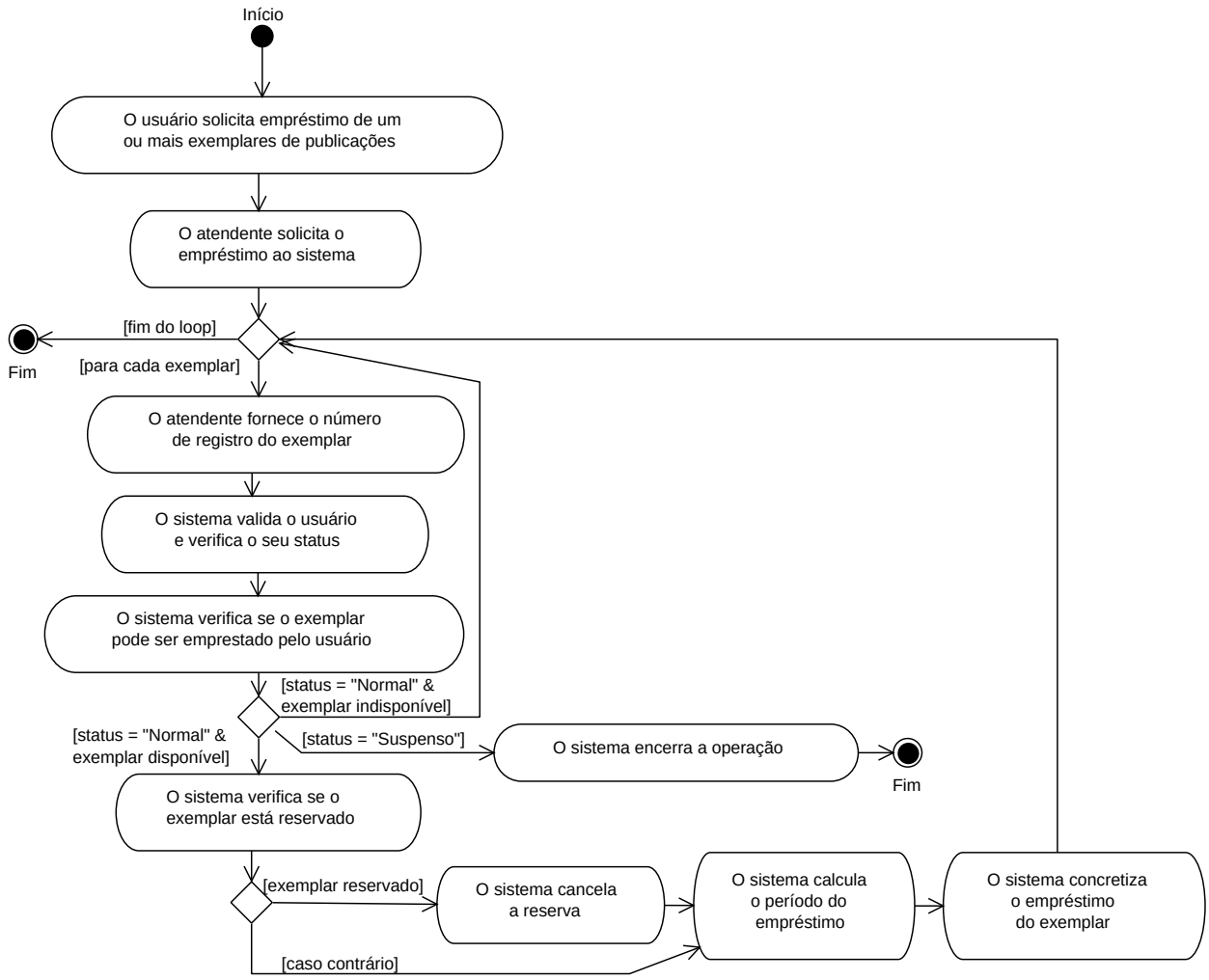


Figura 2.14: Diagrama de Atividades para o Caso de Uso Emprestar Exemplar

2.9.11 Diagramas de Interação de Sistema

As interações entre os atores e o sistema podem ser representadas através de **diagramas de interação de sistema**. A principal característica desses diagramas é a representação do sistema como uma caixa-preta. Assim, a semântica proporcionada pelos diagramas de interação de sistema descreve o que o sistema faz sem se preocupar com o como é feito [29]. A representação da interação do sistema pode ser feita tanto através de um **diagrama de seqüência**, quanto através de um **diagrama de colaboração**, que são diagramas dinâmicos da UML. Esses diagramas, descritos a seguir, serão detalhados no Capítulo 5.

De uma maneira geral, os diagramas de interação de sistema são modelos que ilustram eventos entre atores e o sistema. Ele mostra, para um cenário particular de um caso de uso, os eventos que os atores geram e a ordem com que esses eventos são executados. Dessa forma, a ênfase desses

diagramas está nos eventos que cruzam a fronteira do sistema [29].

A principal diferença entre os diagramas de seqüência e os diagramas de colaboração está na ênfase dada por cada um deles. Enquanto os **diagramas de seqüência** enfatizam a ordem em que os participantes da interação se comunicam, os **diagramas de colaboração** enfatizam as conexões entre os participantes.

A Figura 2.15 mostra um diagrama de seqüência de sistema com as interações em um cenário do caso de uso Emprestar Exemplar, onde o usuário realiza o empréstimo de um exemplar com sucesso. O diagrama apresenta os atores do sistema da Biblioteca e o objeto (:SistemaBiblioteca) que representa o próprio sistema computacional.

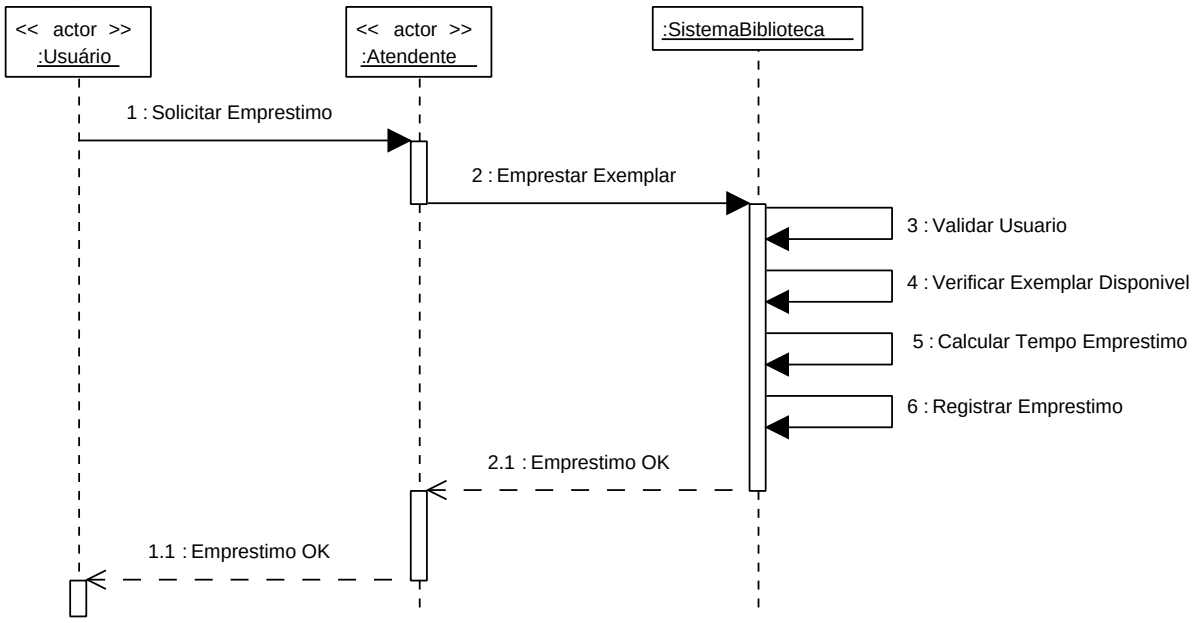


Figura 2.15: Diagrama de Seqüência de Sistema para um Cenário de Emprestar Exemplar

Em um diagrama de seqüência, os participantes da interação aparecem no topo do diagrama, como retângulos. Os nomes associados aos participantes indicam seus tipos e são sublinhados para ressaltar que eles são instâncias de classes, isto é, objetos e não classes.Como visto no Capítulo 1 deste livro (Seção 1.3.2), classe é a descrição de um molde que especifica as propriedades e o comportamento relativo a um conjunto de objetos similares

As mensagens entre os participantes são representadas pelas setas entre as linhas verticais abaixo dos mesmos. Além de representar as mensagens em si com sua origem e destino, os diagramas de seqüência enfatizam a ordem cronológica entre todas as mensagens da interação. Essa ordem é definida através da posição que a mensagem ocupa na linha de tempo do objeto (linha vertical tracejada). Por exemplo, na Figura 2.15, "Emprestar Exemplar" ocorre antes de "Registrar empréstimo", já que se encontra em uma posição mais alta no diagrama.

Além de enviar mensagens para outros participantes, um participante pode enviar uma men-

sagem para si mesmo ("auto-referenciação") para indicar que está realizando algum trabalho interno. Na figura, um exemplo disso é a mensagem "Validar usuário", que o objeto :SistemaBiblioteca envia para si mesmo quando recebe uma mensagem "Emprestar exemplar", advinda do objeto :Atendente.

Você pode também construir um diagrama de colaborações de sistema (Figura 2.16) para uma melhor visualização das associações entre os diversos objetos (no caso, atores e o sistema).

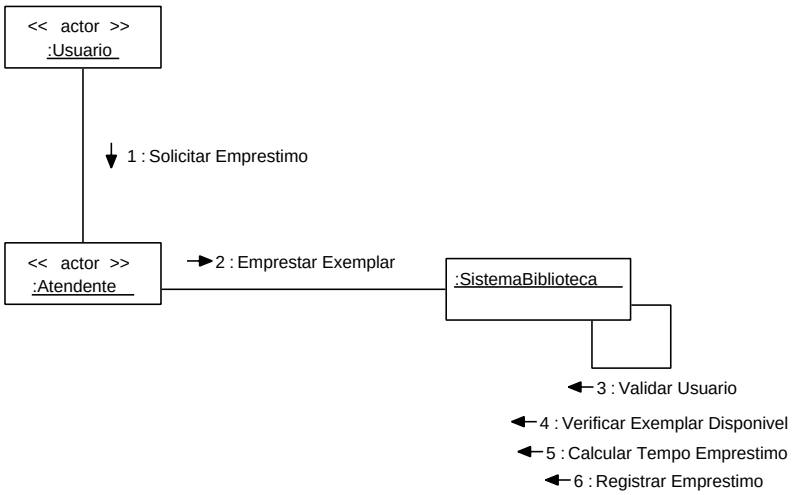


Figura 2.16: Diagrama de Colaboração de Sistema para um Cenário de Emprestar Exemplar

Em um diagrama de colaboração, os participantes da interação são representados da mesma maneira como aparecem em um diagrama de seqüência, porém, não é necessário dispor os objetos de uma maneira pré-definida. Associações entre os objetos são indicadas explicitamente no diagrama através de linhas que ligam dois participantes. As mensagens, por sua vez, são representadas por setas. Diferentemente do que ocorre em diagramas de seqüência, nos quais a ordem entre as mensagens é explícita, num diagrama de colaboração ela só pode ser representada através do artifício de enumeração das mensagens.

2.9.12 Início da Análise

Nos diagramas anteriores, o sistema foi tratado de forma monolítica (como uma caixa preta); a fase de análise é responsável por particionar/identificar os objetos que compõem o sistema internamente. Nessa fase, os componentes do sistema são identificados gradativamente a partir dos casos de uso e representados num **diagrama de classes** UML. Como apresentado na Figura 2.17, esse diagrama é construído a partir da identificação das principais entidades conceituais do sistema e da especificação da maneira como elas se relacionam. Como visto na Seção 1.3.4, cada entidade, conhecida como **classe**, encapsula os seus dados através de **atributos** e o seu comportamento através de **operações**.

A identificação das classes do sistema baseia-se principalmente na análise do domínio do negócio,

definido na especificação dos casos de uso. No contexto do sistema para controle de bibliotecas, nós sabemos que a entidade "publicação" faz parte do seu domínio. Além disso, sabemos que publicações são tipos de "item emprestável". A decisão de modelar o conceito de publicação separado do conceito de item emprestável se baseia na possibilidade da biblioteca disponibilizar outros itens para empréstimo, como por exemplo, DVDs, CDs de música , etc., como mostrado na Seção 2.9.3. As operações `emprestar()` e `devolver()`, comuns a todos os itens emprestáveis, são identificadas na classe ItemEmprestável.

O diagrama de classes do sistema da biblioteca mostrado na Figura 2.17 é parcial. No Capítulo 3 veremos uma forma sistemática de construir um diagrama de classes mais elaborado, a partir das especificações dos casos de uso.

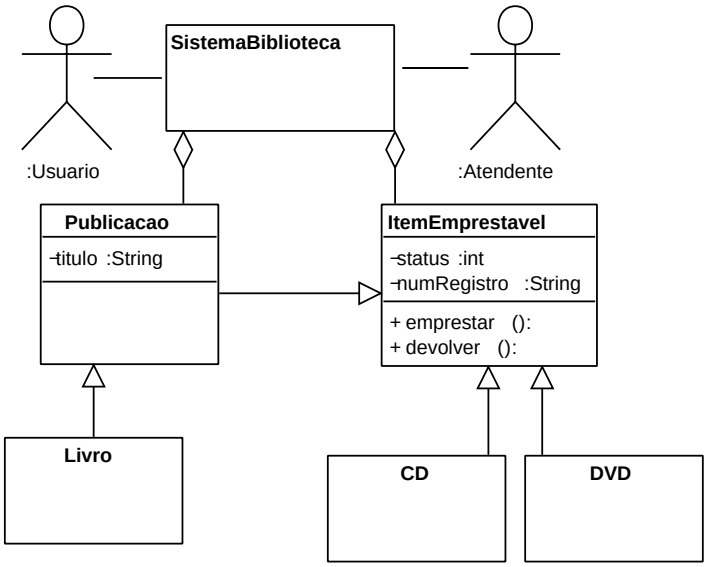


Figura 2.17: Modelo de Classes Preliminar para o Sistema da Biblioteca

2.10 Resumo

Os requisitos de um software representam a idéia do usuário a respeito do sistema que será desenvolvido. Em geral, os requisitos podem ser vistos como condições ou capacidades necessárias para resolver um problema ou alcançar os objetivos pretendidos. Durante a fase de especificação dos requisitos é feita uma distinção clara entre as funcionalidades especificadas para o sistema (requisitos funcionais) e os atributos de qualidade desejados, que são materializados pelos requisitos não-funcionais. Apesar de não representarem funcionalidades diretamente, os requisitos não-funcionais podem interferir na maneira como o sistema deve executá-las.

Os requisitos de um sistema são representados através de um documento, conhecido como documento de requisitos. Esse documento deve registrar a necessidade real dos usuários e suas

expectativas, dimensionar a abrangência do sistema e delimitar o seu escopo. Em UML, essa representação é feita através do conceito de casos de uso.

A modelagem de casos de uso é uma técnica que auxilia o entendimento dos requisitos de um sistema computacional através da criação de descrições narrativas dos processos de negócio (casos de uso), além de delimitar claramente o contexto do sistema computacional (atores).

Existem três maneiras de representar a forma como dois casos de uso se relacionam entre si: (i) relacionamento de inclusão (<< *include* >>), que indica a inclusão do comportamento de um caso de uso em outro; (ii) relacionamento de extansão (<< *extend* >>), que indica uma inclusão condicional; e (iii) relacionamento de generalização/especialização (Seção 1.3.5), que representa o conceito de subtipo.

O comportamento de um caso de uso é detalhado através da descrição de fluxos de eventos. Esses fluxos definem uma seqüência de comandos declarativos, incluindo desvios condicionais e iterações. Instâncias desses fluxos, onde as decisões e iterações são pré-definidas são conhecidas como cenários de uso. Em UML, a forma indicada para representar graficamente os fluxos é através de diagramas de atividades. Já os cenários de uso são normalmente representados através de diagramas de seqüência ou diagramas de colaboração. Por ocultar os detalhes internos do sistema e se preocupar unicamente com a representação da interação do sistema com o meio externo, os diagramas de seqüência e colaboração feitos durante a especificação dos casos de uso são classificados como diagramas "de sistema", isto é, diagramas de seqüência de sistema e diagramas de de colaboração de sistema, respectivamente.

Com os casos de uso especificados, o próximo passo é detalhar o sistema internamente. Do ponto de vista da análise orientada a obetos, esse detalhamento consiste na identificação das classes internas que compõem o sistema e do relacionamento e regras de interação entre elas.

2.11 Exercícios

1. Dado o seguinte enunciado de um sistema de Ponto de Venda (PDV):

   > Um sistema de ponto de vendas é um sistema computacional usado para registrar vendas e efetuar pagamentos. Ele inclui componentes de hardware, como um computador e um *scanner* de código de barras, além dos componentes de software para o controle do sistema. Nesse exemplo, estamos interessados na compra e pagamento de produtos. Os requisitos básicos de funcionamento do sistema são nove: (i) registrar os itens vendidos em cada venda; (ii) calcular automaticamente o total de uma venda, incluindo taxas; (iii) obter e apresentar as informações sobre cada produto mediante a leitura de seu código de barras; (iv) reportar ao estoque a quantidade de cada produto vendido quando a venda é completada com sucesso; (v) registrar cada venda completada com sucesso; (vi) exigir que o atendente forneça sua senha pessoal para que possa operar o sistema; (vii) o sistema deve ter um mecanismo de armazenamento de memória estável; (viii) receber pagamentos em dinheiro ou

cartão de crédito; e (ix) emitir mensalmente o balanço do estoque.

(a) Classifique os nove requisitos do sistema de pontos de venda entre funcionais e não-funcionais. Os requisitos não-funcionais devem ser categorizados de acordo com a nomenclatura ABNT/ISO 9126, apresentada na Seção 2.1.

(b) Especifique o diagrama de casos de uso do sistema;

(c) Identifique os relacionamentos entre os diversos casos de uso do sistema usando $<< include >>$ e $<< extend >>$.

(d) Descreva um dos casos de uso do sistema usando o formato da Seção 2.6.

2. Dado o seguinte enunciado de um sistema de Controle de Videolocadora:

> Um sistema de controle para uma videolocadora tem por objetivo automatizar o processo de locação e devolução de fitas de vídeo. Deve-se manter um controle dos cadastros de clientes e seus respectivos dependentes e também um controle sobre o acervo de fitas e sua movimentação.
>
> Os clientes podem executar operações que envolvem locação, devolução e compra de fitas. Caso a fita não seja devolvida no prazo previsto, uma multa será cobrada. Caso o cliente perca ou danifique uma fita alugada, ele deve ou pagar uma multa equivalente ao preço de uma fita nova, ou comprar uma nova fita para substituir a que foi danificada.

(a) Especifique o diagrama de casos de uso do sistema;

(b) Identifique os relacionamentos entre os diversos casos de uso do sistema usando $<< include >>$ e $<< extend >>$.

(c) Atualize o diagrama de casos de uso especificado para contemplar as seguintes restrições, adicionadas ao sistema:

- O cliente *VIP* pode alugar um número ilimitado de fitas; caso contrário, o número máximo de fitas é limitado a três.
- O pagamento pode ser efetuado no ato da locação ou da devolução e pode ser feito em dinheiro, com cartão de crédito, ou através de "cheque-vídeo", que é comprado antecipadamente. Se pagar com "cheque-vídeo", o cliente recebe um desconto especial.

3. Considere os seguintes requisitos de um sistema para gestão de um estacionamento:

- O controle é efetuado com base na placa do veículo, que deve estar cadastrado no sistema.
- Na entrada do estacionamento deve existir um funcionário inserindo os números das placas no sistema, que registra automaticamente a data e a hora de início.
- Se o veículo não estiver cadastrado, o funcionário deve cadastrá-lo. O cadastro é feito em duas etapas: (i) cadastro do responsável; e (ii) associação do veículo ao responsável.
- O funcionário também pode editar os dados do responsável ou alterar os veículos associados a ele.

- Na saída do estacionamento, o funcionário registra novamente o número da placa do veículo. Nesse momento, o sistema calcula o valor a pagar pelo serviço.
- O estacionamento tem clientes avulso e clientes mensalistas. Para os clientes mensalistas, o sistema deve oferecer a possibilidade de adicionar o valor na conta do usuário, que é paga mensalmente.
- O gerente do estacionamento consulta diariamente um relatório do sistema. Em algumas situações, o gerente poderá desempenhar as funções de atendimento, no entanto, apenas o gerente pode solicitar o relatório.

Construa um diagrama de casos de uso para esse sistema. Tente aplicar os relacionamentos de << *include* >> e << *extend* >> no seu modelo.

4. Modele o diagrama de casos de uso de um sistema de gerenciamento de hotel descrito a seguir, seguindo as diretrizes propostas na Seção 2.9:

   Um grupo de empresários deseja que sua equipe desenvolva um sistema para gerenciar reservas e ocupações de apartamentos em uma rede de hotéis. O sistema será utilizado para controlar serviços internos de cada hotel e para a comunicação entre hotéis da rede de forma que seja possível que uma unidade da rede faça consultas sobre a disponibilidade de vagas em outras unidades da mesma cidade ou região. Os serviços básicos a ser considerados são:

   - Entrada para cadastro de cliente (nome, endereço, e-mail, data de chegada, data de saída, classificação do cliente, documento);
   - Consultas, reservas e cancelamento de reserva através da Web;
   - Caso uma reserva não seja cancelada e nem ocupada no prazo especificado, é cobrada uma tarifa de *no show*, correspondente a 30% do valor da diária;
   - Cadastro de apartamento: tipo de apartamento (suíte, standard, duplo, ar-condicionado), cidade ou local;
   - Cadastro de salas e auditório;
   - Cadastro de despesas;
   - Serviços adicionais são também incluídos no sistema: telefone, TV paga, acesso à internet, "frigobar", lavanderia, serviço de lanche e café da manhã;
   - Conexão para consultas e reservas de vagas em outros hotéis do grupo;
   - Controle de ocupação de apartamentos (reservado ou entrada do hóspede);
   - Controle de ocupação de salas e auditório;
   - Controle de limpeza dos apartamentos;
   - Descontos para clientes VIP e grupos;
   - Recebimento de pagamento (tipo de pagamento cheque, dinheiro, cartão, parcelado, moeda estrangeira);
   - Emissão de nota fiscal (podendo ser separado por itens: hospedagem, restaurante, lavanderia, etc);
   - Emissão da fatura parcial (somente para consulta);

- Emissão de relatórios contábeis;
- Emissão de relatórios de ocupação;
- Emissão de relatórios de hóspedes em débito;
- Relatórios parciais de consulta;
- Gerar relatórios estatísticos (média de dias que o cliente hospeda, gastos médios, itens mais consumidos nos restaurantes);
- Esse sistema deve ser interligado a um sistema contábil, que é responsável pelo pagamento dos serviços consumidos no hotel.

Capítulo 3

Análise Orientada a Objetos: Modelagem Estática

Este capítulo apresenta uma série de passos e técnicas para a elaboração de modelos conceituais, utilizando a notação UML. Essa seqüência de passos bem definidos auxilia a especificação do modelo de análise de sistemas de software a partir dos requisitos estabelecidos. Assumimos que esses requisitos são especificados através de casos de uso (Capítulo 2), embora as idéias apresentadas sejam genéricas o suficiente para possibilitar a sua utilização conjunta com outros artefatos de descrição de requisitos.

3.1 Análise OO x Projeto OO

Um processo de desenvolvimento de software tem por objetivo enfatizar princípios de boas práticas de desenvolvimento e proporcionar diretrizes que o desenvolvedor possa adotar. Entretanto, as atividades envolvidas nessa tarefa requerem habilidades criativas e individuais, experiências passadas e bom senso. Uma das partes mais importantes de um processo de desenvolvimento de software é o conjunto de diretrizes que sugere como decompor sistemas grandes e complexos em componentes menores que possam ser manipulados mais facilmente.

A aplicação de conceitos de orientação a objetos para o desenvolvimento de software traz novas soluções para velhos problemas, mas também demanda a confecção de novos métodos e processos que adotem uma abordagem completamente nova. Sendo assim, um processo de desenvolvimento de software orientado a objetos difere dos processos tradicionais na maneira particular como decompõe o sistema em partes menores e na natureza dos relacionamentos entre essas partes. Além disso, numa abordagem clássica, as fases de análise, projeto, implementação e testes são geralmente

vistas como fases globais e separados a serem executados seqüencialmente no ciclo de vida do sistema inteiro (veja a discussão da Seção 1.4.1). Na abordagem orientada a objetos, essa idéia não se aplica. A seqüência de fases de análise, projeto, implementação e testes pode ser adotada para o ciclo de vida de classes individuais, e não necessariamente para o sistema inteiro como na abordagem clássica.

Embora não haja uma padronização de termos na área de análise e processos de desenvolvimento orientado a objetos, existe uma distinção comum feita entre análise orientada a objetos e projeto orientado a objetos. Em geral, a fase de análise lida com o domínio do problema, enquanto que a fase de projeto lida com o domínio da solução. A seguir, é mostrada uma descrição mais precisa desses termos:

- **Análise OO.** Modela o mundo real de tal modo que ele possa ser compreendido. Durante a análise, a ênfase está em encontrar e descrever as entidades do domínio do problema que sejam relevantes para o sistema que se pretende construir.
- **Projeto OO.** Define as entidades de software que fazem parte do domínio da solução e que serão implementadas em uma linguagem de programação orientada a objetos.

O modelo de análise é uma abstração do modelo de projeto. Graças à sua abstração, o modelo de análise omite vários detalhes de como o sistema funciona e proporciona um panorama geral da funcionalidade do sistema. Por ser um refinamento do modelo de análise, o modelo de projeto consiste de um conjunto de colaborações que representam a estrutura e o comportamento do sistema, de forma mapeável a pelo menos uma linguagem de programação orientada a objetos. A maneira como o sistema se comporta é derivada do modelo de casos de uso e dos requisitos não-funcionais especificados para ele.

3.2 Modelagem Estática x Modelagem Dinâmica

A análise orientada a objetos cria uma especificação que utiliza termos e entidades do domínio do problema para a representar principalmente os requisitos funcionais do sistema (Seção 2.1). Durante a análise, são modelados aspectos estáticos e dinâmicos do domínio do problema. Na **modelagem estática**, conceitos do mundo real relevantes para o sistema a ser construído são incluídos em um diagrama de classes de análise, também conhecido como **modelo de objetos** ou **modelo conceitual** [29]. Basicamente, esse modelo é composto das principais entidades do sistema (classes), os atributos dessas classes e os relacionamentos entre elas. A modelagem estática é o foco deste capítulo.

A **modelagem dinâmica** descreve os aspectos do sistema de software que podem mudar com o tempo devido à ocorrência de eventos e que dizem respeito ao seu fluxo de controle. A modelagem dinâmica usa diagramas dinâmicos de UML, como diagramas de seqüência, de atividades e de colaboração, para modelar as interações entre objetos no sistema. Esses objetos são instâncias das

classes de análise e estão contidos no domínio do problema. A modelagem dinâmica também afeta o modelo de classes de análise, enriquecendo-o com elementos ligados ao comportamento do sistema, como as operações. A modelagem dinâmica é apresentada em detalhes no Capítulo 5.

3.3 Metodologias para Análise Orientada a Objetos

Em geral, programas bem estruturados fazem mais do que simplesmente satisfazer seus requisitos funcionais. Um software que segue práticas apropriadas para a sua construção tem maiores chances de estar correto e ser reutilizável, extensível e fácil de depurar. Vale enfatizar que as diretrizes de projeto advindas da abordagem convencional, tais como particionamento, hierarquia, modularidade, etc., que foram vistas na Seção 1.1.1, também são aplicáveis ao desenvolvimento orientado a objetos.

Como discutido na Seção 1.2, os principais benefícios do desenvolvimento orientado a objetos são: (i) aumentar a produtividade no desenvolvimento e (ii) facilitar a manutenção (adição de melhorias e modificações) de sistemas de software. Esse aumento da produtividade é uma conseqüência tanto do melhor entendimento do sistema, quanto do maior grau de reutilização de suas partes. Por ser um tema atual de pesquisa, existem muitos trabalhos na literatura que enfatizam a reutilização de software de maneira sistemática. A maioria desses trabalhos destacam a importância dos processos de desenvolvimento e do paradigma de objetos para reduzir o acoplamento entre as diversas partes que compõem o sistema. Além de maximizar a reutilização, a redução do acoplamento entre os módulos do sistema melhora consideravelmente o seu requisito de manutenibilidade.

Vários processos orientados a objetos diferentes podem ser encontrados na literatura. Cada um desses processos introduziu um método para a análise de sistemas de software, um conjunto de diagramas que evoluiu a partir do método, e uma notação para representar os diagramas e modelos de uma maneira consistente. Apesar dessa diversidade, desde o final dos anos 90, um processo vem se tornando padrão para o desenvolvimento de software orientado a objetos: o *Rational Unified Process* (RUP), da IBM. Esse processo surgiu da união de três outras bastante populares: Booch [8] de Grady Booch, *Object Modelling Technique* (OMT) de James Rumbaugh [41] e OOSE [27] de Ivar Jacobson.

Pode-se dizer que os processos existentes para desenvolvimento orientado a objetos são, na sua macro-estrutura, muito parecidos. Esta seção, descreve sucintamente como os processos OMT e RUP conduzem a fase de análise orientada a objetos. Em seguida, é apresentado um processo composto dos passos principais, comuns a vários processos de desenvolvimento orientado a objetos. Esse processo, que pode ser considerado uma instância do RUP, será utilizado nos exemplos deste livro.

3.3.1 OMT

O processo OMT foi desenvolvido por Rumbaugh *et.al.* [40] para a análise e projeto de sistemas de software no início dos anos 90. Como pode ser visto na Figura 3.1, a atividade de análise nesta metodologia cria três modelos: (i) modelo de objetos; (ii) modelo dinâmico; (iii) modelo funcional.

- **O modelo de objetos** descreve os aspectos de um software em termos de um diagrama de classes. Esse modelo é uma abstração da implementação de um sistema e é representado por grafos cujos nós são classes, e arcos que denotam os relacionamentos de generalização/especialização, agregação/decomposição ou associações entre as classes. O modelo de objetos da OMT corresponde ao diagrama de classes UML.
- **O modelo dinâmico** descreve os aspectos do sistema que podem mudar com o tempo devido à ocorrência de eventos e é usado para entender o fluxo de controle do software. Ele pode ser ou um diagramas de fluxo de eventos, equivalente aos diagramas de seqüência e de colaboração UML, ou por diagramas de transições de estados, cujos nós são estados e arcos são transições causadas por eventos entre os estados.
- **O modelo funcional** é um reflexo da visão centrada em dados dos processos contemporâneos ao OMT, que seguem o paradigma de programação estruturada. O modelo funcional, que não tem um equivalente em UML, descreve as transformações de dados dentro do sistema e emprega Diagramas de Fluxo de Dados (DFD) para representar computações dos valores de saída a partir de valores de entrada.

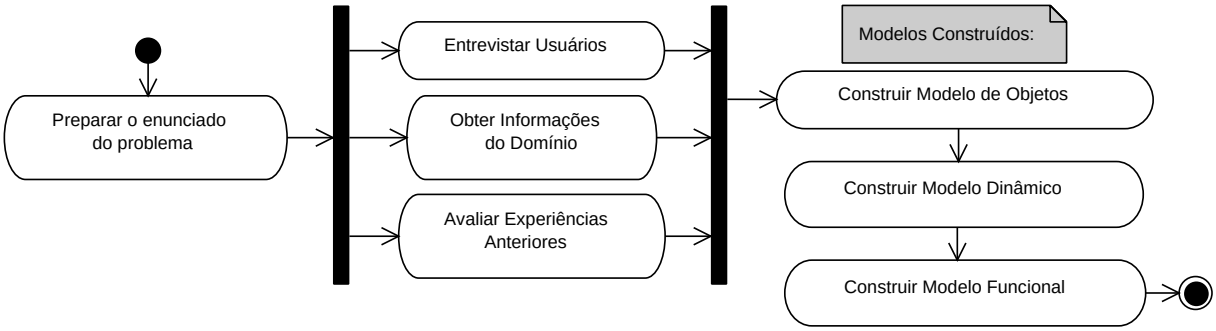


Figura 3.1: Estágio da Análise

Os modelos de objetos, dinâmico e funcional são visões ortogonais da descrição de um sistema. Entretanto, o modelo de objetos é o mais importante porque ele descreve os elementos principais do sistema, independentemente de quando e como eles mudam. O produto da análise é um documento que consiste do (i) enunciado do problema, (ii) modelo de objetos, (iii) modelo dinâmico e (iv) modelo funcional. A partir desses artefatos, é possível identificar: objetos e seus atributos, operações, a visibilidade de cada objeto em relação aos outros e a interface de cada objeto.

Para sistematizar a construção desses artefatos de análise, o processo OMT apresenta oito atividades a serem executadas: (i) identificar objetos e classes; (ii) preparar dicionário de dados;

(iii) identificar associações (agregações); (iv) identificar atributos; (v) refinar com herança; (vi) testar os caminhos de acesso aos atributos usando cenários; (vii) iterar e refinar; e (viii) agrupar classes em módulos.

Primeiro, um conjunto de classes candidatas é encontrado e refinado, a partir de uma descrição textual dos requisitos do sistema. Entre as classes encontradas, são eliminadas aquelas que não são importantes para o domínio do problema e as que não são relevantes para o sistema que se pretende construir. As remanescentes são então descritas em um glossário de termos específicos do problema, que é usado pelos desenvolvedores. Depois que as classes de análise são identificadas, associações e atributos são encontrados a partir da especificação de requisitos e as classes são organizadas em hierarquias de herança. Cenários são usados para verificar se os atributos e associações encontrados são realmente relevantes e para definir o comportamento dos objetos. Esse processo é repetido até que o modelo de objetos esteja estável. Neste momento, as classes do modelo são agrupadas em módulos.

3.3.2 RUP

Na segunda metade da década de 90, Grady Booch [8], James Rumbaugh [40] e Ivar Jacobson [27] colaboraram para combinar as melhores características de suas metodologias de análise e projeto orientados a objetos. O resultado final foi a elaboração da UML, uma linguagem concebida desde o princípio para se tornar um padrão em modelagem orientada a objetos. Embora a UML independa do processo de desenvolvimento utilizado, seus criadores também se preocuparam em elaborar um processo para desenvolvimento de software baseado nela. Este processo foi batizado, inicialmente, de *Unified Software Development Process* [26]. Posteriormente, foi rebatizado para *Rational Unified Process* (RUP), pois a *Rational Software Corporation* contratou seus três criadores. Mais que um processo, o RUP é um imenso conjunto de métodos, técnicas, documentos e procedimentos que visam ser o mais genérico possível. Isso o torna muito complexo para ser usado diretamente, sendo necessário adaptá-lo às necessidades de cada organização, eliminando-se partes que não sejam relevantes em um determinado contexto.

Segundo o RUP, a análise visa (i) identificar as classes que executam o fluxo de eventos de um caso de uso e os relacionamentos entre essas classes (modelagem estática) e (ii) distribuir o comportamento do caso de uso entre essas classes, através de realizações de casos de uso (modelagem dinâmica). A realização de um caso de uso consiste de um conjunto de diagramas dinâmicos da UML, tais como diagramas de seqüência e de colaboração, que modelam o fluxo de eventos e alguns cenários do caso de uso. O RUP recomenda que durante a fase de análise deve-se seguir as sete atividades seguintes:

1. **Definir a estrutura do sistema**. Esta etapa consiste na definição dos módulos principais do sistema e nas restrições de comunicação entre eles. Essa definição estrutural influenciará todas as atividades seguintes, isto é, as restrições especificadas aqui deverão ser satisfeitas adiante.
2. **Complementar as descrições dos casos de uso**. No decorrer do desenvolvimento, é

comum sentir a necessidade de ajustar a especificação dos casos de uso. Essas mudanças são fruto de uma maior contextualização com o problema e uma maior interação com o cliente/usuário.

3. **Identificar classes de análise**. Essas classes devem ser identificadas a partir das especificações dos casos de uso.

4. **Distribuir comportamento entre as classes de análise**. Isso acontece através de simulações dos cenários dos casos de uso, através dos diagramas dinâmicos da UML.

5. **Descrever responsabilidades das classes**. Após a distribuição do comportamento entre as classes, as responsabilidades das classes já estão identificadas. A descrição dessas responsabilidades consiste na atualização do glossário, apresentado na Seção 2.9.4.

6. **Identificar os atributos**. Assim como a descoberta das classes, a identificação dos atributos se baseia na análise da especificação dos casos de uso do sistema.

7. **Estabelecer associações entre classes de análise**. As associações entre as classes de análise podem ser identificadas a partir dos diagramas dinâmicos já especificados. Nesse sentido, a necessidade de interação entre duas classes durante a execução de operações pode indicar a existência de associações entre elas. O conceito de associação foi apresentado na Seção 1.3.7.

Como apresentado após a definição da estrutura do sistema, as descrições de casos de uso são complementadas. O objetivo desse detalhamento é capturar informações adicionais necessárias para entender o comportamento interno que o sistema deve apresentar. Depois, um conjunto de classes de análise candidatas é encontrado a partir das descrições estendidas dos casos de uso.

A fim de determinar as responsabilidades das classes de análise, cenários (diagramas de seqüência e colaboração) são construídos com base nas classes de análise encontradas e nas especificações dos casos de uso. Responsabilidades são extraídas a partir das mensagens trocadas entre os objetos nos diagramas de interação. Descrições curtas podem ser associadas a cada responsabilidade identificada, caso as descrições existentes nos casos de uso não sejam satisfatórias. Durante a etapa de projeto, as responsabilidades das classes de análise são refinadas para operações.

Finalmente, as informações que devem ser mantidas pelas classes de análise e as associações entre essas classes são extraídas das descrições dos casos de uso e dos diagramas de interação especificados.

Uma característica importante do processo RUP é a preocupação constante com a arquitetura de software, que define o sistema em termos dos seus componentes conceituais e das restrições de comunicações entre eles. A arquitetura inicial, proposta pelo RUP, define uma visão modular do sistema, sem detalhar os componentes envolvidos. Mais detalhes sobre arquitetura de software está disponível na Seção 7.2.

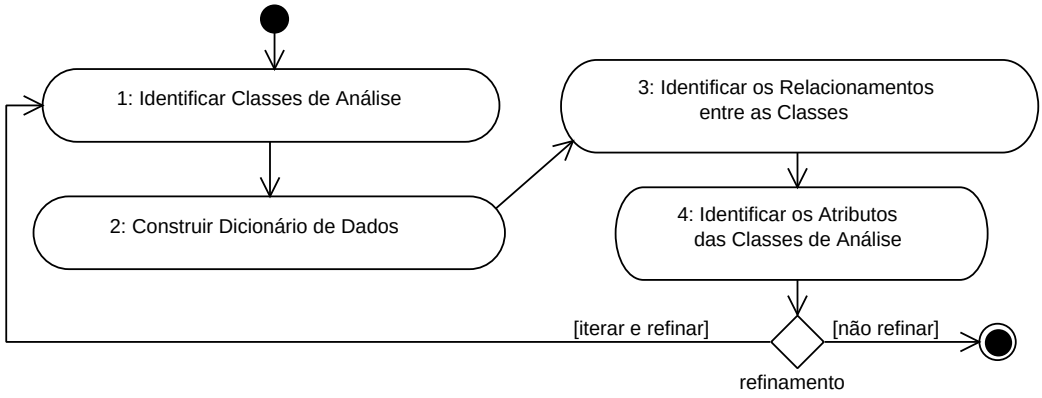


Figura 3.2: Atividades da Modelagem Estática

3.4 Um Método para Análise Orientada a Objetos Usando UML

As duas metodologias apresentadas nas Seções 3.3.1 e 3.3.2 fornecem um panorama útil, ainda que limitado, das metodologias para análise orientadas a objetos. A OMT, apesar de já ter sido bastante popular e de englobar diversas boas idéias, hoje em dia é considerada ultrapassada (por exemplo, ela não explora a noção de casos de uso). O RUP, por outro lado, é moderno e centraliza as qualidades existentes em várias outras metodologias. Por outro lado, é complexo demais para ser usado sem modificações, o que limita um pouco sua aplicabilidade.

Apesar das diferenças entre as diversas metodologias existentes, os passos necessários para realizar a análise orientada a objetos são bastante parecidos em todas elas [39]. Isso pode ser percebido facilmente analisando-se as listas de atividades da OMT e do RUP, apresentadas nas seções anteriores. Essas atividades "genéricas" são apresentadas na Figura 3.2:

As atividades apresentadas na Figura 3.2 compreendem o que é normalmente chamado de modelagem estática. Na literatura, a atividade de encontrar as operações das classes de análise pode ser considerada parte da modelagem estática [26][39], ou parte da modelagem dinâmica [29][40]. Neste livro adotamos a segunda abordagem e deixamos esse passo para ser tratado no Capítulo 5, que descreve a modelagem dinâmica.

Conforme mencionado na Seção 3.3.2, a UML foi concebida para se tornar um padrão em linguagem de modelagem. Conseqüentemente, ela não depende de uma metodologia específica. Apesar disso, seus criadores tinham em mente quatro características que qualquer processo de desenvolvimento baseado na UML deveria seguir: (i) direcionamento por casos de uso; (ii) centrado na arquitetura; (iii) iterativo; e (iv) incremental. Essas características são detalhadas a seguir:

- **Direcionado por casos de uso.** Na UML, casos de uso capturam os requisitos funcionais do sistema. Como os casos de uso contém descrições das funcionalidades que o sistema deve implementar, eles afetam todas as etapas do desenvolvimento.

- **Centrado na arquitetura.** A arquitetura de um sistema computacional é formada pelo conjunto das suas estruturas. Essas estruturas englobam elementos de software e hardware, focando nas propriedades visíveis externamente desses elementos e nos relacionamentos entre eles [5]. Em um processo centrado na arquitetura, a definição da arquitetura do sistema é a primeira atividade realizada durante a etapa de projeto.

- **Iterativo.** Em um processo interativo, sistemas são construídos através de diversas iterações, onde cada uma passa por todas as etapas do desenvolvimento e acrescenta algum elemento ao produto final. Um exemplo desse refinamento é o fato de no final da análise as classes serem refinadas com modelos mais próximos do projeto, como por exemplo o MVC, que será apresentado na Seção 3.10.1.

- **Incremental.** Em um processo iterativo-incremental, cada iteração pode ser usada para construir uma parte do sistema, isto é, um resultado que pode ser testado e avaliado. Casos de uso oferecem uma maneira simples e natural de definir incrementos, já que podem ser priorizados de modo que os mais importantes sejam desenvolvidos primeiro e os menos importantes depois.

A seguir, são apresentadas algumas técnicas para auxiliar na identificação das classes de análise e os respectivos atributos. Em seguida, cada uma das atividades apresentadas na Figura 3.2 será detalhada e exemplificada no contexto do estudo de caso do sistema de bibliotecas, apresentado no Capítulo 2.

3.4.1 Técnicas para Extração de Informações

Durante a identificação de conceitos relevantes para o domínio do problema, é importante tentar extrair o máximo de informação a partir das descrições dos casos de uso. De um modo geral, uma boa prática neste estágio inicial da modelagem é identificar mais classes de análise do que realmente necessário. A razão para isso é o fato de ser mais fácil descartar as classes desnecessárias do que depois precisar analisar os mesmos casos de uso novamente para identificar as classes ausentes. As situações em que é necessário "voltar atrás" acarretam desperdício de tempo no ciclo de desenvolvimento do software. Para agilizar a construção do software, os processos de desenvolvimento buscam antecipar o máximo possível as informações necessárias de maneira a evitar ciclos muito curtos.

Uma outra maneira de agilizar o processo de desenvolvimento é identificar claramente os limites do sistema. Dessa forma, é possível excluir do modelo qualquer aspecto que não esteja diretamente relacionado com o problema que o sistema deve resolver. Um usuário ou cliente é tipicamente um objeto que está fora do escopo sistema que estamos tentando construir, a não ser que o sistema precise manter um registro dos seus dados, por exemplo, para controle de acesso. Embora devamos estar atentos às mensagens enviadas pelos usuários, não existe nenhuma razão para modelá-los como objeto. É fundamental entender que, durante a análise, existem objetos externos ao sistema que fazem parte do domínio do problema, mas que não estarão representados no domínio da solução.

Diversas estratégias podem ser empregadas para se obter as informações relevantes para a construção do diagrama de classes de análise a partir de especificações de casos de uso. Neste livro, descrevemos três técnicas que auxiliam a identificação de classes candidatas para o modelo. São elas: (i) lista de categorias conceituais [29]; (ii) análise textual; e (iii) análise de domínio, apresentada na Seção 2.9.3, no contexto de casos de uso. Essas técnicas são complementares e, sempre que possível, devem ser usadas em conjunto.

Lista de Categorias Conceituais

Nessa estratégia, classes candidatas podem ser identificadas com o auxílio de uma lista de categorias. Essa lista contém diversas categorias que normalmente devem ser levadas em consideração para a construção de sistemas de software. A seguir, são apresentadas as oito principais categorias apresentadas por Larman [29]: (i) objetos físicos ou tangíveis; (ii) especificações ou descrições de coisas; (iii) locais; (iv) transações; (v) itens de transações; (vi) papéis de pessoas; (vii) outros sistemas ou dispositivos externos ao sistema; e (viii) eventos. As categorias não estão organizadas de acordo com qualquer ordem de importância.

Análise Textual

Por se basear unicamente em descrições textuais a respeito do problema, a abordagem de análise textual pode ser utilizada desde os primeiros estágios da análise. Vários processos de desenvolvimento se utilizam dessa característica e propõem formas sistemáticas de analisar artefatos como por exemplo o documento de uma entrevista, uma especificação dada por um especialista do domínio, ou até mesmo as especificações dos casos de uso. Na análise textual, os substantivos identificados correspondem a objetos ou atributos e os verbos correspondem normalmente a associações ou operações (Figura 3.3). Além disso, adjetivos e indicações de estado (palavras após verbo de ligação) podem indicar a necessidade de um atributo. Por exemplo, pela frase "O profuto possui um código r deve estar liberato para consumo", poderia ser identificada a classe candidata Produto, com os atributos "código" e "status da liberação". Como pode ser percebido, a eficácia da análise textual é relativa, uma vez que sua precisão depende da clareza do texto analisado. Mesmo assim, ela é um passo inicial útil para a identificação de classes candidatas e seus comportamentos.

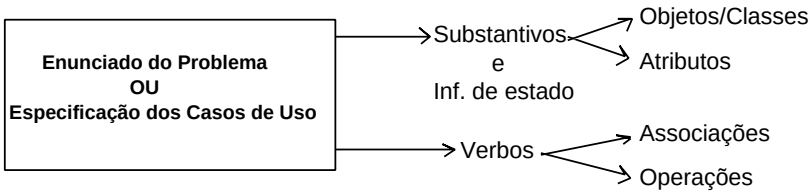


Figura 3.3: Extração de Informações a partir do Enunciado do Problema

A partir da próxima seção, cada uma das atividades da análise orientada a objetos é apresentada

em detalhes. Para exemplificar o papel de cada uma delas, será utilizado o sistema de controle de bibliotecas, apresentado no Capítulo 2.

3.5 Atividade 1: Identificar Classes de Análise

O primeiro passo para analisar os requisitos da aplicação é identificar os conceitos relevantes para o sistema que se pretende construir, no contexto do domínio do problema. A atividade de extração dessas entidades consiste basicamente na análise dos casos de uso e dos outros documentos de requisitos do sistema, a fim de identificar as entidades conceituais relevantes para o domínio do problema. Uma vez obtidos, esses conceitos são transformados em classes, constituindo assim a primeira versão do diagrama de classes de análise. Esse diagrama deve ser revisado até que todos os elementos redundantes ou irrelevantes sejam removidos.

Além de decompor o domínio do problema em unidades compreensíveis para não-especialistas, a criação do diagrama de classes de análise contribui para o entendimento da terminologia ou vocabulário desse domínio. O diagrama de classes de análise oferece meios para que desenvolvedores e especialistas no domínio (usuários, por exemplo) se comuniquem usando uma terminologia uniforme. Além disso, esse diagrama, juntamente com as especificações dos casos de uso, são as principais fontes de informação usadas por desenvolvedores durante a etapa de projeto do sistema.

Durante a análise, assim como em todas as outras etapas de um processo de desenvolvimento orientado a objetos, não há a obrigação de "acertar" tudo na primeira tentativa. É perfeitamente aceitável e até esperado que o analista deixe escapar alguns conceitos relevantes durante a análise e só perceba isso na fase de projeto, ou até mesmo depois. É importante ter em mente que o desenvolvimento deve ser iterativo (Seção 3.4) e que na próxima iteração haverá uma nova etapa de análise. Não vale a pena atrasar todo o cronograma do projeto tentando encontrar todo e qualquer conceito que tem alguma probabilidade, ainda que mínima, de ser relevante para o problema em questão.

Um outra característica da análise orientada a objetos é o fato do seu principal produto, o diagrama de classes de análise, ser uma descrição de elementos do domínio do problema do mundo real, não do domínio da solução, que é o foco do projeto de software. Conseqüentemente, artefatos de software, tais como janelas e bancos de dados, não devem ser levados em consideração na análise, a não ser que essas entidades façam parte do domínio em questão, por exemplo, na modelagem de conceitos de software [29].

3.5.1 Atividade 1.1: Extrair Classes Candidatas

Para identificar classes candidatas em nosso estudo de caso, foram utilizadas as técnicas apresentadas na Seção 3.4.1: (i) lista de categorias conceituais; e (ii) análise textual. A seguir, são mostradas as classes candidatas identificadas com cada uma dessas técnicas.

Lista de Categorias Conceituais

A Tabela 3.1 mostra as entidades descobertas no contexto do sistema de bibliotecas, para cada uma das categorias sugeridas.

Tabela 3.1: Categorias e Entidades identificadas para o sistema da biblioteca

CATEGORIA	EXEMPLO
Objetos físicos ou tangíveis	livro, periódico, tese, manual, cartão da biblioteca
Locais	biblioteca
Transações	empréstimo, reserva, cadastro de usuário, inclusão de nova obra no acervo
Itens de transações	exemplar, publicação
Papéis de pessoas	atendente, usuário, bibliotecária
Outros sistemas ou dispositivos externos ao sistema	sistema de cadastro de publicações e usuários
Eventos	devolução com atraso, reserva, empréstimo, perda, bloqueio, desbloqueio

Como resultado, as seguintes classes candidatas foram identificadas: Livro, Periódico, Tese, Manual, Cartão da biblioteca, Biblioteca, Empréstimo, Reserva, Cadastro de Usuário, Inclusão de nova obra no acervo, Publicação, Atendente, Usuário, Reserva, Empréstimo e Perda.

Análise Textual

A identificação de classes candidatas a partir de análise textual consistiu basicamente na procura de substantivos e referências a mudanças de estado nas especificações dos casos de uso do sistema, que foram identiicados no Capítulo 2. Posteriormente, para definir as associações e operações, será enfatizada a identificação de verbos.

A seguir, apresentamos novamente a descrição do caso de uso Emprestar Exemplar, destacando

os substantivos e referências a estado encontrados.

Caso de Uso: Emprestar Exemplar
Atores: Usuário, Atendente, Sistema de Cadastro.
Descrição: Este caso de uso representa o processo de empréstimo de um ou vários exemplares da biblioteca. O empréstimo se inicia com a solicitação feita pelo usuario ao atendente. Em seguida, através de um terminal, o atendente solicita ao sistema o empréstimo dos respectivos exemplares.
Pré-condição: O exemplar da publicação está disponível, o usuário está cadastrado no sistema e o cliente não está suspenso.
Pós-condição: O exemplar está emprestado.
Restrições Especiais: nenhuma.
Fluxo Básico de Eventos:

1. O usuário solicita empréstimo de um ou mais exemplares de publicações (livro, periódico, tese ou manual), fornecendo o seu código e os exemplares desejados;
2. O atendente solicita o empréstimo ao sistema, fornecendo o código do usuário;
3. Para cada exemplar a emprestar:
   3.1 O atendente fornece o número de registro do exemplar.
   3.2 O sistema valida o usuário e verifica o seu *status* ("Normal" ou "Suspenso") através de seu número de registro. (<< *include* >> Validar Usuário);
   3.3 O sistema verifica se o exemplar pode ser emprestado pelo usuário em questão;
   3.4 Se o *status* do usuário for "Normal" e o exemplar estiver disponível:
      3.4.1. O sistema verifica se a publicação está reservada. Se estiver reservada:
         A. O sistema cancela a reserva, passando o número de tombo da publicação (<< *extend* >> Cancelar Reserva)
      3.4.2. O sistema calcula o período do empréstimo, que depende do tipo de usuário - 7 dias para alunos e 15 para professores (<< *include* >> Calcular Tempo de Empréstimo)
      3.4.3. O sistema registra o empréstimo do exemplar;
      3.4.4. O sistema atualiza seu banco de dados com a informação de que o exemplar não irá se encontrar na biblioteca até completar o período.

Fluxos Alternativos

Fluxo Alternativo 1:

No Passo 3.4, se o usuário estiver suspenso, este é informado de sua proibição de retirar exemplares e o empréstimo não é realizado.

Fluxo Alternativo 2:

No Passo 3.4, se todas as cópias da publicação estiverem emprestadas ou reservadas, o sistema informa ao atendente que não será possível realizar o empréstimo daquele exemplar. Se tiver outros exemplares para emprestar, vá para o Passo 3.1 do fluxo básico.

Entidades identificadas:

exemplar	processo de empréstimo	biblioteca
empréstimo	usuário	atendente
terminal	sistema	publicação
está disponível	está cadastrado	está suspenso
está emprestado	livro	periódico
tese	manual	código do usuário
número de registro	status	está reservada
reserva	número de tombo	período de empréstimo
tipo de usuário	dias	alunos
professores	banco de dados	informação
período	proibição	cópias

3.5.2 Atividade 1.2: Eliminar Classes Inapropriadas

Depois que a lista inicial de classes candidatas é obtida, é necessário refiná-la para eliminar o que não é classe, conceitos redundantes ou irrelevantes para o escopo do sistema. As principais justificativas para a eliminação de classes candidatas são: (i) itens sinônimos; (ii) itens irrelevantes para o contexto do sistema; (iii) itens que representam atributos de outra classe; e (iv) itens que representam conceitos vagos. Entre as classes candidatas listadas anteriormente, podemos eliminar as seguintes:

- **processo de empréstimo**: sinônimo de empréstimo.
- **está disponível**: estado de exemplar (atributo).
- **está cadastrado**: estado de usuário (não é atributo).
- **está suspenso**: estado de usuário (atributo).
- **está emprestado**: estado de exemplar (atributo).
- **código do usuário**: atributo de usuário.
- **número de registro**: atributo de usuário.
- **status**: atributo de usuário.
- **está reservada**: estado de publicação (não é atributo).

- **número de tombo**: atributo de publicação.
- **período de empréstimo**: sinônimo de período.
- **tipo de usuário**: essa informação é capturada pela hierarquia formada entre usuario, aluno e professor.
- **dias**: termo vago.
- **banco de dados**: faz parte do domínio da solução.
- **informação**: termo vago.
- **período**: atributo de empréstimo.
- **proibição**: representa um papel (processo dinâmico).
- **cópias**: sinônimo de exemplar.

Note que poderíamos ter eliminado também as classes candidatas professor e aluno, já que ambas poderiam ser representadas pela classe usuário. Resolvemos deixá-las, porém, porque são conceitos relevantes e a decisão de uni-las faz parte do domínio da solução, mais precisamente na fase de projeto do sistema. O mesmo vale para livro, periódico, tese e manual com relação a publicação.

3.5.3 Atividade 1.3: Refinar a Lista de Classes Candidatas

Após eliminar algumas classes inapropriadas, a nova lista de classes identificadas é apresentada a seguir. As classes em destaque são as que não foram eliminadas:

exemplar	processo de empréstimo	**biblioteca**
empréstimo	**usuário**	**atendente**
terminal	**sistema**	**publicação**
está disponível	está cadastrado	está suspenso
está emprestado	**livro**	**periódico**
tese	**manual**	código do usuário
número de registro	status	está reservada
reserva	número de tombo	período de empréstimo
tipo de usuário	dias	**alunos**
professores	banco de dados	informação
período	proibição	cópias

A classe sistema é uma representação semântica do sistema e em fases posteriores do desenvolvimento poderá ser eliminada.

3.6 Atividade 2: Atualizar Dicionário de Dados

Depois que as classes de análise foram identificadas, é conveniente descrevê-las sucintamente em um **dicionário de dados**. Este serve para aumentar o entendimento dos desenvolvedores sobre o domínio do problema. O dicionário de dados para o nosso estudo de caso é apresentado a seguir.

- **Classe** `Publicação`: representa as publicações pertencentes ao acervo da biblioteca. A classe publicação na realidade representa um conjunto de outras classes que são livros, periódicos, manuais e teses.

- **Classe** `Exemplar`: representa as cópias de uma publicação; representa o objeto físico que é emprestado.

- **Classe** `Biblioteca`: representa a biblioteca em si e por isso guarda os dados relativos a ela, tais como, nome, cnpj e endereço.

- **Classe** `Empréstimo`: representa o empréstimo de uma ou mais publicações a um usuário. Essa classe guarda informações como quais foram as obras emprestadas, quem as pegou emprestadas, quando o empréstimo foi realizado e qual o prazo para devolução.

- **Classe** `Usuário`: representa indivíduos que podem pegar livros emprestados na biblioteca. Dois tipos de usuários são definidos pelo sistema: alunos e professores.

- **Classe** `Atendente`: representa o atendente, um tipo de usuário que opera o sistema no estabelecimento da biblioteca.

- **Classe** `Terminal`: representa o ponto de acesso entre o usuário e o sistema.

- **Classe** `Sistema`: representa o sistema como um todo e engloba todas as classes identificadas.

- **Classes** `Livro`, `Periódico`, `Tese` **e** `Manual`: representam tipos de publicações existentes na biblioteca.

- **Classe** `Reserva`: representa uma reserva que um usuário pode fazer a uma lista de publicações.

- **Classes `Aluno` e `Professor`**: representam tipos de usuário que podem pegar exemplares emprestados na biblioteca.

- **Classe `Item Emprestável`**: representa as características comuns de qualquer item que pode ser emprestado na biblioteca. Como discutido na Seção 2.9.12 do Capítulo 2, a decisão de modelar o conceito de publicação separado do conceito de item emprestável se baseia na possibilidade da biblioteca disponibilizar outros itens para empréstimo, como por exemplo, DVDs, CDs de música , etc.

Vale a pena salientar que após a descrição do dicionário de dados, pode ser conveniente atualizar o glossário, criado inicialmente durante a especificação dos casos de uso (Seção 2.9.4).

3.7 Atividade 3: Identificar os relacionamentos entre as classes

Neste estágio, realiza-se a identificação de associações, agregações e relacionamentos de herança entre as classes. A identificação das associações e agregações torna clara a responsabilidade de cada objeto e é o primeiro passo em direção à especificação das operações das classes do sistema. A identificação de relacionamentos de herança, por sua vez, promove um melhor particionamento dessas responsabilidades, através do agrupamento das características comuns a alguns subconjuntos de entidades.

3.7.1 Atividade 3.1: Identificar Associações

Como apresentado na Seção 1.3.7, uma associação em UML é um relacionamento estrutural entre objetos. Assim como as classes, associações podem ser identificadas tanto através das características do domínio, quanto analisando-se os documentos de requisitos e as descrições dos elementos do dicionário de dados. Expressões do tipo "conhece" freqüentemente indicam a existência de associações. A seguir, é apresentada uma lista com alguns critérios que podem ser utilizados para auxiliar na descoberta de associações entre as classes de análise `A` e `B` [29]:(i) `A` é uma parte física ou lógica de `B`; (ii) `A` está contida em `B`; (iii) `A` é uma descrição de `B`; (iv) `A` é um ítem de uma transação ou relatório `B`; (v) `A` é um membro de `B`; (vi) `A` é uma sub-unidade organizacional de `B`; (vii) `A` usa ou gerencia `B`; (viii) `A` se comunica com `B`; (ix) `A` está relacionada com uma transação `B`; e (x) `A` é possuído por `B`.

Apesar de auxiliar na descoberta de associações, a satisfação de um dos critérios anteriores pode não implicar necessariamente numa associação entre entidades. Mas de uma maneira geral, relacionamentos entre duas classes cujo conhecimento precisa ser preservado durante algum tempo normalmente produzem associações.

Antes de identificar as associações, vale a pena observar que associações demais tendem a tornar o modelo confuso, ao invés de deixá-lo mais claro. Descobrí-las pode ser muito trabalhoso e trazer

poucos benefícios concretos. Por essa razão, evite colocar no modelo associações que podem ser derivadas a partir de outras. Em resumo, na etapa de modelagem estática, é mais importante identificar classes do que associações. A forma como as classes se relacionam entre si é identificada naturalmente durante a modelagem dinâmica (Capítulo 5), uma vez que a troca de mensagens entre objetos podem significar a existência de uma associação entre as classes.

3.7.2 Atividade 3.2: Identificar Agregações

Os relacionamentos de agregação e composição, apresentados na Seção 1.3.6, são associações que indicam relações parte-todo entre duas ou mais classes distintas. Por exemplo, lendo a descrição da classe `Sistema` (Seção 3.6), é possível perceber que há uma agregação entre esta e todas as outras classes de análise, já que ela "engloba todas as classes identificadas".

De uma maneira geral, dadas duas classes (`A` e `B`), podemos dizer que `B` agrega `A` quando: (i) `A` é uma parte física ou lógica de `B`; (ii) `A` está contida em `B`; (iii) `A` é um ítem de uma transação ou relatório `B`; (iv) `A` é um membro de `B`; (v) `A` é uma sub-unidade organizacional de `B`; e (vi) `A` é possuído por `B`.

3.7.3 Atividade 3.3: Identificar Herança

A identificação de relacionamentos de herança é mais simples do que a de associações. Para encontrar relacionamentos de herança entre as classes de análise identificadas, basta estudá-las e procurar por relações do tipo "é-um-tipo-de" entre elas. Um exemplo claro no sistema de biblioteca é o relacionamento entre a classe `Usuário` e as classes `Atendente`, `Aluno` e `Professor`. Além disso, as classes `Periódico`, `Livro`, `Tese` e `Manual` podem ser consideradas como sub-tipos de `Publicação`. A relação das classes identificadas para o sistema pode ser identificada a partir do domínio do problema ou do dicionário de dados, apresentado na Seção 3.6.

Após a identificação dos relacionamentos de herança, pode ser necessário refatorar as associações entre as classes. Essa refatoração tem o objetivo principal de mover as associações comuns a todas as classes derivadas ("filhas") para a classe base ("superclasse"). Além disso, a existência de hirarquias de generalização/especialização bem estruturadas possibilitarão outros benefícios ao modelo. Um exemplo é a utilização do conceito de polimorfismo de inclusão, que será visto na Seção 4.5.5 do Capítulo 4.

A Figura 3.4 apresenta o diagrama de classes de análise depois da adição de associações, agregações e relacionamentos de herança. Devido ao excesso de agregações entre a classe `Sistema` e as demais classes modeladas, essa informação conceitual de composição das partes do sistema pode ser expressada através do conceito de módulo, que pode ser representado por pacotes UML (ver Seção 1.2.2). A Figura 3.5 apresenta o mesmo diagrama de classes de análise da Figura 3.4 utilizando um pacote para expressar a composição de todo o sistema.

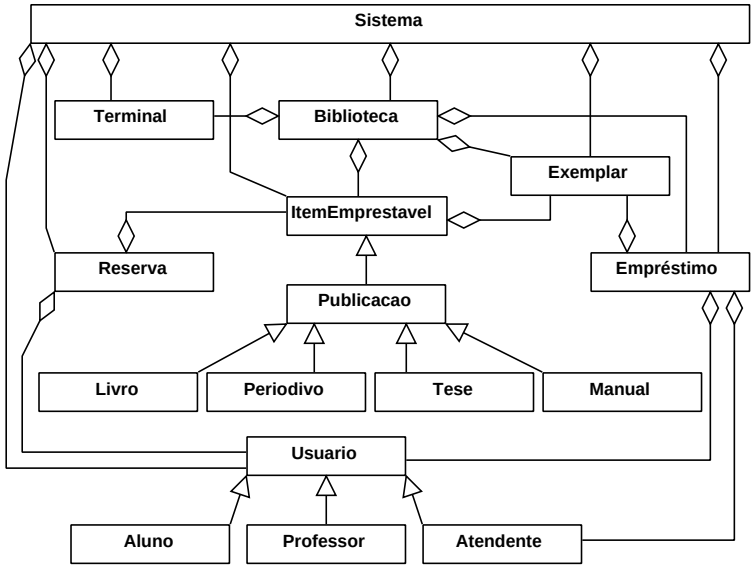


Figura 3.4: Diagrama de Classes Inicial de Análise

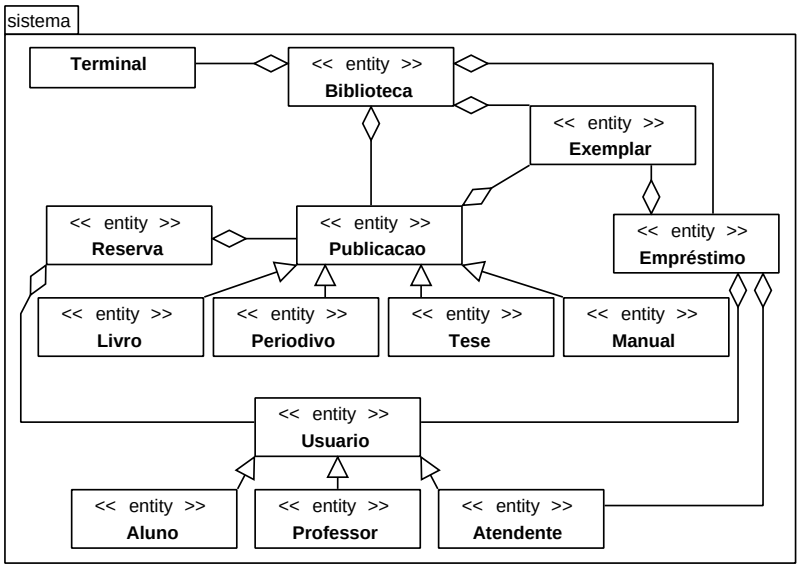


Figura 3.5: Diagrama de Classes Inicial de Análise (com Pacotes)

3.8 Atividade 4: Identificar/Atributos das Classes de Análise

No último estágio da modelagem estática, são identificados os atributos das classes de análise. Porém, nem todos os atributos precisam ser encontrados durante a análise; é esperado que uma parte deles pertença à solução e por isso devem ser identificadas durante a fase de projeto. Os atributos que são identificados durante a fase de análise, ou são inerentes ao domínio da aplicação,

ou são aqueles que representam informações relevantes para a realização dos casos de uso, isto é, aqueles que representam informações explicitadas na especificação dos casos de uso. Sendo assim, esses atributos quase sempre estão associados a classes de entidade. Como será visto a seguir (Seção 3.11), podem existir classes responsáveis unicamente pela coordenação da execução dos serviços. Essas classes normalmente só precisam ter estado se for necessário guardar informações relativas à interação com um certo usuário, por exemplo informações de seções, utilizadas no caso de vários usuários acessarem o sistema ao mesmo tempo. Mas normalmente esse tipo de informação só é levado em consideração e identificado a partir da fase de projeto.

A identificação dos atributos é feita de maneira análoga à identificação das classes, isto é, os atributos normalmente são identificados através da análise do domínio e do estudo das especificações dos casos de uso do sistema, do enunciado do problema e do dicionário de dados. De uma maneira geral, eles podem ser identificados a partir das classes candidatas descobertas através da análise textual desses artefatos, mais especificamente, a partir das classes descartadas no refinamento do diagrama de classes de análise, realizado pela Atividade 1.2 (Seção 3.5.2). Esses atributos normalmente representam conceitos simples que podem ser expressados usando-se tipos primitivos, como inteiros e caracteres. Quando não é o caso, é comum criar novas classes que representam registros de dados, onde cada atributo representa um campo do registro. Para enfatizar a sua importância, alguns autores dão a essas classes o estereótipo << *datatype* >>. Exemplos de classes que normalmente definem tipos de dados são `Endereço`, `Cor`, `Telefone`, `Ponto cartesiano`, etc. [29].

De um modo geral, se um atributo de uma classe é muito complexo, provavelmente ele deveria ser definido como uma classe à parte. Depois, se for necessário, as classes de atributos podem ser facilmente transformadas em atributos na fase de projeto do sistema.

3.9 Atributos das Classes de Análise no Estudo de Caso

Olhando rapidamente a lista de classes eliminadas na Seção 3.5.1, percebemos que várias delas não se tornaram classes de análise por representarem, na verdade, atributos de outras classes. Além dessas informações, é possível extrair algumas outras examinando a especificação do dicionário de dados e os atributos inerentes ao domínio da aplicação. A Tabela 3.2 associa os atributos identificados às classes correspondentes.

Algumas informações foram modificadas para se tornarem mais simples e de acordo com as orientações descritas na seção anterior. Por exemplo, o período de um empréstimo foi definido através de dois atributos: a data em que o empréstimo foi realizado (`data de empréstimo`) e a data esperada para devolução `data de devolução`.

Também podem ser adicionados ao modelo de análise atributos "óbvios", relativos ao domínio e que não apareceram na descrição dos casos de uso analisados. Deve-se ter cuidado, porém, com a definição do que é ou não inerente ao domínio. Por exemplo, no nosso estudo de caso, é perfeitamente coerente adicionar o atributo "Título" à classe `Publicação`, já que qualquer publicação tem um título. Por outro lado, apesar de tentador, não é adequado adicionar um atributo "Au-

Tabela 3.2: Atributos identificados para o sistema de bibliotecas

Classes de Análise	Atributos
Empréstimo	data de empréstimo data de devolução
Usuário	status número de registro
Publicação	número de tombo
Item Emprestável	identificador status
Reserva	data inicial data final

tor" a essa classe, já que, entre as publicações com as quais o sistema deve lidar, estão periódicos, publicações para as quais o conceito de autor normalmente não faz sentido.

A Figura 3.6 apresenta o diagrama de classes de análise depois de adicionados os atributos identificados nesta seção.

3.10 Atividade 5: Iterar e Refinar

Por se tratar de um processo de desenvolvimento iterativo, o sistema é construído gradativamente, a partir de refinamentos sucessivos dos modelos produzidos. Essa construção gradual do sistema proporciona uma distribuição da sua complexidade, o que além de permitir mudanças tardias dos requisitos, aumenta a qualidade final do produto final.

Nesta seção, serão mostradas as atividades onde houve algum refinamento do diagrama inicial de classes produzido na primeira iteração (Figura 3.6).

3.10.1 Atividade 1 (iteração 2): Identificar Classes de Análise

O processo RUP sugere que classes de análise sejam divididas em três grupos, a fim de classificar os elementos de acordo com o seu papel no modelo. O principal objetivo dessa classificação é simplificar a transição da análise para o projeto. Os três grupos definidos são: classes de entidade, classes de controle e classes de fronteira. Essa "classificação" das classes de análise pode ser adotada desde

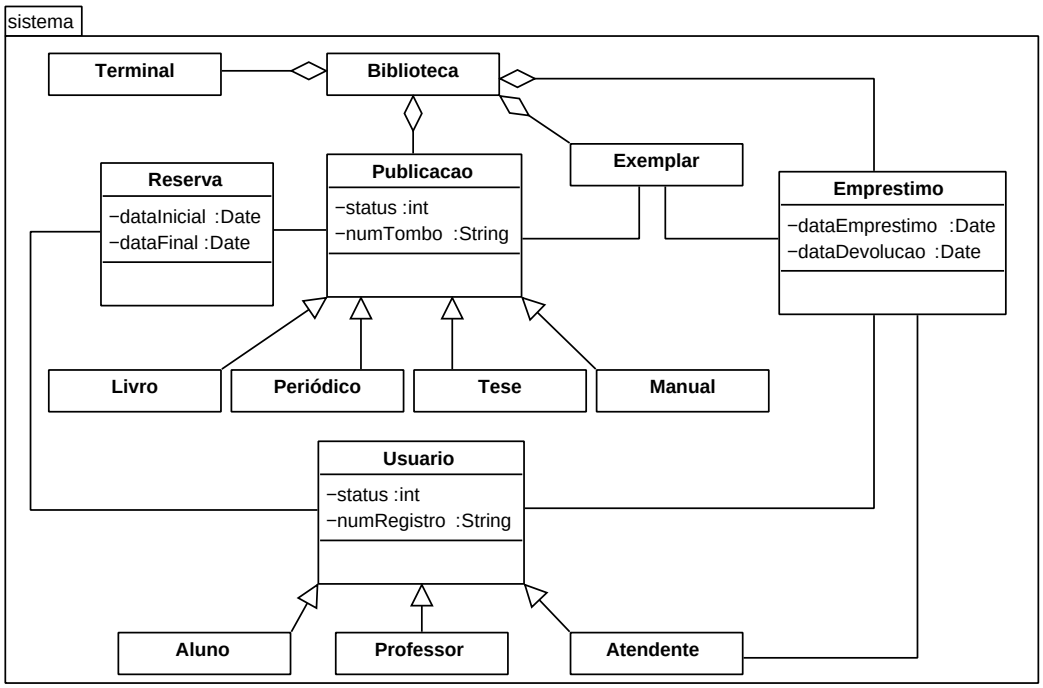


Figura 3.6: Diagrama de Classes Inicial de Análise com Relacionamentos e Atributos

o início da modelagem estática, embora seja especialmente útil durante a modelagem dinâmica, quando diagramas de interação envolvendo as classes de análise são construídos. Em um contexto mais geral, a divisão entre entidade, controle e fronteira independe do processo utilizado e no contexto da orientação ao objetos é considerado um padrão de análise chamado MVC (do inglês *Model-View-Controller*). Um padrão é uma solução para um problema bem conhecido que já foi empregada e validada em diversas ocasiões distintas. O padrão MVC pode ser usado em diversos níveis de abstração diferentes, como projeto arquitetural, projeto detalhado e implementação. O objetivo principal desse padrão é separar explicitamente os dados (entidades), os controladores (controle) e as interfaces (fronteiras). No contexto do MVC, as interfaces representam os pontos de acesso entre o sistema e os seus elementos externos (usuários, dispositivos de hardware e outros sistemas).

Classes de entidade modelam informações que devem ser armazenadas e o comportamento associado a elas. Objetos de entidade (instâncias de classes de entidade) são usados para guardar e atualizar informações sobre os elementos de dados relevantes para o sistema, como por exemplo um evento, uma pessoa, ou algum objeto da vida real. Os dados dessas entidades normalmente são persistentes, isto é, são armazenados em repositórios permanentes, por exemplo, um banco de dados. Porém, os detalhes relacionados à persistência dos dados em disco são específicos de tecnologia e por isso só devem ser considerados durante a fase de projeto.

As classes de controle são os locais onde é implementada a lógica do negócio, uma vez que elas modelam o comportamento de controle para um ou mais casos de uso. Essas classes podem contribuir para o entendimento do sistema porque representam seus aspectos dinâmicos, coordenando

as principais tarefas e fluxos de controle entre as classes do sistema.

Uma classe de fronteira, como o próprio nome indica, representa um ponto de acesso para o sistema a partir do meio externo. Por possibilitar esse ponto de acesso, as classe de fronteira modelam a interação entre os elementos internos do sistema e os atores que se comunicam com ele. Essa interação envolve entre outras coisas, a transformação e a tradução de eventos. Sendo assim, são as únicas partes do sistema que conhecem e dependem dos elementos externos. Conseqüentemente, uma mudança na interface com o usuário ou no protocolo de comunicação deve implicar em mudanças apenas nas classes de fronteira e não nas de controle ou entidade. Posteriormente, na fase de projeto, o papel das classes de fronteira é normalmente atribuído à própria interface com o usuário. Mas devido à sua relevância do ponto de vista conceitual, essas classes são representadas desde o diagrama de classes de análise.

Para cada tipo de classe de análise, o RUP define um estereótipo. Classes de entidade são identificadas pelo estereótipo $<< entity >>$, classes de controle pelo estereótipo $<< control >>$ e classes de fronteira pelo estereótipo $<< boundary >>$. Cada um desses estereótipos tem uma representação gráfica, conforme mostrado na Figura 3.7.

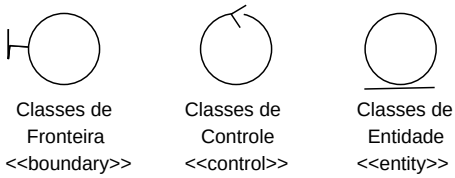


Figura 3.7: Representações Gráficas para Estereótipos de Classes de Análise

Além de definir as três categorias de classes já apresentadas, o padrão MVC define algumas restrições de relacionamento entre elas. A Figura 3.8 mostra as ligações possíveis entre classes estruturadas de acordo com o padrão MVC. As classes de fronteira só podem se associar às classes de controle. Essa restrição visa impedir, por exemplo, que uma classe de fronteira coordene a execução de várias classes de entidade, de modo a implementar as regras de negócio da aplicação, o que é papel exclusivo das classes de controle. Classes de entidade, de modo geral, também só se associam às classes de controle. Há casos especiais, porém, em que vale a pena criar uma associação direta entre duas classes de entidade. Normalmente, essa associação entre entidades pode ser justificada quando representa um relacionamento de agregação. Um exemplo bastante comum é a ligação entre um cliente e suas contas em um sistema bancário. Nessa situação, o fato de um `Cliente` poder "possuir" `Conta`s pode ser interpretado como uma agregação, o que justifica a permanência da associação. Esse tipo de decisão deve ser tomado com cuidado, porém, já que (i) a classe `Cliente` fica responsável por manter a consistência da associação, o que pode torná-la muito complexa, e (ii) essa ligação pode exigir uma associação bidirecional entre as classes `Cliente` e `Conta`. Se a complexidade de uma associação entre duas classes de entidade for muito grande, vale a pena pensar na possibilidade de criar uma nova classe controladora só para gerenciar essa interação.

Revisando a lista de classes de análise da iteração anterior, percebe-se a necessidade de se

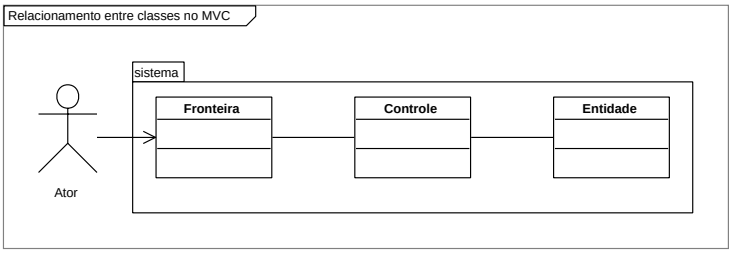


Figura 3.8: Ligações Possíveis entre as Classes do Padrão MVC.

adicionar uma nova classe ao modelo: a classe `Controlador`, que encapsulará as regras de negócio que o sistema deve implementar. Na especificação de requisitos vista no Capítulo 2, essas regras são representadas pelos fluxos dos casos de uso do sistema.

3.10.2 Atividade 3 (Iteração 2): Identificar os Relacionamentos entre as Classes

Após refinar o modelo de análise com o padrão MVC, é necessário verificar a necessidade de refinar as associações entre as classes, de acordo com as diretrizes apresentadas na Seção 3.7.1. Para isso, é necessário analisar o modelo de duas formas complementares: (i) verificar a necessidade de ligação entre as novas classes (no nosso exemplo, a classe `Controlador`) e as classes já existentes; e (ii) verificar a necessidade das associações anteriores, após a adição das classes novas. A ligação entre `Terminal` e `Controlador` pode ser compreendida, uma vez que o terminal, que faz a interface com os usuários, precisa estar ligado à parte do sistema responsável por suas regras de negócios, para que operações, tais como empréstimo e consulta de fato sejam efetuadas. Além disso, tanto o enunciado do problema quanto as descrições dos casos de uso deixam claro a necessidade de se ter os dados relativos à `Publicacao, Exemplar, Emprestimo, Reserva` e `Usuario` para efetivar operações de reserva e empréstimo. Por esse motivo, a classe `Controlador` necessita estar associada às classes de entidade citadas.

A Figura 3.9 apresenta o diagrama de classes da iteração anterior (Figura 3.6), com algumas modificações. A classe `Controlador` foi adicionada ao modelo, assim como as novas associações e os símbolos correspondentes aos estereótipos do padrão MVC.

3.11 Padrões de Modelagem

Desenvolvedores experientes de sistemas orientados a objetos construíram ao longo dos anos um repertório de soluções para problemas comuns encontrados durante o desenvolvimento. Para serem consideradas como padrões, essas soluções, devem ser validadas através do uso em diversas circunstâncias distintas, especificadas de uma maneira estruturada que descreve tanto a solução quanto o problema que ela se propõe a resolver.

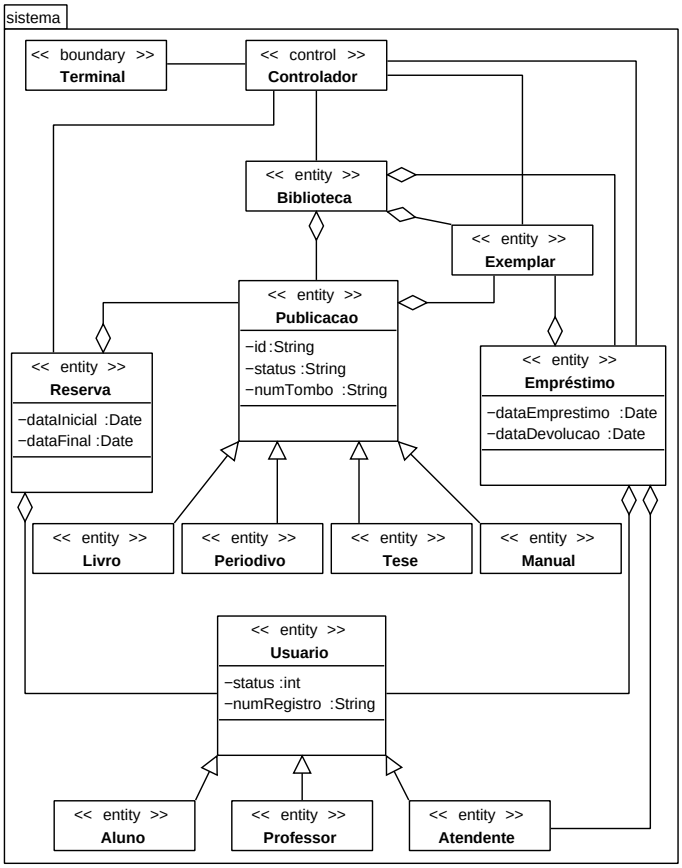


Figura 3.9: Diagrama de Classes de Análise Refinado

Inicialmente, os padrões surgiram no contexto do projeto arquitetônico, na construção civil [1]. Segundo Christopher Alexander, cada padrão descreve um problema que ocorre recorrentemente em nosso ambiente e uma solução para ele, de tal maneira que se pode usar essa solução várias vezes e de diversas maneiras [1].

No desenvolvimento de software, padrões podem ser encontrados em diversas circunstâncias diferentes. Padrões de projeto foram os primeiros a aparecer [19] e não demorou muito para que uma miríade de outros, como padrões arquiteturais [10], análise [18] e de implementação [29] surgissem. Atualmente, padrões de variados tipos são empregados em projetos de desenvolvimento para tornar os sistemas construídos mais fáceis de entender e manter, além de facilitar a reutilização de suas partes.

Em geral, um padrão tem quatro elementos fundamentais:

1. **Nome do padrão**: um identificador que podem ser utilizados para descrever um problema, sua solução e suas conseqüências em uma ou duas palavras.

2. **Problema**: uma descrição de quando aplicar o padrão. Ele explica o problema que o padrão

resolve, bem como seu contexto.

3. **Solução**: uma descrição dos relacionamentos, responsabilidades e colaborações que compõem o padrão e explica como esses elementos são combinados, a fim de resolver o problema.

4. **Conseqüências**: os resultados de se utilizar o padrão. Essas conseqüências normalmente envolvem escolhas que influenciam na decisão de utilizar ou não o padrão.

Um exemplo de padrão de análise que discutimos neste capítulo foi o padrão Controlador [29], apresentado como parte do padrão de análise MVC (Seção 3.10.1). Este padrão consiste em atribuir a uma classe a responsabilidade de tratar os eventos do sistema, sendo responsável por implementar a lógica do negócio. Esse padrão de análise é descrito resumidamente a seguir:

> **Nome**: Controlador
> **Problema**: definir qual classe implementa as regras do negócio e é responsável por lidar com os eventos do sistema.
> **Solução**: atribuir essa responsabilidade a uma classe cujo propósito é justamente lidar com esses eventos e implementar as regras de negócios. Essa classe representa a "inteligência" do sistema.
> **Conseqüências**: (i) aumenta o potencial para reutilização, já que separa o controle de outras características do sistema, como a interface com o usuário e a representação dos dados; e (ii) melhorar a manutenibilidade do sistema, uma vez que a rápida localização do controle facilita evolução das regras de negócio.

3.12 Resumo

O diagrama de classes de análise é o principal artefato da fase de análise orientada a objetos. O primeiro passo para a especificação desse diagrama é modelar a sua estrutura, através da identificação das classes que compõe o sistema, os atributos dessas classes e os relacionamentos entre elas. Este capítulo apresentou uma maneira sistemática de partir dos documentos de requisitos apresentados no Capítulo 2, como por exemplo o modelo de casos de uso, e através de diretrizes e refinamentos sucessivos, construir o diagrama de classes de análise. Por se preocupar prioritariamente com os aspectos estruturais do problema a ser resolvido, essa etapa da análise orientada a objetos é conhecida como modelagem estática do sistema.

A partir do enunciado do problema e das descrições dos casos de uso, o processo apresentado neste capítulo propõe três maneiras complementares de identificar as entidades relevantes para representar o problema: (i) lista de categorias conceituais, que consiste na utilização de uma lista de categorias comuns a vários sistemas, como *checklist* para ajudar na identificação de algumas entidades básicas; (ii) análise textual, através da análise dos documentos de requisitos, com o objetivo de identificar substantivos (possíveis classes ou atributos) e informações de estado (possíveis atributos); e (iii) análise de domínio, que possibilita a identificação de entidades e atributos inerentes a um conjunto de sistemas de um mesmo domínio.

Com as entidades identificadas, o próximo passo é identificar os relacionamentos entre elas. Conforme apresentado na Seção 1.3 do Capítulo 1, os possíveis relacionamentos entre classes são: generalização/especialização, agregação/decomposição, associações e dependências. Apesar de todos os relacionamentos serem importantes, do ponto de vista da análise estática, existem dois principais: generalização/especialização e agregação/decomposição. A importância desses dois relacionamentos é relativa ao seu caráter conceitual, que um enfoque fortemente semântico. Os demais relacionamentos podem ser identificados naturalmente na fase de modelagem dinâmica do diagrama de classes, que será apresentada no Capítulo 5.

Com a identificação do relacionamento entre as classes, está construído o diagrama inicial de classes de análise, também conhecido como modelo conceitual. Para finalizar a etapa de modelagem estática do sistema, o modelo conceitual deve ser refinado com padrões de análise. O processo apresentado neste capítulo recomenda a utilização dos padrões MVC (do inglês *Model-View-Controller*) e Controlador. Juntos, esses padrões possibilitam uma melhor separação de interesse no modelo de análise. Essa separação é realizada através de duas restrições importantes: (i) classificação das classes do modelo em três categorias (entidade, controle e fronteira); e (ii) definição de regras de relacionamento entre as categorias.

As classes de entidade, que recebem o estereótipo << *entity* >>, são os elementos do modelo conceitual que representam entidades da solução, diferentemente de processos ou da lógica do negócio. As classes com esse perfil de controle das ações e lógica do negócio são classificadas com o estereótipo << *control* >>. Finalmente, para explicitar os pontos de interação entre o sistema e os seus atores relacionados (humanos ou outros sistemas), as classes que desempenham esse papel recebem o estereótipo << *boundary* >>. Segundo o padrão de análise MVC, um diagrama de classes deve possuir pelo menos uma classe de cada uma dessas categoria.

Após essa classificação é feito um refinamento dos relacionamentos entre as classes, de forma a atender as três restrições seguintes: (i) as classes de entidade só possuem associação simples com classes de controle e só podem se relacionar com agregação com outras classes de entidade; (ii) as classes de fronteira só se comunicam com classes de controle, o que acontece através de associações simples; e (iii); as classes de controle podem possuir associações simples para qualquer outra classe, inclusive outros controladores.

Com o modelo estátido de análise pronto, o analista está pronto para iniciar a modelagem dinâmica, que será o foco de estudo do Capítulo 5. Mas antes é importante compreender alguns conceitos mais avançados do paradigma de objetos. Esses conceitos, apresentados nos Capítulos 4 e 4, podem ser utilizados para refinar o diagrama de classes no seu estado atual.

3.13 Exercícios

1. Complemente o modelo do sistema da biblioteca exibido na Figura 3.9, de modo a contemplar todos os requisitos do sistema de gerenciamento de bibliotecas, descrito no Capítulo 2.

2. A seguir, é apresentada a descrição de um sistema de controle de reservas e ocupações, que

será utilizado pelo hotel Astória. Construa um modelo de objetos que melhor represente esse sistema. Identifique as classes, as hierarquias, e possíveis atributos e operações das suas classes.

Descrição:
O Hotel Astória tem 5 salas de palestras (numeradas de 1-5) e 40 quartos (numerados de 6-45). Os quartos de 6-15 são "single" e os quartos de 16-45 são "double". Quando o cliente entra no hotel, eles são alocados para o primeiro quarto disponível do tipo requerido por ele. Além disso, o cliente preenche uma ficha com seus dados pessoais juntamente com o nome do pagador (isto é, quem efetivamente está pagando pelo quarto). Se é o próprio cliente quem vai pagar pelo seu quarto, este campo é preenchido como "privado"; caso contrário, o nome da companhia ou organização é anotado. As tarifas para um quarto "single" é de R$100,00, para um quarto "double" é de R$200,00 e para uma sala de palestras é de R$300,00. Existe apenas 1 conjunto de equipamentos de apresentação, que inclui 1 canhão, 1 laptop, e 1 microfone, no hotel que pode ser movido entre as salas de palestras.

O sistema de controle de reservas dos cômodos do hotel permite que um cliente seja alocado para um quarto disponível e garante que o cômodo esteja disponível para futuras reservas assim que o cliente sai do hotel. Suponha que o sistema não lide com datas, de forma que as entradas/saídas dos clientes e as mudanças do equipamento entre as salas sejam puramente eventos que ocorrem em tempo de execução. O sistema é capaz de fornecer as seguintes informações na tela: (i) quantos cômodos estão sendo correntemente ocupados; (ii) os números dos quartos correntemente ocupados e detalhes dos seus hóspedes; (iii) os números das salas de palestras correntemente ocupadas; e (iv) qual a sala de palestra que contém o equipamento.

3. O enunciado abaixo descreve um sistema de controle de tráfego aéreo simplificado. Nós assumimos que o enunciado do problema foi obtido através de uma análise de domínio envolvendo entrevistas com especialistas e usuários finais.

   Leia cuidadosamente o enunciado e especifique o caso de uso Pousar Avião. A partir dessa especificação, realize a análise do caso de uso. O resultado dessa atividade deve ser um diagrama de classes de análise contendo as classes, relacionamentos entre elas e atributos de cada uma.

   Enunciado do Problema:
   Aeronaves fazem vôos entre aeroportos, e um vôo é sempre entre 2 aeroportos, um origem e um destino. Cada vôo tem um número de vôo. Aeroportos variam o seu número de pistas, heliportos, terminais e hangares, mas devem ter pelo menos uma pista e também uma torre de controle. Cada aeronave, aeroporto, pista e terminal devem ter um identificador único para permitir que o controle de tráfego aéreo gerencie. Alguns hangares são puramente para armazenamento de aeronaves, enquanto que outros são hangares de manutenção e tem facilidades adicionais, como por exemplo, a habilidade de servir as aeronaves.

   Um aeronave pode ser um helicóptero ou um avião, e pode ter comportamentos diferentes, como por exemplo, a habilidade do helicóptero decolar e aterissar em um

heliporto, sem precisar de uma pista. Entretanto, eles também têm características em comum, como por exemplo, necessidade de um piloto antes do vôo começar. Vôos carregam pessoas e suas bagagens, e uma aeronave que carrega passageiros deve ter uma lista de passageiros detalhando todos os passageiros e os respectivos números da passagem num arquivo indexado.

Todos os vôos são registrados numa base central onde eles podem ser acessados pelos controladores (ou operadores) do tráfego aéreo nas torres de controle dos aeroportos. Aeronaves são rastreadas por radar e são dadas instruções de decolagem e aterissagem quando estão dentro das áreas controladas por aeroportos específicos. Uma aeronave chegando pode ser colocado numa fila caso uma pista ou um heliporto não esteja disponível. Aeroportos podem ser fechados devido ao mau tempo; nesse caso as aeronaves que chegam são desviados para outros aeroportos.

4. Um proprietário de restaurante acredita que a automação poderá fazer com que seu negócio se torne mais eficiente, atraindo mais clientes. Ele deseja automatizar a maior quantidade possível de operações. Devem ser mantidos manuais dos processos de preparar os pratos e todos os pedidos devem ser anotados. Construa o diagrama de classes de análise, sabendo que através de entrevista e questionários, foram identificadas as seguintes informações:

   - **Entidades envolvidas**:
     - Proprietário do restaurante;
     - Cliente (garçons/ operador de caixa);
     - Fornecedor.
   - **Objetivos do sistema**:
     - Processar os pedidos e emitir notas;
     - Automatizar contabilidade (pagamento de fornecedores, relatórios contábeis);
     - Fazer pedido de compras aos fornecedores da maneira mais eficiente, de forma que os produtos não faltem e que também não "encalhem" no estoque.
   - O proprietário deseja que o sistema controle o estoque, para facilitar a verificação do balanço entre os valores real e esperado.
   - O cliente também deseja ter algumas estatísticas sobre vendas de diferentes itens.
   - **Idéias preliminares**:
     - Manter informações sobre fornecedores, contabilidade, pedidos e menu;
     - O controle de estoque deve usar as informações sobre o total da compra e sobre as vendas realizadas, baseado na quantidade estimada dos produtos em cada prato, que é obtida a partir de informações do menu.

Capítulo 4

Modelagem Estrutural em UML

4.1 Objetos e Classes

Conforme apresentado no primeiro capítulo deste livro (Seção 1.3.1), um objeto é uma entidade utilizada para representar elementos relevantes para modelar um problema. Cada objeto possui um estado, definido através de um conjunto de atributos, e um comportamento, definido através de um conjunto de operações. Essas operações são implementadas através de **métodos** e o conjunto de operações e atributos públicos constituem a interface pública de um objeto. A comunicação entre dois objetos de um sistema acontece através do envio de mensagens. Como um objeto não deve possuir atributos públicos, o envio de uma mensagem por um objeto deve implicar na execução de uma operação da interface pública do objeto recipiente. Esse envio de mensagem consiste na única maneira de comunicação ente os objetos do sistema.

Como visto na Seção 1.3.2 do Capítulo 1, uma classe é a descrição de um molde que especifica as propriedades e o comportamento relativo a um conjunto de objetos similares. No modelo de objetos, atributos e operações são usualmente parte da definição de uma classe.

4.1.1 Classes Concretas

Como visto no Capítulo 1 deste livro (Seção 1.3.4), as unidades básicas de desenvolvimento do modelo de objetos são classes e objetos.

A Figura 4.1 mostra a representação em UML para a classe Publicacao, que representa uma publicação de uma biblioteca. Como pode ser visto, o estado de uma publicação é definido através de dois atributos: numeroTombo e nome. O comportamento, por sua vez, é definido através de quatro operações: Publicacao(), responsável por criar objetos da classe, o que é conhecido por

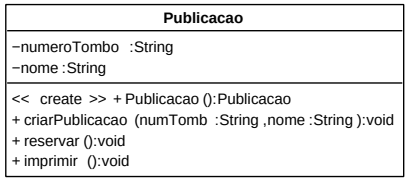


Figura 4.1: Classe que representa as publicações de uma biblioteca

instanciação da classe, além das operações criarPublicacao(numTomb:String, nome:String), que inicializa os valores do estado do objeto criado, reservar() e imprimir().

4.1.2 Instanciação de Objetos em UML

Como apresentado na Seção 1.3.3 do Capítulo 1, é possível criar um número qualquer de objetos relacionados a uma classe. Para isso, o modelo de objetos define o relacionamento de classificação/instanciação.

A Figura 4.2 mostra a representação em UML para classificação/instanciação. Como pode ser percebido, um objeto é instância de apenas uma classe, enquanto uma classe pode ter várias instâncias associadas a ela. Neste exemplo, os objetos p1 e p2 são instâncias da classe Publicacao. O fato dos objetos dependerem da sua classe é representada através de dependências UML (setas pontilhadas). O relacionamento de dependência foi apresentado na Seção 1.3.8 do Capítulo 1.

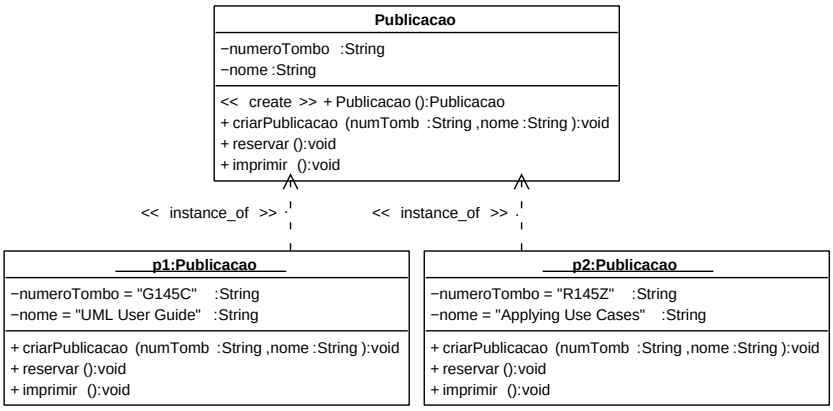


Figura 4.2: Instanciação de dois objetos da classe Publicacao

Nas linguagens de programação modernas, os objetos podem ser alocados dinamicamente, isto é, os endereços de memória são referenciados em tempo de execução. Em Java, por exemplo, a criação de um objeto é sinalizada através da palavra reservada `new`.

Apesar do ganho de flexibilidade e expressividade oferecido pela alocação dinâmica, ela obriga que o programador se preocupe explicitamente com a alocação e desalocação dos recursos da

memória. Exceções à obrigatoriedade de desalocação explícita são as linguagens que implementam mecanismos de coleta automática de lixo (do inglês *garbage collection*), como por exemplo a linguagem Java.

O exemplo a seguir mostra a alocação dinâmica de um objeto em Java:

```
1     ...
2 Publicacao  p1;
3 p1 = new Publicacao();
4 p1.criarPublicacao(''G145C'',''UML User Guide'');
5 p1.imprimir();
6     ...
```

No trecho de código anterior, `p1` é um apontador para um objeto do tipo `Publicacao` (Linha 2). Inicialmente, ele aponta para `null` (endereço nulo). Na Linha 3 é criado um novo objeto do tipo `Publicacao` e `p1` passa a apontar para ele através da operação de atribuição (`''='`'). A mensagem `criarPublicacao()` é enviada para `p1` (Linha 4) a fim de atribuir valores iniciais aos atributos `numeroTombo` e `nome`.

Como pôde ser percebido no exemplo em Java, a sintaxe para a invocação de uma operação é uma combinação de acesso a uma estrutura de dados (o objeto) e a invocação de um procedimento. A razão disso é que uma operação está associada a uma instância de classe específica e tem que existir uma maneira de dizer para a operação em qual instância ela deve operar. Isto é feito como segue:

$< nomeObjeto > . < nomeOperacao > (argumentos)$

Para fixar melhor o conceito, a Figura 4.3 apresenta a relação entre a criação da variável (o apontador `p1` do exemplo de Java) e a instanciação do objeto.

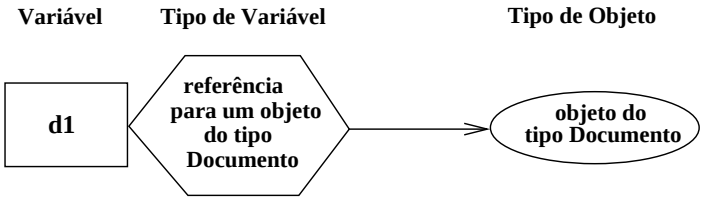


Figura 4.3: Criação de um Objeto

Cada objeto criado contém seus próprios valores dos atributos `numeroTombo` e `nome`, declarados na classe `Publicacao`. Objetos são manipulados através do envio de mensagens que acionam operações realizadas sobre o objeto alvo. Na Linha 5 do trecho de código Java, a mensagem `imprimir()` é enviada para o objeto referenciado por `p1`.

Na maioria das linguagens orientadas a objetos, existe uma distinção clara entre as variáveis e os objetos referenciados por elas. Essas linguagens adotam a alocação dinâmica, onde variáveis são

apenas nomes que contém referências (endereços) para objetos.

4.1.3 Operações Construtoras

Um **construtor** é uma operação especial, ativada automaticamente no momento da criação de um objeto da classe. Como apresentado na Figura 4.4, uma operação construtora possui nome idêntico ao da sua classe e não pode retornar nenhum resultado (nem mesmo `void`), embora possa receber valores através de um ou mais parâmetros. Quando não recebe nenhum parâmetro, dizemos que a operação representa um **construtor padrão** (primeira operação da Figura 4.4). Caso contrário, trata-se de um **construtor não-padrão** (segunda operação da Figura 4.4).

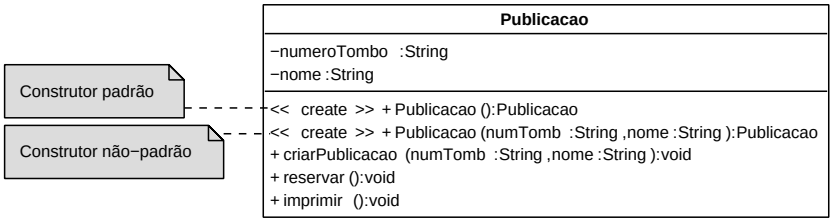


Figura 4.4: Classe Publicacao com duas Operações Construtoras

Na Linha 4 da listagem mostrada a seguir, é apresentado o método construtor padrão (implementação da operação) da classe Publicacao. A implementação do construtor não padrão é apresentada nas Linhas 5 a 8. Ao implementar as operações de uma classe, pode ser necessário enviar mensagens ao objeto que está sendo executado. Para isso, as linguagens de programação possuem variáveis especiais, definidas implicitamente e visíveis em todos os métodos da classe. Em Java, essa variável é denominada `this` (Linha 7).

Os métodos construtores são utilizados normalmente para atribuir valores iniciais aos atributos. Na maioria das linguagens orientadas a objetos, inclusive em Java, se o programador não especificar nenhum método construtor, a linguagem fornece um construtor padrão para a classe. Esse método padrão não possui nenhum parâmetro e atribui valores iniciais padrões aos atributos, de acordo com as suas definições de tipos.

```
1   public class Publicacao {
2     private String numeroTombo;
3     private String nome;
4     public Publicacao(){ }
5     public Documento(String numTomb, String nome) {
6       numeroTombo= numTomb;
7       this.nome = nome;
8     }
9     public void reservar() {
```

```
10          System.println(``Reserva a Publicacao'');
11        }
12        public void imprimir() {
13          System.out.println(``Imprime o conteúdo da Publicacao'');
14        }
15      }
```

Agora que `Publicacao` tem um construtor que recebe parâmetros, o método `criarPublicacao()` tornou-se desnecessário. Assim, a criação de uma publicacao passa a exigir apenas os dois passos a seguir:

```
1     Publicacao p1;
2     p1 = new Publicacao(``G145C'',``UML User Guide'');
```

4.1.4 Visilibilidade de Atributo e Operação

O modelo de objetos apresenta estratégias variadas para acesso e visibilidade de atributos e operações. Atributos e operações definidas como públicas (``+'') possuem acesso é irrestrito, isto é, podem ser acessadas por qualquer objeto do sistema. No caso dos atributos e operações com visibilidade privada (``-''), eles só são visíveis pelo próprio objeto. Em outras palavras, esses atributos e operações só são visíveis na construção dos métodos da classe.

Apesar do modelo permitir a definição de atributos públicos, uma recomendação para garantir o encapsulamento dos dados do objeto é a atribuição da visibilidade privada para todos os atributos da classe. Dessa forma, a interface pública do objeto deve ser constituída apenas de operações.

4.2 Tipo Abstrato de Dados

4.2.1 Tipos e Classes

Como apresentado na Seção 1.2.3 do Capítulo 1, o conceito de **tipo** pode ser visto como uma especificação de um conjunto de valores que podem ser atribuídos a uma variável, juntamente com as operações que podem ser usadas legalmente para criar, acessar e modificar tais valores. Por exemplo, o tipo `boolean` é associado aos valores *true* e *false* e às operações `AND`, `OR` e `NOT`.

No contexto de orientação a objetos, a definição de tipo deve ser apresentada em termos de objetos. Sendo assim, um **tipo** pode ser definido como as características comuns de uma coleção de objetos do sistema que respondem ao mesmo conjunto de mensagens. Ou seja, o tipo é o conceito

que agrupa uma coleção de objetos com a mesma interface pública [4]. Portanto, essa definição se alinha com o conceito de classes no modelo de objetos.

Entretanto, os conceitos de tipo e classe são distintos [7]. Um tipo é essencialmente a descrição de uma interface, que especifica o comportamento comum a um grupo de objetos. Enquanto isso, uma classe especifica uma implementação particular de um tipo. A definição de uma classe, além de conter a implementação das operações dos objetos, inclui uma descrição dos estados internos desses métodos. Portanto, apesar de uma classe apresentar implicitamente o conceito de tipo através dos seus métodos públicos, o papel da classe é implementar o(s) tipo(s) associado(s) a ela.

Embora tipo e classe não sejam a mesma coisa, muitos autores usam esses termos como sinônimos; outros acham que a diferença entre os dois é primordial. Algumas linguagens orientadas a objetos, como por exemplo Arche, POOL, Guide e Java, separam a definição de tipo e sua implementação. C++, Eiffel, e Modula-3 [22] não oferecem essa separação. A linguagem de modelagem UML também separa explicitamente tipos e classes, através do conceito de interface (`interface` em Java), que será discutido em detalhes na Seção 4.7.

4.2.2 Tipos Abstratos de Dados

As Seções 1.3.1 e 1.3.2 do Capítulo 1 e a Seção 4.1 deste capítulo enfatizaram a importante distinção entre tipos e classes, que é uma conseqüência da separação de interesse (do inglês *separation of concerns*) entre a especificação do comportamento e a sua implementação. O conceito de **tipo abstrato de dados** (TAD) utilizado neste livro (Seção 1.3.4 do Capítulo 1) considera que todos os atributos possuem visibilidade privada e por não serem visíveis externamente, não fazem parte da definição do tipo. Dessa forma, um tipo abstrato de dados consiste de um conjunto de operações, que devem ser implementadas por todos os elementos que pertençam a esse tipo.

Estruturalmente, um tipo abstrato de dados é constituído de duas partes (Figura 4.5): (i) especificação, isto é, a interface; e (ii) implementação. Cada parte, por sua vez, é subdividida. A especificação é denotada pela sua sintaxe (ou assinatura) juntamente com a sua semântica, enquanto que a implementação é denotada pela representação (ou estruturas de dados) e os seus algoritmos associados. Em UML, a especificação de um TAD é materializada através de interfaces, enquanto que a implementação é materializada através de classes.

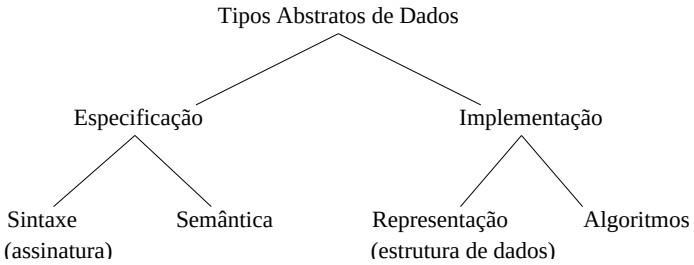


Figura 4.5: Estrutura de um Tipo Abstrato de Dados

O principal benefício decorrente da separação explícita entre a especificação de um tipo e a sua implementação é a garantia de abstração por parte do cliente. Sendo assim, o cliente pode fazer uso do TAD, mesmo desconhecendo os aspectos técnicos envolvidos na sua implementação. Outra conseqüência dessa separação é o baixo acoplamento existente entre esses conceitos, o que possibilita que um TAD tenha mais de uma implementação associada a ele.

Considere a classe Pilha, apresentada na Figura 4.6 e a especificação informal das suas operações, apresentada na Tabela 4.1.

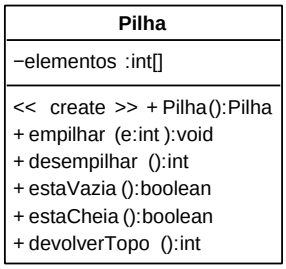


Figura 4.6: Classe Pilha, que define um TAD

Tabela 4.1: Especificação Informal do TAD `Pilha`.

Linguagem	Descrição
`Pilha()`	O construtor não recebe parâmetros de entrada, não devolve nenhum resultado e cria uma pilha vazia.
`empilhar(int e)`	Recebe um elemento inteiro para ser empilhado e não retorna nenhum resultado.
`int desempilhar()`	Não recebe parâmetros de entrada; desempilha o elemento do topo da pilha, do tipo `int` e o retorna.
`estaVazia()`	Não recebe parâmetros de entrada e devolve *true* se a pilha estiver vazia e *false*, caso contrário.
`estaCheia()`	Não recebe parâmetros de entrada e devolve *true* se a pilha estiver cheia e *false*, caso contrário.
`int devolverTopo()`	Não recebe parâmetros de entrada; inspeciona o topo da pilha e devolve uma cópia do elemento como resultado de retorno, que é do tipo `int`.

Um tipo abstrato de dados é normalmente definido através de uma interface, que no contexto

do modelo de objetos, representa as operações públicas de uma classe. Entretanto, apenas a especificação sintática não é suficiente para descrever o comportamento de um tipo abstrato de dados. Também é necessário algum suporte para especificar a sua semântica. Uma das técnicas mais bem sucedidas para se especificar a semântica de um tipo abstrato de dados é a especificação algébrica. Esta técnica requer que o projetista defina um conjunto de equações lógicas que expressem a semântica desempenhada pelas operações do tipo abstrato de dados.

A especificação sintática do TAD `Pilha` é apresentada a seguir, em forma de classe Java:

```
1  class Pilha {
2    // ... atributos
3    public Pilha() { ... }
4    public void empilhar(int e) { ... }
5    public int desempilhar() { ... }
6    public boolean estaVazia() { ... }
7    public boolean estaCheia() { ... }
8    public int devolverTopo() { ... }
9  } // fim da classe Pilha
```

Uma possível especificação semântica para o TAD `Pilha` usando a técnica de especificação algébrica é apresentada a seguir. Nessa especificação, assumimos que `p` é uma instância de `Pilha` e `e` é um elemento que pode ser empilhado. Além disso, é utilizada uma sintaxe semelhante ao OCL (do inglês *Object Constraint Language*), a linguagem formal da UML. Por exemplo, `p@pre` indica o estado inicial da pilha `p`, antes de executar qualquer operação representada:

(1) - `[Pilha()].estaVazia()` é verdadeiro.
(2) - `[p.empilhar(e)].estaVazia()` é falso.
(3) - `[p.empilhar(e)].desempilhar() == e AND [[p.empilhar(e)].desempilhar()] ⇒ [p == p@pre]]`
(4) - `[p.empilhar(p.desempilhar())] ⇒ [p == p@pre]`
(5) - `[p.devolverTopo()] ⇒ [p == p@pre]`

Outra técnica que auxilia o programador a especificar o comportamento de tipos abstratos de dados é baseada na definição de expressões que constituem as pré e pós-condições das operações, da mesma forma como apresentado no Capítulo 2 em relação aos casos de uso. Predicados (ou expressões booleanas) podem ser associadas a operações para descrever o estado desejável do objeto. Uma pré-condição para uma operação descreve o estado do objeto desejado antes do método ser executado. Se a pré-condição é verdadeira, o método pode ser executado. Se for falsa, o método não pode ser executado. Uma pós-condição, por sua vez, descreve o estado do objeto após a execução do método.

Por exemplo, um programador pode associar pré e pós-condições às operações `empilhar()` e `desempilhar()` para garantir a consistência da pilha, de acordo com a semântica esperada para

cada operação. Uma pré-condição para `empilhar()` é que a pilha não pode estar cheia, enquanto que uma pré-condição para `desempilhar()` é que a pilha não pode estar vazia. Similarmente, uma pós-condição para `empilhar()` é que a pilha não seja mais vazia e, para `desempilhar()`, que o número total de elementos seja decrescido de 1. A especificação das operações do TAD `Pilha`, usando pré e pós-condições, é apresentada na Tabela 4.2.

Tabela 4.2: Especificação do TAD `Pilha` Utilizando Pré e Pós-condições

Operação	Pré-condição	Pós-condição
empilhar()	!estaCheia()	!estaVazia()
desempilhar()	!estaVazia()	!estaCheia()
estaVazia()	TRUE*	Nenhuma mudança no estado da pilha
estaCheia()	TRUE	Nenhuma mudança no estado da pilha
devolverTopo()	!estaVazia()	Nenhuma mudança no estado da pilha
construtorPilha()	TRUE	estaVazia()

*TRUE significa que a operação pode ser chamada incondicionalmente.

Apesar de ser um conceito útil do ponto de vista da modelagem, poucas linguagens de programação dão apoio à especificação semântica do comportamento. O exemplo mais conhecido de uma linguagem de programação que apóia essa especificação é a linguagem Eiffel, que permite a especificação de pré e pós-condições associadas aos métodos das classes. Bar-David[4] descreve o comportamento de tipos abstratos de dados através de um conjunto de equações que relacionam a interface pública do tipo no contexto de C++. A fim de oferecer suporte à especificação dessas assertivas, a partir da sua versão 1.4, a linguagem Java incorporou a palavra reservada `assertion`. Uma *assertion* é utilizada para associar expressões lógicas que devem ser satisfeitas. Sendo assim, de acordo com o local da sua inserção, a assertiva pode significar tanto uma pré-condição (início do método), quanto uma pós-condição (final do método).

4.3 Relacionamento de Agregação e Associação

Agregação é um relacionamento estrutural entre o todo e suas partes [9]. Como visto na Seção 1.3.6, as hierarquias de agregação/decomposição devem ser empregadas para modelar relacionamentos do tipo parte/todo. Sendo assim, agregações são utilizadas para modelar entidades a partir das suas partes, o que permite a reutilização de comportamento entre elas.

A Figura 4.7 apresenta uma hierarquia de agregação na qual instâncias da classe `Biblioteca`

são compostas por instâncias das classes `Usuario` e `Publicacao`.

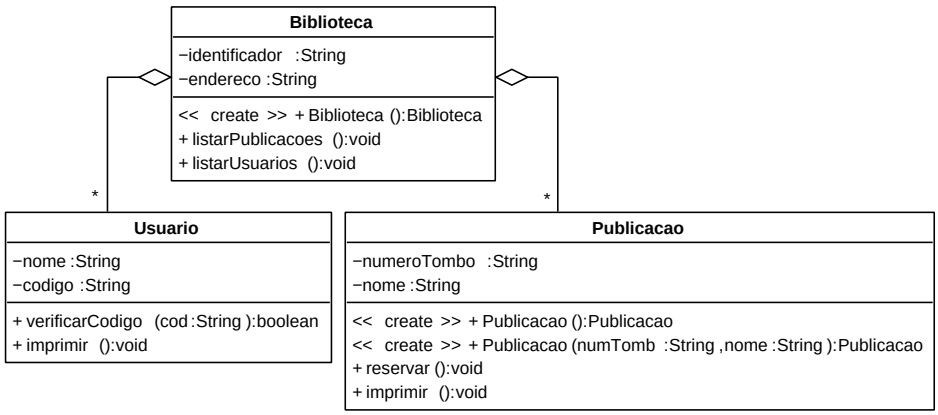


Figura 4.7: Hierarquia de Agregação de uma Biblioteca

Para implementar numa linguagem de programação orientada a objetos a hierarquia da Figura 4.7, uma classe estruturalmente descrevendo uma biblioteca é definida com atributos referenciando seus vários componentes. Esses atributos são iniciados com as instâncias correspondentes das classes Usuario e Publicacao. Esta abordagem também permite ligações com várias instâncias de uma classe para descrever um objeto agregado. Por exemplo, pode ser necessário referenciar mais de uma instância de Publicacao para definir o acervo da Biblioteca. O princípio de herança geralmente não permite que uma informação seja herdada mais do que uma vez a partir de uma mesma classe, ao contrário da agregação. O seguinte trecho de código apresenta uma implementação parcial para a classe Biblioteca.

```
1  public class Biblioteca {
2     //Atributos
3      private String identificador;
4      private String endereco;
5
6      //Agregações
7      public Publicacao[] publicacao;
8      public Usuario[] usuario;
9
10     //Metodos
11     public Biblioteca(){...} // Construtor padrão
12     public void listarPublicacoes() {
13        for(int i = 0; i<publicacao.length; i++)
14           publicacao[i].imprimir();
15     }
16     public void listarUsuarios() {
```

```
17          for(int i = 0; i<usuario.length; i++)
18              usuario[i].imprimir();
19      }
20  }
```

Como pode ser percebido na implementação das operações `listarPublicacoes()` e `listarUsuarios()` da classe Biblioteca, o objeto agregador pode propagar mensagens para os seus agregados (Linhas 14 e 18).

Para apresentar uma visão mais clara da estrutura dos objetos instanciados, a Figura 4.8 apresenta um diagrama de colaboração (Seções 2.9.11 do Capítulo 2) entre objetos dos tipos Biblioteca e Publicacao. Após o recebimento da mensagem listarPublicacoes() pelo objeto bc, ele imediatamente propaga a mensagem imprimir() para os objetos dp e cj.

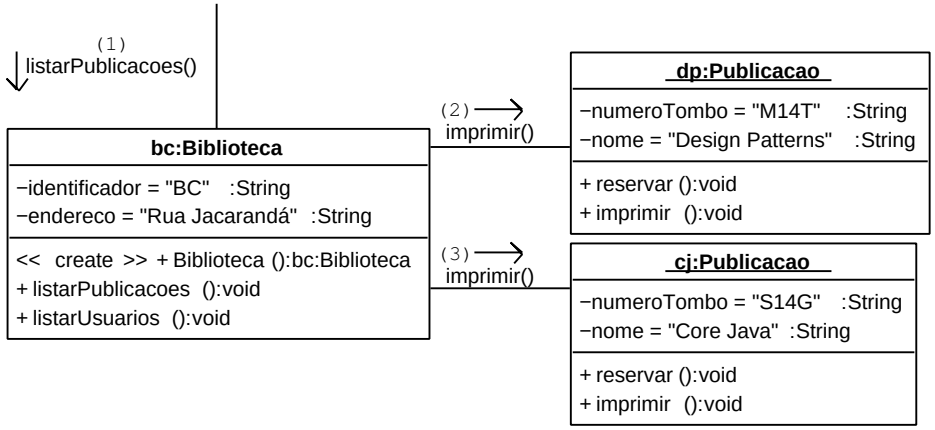


Figura 4.8: Diagrama de Colaboração com Propagação de Mensagem

4.4 Relacionamento de Generalização

4.4.1 Herança Simples de Classes

Herança (do inglês *inheritance* ou *subclassing*) é um mecanismo para derivar novas classes a partir de classes existentes através de um processo de refinamento. Uma classe derivada herda a representação de dados e operações de sua classe base, mas pode seletivamente adicionar novas operações, estender a representação de dados ou **redefinir** a implementação de operações já existentes. Uma classe base proporciona a funcionalidade que é comum a todas as suas classes derivadas, enquanto uma classe derivada proporciona a funcionalidade adicional que especializa o seu comportamento. Para uma classe derivada acessar as operações e atributos visíveis da classe base, ela pode se utilizar da

variável implícita `super`. De forma análoga ao `this` apresentado na Seção 4.1.3, o `super` é uma variável implícita que fica visível na implementação de todos os métodos da classe.

A hierarquia da Figura 4.9 especifica três tipos diferentes: (i) `Publicacao`, com as operações `reservar()` e `imprimir()`; (ii) `Livro`, com as operações `reservar()`, `imprimir()` e `bloquear()`; e (iii) `Periodico`, com as operações `reservar()`, `imprimir()` e `indexar()`. Além das classes que definem os tipos, foram instanciados dois objetos: l1, do tipo Livro e p1, do tipo Periódico.

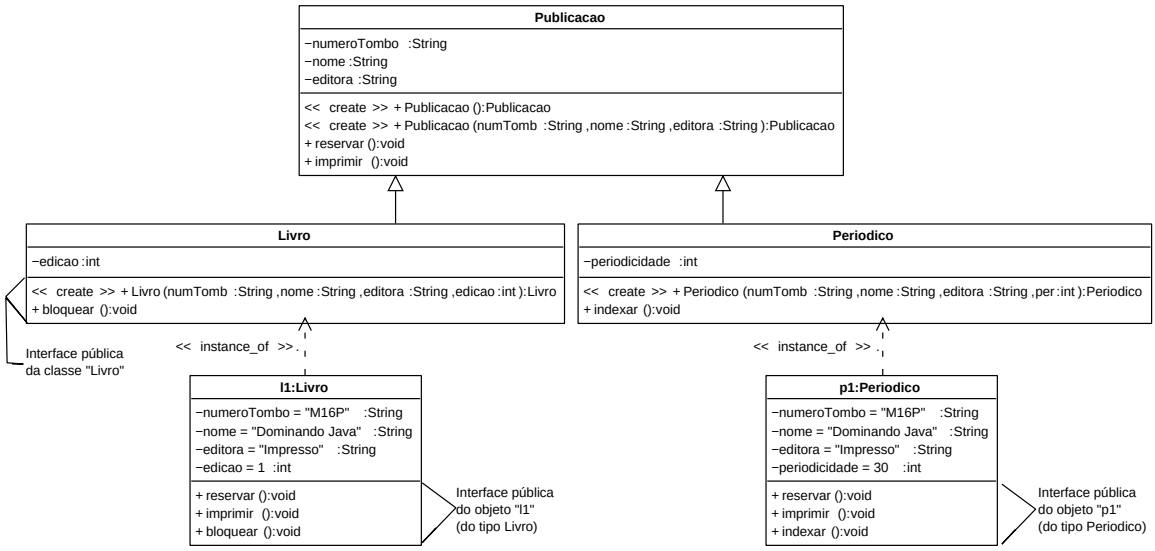


Figura 4.9: Exemplo de Herança de Classes

O estado do objeto `l1` contém quatro atributos: `numeroTombo`, `nome`, `editora` e `edicao`. Os atributos `numeroTombo`, `nome` e `editora` devem ser alterados apenas por operações da classe Publicacao, uma vez que são definidos nessa classe e possuem visibilidade privada. Por uma razão análoga, o atributo `edicao` só pode ser alterado por operações da classe Livro. As operações da classe Livro não "enxergam" os atributos e operações privadas da classe base e, portanto, ela não pode alterá-los. Para isso, é necessário utilizar as operações visíveis da classe Publicação, através da variável `super`. A interface pública do tipo Livro é estabelecida pelas operações válidas para objetos desse tipo, isto é, tanto a operação `bloquear()`, definida pela classe Livro, quanto as operações `reservar()` e `imprimir()`, definidas na classe Publicacao.

As operações construtoras são uma exceção à regra de herança, pois não são herdadas pelas classes derivadas. O construtor de uma classe tem que obrigatoriamente chamar um construtor de sua classe base. Em Java, a palavra reservada `super` significa "minha super classe" e o método `super(x)` significa "o método construtor da minha superclasse", onde `''x''` são os parâmetros passados para o construtor. Esses parâmetros são usados normalmente para inicializar os atributos da superclasse.

A terminologia usada para herança engloba vários termos que são usados como sinônimos:

- **Classe derivada** ou **subclasse** ou **classe filha**: é uma classe que herda parte dos

seus atributos e métodos de outra classe.

- **Classe base** ou **superclasse** ou **classe pai**: é uma classe a partir da qual classes novas podem ser derivadas.
- **Classes ancestrais** são todas as classes das quais se herdam atributos e métodos, independentemente da posição que elas ocupam na hierarquia de classes. As superclasses são exemplos de classes ancestrais.
- **Classes descendentes** são todas as classes que herdam o comportamento de uma classe específica, independentemente da posição que elas ocupam na hierarquia de classes. As subclasses são exemplos de classes descendentes.

4.4.2 Visibilidade Protegida de Atributos e Operações

A Figura 4.10 mostra uma hierarquia de classes de usuários de uma biblioteca. Na raiz dessa hierarquia está uma classe base genérica chamada Usuario, a partir da qual as classes Aluno e Professor herdam os atributos `nome`, `codigo` e `status`, além das operações `verificarCodigo()`, `imprimir()` e `mudarStatus()`.

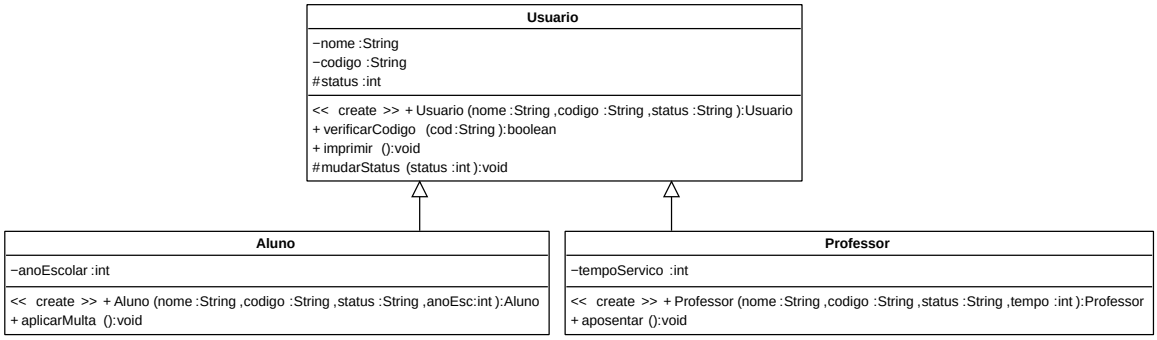


Figura 4.10: Hierarquia de Classes de Usuários

O código parcial da implementação dessa hierarquia em Java pode ser o seguinte:

```
1  public class Usuario {
2      //Atributos
3      private String nome;
4      private String codigo;
5      protected int status; // 1:inativo; 2:suspenso; 3:normal.
6
7      //Operações
8      public  Usuario(String nome, String codigo, String status) {...}
9      public boolean verificarCodigo(String cod) {...}
10     public void imprimir() {...}
```

```
11    public mudarStatus(int status){
12       this.status = status;
13    }
14 }
15
16 class Aluno extends Usuario {
17       private int anoEscolar;
18       public Aluno(String nome, String codigo, String status, int anoEsc){
19          super(nome, codigo, status);
20          anoEscolar = anoEsc;
21       }
22       public double aplicarMulta()  { // se estiver em atraso, suspende
23          double multa = ``valorMulta'' * ``diasAtraso'';
24          if (multa > 0)
25             status = 2; // alterando um atributo da superclasse
26          return multa;
27       }
28 }
29
30 class Aluno extends Professor {
31       private int tempoServico;
32       public Aluno(String nome, String codigo, String status, int tempo){
33          super(nome, codigo, status);
34          tempoServico = tempo;
35       }
36       public void aposentar()  { // se estiver em atraso, suspende
37          if (tempoServico >= 30)
38            status = 1; // alterando um atributo da superclasse
39       }
40 }
```

As classes Aluno e Professor estão alterando um atributo definido pela classe base. Isso somente é possível porque a visibilidade do atributo saldo é estabelecida como protegida (``#'' em UML, ou `protected` em Java) na classe Usuario. Se a visibilidade do atributo `status` fosse alterada para privada (`private`), ao invés de protegida, a implementação da operação `aplicarMulta()` da classe `Aluno` deveria ser modificada, de modo a evitar o acesso direto ao atributo. Uma possibilidade dessa nova versão da operação `aplicarMulta()` é apresentada a seguir:

```
1       public double aplicarMulta()  { // se estiver em atraso, suspende
```

```
2          double multa = ``valorMulta'' * ``diasAtraso'';
3          if (multa > 0)
4             this.mudarStatus(2); // executando um método da superclasse
5          return multa;
6       }
```

A alteração se faz necessária porque com a visibilidade privada, a classe `Aluno` passa a não enxergar o atributo `status` diretamente. Seu acesso deve então ser realizado por meio da interface pública da classe `Usuario`, neste caso, através da operação `mudarStatus()`.

4.4.3 Clientes por Herança e por Instanciação

Como visto na Seção 1.2.2 do Capítulo 1, o encapsulamento é uma técnica utilizada para minimizar as interdependências entre os módulos do sistemas. Uma das formas mais utilizadas para melhorar o encapsulamento dos módulos é restringir a comunicação entre eles através de interfaces externas bem definidas

Essa interface pode ser vista como uma espécie de contrato entre o módulo e os seus clientes; dessa forma, o encapsulamento garante aos projetistas que mudanças podem ser feitas de modo seguro, facilitando a manutenção e a evolução do software. Esses benefícios são especialmente importantes para sistemas de software complexos, grandes e de vida longa. Do ponto de vista da orientação a objetos, a interface externa de um módulo nada mais é do que o conjunto de operações públicas de todas as classes públicas do módulo; por essa razão, até mesmo uma classe OO isolada pode ser vista como um módulo de granularidade pequena.

Em Orientação a Objetos, a definição de uma classe pode ser vista como espécie de "módulo" cuja interface externa consiste de um conjunto de operações; mudanças na implementação da classe que preservam sua interface externa não afetam o código do resto do sistema.

O comportamento de uma classe pode ser acessado de duas maneiras: através do relacionamento de generalização/especialização, onde a subclasse herda o comportamento da superclasse, ou através da instanciação da classe e execução dos seus métodos. Em relação à instanciação, uma classe pode oferecer interfaces distintas, dependendo do cliente que a instanciou.

A diferença das interfaces públicas apresentadas a cada uma dessas duas categorias de clientes é definida através do controle de visibilidade.

Para recapitular, as duas categorias básicas de clientes que podem acessar uma classe são:

- **Clientes por herança** são as subclasses que herdam os métodos e a estrutura da classe.
- **Clientes por instanciação** são as classes que criam instâncias da classe e acessam o seu comportamento através do envio de mensagens para essas instâncias.

4.4.4 Herança de Implementação

Apesar do relacionamento de generalização/especialização possuir uma semântica de subtipo, algumas linguagens de programação orientadas a objetos utilizam artifícios para diversificar as formas de herança. Por essa razão, dependendo da maneira como o mecanismo de herança é usado, ele pode ser classificados de duas maneiras diferentes(ii) herança de implementação. A seguir, cada uma dessas formas de utilização do mecanismo de heraça é discutida em maiores detalhes.

Mas há casos em que o programador deseja reutilizar comportamento, mas as classes envolvidas não possuem uma relação de subtipos entre elas.

Nesses casos a herança de comportamento não pode ser utilizada. Um exemplo dessa situação é apresentado na Figura 4.11, onde a classe `Pilha`, que está sendo desenvolvida, herda o comportamento da classe `Lista`, já existente. O problema dessa hierarquia é o fato de uma pilha possuir restrições quanto à ordem de inserção e remoção de elementos, o que não existe em uma lista. Como pode ser visto no objeto `p1` do tipo `Pilha`, além das operações `empilhar()` e `desempilhar()`, definidas na classe `Pilha`, ele herda operações que podem comprometer a consistência da ordem da pilha, que nesse caso são `adicionarInicio()` e `removerInicio`.

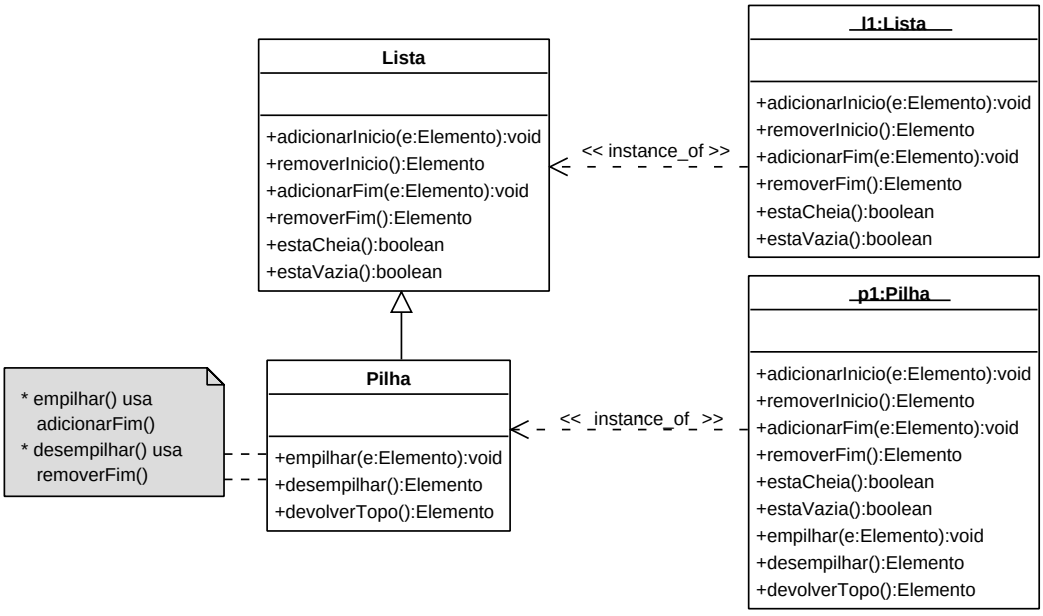


Figura 4.11: Hierarquia de Lista e Pilha com Herança

A seguir, é apresentada a implementação da hierarquia apresentada na Figura 4.11 em Java:

```
1  public class Lista {
2      //Atributos
3      //...
4      //Operações
```

```
5         public void adicionarInicio(Elemento e){...}
6         public Elemento removerInicio(){...}
7         public void adicionarFim(Elemento e){...}
8         public Elemento removerFim(){...}
9         public boolean estaVazia(){...}
10          // retorna TRUE se a lista estiver vazia.
11          // Senão, retorna FALSE
12        public boolean estaCheia(){...}
13          // retorna TRUE se a lista estiver cheia.
14          // Senão, retorna FALSE
15    }
16
17    public class Pilha extends Lista {
18        //Atributos
19        //...
20        //Operações
21        public void empilhar(Elemento e) {adicionarFim(e);}
22        public Elemento desempilhar()    {return removerFim();}
23        public Elemento devolverTopo() {
24           Elemento e = desempilhar();
25           empilhar(e);
26           return e;
27        }
28    }
29    ...
30    public static void main(String[] args){
31       Pilha plh = new Pilha();
32       plh.adicionarInicio(elemento); // corrompe o estado da pilha
33    }
```

Para contornar esse problema, é possível utilizar uma variação de herança, existente em UML: **herança de implementação**. Diferentemente da herança que materializa o relacionamento de generalização/especialização, apresentada na Seção 1.3.5 do Capítuilo 1, a herança de implementação não representa o conceito de tipo/subtipo. Isso acontece pelo fato dela especificar que a subclasse herda apenas a implementação das operações, mas não essas operações não fazem parte da sua interface pública [9]. A Figura 4.12 apresenta a hierarquia de Lista e Pilha, utilizando herança de implementação. Perceba que o relacionamento de herança recebe o estereótipo $\ll$*implementation* $\gg$. Nessa hierarquia, Pilha não é um tipo de Lista.

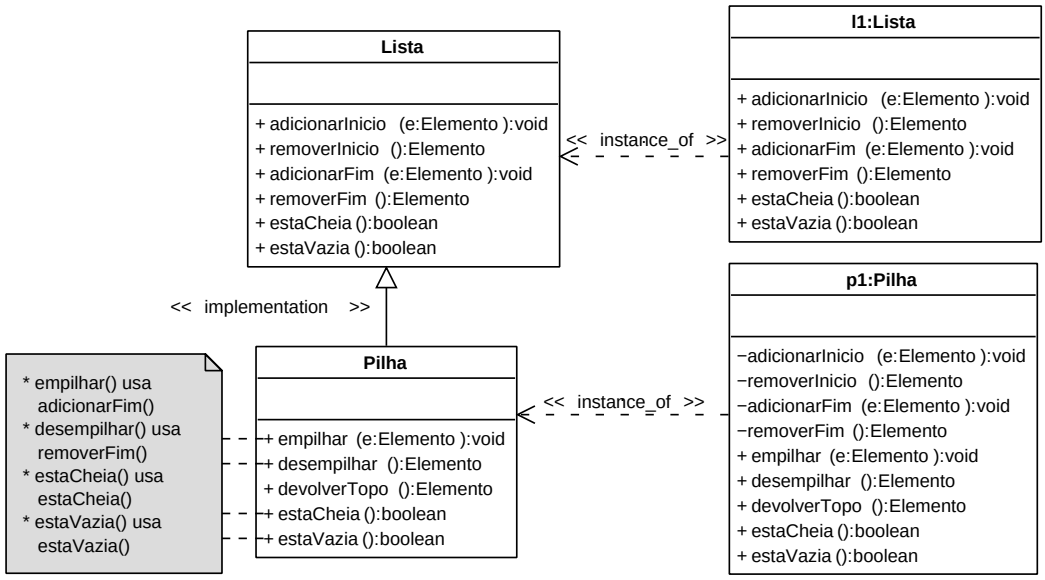


Figura 4.12: Hierarquia de Lista e Pilha com Herança de Implementação

A seguir, é apresentada a implementação da hierarquia apresentada na Figura 4.12 em C++, utilizando herança de implementação, que em C++ é implementada através de derivação privada:

```
1   class Pilha : private Lista {
2     //Atributos
3     //...
4     //Operações
5     public:
6       Lista::estaCheia(); // visibilidade pública explicita
7       Lista::estaVazia();  // visibilidade públia explicita
8       //operações de Pilha
9       void empilhar(Elemento e) {adicionarFim(e);}
10      Elemento desempilhar()    {return removerFim();}
11      Elemento devolverTopo() {
12          Elemento e = desempilhar();
13          empilhar(e);
14          return e;
15      }
16  };
17  ...
18  int main(){
19      Pilha plh;
```

```
20        plh.adicionarInicio(elemento); //erro de compilação
21    }
```

Solução Recomendada Usando Agregação

Apesar da herança de implementação ser uma possível solução para o problema da herança de operações indesejadas, a sua utilização pode tornar o modelo mais complexo e propenso a erros, uma vez que não representa um relacionamento de generalização/especialização entre as classes. Uma solução alternativa para a hierarquia de Lista apresentada na Figura 4.11 consiste em criar uma hierarquia de agregação envolvendo Lista e Pilha. Neste caso, podemos tratar Lista como uma parte de Pilha, conforme ilustrado na Figura 4.13.

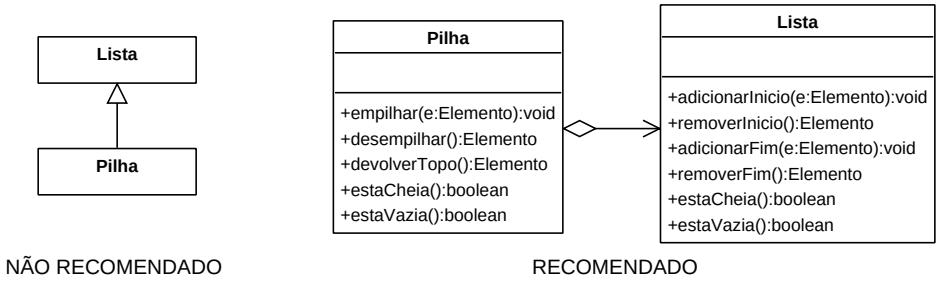


Figura 4.13: Solução Alternativa com Agregação

O trecho de programa a seguir mostra a implementação parcial dessa solução alternativa em Java:

```
1     class Pilha {
2         private Lista l; // apontador para um objeto Lista
3
4       public void empilhar(Elemento e) {l.adicionarFim(e);}
5       public Elemento desempilhar() {return l.removerFim();}
6       public Elemento devolverTopo() {
7           Elemento e = desempilhar();
8           empilhar(e);
9           return e;
10      }
11      public boolean estaCheia() {return l.estaCheia();}
12      public boolean estaVazia() {return l.estaVazia();}
13    };
```

4.4.5 Herança de Comportamento

A herança de comportamento é a maneira utilizada pelas linguagens de programação para implementar o relacionamento de generalização/especialização, onde há a idéia de uma hierarquias de tipos.

Essa forma de uso do mecanismo de herança é a recomendada, uma vez que a noção de subtipo tem um papel fundamental no modelo de objetos.

Para fixar melhor o conceito, a seguir é apresentada uma definição mais clara de subtipo.

Dados dois tipos `S` e `T`. `S` é considerado um subtipo de `T` se e somente se `S` possui pelo menos o comportamento de `T`. Por essa razão, um objeto do tipo `T` pode ser substituído por um objeto do tipo `S`, uma vez que ele possui todas as operações definidas em `T`. Mas apesar de herdar o comportamento do(s) seu(s) supertipo(s), um subtipo pode ter propriedades e operações adicionais.

Muitos desenvolvedores costumam utilizar o mecanismo de herança como forma de reutilização de código através do compartilhamento de comportamento. Mas em orientação a objetos, o compartilhamento de comportamento através de herança só é justificável quando existe um relacionamento verdadeiro de generalização/especialização entre as duas classes, isto é, quando a subclasse "é um tipo" da superclasse.

A Figura 4.10 apresentou um exemplo da utilização do mecanismo de herança para indicar um relacionamento de generalização/especialização. Como um Aluno e um Professor são considerados "um tipo de" Usuario, a herança de comportamento faz todo o sentido. Nessa hierarquia, as classes Aluno e Professor herdam todo o comportamento de um usuário da biblioteca: `verificarCodigo()`, `imprimir()` e `mudarStatus()` e por isso podem ser utilizadas onde a classe `Usuario` é esperada. O fato de haver um supertipo de usuários facilita a evolução do modelo, no intuito de adicionar novos tipos de usuários, como por exemplo: Funcionários e PesquisadoresVisitantes.

Sabendo que os elementos de um subconjunto estão contidos em um conjunto maior e mais abrangente, podemos concluir intuitivamente, que a noção de supertipo/subtipo assemelha-se à noção de conjunto/subconjunto. Por exemplo, todos os objetos do tipo Aluno formam um subconjunto dos objetos do tipo `Usuários` (Figura 4.14).

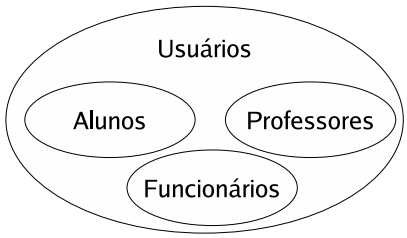


Figura 4.14: Analogia entre Supertipo/Subtipo e Conjunto/Subconjunto

4.4.6 Herança Múltipla

Em uma hierarquia de classes que utiliza apenas herança simples, qualquer classe deriva de no máximo uma classe base. Porém, em muitas situações é desejável que uma classe herde de mais de uma classe, o que é possível com a utilização de **herança múltipla**.

Por exemplo, a Figura 4.15 apresenta uma hierarquia de figuras geométricas. A classe Figura define um tipo abstrato de dados de uma figura de duas dimensões. A classe Retangulo herda diretamente de Figura e redefine a operação `calcularArea()`. A hierarquia também contém uma classe denominada Centrado, que define o tipo abstrato de dados relativo a elementos que possuem um centro definido, e que por isso podem girar em torno dele. Finalmente, a classe RetanguloCentrado representa uma entidade que ao mesmo tempo é um tipo de retângulo que possui um centro determinado, sendo também um tipo de elemento centrado.

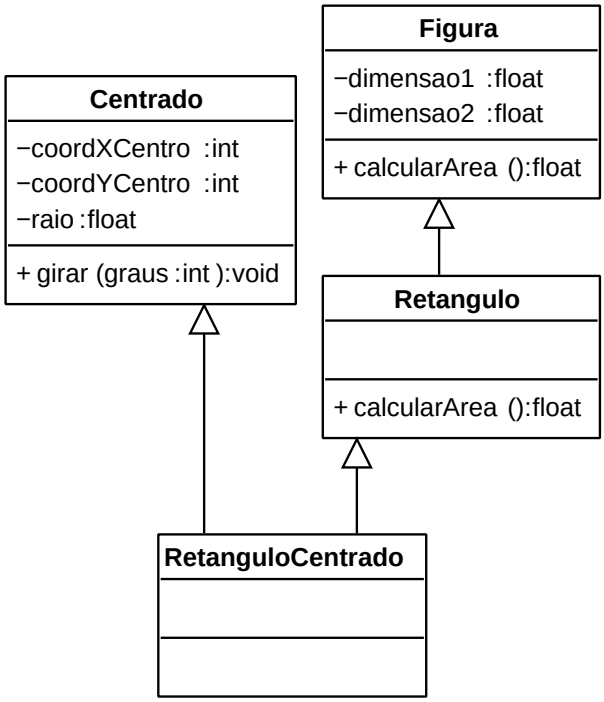


Figura 4.15: Hierarquia de Figuras Geométricas

Estratégias para Resolução de Conflitos

Quando uma classe herda de mais de um pai, existe a possibilidade de conflitos: operações e atributos com mesmo nome, mas com semânticas diferentes, que são herdadas das diferentes superclasses. Ao utilizar herança múltipla de classes, um tipo particular de ambigüidade é conhecido como **problema do diamante**. A Figura 4.16 ilustra esse problema. Quando duas classes (Aluno e Professor) herdam de um mesmo pai (Usuario), e uma outra classe (ProfessorAluno) herda ao mesmo tempo

das classes intermediárias, se um objeto do tipo ProfessorAluno receber uma chamada referente a uma operação definida em Usuario (`verificarCodigo`, por exemplo), qual a implementação a ser executada? de Usuario, de Aluno, ou de Professor?

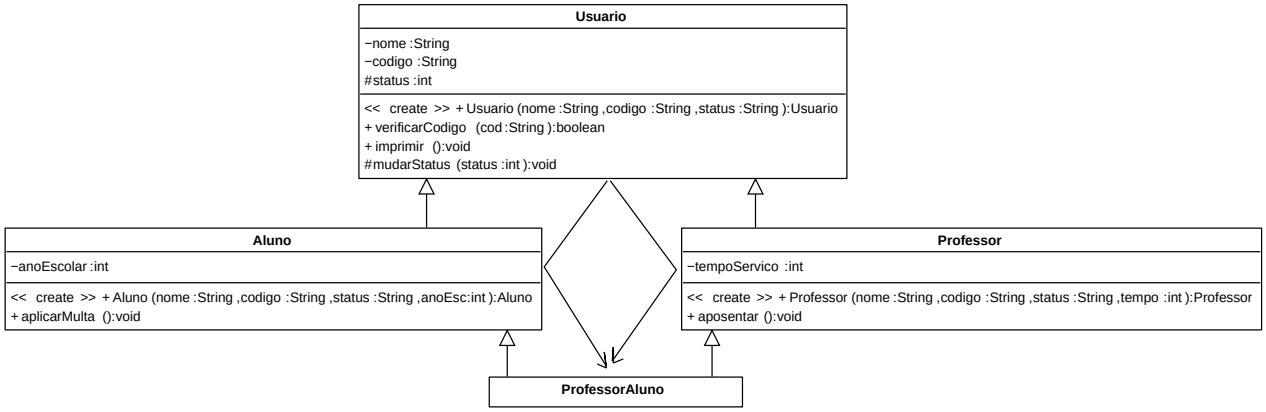


Figura 4.16: Problema do Diamante em Herança Múltipla

O nome "problema do diamante" é devido ao formato de losango, estabelecido entre as classes (ver Figura 4.16), que lembra um diamante.

Muitas soluções têm sido propostas para resolver esse problema. A seguir, são apresentadas as três principais estratégias existentes:

1. **Linearização.** Esta estratégia especifica uma ordem linear e total das classes, e determina que a busca do atributo ou método seja efetuada na ordem estabelecida. Essa abordagem é implementada pelas linguagens Flavors e CommonLoops.

2. **Renomeação.** Ela requer a alteração dos nomes de atributos e operações que sejam conflitantes. Essa abordagem é implementada por Eiffel.

3. **Qualificação.** Sempre que ocorrer ambigüidade, deve-se redefinir a operação e utilisar um operador de escopo, como por exemplo a identificação da classe seguida de quatro pontos ("::"), presente em C++. Dessa forma, esse operador permite a identificação única do atributo ou método conflitante. Essa abordagem é implementada pela linguagem C++.

A estratégia de linearização torna o problema da resolução de conflitos transparente ao programador, mas introduz uma ordem obrigatória na herança das classes, o que inviabiliza por exemplo a seleção de uma operação conflitante de cada classe. A renomeação e a qualificação são estratégias que promovem uma maior flexibilidade para o programador decidir a aplicabilidade dos métodos e/ou atributos herdados.

Várias discussões a respeito da viabilidade da herança múltipla têm sido discutidas na literatura. Herança múltipla é um assunto controverso. Apesar disso, esse tipo de herança é uma ferramenta poderosa para a especificação de classes e para o aumento da oportunidade de reutilização de

comportamento. As suas principais desvantagens são o aumento do acoplamento entre as classes do sistema e o aumento da complexidade do modelo e da implementação.

4.4.7 Exemplo de Herança Múltipla em UML

A Figura 4.17 mostra um exemplo de uso de herança múltipla.

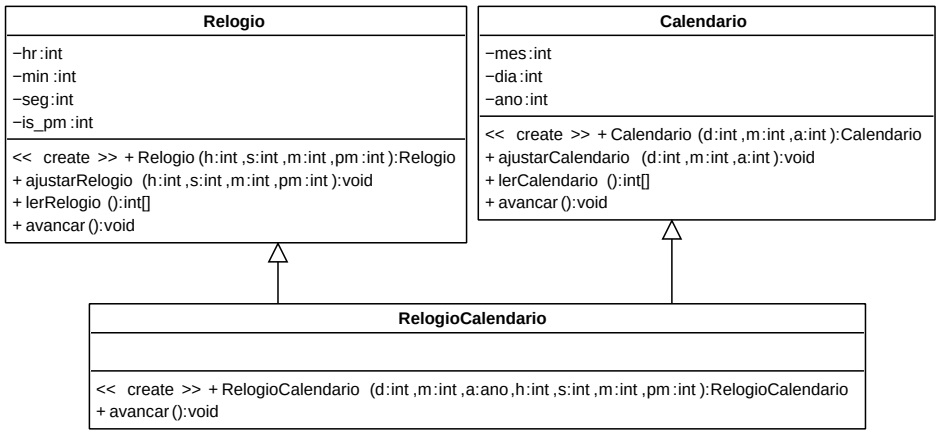


Figura 4.17: Hierarquia de um Relógio-Calendário

Neste exemplo, tanto a classe `Relogio` quanto a classe `Calendario` apresentam o método `avancar()` nas suas interfaces. Embora essas operações tenham o mesmo nome, as suas semânticas são diferentes pois a operação `avancar()` da classe `Relogio` avança o ponteiro de um relógio, enquanto a operação `avancar()` da classe `Calendario` avança um dia do calendário e eventualmente também avança o mês e o ano, caso seja necessário. A classe `RelogioCalendario` implementa a sua própria operação `avancar()` cuja semântica é uma fusão das outra duas operações `avancar()` herdadas. O conflito de nomes repetidos das operações é resolvido com a estratégia de qualificação, usando-se o operador de escopo de C++ "nomeDaClasse::nomeDaOperação". Dessa forma a classe `RelogioCalendario` pode fazer uso das duas implementações herdadas como lhe for conveniente.

O trecho do código a seguir, apresenta uma implementação parcial em C++ para a hierarquia apresentada na Figura 4.17.

```
1    class Relogio {
2      protected:
3        int hr;  int min;
4        int seg; int is_pm;
5      public:
6        Relogio( int h, int s, int m, int pm){...}
7        void ajustarRelogio(int h, int s, int m, int pm){...}
8        int[] lerRelogio(){...}
```

```
9         void avancar(){...} // ----------> conflito de nome
10    };
11
12    class Calendario {
13      protected:
14        int mes;
15        int dia; int ano;
16      public:
17        Calendario( int d, int m, int a){...}
18        void ajustarCalendario(int d, int m, int a){...}
19        int[] lerCalendario(){...}
20        void avancar(){...} // ----------> conflito de nome
21    };
22
23    class RelogioCalendario : public Relogio, public Calendario {
24      public:
25    RelogioCalendario (int d, int m, int a, int h, int mn, int s, int pm){...}
26        void avancar(){
27                  Relogio::avancar();
28                  Calendario::avancar();
29                //----------> Métodos Qualificados
30        }
31    };
```

Herança Múltipla vs. Agregação

Apesar da expressividade do relacionamento de herança múltipla, a sua utilização aumenta o acoplamento entre as classes, uma vez que a classe derivada passa a ser afetada diretamente por alterações em qualquer uma de suas classes base. Além disso, como já discutido anteriormente, o uso de herança múltipla aumenta consideravelmente a complexidade dos sistemas e por isso deve ser empregado de forma disciplinada.

Devido à dificuldade em se utilizar corretamente o conceito de herança múltipla e principalmente ao aumento da complexidade do sistema decorrente dessa utilização, algumas linguagens de programação orientada a objetos, como por exemplo Java e C#, não oferecem meios de se implementar herança múltipla entre classes. Em contrapartida, a literatura de engenharia de software apresenta técnicas de modelagem que podem ser empregadas como formas alternativas a esse conceito. Essas técnicas visam tanto suprir as necessidades de reutilização de código, quanto a idéia

de possuir mais de um supertipo, separando explicitamente esses dois conceitos.

Como mostrado na Figura 4.4.7, o relacionamento de agregação pode ser utilizado como forma alternativa de se alcançar a reutilização de código, existente na implementação da classe `RelogioCalendario`, mostrada na Figura 4.17. Essa reutilização acontece através da comunicação com as classes `Relogio` e `Calendario`, que é possível graças ao relacionamento de agregação, tornando a classe `RelogioCalendario` um cliente por instanciação das classes `Relogio` e `Calendario`.

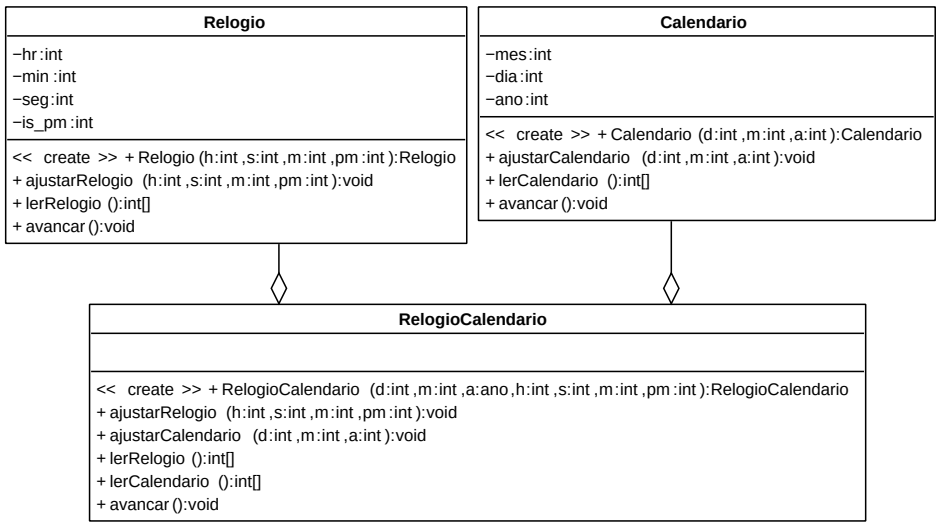


Figura 4.18: Alternativa à Herança Múltipla com Agregação

Apesar da agregação não expressar o conceito de supertipos/subtipo envolvido no relacionamento de herança, isso pode ser resolvido de uma maneira elegante utilizando interfaces. A Seção 4.7 abordará essa questão em detalhes. Por representar unicamente o conceito de tipo, as interfaces podem ser utilizadas de forma ortogonal e complementar ao conceito de composição de classes, presente na agregação.

4.5 Polimorfismo

Esta seção examina os conceitos de polimorfismo, concentrando-se nas várias formas de polimorfismo existentes. Polimorfismo é um conceito chave no contexto de orientação a objetos.

4.5.1 O que é polimorfismo?

A palavra "polimorfismo" é derivada do grego e significa "muitas formas" ou "tendo muitas formas". Quando usada no contexto de orientação a objetos, **polimorfismo** significa que variáveis podem

referenciar mais do que um tipo. Como conseqüência, essas variáveis podem fazer referências a diferentes tipos em diferentes contextos.

O conceito de polimorfismo não é novo; por volta de 1976, a linguagem ML apresentou uma disciplina de tipo polimórfica uniformizada para todos os tipos da linguagem. Atualmente, existem funções e tipos polimórficos aplicados a diversas linguagens de programação. Uma função é considerada polimórfica quando seus parâmetros podem ter mais do que um tipo. Em tipo, por sua vez, é dito polimórfico se o seu conjunto de operações pode ser aplicada a operandos de mais de um tipo; caso contrário são ditos monomórficos. Por exemplo, numa linguagem polimórfica, pode-se definir uma única função `obterComprimento()` do tipo: `obterComprimento::[A] -> NUM`, para todos os tipos `A`, significando que (i) `obterComprimento()` é uma função cujo parâmetro de entrada é uma lista de elementos (simbolizada pelos colchetes); (ii) o tipo de cada elemento da lista pode ser qualquer um, simbolizado por (`A`); e (iii) a função devolve um valor numérico como saída.

Em contraste, uma linguagem com tipos monomórficos força o programador a definir diferentes funções para retornar o comprimento de uma lista de inteiros, de uma lista de reais, e assim por diante. Exemplos de linguagens com tipos monomórficos são C, Pascal e Algol-68.

Em particular, no contexto do modelo de objetos, polimorfismo significa que diferentes tipos de objetos podem responder a uma mesma mensagem de maneiras diferentes (Figura 4.19). Por exemplo, num programa, podemos definir uma operação chamada `imprimir()` em diversas classes diferentes, mas cada "versão" de `imprimir()` é adaptada para o tipo específico de objeto que será impresso. Um objeto do tipo Publicacao responderá à mensagem `imprimir()` de uma maneira, um objeto Livro responderá de outra forma e um objeto do tipo Periodico responderá de uma outra forma ainda. Isto significa que não temos que definir operações com nomes completamente diferentes para cada um dos tipos, como por exemplo, ‘‘`imprimirPublicacao()`’’, ‘‘`imprimirLivro()`’’ e ‘‘`imprimirPeriodico()`’’, o que tornaria o programa muito menos flexível. Dizemos que `imprimir()` é uma operação polimórfica pois ela é implementada diferentemente por diferentes tipos de objetos.

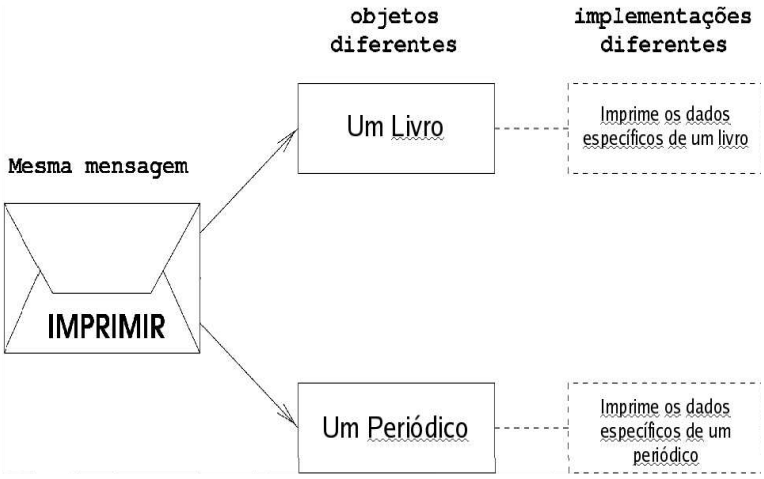


Figura 4.19: Método Polimórfico

A Figura 4.20 mostra uma hierarquia de classes em UML correspondente ao exemplo da Figura 4.19. A classe Publicacao está no topo da hierarquia e define uma implementação padrão para a implementação `imprimir()`. Suas subclasses: Livro e Periodico, definem implementações específicas.

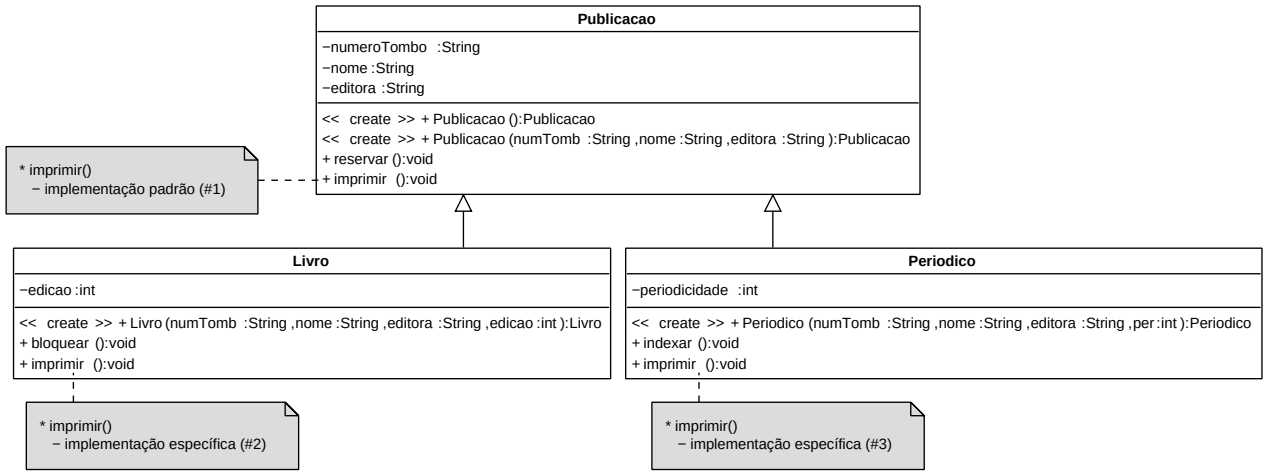


Figura 4.20: Uma Hierarquia de Tipos e Subtipos Polimórficos

Em sistemas de software, mais especificamente nas linguagens de programação, a multiplicidade de formas pode ser obtida de diversas maneiras. Cardelli e Wegner [13] criaram uma taxonomia que classifica o polimorfismo em categorias e subcategorias. Como pode ser visto na Figura 4.21, existem duas categorias principais: (i) *ad hoc*; e (ii) universal. O **polimorfismo *ad hoc*** é caracterizado pela ausência de um modo uniforme no comportamento de uma função, que pode variar radicalmente, de acordo com os tipos dos seus argumentos de entrada. No **polimorfismo universal**, também conhecido como **polimorfismo verdadeiro**, uma função ou tipo trabalha uniformemente para qualquer um dos tipos que ele seja compatível. Sendo assim, enquanto o polimorfismo *ad hoc* trabalha com um número limitado de tipos de um modo não sistemático, o polimorfismo universal trabalha potencialmente com um conjunto infinito de tipos de modo disci plinado. As duas principais formas de polimorfismo *ad hoc* são: **coerção** e **sobrecarga**. Em relação ao polimorfismo universal, as duas formas básicas são: **polimorfismo paramétrico** e **polimorfismo de inclusão**. A seguir, as Seções 4.5.2 a 4.5.5 detalham cada uma dessas quatro formas de polimorfismo.

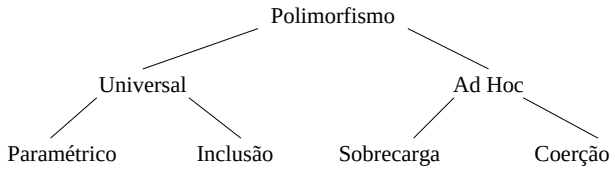


Figura 4.21: Taxonomia de Cardelli e Wegner

4.5.2 Coerção

A coerção proporciona um meio de contornar a rigidez de tipos monomórficos. Linguagens que dão suporte a coerção realizam um mapeamento interno entre tipos, de acordo com relações de equivalência existentes entre eles. Imagine que uma determinada chamada de função passa um tipo diferente do esperado; nesse caso, a própria linguagem verifica se existe uma coerção adequada, baseada na conversão implícita entre tipos de dados equivalentes. O trecho de código seguinte apresenta um exemplo desse cenário, implementado em Java.

```
1  public class Professor {
2    //Atributos
3     private float salario;
4     ...
5
6    //Operacoes
7     public void aumentarSalario(float abono){
8        salario = salario + abono;
9     }
10    ...
11
12    //...utilização do método
13    public static void main(String[] args) {
14       Professor p1 = new Professor();
15       int i1 = 400;
16       float f1 = 712.5;
17       p1.aumentarSalario(i1);
18       p1.aumentarSalario(f1);
19    }
20 }
```

No exemplo anterior, o método `aumentarSalario()` é definido como tendo um parâmetro do tipo `float`. Devido ao uso de coerção, é possível chamar esse procedimento passando como argumento um parâmetro do tipo `int` (Linha 17) ou qualquer outro tipo compatível com o tipo ("que caiba em") `float`. Se não houvesse coerção, seria necessário definir dois métodos:

```
1     public void aumentarSalario1(float abono){...};
2     public void aumentarSalario2(int abono){...};
```

Note que, no trecho de código acima, todos os métodos têm nomes diferentes, mas suas implementações seriam a mesma. Dessa forma, além de melhorar a legibilidade do programa, a coerção também proporciona reutilização de código.

4.5.3 Sobrecarga

O polimorfismo de sobrecarga (do inglês *overloading*) permite que um nome de função seja usado mais do que uma vez, com diferentes tipos de parâmetros. A Figura 4.22 apresenta a especificação da classe Publicacao, que possui duas operações construtoras recebendo argumentos diferentes. Nesse caso, dizemos que o construtor da classe Publicacao foi sobrecarregado.

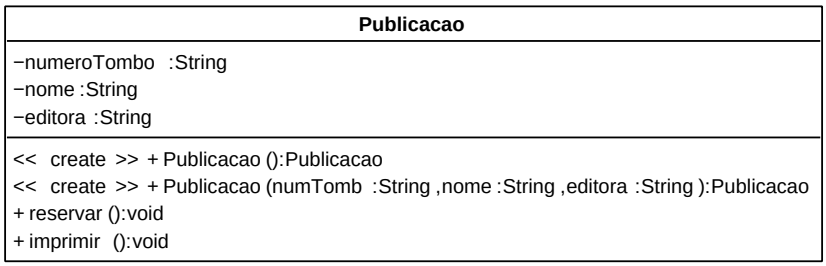


Figura 4.22: Polimorfismo de Sobrecarga na Classe Publicacao

4.5.4 Polimorfismo Paramétrico

O uso do polimorfismo paramétrico possibilita que uma única função possa ser codificada, de modo a funcionar uniformemente com vários tipos distintos. Por esse motivo, funções paramétricas são também chamadas de funções genéricas. Como exemplo de uma função paramétrica, considere a função `obterComprimento()` discutida na Seção 4.5.1. Como exemplo de tipo paramétrico ou classe paramétrica, considere uma classe Pilha$< Tipo >$, onde `Tipo` é o tipo do elemento que será manipulado. Desse modo, uma classe "genérica" que implementa uma pilha pode ser escrita independentemente do tipo dos itens que são armazenados nela. A Figura 4.23 mostra a representaçã da pilha genérica em UML. Nessa figura, o tipo paramétrico Pilha$< Tipo >$, exemplificando a sua instanciação através da definição de três objetos que representam pilhas de tipos específicos e distintos: uma pilha de livros (`PilhaLivro` ou `Pilha<Livro>`), uma pilha de publicações (`PilhaPublicacao` ou `Pilha<Publicacao>`) e uma pilha de strings (`PilhaString` ou `Pilha<String>`).

Note que o estereótipo $<< bind >>$ se aplica a um relacionamento de dependência, especificando que o tipo genérico fonte (no caso, a classe `Pilha`) instancia o tipo alvo usando os parâmetros dados.

O trecho de código a seguir apresenta parte da implementação em Java da pilha polimórfica apresentada na Figura 4.23. A partir da versão 1.5.0 [11] lançada em 2005, a linguagem Java incorporou o conceito de tipos genéricos, uma forma de implementar polimorfismo paramétrico. As Linhas 12 a 16 apresentam como a classe Pilha pode ser instanciada para tipos distintos.

```
1  public class Pilha<Tipo>{
2      private Tipo itemAtual;
3      private Tipo proximoItem;
4      private int maxLen;
```

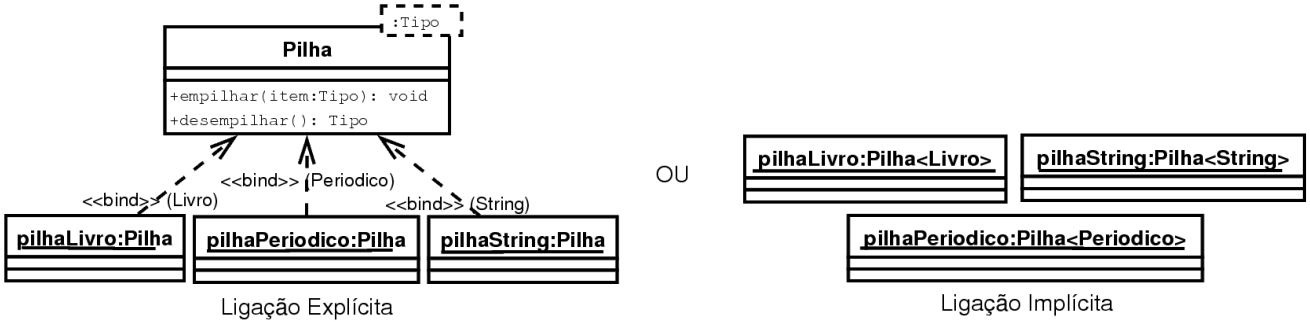


Figura 4.23: Polimorfismo Paramétrico

```
5       private int top;
6
7       public void empilhar(Tipo i){...}
8       public Tipo desempilhar(){...}
9       public boolean estaCheia(){...}
10      public boolean estaVazia(){...}
11
12      public static void main(String[] args){
13          Pilha<Livro> pilhaLivro = new Pilha<Livro>(); // pilha de ''Livros''
14          Pilha<Periodico> pilhaPeriodico = new Pilha<Periodico>(); // pilha de ''
                Periodicos''
15          Pilha<String> pilhaString = new Pilha<String>(); // pilha de ''Strings''
16      }
17  }
```

Um detalhe importante da implementação de polimorfismo paramétrico em Java é que o conceito de "classes genéricas" só pode ser instanciada com tipos relativos a classes, (subtipos de `Object`). Dessa forma, no exemplo da pilha genérica, não é possível definir uma pilha de nenhum tipo nativo da linguagem: `boolean`, `byte`, `char`, `double`, `float`, `int`, `long` e `short`.

4.5.5 Polimorfismo de Inclusão

Por estar relacionado com a noção de supertipo/subtipo decorrente do relacionamento de generalização/especialização, o polimorfismo de inclusão só é encontrado em linguagens orientadas a objetos. O princípio básico do polimorfismo de inclusão é uma conseqüência do fato das subclasses (ou subtipos) herdarem automaticamente todas as operações das suas respectivas superclasses (ou

supertipos). Por exemplo, na hierarquia da Figura 4.20 da Seção 4.5.1, a classe Publicacao tem duas subclasses: Livro e Periodico. Além dos seus construtores, que não são herdados, a classe Publicacao define duas outras operações: `reservar()` e `imprimir()`. `Livro` define uma operação adicional: `bloquear()`, enquanto `Periodico` adiciona a operação: `indexar()`. O trecho de código seguinte apresenta alguns exemplos de uso dessas classes, levando-se em consideração a existência de polimorfismo de inclusão. Perceba que as Linhas 8 e 12 apresentam um erro de inconsistência de tipos.

```
1   ...
2   Publicacao p = new Publicacao();
3   p.imprimir();
4   p.reservar();
5   p = new Livro();
6   p.imprimir();
7   p.reservar();
8   p.bloquear();   // ERRO
9   p = new Periodico();
10  p.imprimir();
11  p.reservar();
12  p.indexar(); // ERRO
```

No código anterior, a variável `p` é do tipo `Publicacao` (Linha 2) e, portanto, independentemente do subtipo do objeto para o qual apontar, essa variável aceita apenas as operações definidas em `Publicacao`. A variável `p` pode referenciar qualquer objetos do tipo `Publicacao`, incluindo as instâncias de seus subtipos. No exemplo, são atribuídos a `p` um objeto do tipo `Livro` (Linha 5) e outro do tipo `Periodico` (Linha 9). Quando as operações `imprimir()` e `reservar()` são acionadas a partir de `p`, tanto os objetos do tipo `Publicacao` quanto os dos tipos `Livro` e `Periodico` são capazes de respondê-las, uma vez que esses dois últimos herdaram essas operações de `Publicacao`. Por outro lado, se chamarmos as operações `bloquear()` (Linha 8) ou `indexar()` (Linha 12), ocorre um erro, já que essas operações não estão declaradas no tipo da variável `p`, que é `Publicacao`.

O polimorfismo de inclusão foi batizado com esse nome porque as operações polimórficas de um tipo estão "incluídas" nos seus subtipos. Em Java, a classe Object é a raiz de todas as classes. Os métodos da classe Object são exemplos de polimorfismo de inclusão, pois eles são capazes de operar uniformemente sobre objetos de qualquer tipo em Java.

Um caso especial de polimorfismo de inclusão ocorre quando uma subclasse define uma forma de implementação especializada para um método herdado da classe base. Nesse caso, diz-se que o método da classe derivada **redefine** o método da classe base. Para que a redefinição ocorra, o novo método deve ter a mesma "assinatura" do método herdado, isto é, eles devem ser idênticos quanto ao nome da operação e à lista de parâmetros (mesmo número de parâmetros, com os mesmos tipos e declarados na mesma ordem). O tipo do resultado de retorno não faz parte da assinatura do método e não pode ser modificado. A noção de redefinição de métodos só faz sentido em linguagens

que dão suporte a acoplamento dinâmico, como Java, C++ e C#.

Na hierarquia de publicações apresentada na Figura 4.20 da Seção 4.5.1, a operação `imprimir()`, definida inicialmente na classe Publicacao, é redefinido pelas suas subclasses Livro e Periodico. Note que a redefinição de métodos é diferente da sobrecarga. Na primeira, o método que redefine e o método redefinido precisam ter assinaturas absolutamente idênticas e esses métodos precisam ser declarados em classes distintas ligadas por um relacionamento de herança. No caso da sobrecarga, as assinaturas precisam ser diferentes (contanto que o nome continue o mesmo) e as diversas implementações do método podem estar todas localizadas na mesma classe.

4.5.6 Exemplo: Pilha de Publicações

Considere uma pilha de publicações formada por vários tipos diferentes de publicações, como livros e periódicos, como ilustrado pela Figura 4.24.

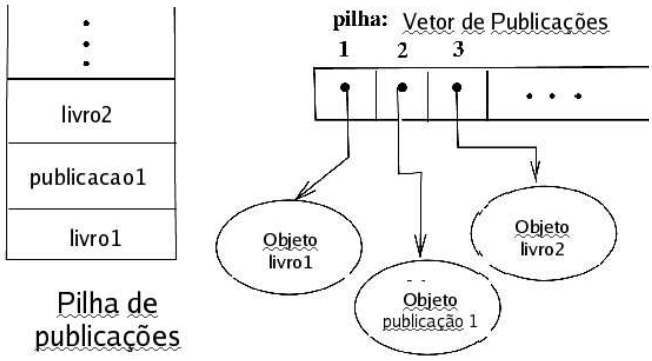


Figura 4.24: Uma Pilha de Publicações

A pilha de publicações da Figura 4.24 poderia ser modelada conforme apresentado na Figura 4.25. Suponha agora que gostaríamos de imprimir o conteúdo de cada um dos objetos armazenados na pilha. Numa linguagem orientada a objetos que dá suporte a polimorfismo de inclusão, podemos modelar os tipos de documentos existentes e a própria pilha seguindo a hierarquia de classes apresentada na Figura 4.25. O trecho de código seguinte apresenta a implementação em Java do método `imprimir()` da classe Pilha.

```
1 public class Pilha{
2       private Publicacao elementosPilha[]; //array de apontadores para
3                                            //objetos do tipo Publicacao
4       ...
5       public void imprimir(){
6          for(int i=0; i < elementosPilha.length; i++) {
7             elementosPilha[i].imprimir();
```

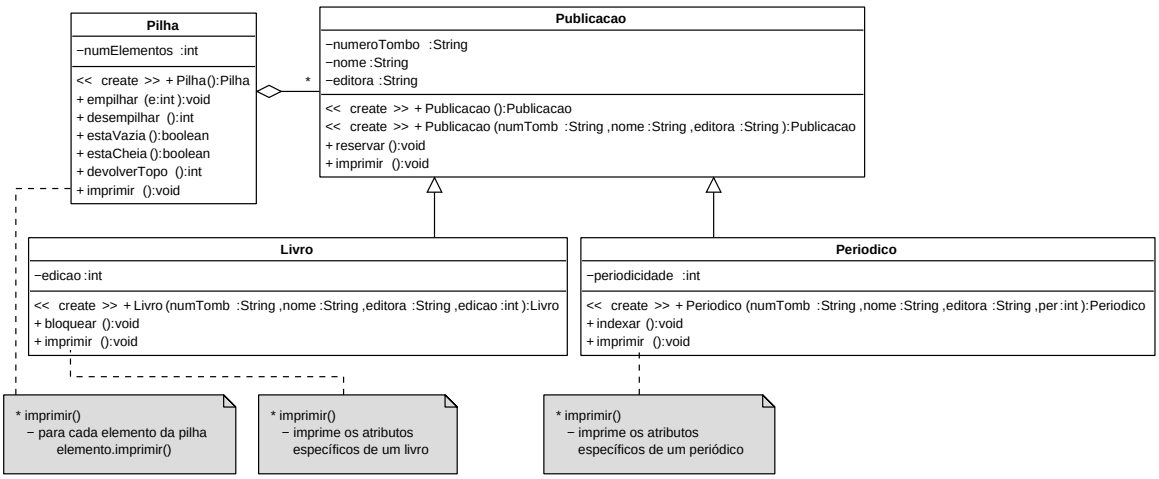


Figura 4.25: Um Exemplo de Polimorfismo de Inclusão com Redefinição de Operações

```
8               }
9           }
10      }
```

No trecho de código acima, a implementação do método `imprimir()`, que é executado a cada iteração do laço for, depende do tipo do objeto apontado por `elementosPilha[i]`. Por exemplo, se for um objeto do tipo Livro, o método imprimir() chamado é o definido pela classe Livro. Por outro lado, se elementosPilha[i] aponta para um objeto do tipo Periodico, a implementação utilizada é a

4.6 Classes Abstratas

Uma classe abstrata é uma classe que não pode ter instâncias diretas, enquanto que uma classe concreta pode ser instanciada. Classes abstratas podem definir o protocolo para uma operação, fornecendo ou não a implementação correspondente. No caso de não oferecer o código referente à sua implementação, a operação é denominada abstrata. Uma operação abstrata define a assinatura de uma operação para a qual cada subclasse concreta deve providenciar sua própria implementação. Por exemplo, a Figura 4.26 mostra uma operação abstrata reservar() da classe Publicacao (apresentada em *itálico*); sua assinatura está definida mas não sua implementação. Dessa forma, cada subclasse concreta deve redefinir o método da operação abstrata. Livro e Periodico são exemplos de classes concretas. Perceba que por possuir um método abstrato, a classe Publicacao obrigatoriamente deve ser abstrata (nome em *itálico*).

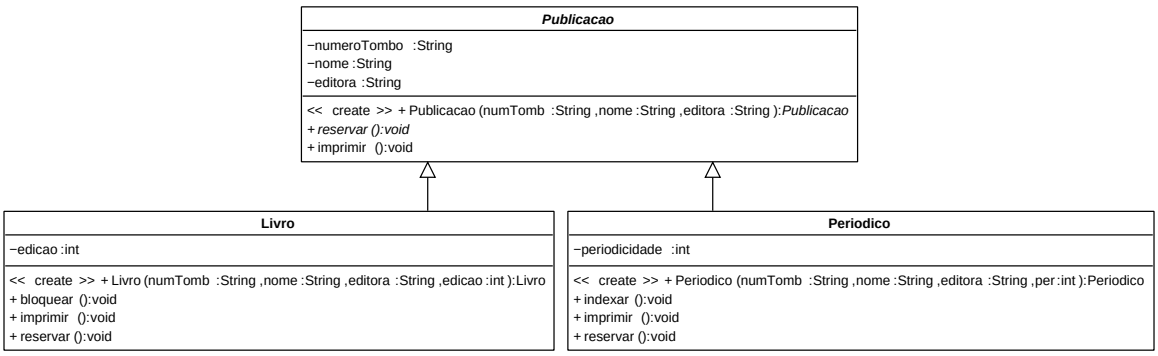


Figura 4.26: Exemplo de Classe Abstrata e Concreta

4.6.1 Operação Abstrata vs. Operação Concreta

Sintaticamente, uma classe abstrata é expressada em UML pela representação do seu nome em *itálico* (veja a classe Publicacao da Figura 4.26). Em Java, tanto uma operação quanto uma classe podem ser declaradas como abstratas. Em ambos os casos, é necessário usar o modificador `abstract`. Por exemplo:

```
1    public abstract class Publicacao {
2        public Publicacao(String numTomb, String nome, String editora){...}
3        public abstract void reservar();
4        public void imprimir(){...}
5
6    }
```

Existem duas idéias principais envolvidas com a noção de classe abstrata, a saber: (i) presença de operações abstratas; e (ii) inviabilidade de se criar instâncias a partir delas. Em Java, uma classe pode ser declarada como abstrata sem que nenhuma das suas operações seja abstrata. O modificador `abstract` atribuído à classe indica apenas que ela não pode ser instanciada. Porém em C++, para que uma classe seja considerada abstrata é necessário que ela tenha pelo menos uma operação abstrata. Mas uma característica essas duas linguagens têm em comum: toda classe que contém operações abstratas precisa, necessariamente, ser declarada como abstrata. Se uma classe contém operações abstratas, não faz sentido permitir que seja instanciada, uma vez que objetos criados a partir dessa classe não serão capazes de responder a todas as mensagens definidas na sua interface pública.

Na Figura 4.27, um outro exemplo, baseado na mesma hierarquia de publicações da Figura 4.26 é apresentado.

Nessa figura, a classe Publicação é abstrata. Isso significa que, apesar de Usuario estar associado a Publicacao, os objetos que as pessoas reservam são instâncias das classes concretas Livro ou Periodico. Quando a operação reservar() é chamada para uma variável do tipo Publicacao, o efeito

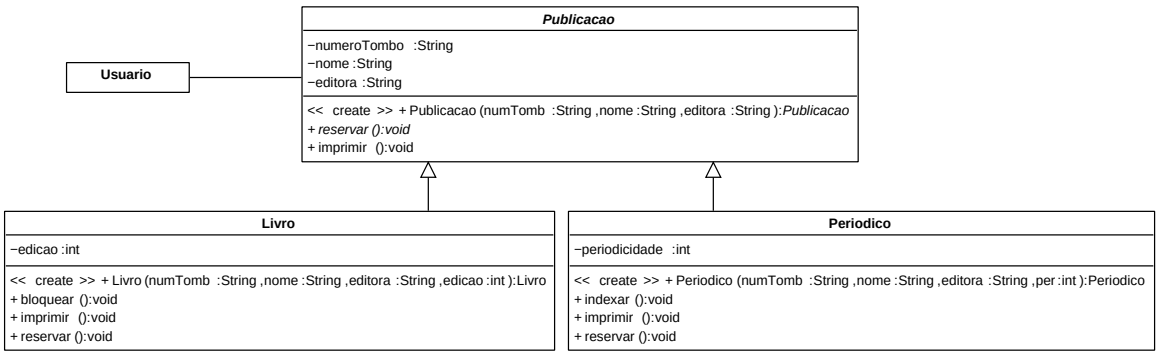


Figura 4.27: Uma Hierarquia de Classes de Reserva de Publicações

obtido depende do tipo do objeto sendo referenciado na ocasião. Se for um objeto da classe Livro, a operação reservar() agenda a reserva de um livro por quinze dias, utilizando a implementação especificada na classe Livro. Se for um objeto da classe Periodico, a operação reservar() agenda a reserva de um periódico por um período de sete dias, utilizando a implementação especificada na classe Periodico.

4.6.2 Classe Raiz, Classe Folha e Operação Folha

Classes abstratas e concretas bem como os seus elementos polimórficos são exemplificados na Figura 4.28. Em UML, uma classe é abstrata se o seu nome está escrito em itálico. Por exemplo, como a Figura 4.28 mostra, as classes A, B e C são todas abstratas, isto é, nenhuma delas pode ter instâncias diretas. Em contraste, as classes D, E e F são concretas, isto é, elas podem ter instâncias diretas criadas a partir delas.

Normalmente em UML, quando uma classe é criada, considera-se que ela possa herdar de outras classes mais gerais e que tenha classes derivadas que herdem seu comportamento. Entretanto, você pode especificar em UML uma classe que não possui classes filhas, escrevendo a propriedade *leaf* ("folha") embaixo do nome da classe. Uma **classe folha** não tem classes derivadas. De forma análoga, podemos especificar uma classe raiz, isto é, que não tem nenhum pai, escrevendo a propriedade *root* embaixo do nome da classe. Uma classe raiz não é derivada de nenhuma outra classe do sistema, e indica o início de uma hierarquia de classes.

Em UML, uma operação é, por definição, polimórfica. Isso significa que você pode especificar operações com a mesma assinatura em diversos pontos da hierarquia de classes (isto é, polimorfismo de inclusão). Nesse caso, quando uma mensagem é enviada em tempo de execução, a operação que é invocada na hierarquia é escolhida polimorficamente – isto é, a operação escolhida depende do tipo do objeto que recebeu a mensagem. Por exemplo, na Figura 4.28, a operação1() e a operação2() são operações polimórficas. Além disso, a operação *A::operação1()* é abstrata. Em UML, você especifica que uma operação é abstrata escrevendo o seu nome em itálico, similarmente ao que é feito com uma classe. Em contraste, A::operação3() é uma operação folha, significando que a operação não é polimórfica e não pode ser redefinida em nenhum lugar da hierarquia.

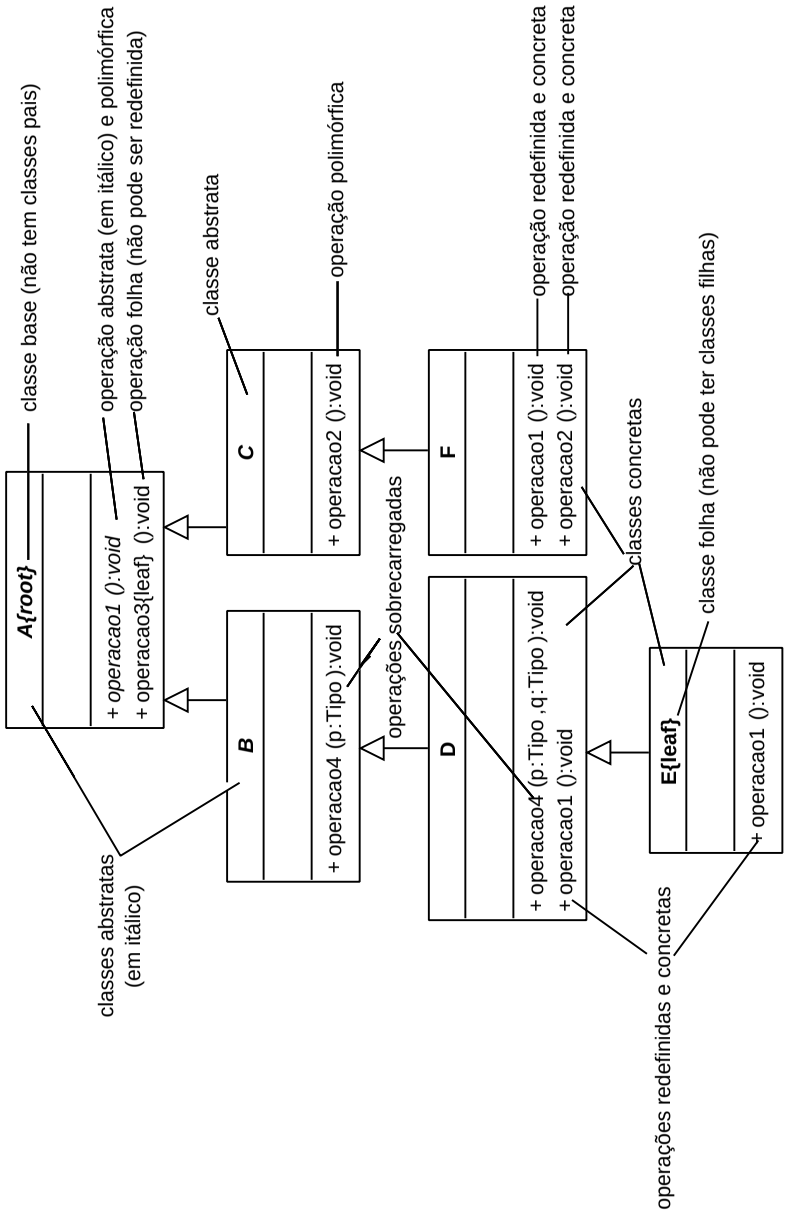


Figura 4.28: Exemplo de Hierarquia de Classes

4.6.3 Exemplo 1: Hierarquia de Raças de Cães

A Figura 4.29 mostra uma hierarquia de classes para cães. A classe Cão é a raiz da hierarquia e implementa o método mostraRaça como sendo *virtual*. As suas subclasses redefinem o método mostraRaça para que imprima o tipo de classe de cão que seja adequada.

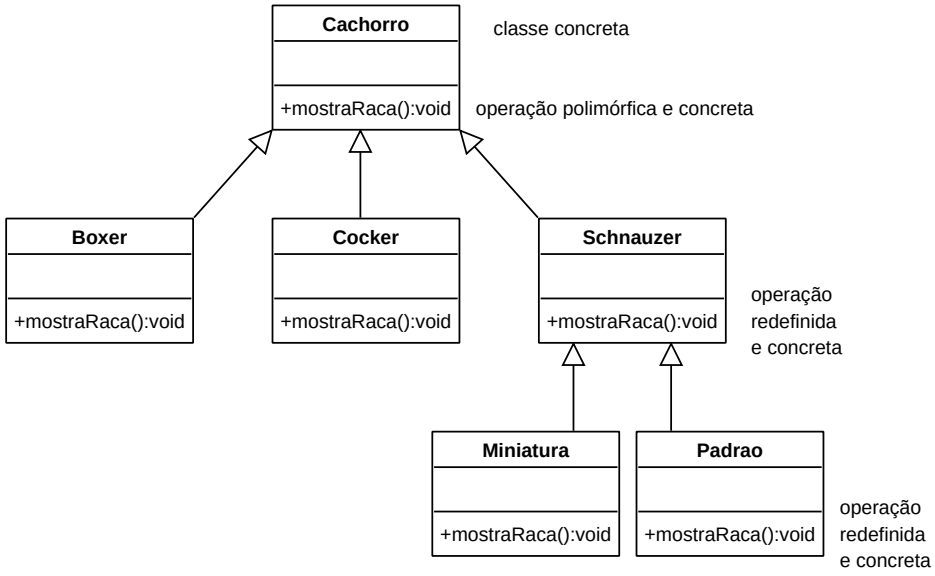


Figura 4.29: Hierarquia de Classe para Cães

Um trecho parcial de código Java que implementa essa hierarquia, é mostrado a seguir:

```
1   class Cachorro{
2     public void mostraRaca() {
3         System.out.println(``Raca não definida'');
4     }
5   }
6
7   class Schnauzer extends Cachorro {
8     public void mostraRaca() {
9         System.out.println(``Raca Schnauzer'');
10    }
11  }
12
13  class Miniatura extends Schnauzer {
14    public void mostraRaca() {
15        System.out.println(``Raca Schnauzer Miniatura'');
16    }
17  }
18
19  public static void main(String args[]) {
```

```
20  Cachorro apCao = new Cachorro(); // apontador para objetos do tipo Cachorro
21      Schnauzer s = new Schnauzer();
22      Miniatura m = new Miniatura();
23      apCao.mostraRaca(); // imprime ''Raça não definida''
24      apCao = s; // recebe endereço do objeto s
25      apCao.mostraRaca(); // imprime ''Raça Schnauzer''
26      apCao = m; // recebe endereço do objeto m
27      apCao.mostraRaca(); // imprime ''Raça Schnauzer Miniatura''
28    }
```

Observe que no exemplo mostrado anteriormente, a classe Cachorro não deveria poder ser instanciada, já que não faz sentido criar objetos que representam cães que não possuem uma raça. O fato do método mostraRaca() definido nessa classe retornar uma mensagem de erro é um forte indicador desse fato.

Pode-se modelar a hierarquia apresentada na Figura 4.29 de uma maneira mais natural usando o conceito de classe abstrata. Nessa abordagem, Cachorro é transformada em uma classe abstrata. Porém isso não é suficiente, uma vez que essa definição não garante a redefinição do método mostraRaca() nas classes derivadas. Daí conclui-se que tanto a classe Cachorro quanto a operação mostraRaca() precisam ser declaradas como abstratas.

Tendo em vista as modificações sugeridas no parágrafo anterior, cada classe derivada de Cão tem duas escolhas: (i) implementar o método mostraRaca() tornando-se uma classe concreta, ou (ii) não implementar o método mostraRaca(), tornando-se uma classe abstrata. Para as classes Boxer e Cocker faz sentido implementar mostraRaça. Para a classe Schnauzer, entretanto, temos duas opções válidas: (i) defini-la como uma classe concreta como anteriormente, ou (ii) não redefinir o método mostraRaca() e deixar que suas subclasses Miniatura e Padrao o façam. O trecho de código seguinte mostra alguns exemplos de uso correto e incorreto da hierarquia modificada:

```
1   public static void main(String args[]) {
2     Cachorro apCao = new Cachorro(); // ERRO. Classe abstrata.
3     Schnauzer s = new Schnauzer(); // ERRO. Classe abstrata.
4     Miniatura m = new Miniatura(); // Ok. Classe concreta.
5     m.mostraRaca(); // imprime ''Raça Schnauzer Miniatura''
6   }
```

4.6.4 Exemplo 2: Hierarquia de Figuras Gráficas

A Figura 4.30 mostra uma hierarquia para figuras gráficas. A classe Figura é uma classe abstrata e estabelece uma semântica para as classes derivadas a partir dela. Os métodos mostra() e esconde()

são declarados como abstratos na classe Figura. As classes concretas Linha, Arco e Triângulo implementam esses métodos. O trecho de código em Java a seguir mostra uma implementação parcial da classe Figura:

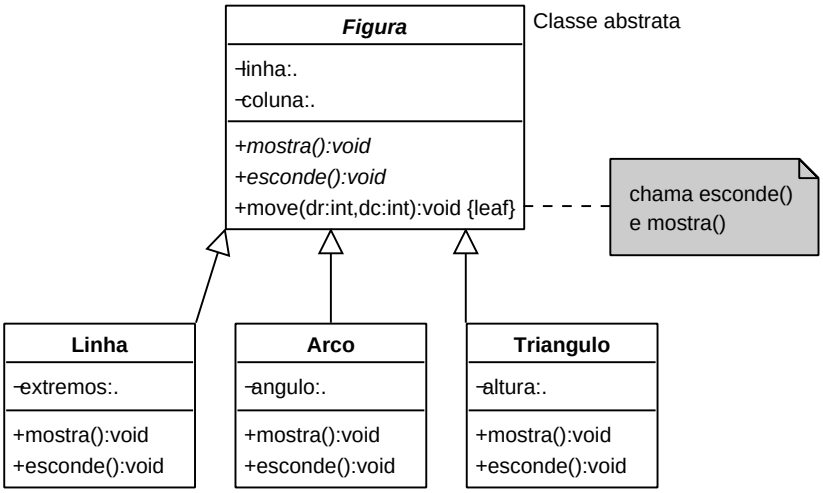


Figura 4.30: Hierarquia de Classes para Figuras Gráficas

```
1   abstract class Figura {
2       protected int linha;
3       protected int coluna;
4       public Figura (int r, int c) { linha = r; coluna = c; }
5       public abstact void mostra();  // abstrata
6       public void esconde(); // abstrata
7       public final void moveAoRedor (int dr, int dc) {
8           esconde();
9           linha += dr;
10          coluna += dc;
11          mostra();
12      }
13  }
```

Note que o método moveAoRedor() não é declarado como abstract na classe Figura. O motivo é que esse método usa outros métodos para executar as ações que dependem de implementações específicas de figuras (como Linha, Arco e Triangulo). No exemplo, essas ações são materializadas pelos métodos esconde() e mostra(). Além disso, o método moveAoRedor() é declarado como final, indicando que é acoplado estaticamente. Por não poder ser redefinido, esse método representa um ponto de congelamento[1] da hierarquia de classes. Por outro lado, os métodos mostra() e esconde()

[1]Do Inglês *frozen spot*.

são chamados de pontos de adaptação[2] da hierarquia de classes, pois são polimórficos e podem ser redefinidos em diversos pontos da hierarquia.

4.7 Interfaces

Em geral, o conceito de interface define uma fronteira que separa duas fases de um mesmo sistema. Uma fase, por sua vez, é definida como sendo uma parte homogênea pertencente a um sistema heterogêneo. Em sistemas computacionais, interfaces podem ser encontradas em diversos domínios (Figura 4.31) , por exemplo: (i) interface humano-computador, que é a camada de um sistema responsável pela fronteira entre o usuário e os processos da aplicação; e (ii) interface de programação ou API[3], que é uma camada composta por um conjunto de funções que podem ser chamadas pelos programas aplicativos para usar os serviços fornecidos pelo sistema operacional.

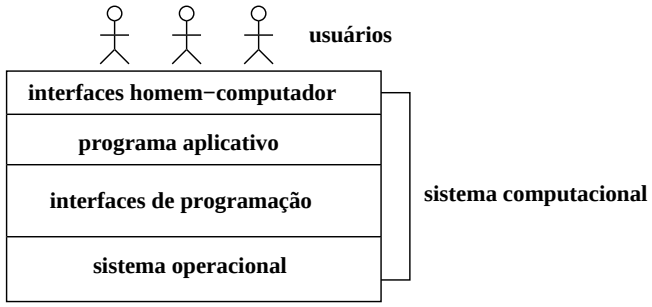


Figura 4.31: Exemplo de Interface em Sistemas Computacionais

Em orientação a objetos, uma interface é usada para definir um tipo que descreve o comportamento externamente visível de uma classe, objeto ou uma outra entidade. Sendo assim, uma interface especifica um TAD através da definição de um conjunto de operações e suas respectivas assinaturas de métodos. No restante desta seção, focaremos nossa atenção no conceito de interface conforme definido por UML. As idéias apresentadas aqui são válidas para qualquer linguagem que possibilite a representação explícita das interfaces, como por exemplo, C# ou Modula 3 e Java, que é uma das linguagens de programação mais populares na qual o conceito de interface está explicitamente materializado.

A implementação de um TAD especificado por uma interface é feita através de uma classe. Uma classe pode implementar várias interfaces e uma interface pode ser implementada por diversas classes. A tarefa de uma interface é separar de forma explícita diferentes grupos de classes de uma aplicação, isto é, estabelecer fronteiras explícitas entre partes homogêneas de um sistema computacional heterogêneo. A Figura 4.32 mostra graficamente como isso pode ser feito.

No caso (a), a classe Carro contém uma referência direta para um objeto de tipo Motor1.0.

[2]Do Inglês *hot spots*.

[3]Do Inglês *Application Programming Interface*

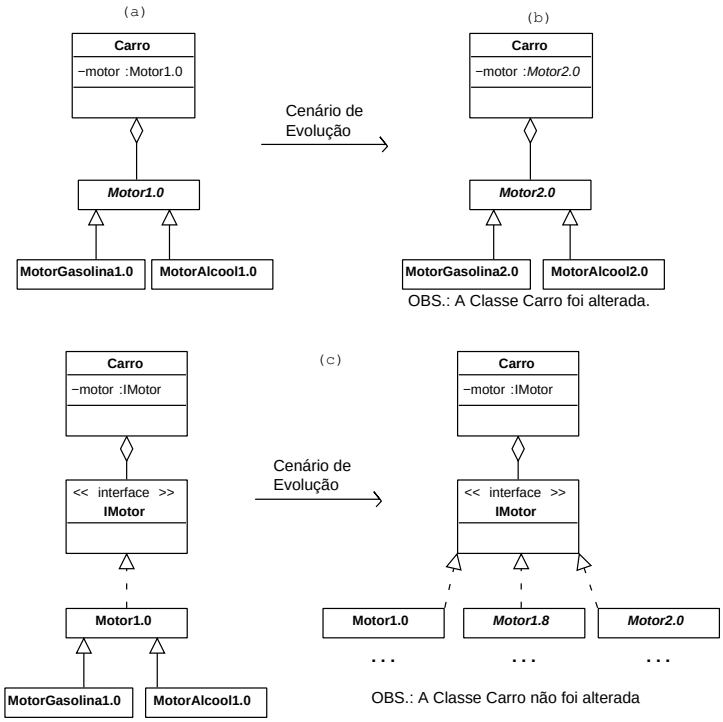


Figura 4.32: Uma Hierarquia de Classes e Interfaces

Uma dependência entre Carro e Motor1.0 é estabelecida. No caso (b), as classes Carro e Motor1.0 se acoplam através de uma interface IMotor. Carro contém uma referência para um objeto de tipo IMotor e as classes Motor1.0 e Motor2.0 implementam essa mesma interface IMotor. A dependência entre Carro e Motor1.0 é reduzida, uma vez que Carro é capaz de operar com qualquer objeto do tipo IMotor, que pode ser instanciado a partir das classes Motor1.0 ou Motor2.0.

4.7.1 Relacionamento de Realização em UML

Uma interface UML contém apenas assinaturas de métodos e definições de constantes. Uma característica essencial é a ausência de implementação, tanto de métodos quanto de estrutura de dados, isto é, atributos. Portanto, uma interface é, por definição, abstrata e possui todas as suas operações igualmente abstratas e com visibilidade pública. Uma classe, em contraste com uma interface, pode conter definições de atributos, métodos concretos e métodos abstratos.

Sendo assim, enquanto uma interface especifica de forma explícita um TAD, uma classe fornece a implementação dessa especificação. Interfaces representam um nível superior de abstração em relação ao das classes, isto é, pode-se projetar todas as interfaces de uma aplicação explicitamente antes de se decidir sobre a forma mais adequada de implementação. Essa separação entre o projeto de interface e o projeto de classes é análoga à separação entre o projeto arquitetônico e o projeto de engenharia de uma construção civil.

A Figura 4.33 apresenta uma interface ICadastro implementada por duas classes diferentes, ListaCad1 e ArrayCad. Note que, em UML, uma interface é representada como uma classe com o estereótipo <<interface>>.

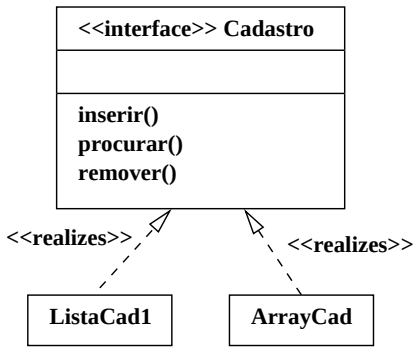


Figura 4.33: Uma Interface Implementada por Duas Classes

O trecho de código abaixo exemplifica a implementação de uma mesma interface por duas classes diferentes, para as interfaces e classes da Figura 4.33. Caso uma classe implemente uma interface mas não forneça uma implementação para algum dos métodos definidos por ela, o método torna-se implicitamente abstrato nessa classe, e conseqüentemente, a classe se torna abstrata.

```
1    public interface ICadastro { ... }
2    class ListaCad1 implements ICadastro { ... }
3    class ArrayCad implements ICadastro { ... }
```

A Figura 4.34 apresenta uma classe ListaCad2, que implementa duas interfaces, I e ICadastro, e estende uma classe ListaEspecial.

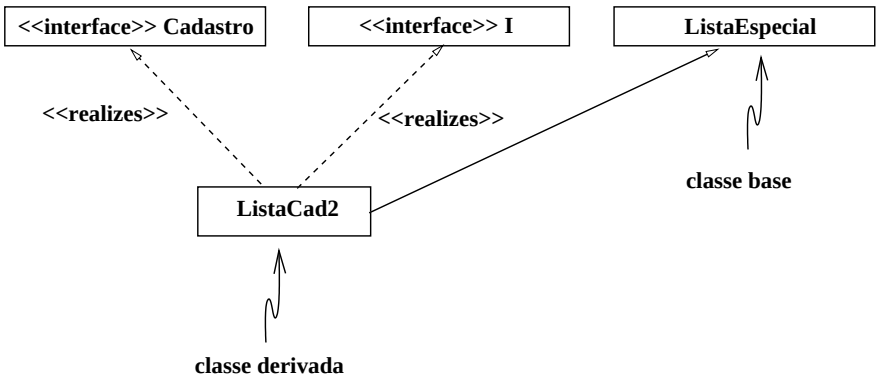


Figura 4.34: Uma Classe que Implementa Duas Interfaces e Estende Outra Classe

O trecho de código seguinte exemplifica a implementação de diversas interfaces por uma mesma classe.

```
1     public interface Cadastro { ... }
2     public interface I { ... }
3    public class ListaCad2 extends ListaEspecial implements ICadastro, I { ... }
```

Em Java, uma classe pode herdar apenas de uma classe base (herança simples), mas pode implementar várias interfaces. Essa política impede que ambigüidades como as descritas na Seção 4.4.6 ocorram, já que elas são causadas por implementações conflitantes de métodos com as mesmas assinaturas.

4.7.2 Herança Simples e Múltipla entre Interfaces

Interfaces podem estender outras interfaces. Uma interface pode estender várias interfaces e uma mesma interface pode ser estendida por várias outras. A Figura 4.35 apresenta exemplos de herança simples e múltipla entre interfaces.

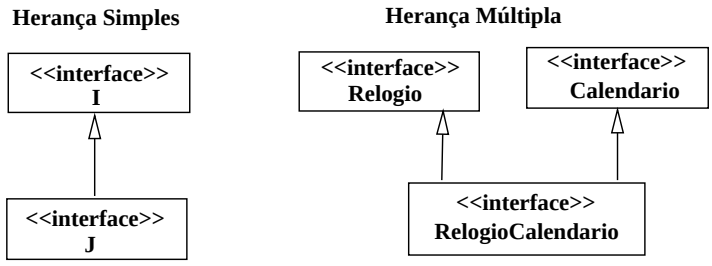


Figura 4.35: Herança Simples e Múltipla entre Interfaces

Em Java, herança entre interfaces é indicada através da palavra chave extends. O trecho de código seguinte apresenta alguns exemplos de herança entre interfaces, baseado nos relacionamentos mostrados na Figura 4.35.

```
1     public interface IMotor { ... }
2 public interface IMotorEletrico extends IMotor { ... } // herança simples de
      interfaces
3     public interface IMotorCombustao extends IMotor { ... } // herança simples de
          interfaces
4
5     public interface Relogio { ... }
6     public interface Calendario { ... }
7     public interface RelogioCalendario extends Relogio, Calendario { ... }
8         // herança múltipla de interfaces
```

Se uma interface IMotorCombustao estende uma interface IMotor, IMotorCombustao herda todas

as operações definidas por IMotor. Conseqüentemente, uma classe concreta MotorGasolina que implemente IMotorCombustao deve fornecer implementações para todas as operações de ambas, IMotor e IMotorCombustao.

4.7.3 Exemplo 1 - Integração Objeto-Relacional

Uma aplicação comum de interfaces em sistemas reais nos quais informações são armazenadas em um banco de dados consiste em separar as camadas de dados e negócios, de modo que essas duas possam evoluir de maneira independente. Classes da camada de negócios são clientes da camada de dados e a usam para abstrair a maneira como os dados são armazenados. A Figura 4.36 apresenta essa organização.

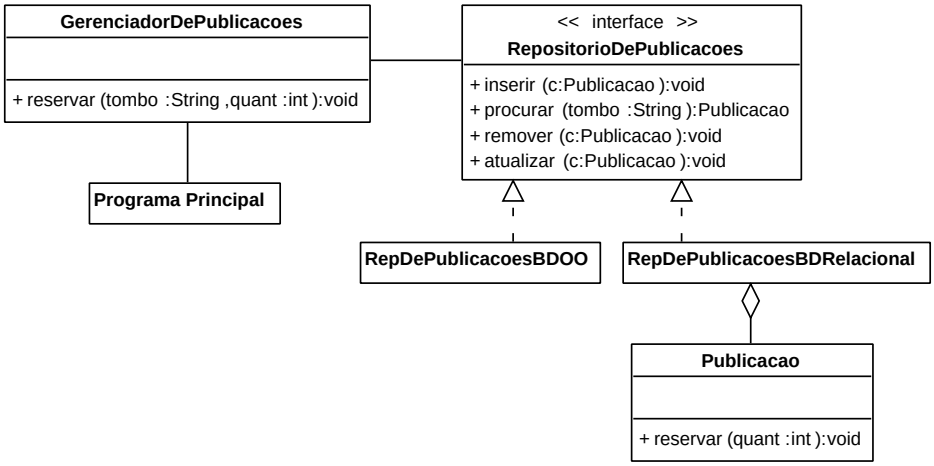


Figura 4.36: Separação Explícita Entre Camadas de Negócios e Dados

O seguinte trecho de código mostra a implementação da classe CadastroDePublicacoes, levando em conta a estrutura definida pela Figura 4.36.

```
1   // arquivo GerenciadorDePublicacoes.java
2   public class GerenciadorDePublicacoes {
3       private RepositorioDePublicacoes rc; // definição de atributo
4           // do tipo RepositorioDePublicacoes\\
5       public GerenciadorDePublicacoes(RepositorioDePublicacoes r) {
6           rc= r; // recebe referencia de um objeto cuja classe
7                  // implementa a interface RepositorioDeContas
8       }
9       void reservar(String tombo, int quant) {
10          Publicacao p = rc.procurar(tombo);
11          p.reservar(quant);
```

```
12              rc.atualizar(p);
13          }
14      } // fim da classe GerenciadorDePublicacoes
```

No código acima, a classe GerenciadorDePublicacoes não faz referência a nenhuma classe que implemente a interface RepositorioDePublicacoes. Ao invés disso, ela referencia apenas RepositorioDePublicacoes. Seu construtor recebe como parâmetro um objeto que implementa a interface RepositorioDePublicacoes e esse objeto é usado pelo método reservar(), independentemente de sua classe de implementação. O trecho de código seguinte mostra como a classe GerenciadorDePublicacoes pode ser usada, em conjunto com alguma implementação de RepositorioDePublicacoes (no exemplo, RepDePublicacoesBDRelacional).

```
1   // arquivo Principal.java
2   public class Principal {
3       public static void main(String args[]) {
4           GerenciadorDePublicacoes gerPublicacoes = new GerenciadorDePublicacoes (
                new
5               RepDePublicacoesBDRelacional());
6           gerPublicacoes.reservar(''G157T'',1);
7           ...
8       }
9       ...
10  } // fim da classe Principal
```

Note que a implementação de RepositorioDePublicacoes só é conhecida no momento em que GerenciamentoDePublicacoes é instanciada (Linhas 4 e 5).

4.7.4 Exemplo 2 - Alternativa à Herança Múltipla de Classes

Interfaces podem ser usadas no lugar de herança múltipla, em linguagens como Java, que não dão suporte a esse recurso. O uso de interfaces nesses casos tem a vantagem de sempre produzir herança por comportamento, já que todas as classes concretas que implementam uma interface devem oferecer implementações para todos os métodos definidos por essa interface. A Figura 4.37 apresenta uma hierarquia de classes que usa herança múltipla. Nessa figura, a classe RetanguloCentrado herda de duas classes distintas: Retangulo e Centrado.

A Figura 4.38 apresenta uma solução alternativa, na qual a classe RetanguloCentrado estende Retangulo e implementa Centrado, que passa a ser uma interface.

O trecho de código seguinte apresenta as implementações da interface Centrado e da classe RetanguloCentrado.

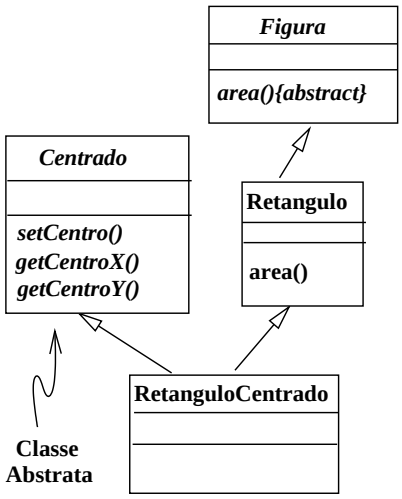


Figura 4.37: Herança Múltipla Entre Classes

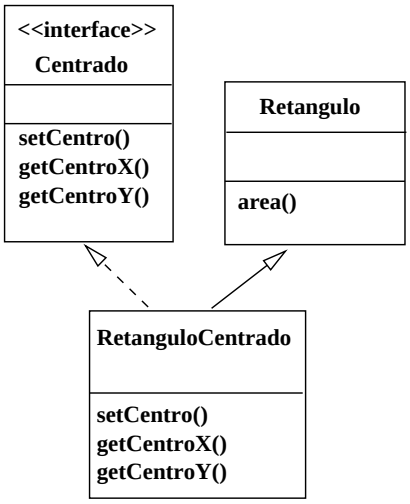


Figura 4.38: Solução em Java Usando Interfaces

```
1   public interface Centrado {
2       public void setCenter(double x, double y);
3       public double getCentroX();
4       public double getCentroY();
5   }
6
7   public class RetanguloCentrado extends Retangulo
8           implements Centrado {
```

```
9         private double cx,cy; // novos atributos
10        public RetanguloCentrado(double cx, double cy, double larg,
11                double alt) {
12            super(larg, alt); // largura e altura do retangulo
13            this.cx = cx;
14            this.cy = cy;
15        }
16        public void setCentro(double x, double y) {
17            cx = x; cy = y;
18        }
19        public getCentroX() {return cx;}
20        public getCentroY() {return cy;}
21    }
```

Suponha que implementemos outras duas classes, CirculoCentrado e QuadradoCentrado, do mesmo modo que RetanguloCentrado. A classe Figura (Figura 4.37) é superclasse de Circulo, Quadrado e Retangulo. Cada classe derivada redefine a operação area() herdada do pai. A Figura 4.39 apresenta essa hierarquia.

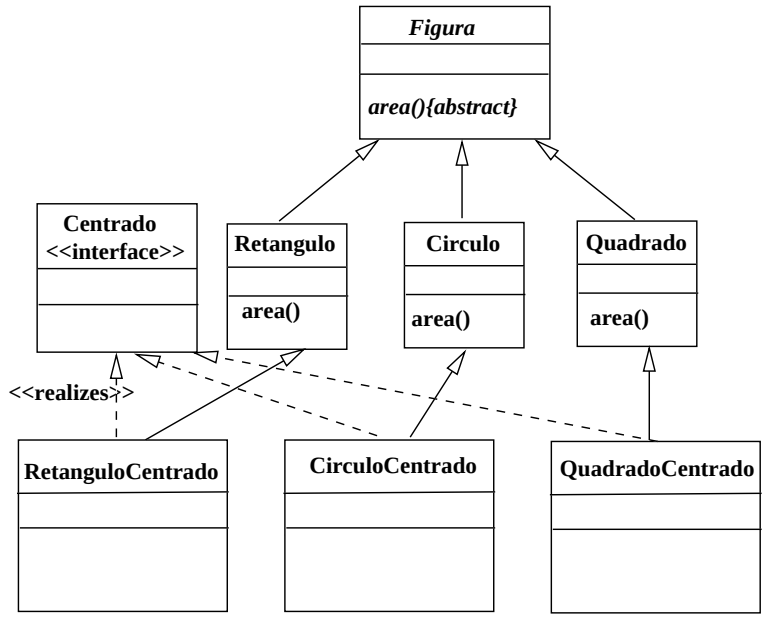


Figura 4.39: Uma Hierarquia de Figuras Geométricas

O seguinte trecho de código utiliza a hierarquia mostrada na figura 4.39 para realizar algumas operações com figuras, abstraindo o tipo específico da figura que é manipulada.

```
1     Figura[ ] shapes = new Figura[3]; // cria um array de referencias
2                                       // para objetos do tipo Figura
3     shapes[0] = new RetanguloCentrado(1.0, 1.0, 1.0, 1.0);
4     shapes[1] = new QuadradoCentrado(2.5, 2.0, 3.0);
5     shapes[2] = new CirculoCentrado(2.3, 4.5, 3.4);
6     double totalArea = 0;
7     double totalDistancia= 0;
8     for (int i=0; i < shapes.length; i++) {
9         totalArea += shapes[i].area();
10        if (shapes[i] instanceof Centrado) {
11            // é necessário ''casting'' do tipo Figura
12            // para uma subclasse do tipo Centrado
13            Centrado c = (Centrado)shapes[i];
14            double cx = c.getCentroX();
15            double  cy = c.getCentroY();
16            double totalDistance+= Math.sqrt(cx * cx + cy * cy);
17               // fórmula para calcular a distância da origem(0,0)
18               // ao centro da figura (cx, cy)
19        } // fecha o if
20    } // fecha o for
21    System.out.println(''Área Média:'' + totalArea/shapes.length);
22    System.out.println(''Distância Média:'' + totalDistance/shapes.length);
```

Interfaces são tipos, tal como as classes. Quando uma classe implementa uma interface, instâncias daquela classe podem ser atribuídas às variáveis do tipo da interface. Conseqüentemente, não é necessário fazer *casting* de irmãos de RetanguloCentrado para Centrado, antes de chamarmos os métodos setCentro(), getCentroX() ou getCentroY(). RetanguloCentrado implementa setCentro(), getCentroX() e getCentroY() e herda o método area() da superclasse Retangulo. No exemplo, o *casting* foi necessário porque o *array* shapes é do tipo Figura e essa classe não implementa Centrado.

4.7.5 Exemplo 3 - Classes, Interfaces e Tipos

Considere a hierarquia de classes da Figura 4.40.

O trecho de código seguinte cria objetos dos tipos C1, C2, C3 e C4 e realiza diversas atribuições válidas.

```
1     C1 c1 = new C1();
2     C2 c2 = new C2();
```

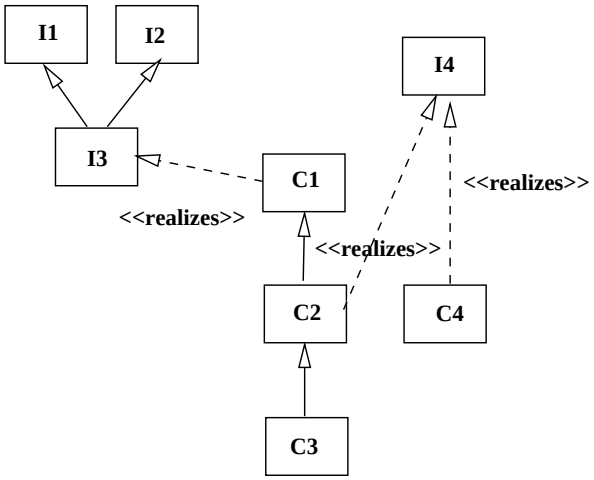


Figura 4.40: Uma Hierarquia Mais Complexa

```
3     C3 c3 = new C3();
4     C4 c4 = new C4();
5
6     I1 i1 = c1; // classe C1 implementa a interface I1
7     I1 i1 = c2; // classe C2 implementa a interface I1
8     I1 i1 = c3; // classe C3 implementa a interface I1
9     I2 i2 = c1; // classe C1 implementa a interface I2
10    I2 i2 = c2; // classe C2 implementa a interface I2
11    I2 i2 = c3; // classe C3 implementa a interface I2
12    I3 i3 = c1; // classe C1 implementa a interface I3
13    I3 i3 = c2; // classe C2 implementa a interface I3
14    I3 i3 = c3; // classe C3 implementa a interface I3
15    I4 i4 = c4; // classe C4 implementa a interface I4
16    I4 i4 = c2; // classe C2 implementa a interface I4
17    I4 i4 = c3; // classe C3 implementa a interface I4
```

4.8 Pacotes em UML

As unidades básicas de abstração e encapsulamento, em orientação a objetos, são os tipos abstratos de dados, representados por definições de classes. A quantidade de classes numa aplicação real pode atingir facilmente a casa das centenas, ou mesmo milhares. Qualquer sistema dessa magnitude exige alguma forma de modularização que abstraia a maior parte da complexidade do sistema, agrupando

os seus elementos internos que podem ser vistos como unidades indivisíveis, em níveis de abstração mais elevados.

Dois princípios fundamentais regem a construção de sistemas computacionais complexos: abstração e modularização. Como visto nas Seções 1.2.2 e 1.1.1 do Capítulo 1, abstrações nos permitem entender e analisar o problema concentrando-se nos aspectos relevantes e ignorando detalhes irrelevantes. Uma das formas mais eficazes de abstrair os detalhes indesejados de sistemas de software é através da do desenvolvimento modular. A modularização nos permite projetar e construir um sistema computacional a partir de partes menores, chamados módulos ou pacotes. No desenvolvimento modular, durante o projeto de construção de um sistema, é criada uma estrutura de módulos para o programa; se esses módulos corresponderem às abstrações descobertas durante a análise do problema, então o sistema será mais facilmente compreendido e gerenciado.

Dada a sua importância, a fase de projeto de software tem como um de seus objetivos encontrar uma decomposição modular que seja adequada para a implementação do sistema. Uma decomposição modular é considerada boa quando ela é composta por módulos que sejam independentes o máximo possível uns dos outros, isto é, que haja um fraco acoplamento entre os módulos. Para isso, cada módulo deve "esconder" uma decisão particular de projeto; de forma que, se essa decisão for mudada, apenas o módulo que "conhece" a decisão tomada será modificado. Os outros módulos permanecem inalterados. Para que isso seja possível, uma decomposição modular (ou fatoração) de boa qualidade deve proporcionar um fraco acoplamento entre os diferentes pacotes e alta coesão entre os elementos de um mesmo pacote. Quando um projeto é composto por módulos altamente independentes entre si, esses módulos podem ser utilizados como unidade para atribuição de trabalho entre os membros do time de programação. Além disso, como discutido na Seção 1.5.3 do Capítulo 1, a existência de elementos independentes entre si facilita as atividades de manutenção do sistema, uma vez que os efeitos das modificações são restritos a módulos individuais.

Um exemplo dessa estrutura pode ser visto na Figura 4.41, que apresenta duas aplicações (`sistemaBibliotecas` e `controleDados`). Nela, as pastas representam pacotes, enquanto os retângulos correspondem a classes na sua visão simplificada (ver Seção 1.3.2 do Capítulo 1). A aplicação `sistemaBibliotecas` é composta pelas classes Publicacao, Controlador, Livro e Periodico e pelo pacote `reservas`, que é utilizado pela classe Publicacao; o pacote `reservas` é composto pelas classes Rerserva e Controlador. Sobre a aplicação controleDados, sabe-se apenas que ela é utilizada pelos dois controladores da aplicação `sistemaBibliotecas`, a sua complexidade interna foi totalmente abstraída.

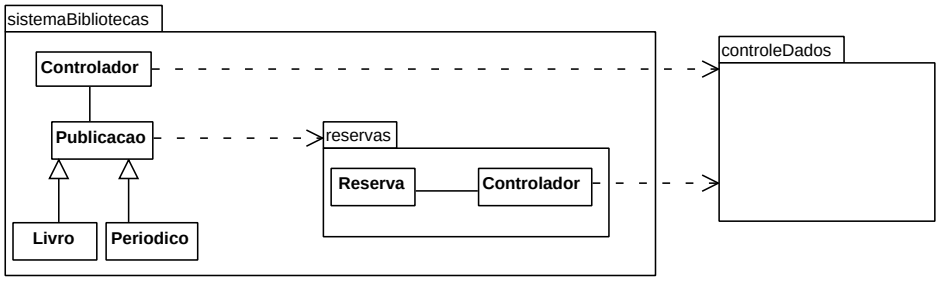


Figura 4.41: Exemplo de Organização em Pacotes

Em Java, um pacote é representado por um diretório do sistema de arquivos, que é utilizado para armazenar o código da aplicação, inclusive outros pacotes subordinados hierarquicamente (subdiretórios). Todas as definições de tipo contidas num pacote precisam identificar a hierarquia completa de pacotes que o contém, que em Java é indicado através da palavra reservada `package`. O trecho de código seguinte mostra as declarações dos pacotes feitas em cada uma das classes apresentadas na Figura 4.41.

```
1  // arquivo sistemaBibliotecas/Publicacao.java
2  package sistemaBibliotecas; // Pertence ao pacote sistemaBibliotecas.
3  ...
4
5  // arquivo sistemaBibliotecas/Controlador.java
6  package sistemaBibliotecas; // Pertence ao pacote sistemaBibliotecas.
7  ...
8
9  // arquivo sistemaBibliotecas/Livro.java
10 package sistemaBibliotecas; // Pertence ao pacote sistemaBibliotecas.
11 ...
12
13 // arquivo sistemaBibliotecas/Periodico.java
14 package sistemaBibliotecas; // Pertence ao pacote sistemaBibliotecas.
15 ...
16
17 // arquivo sistemaBibliotecas/reservas/Reserva.java
18 package sistemaBibliotecas.reservas; // Pertence ao pacote reservas, que
19 ...                                  // pertence ao pacote sistemaBibliotecas.
20
21 // arquivo sistemaBibliotecas/reservas/Controlador.java
22 package sistemaBibliotecas.reservas; // Pertence ao pacote reservas, que
23 ...                                  // pertence ao pacote sistemaBibliotecas.
```

Em algumas linguagens, como C# [24], pacotes são conhecidos como espaços de nomes (do inglês *namespaces*). Essa nomenclatura é coerente, já que classes com o mesmo nome podem ser empregadas em um mesmo sistema, através do uso de pacotes. No exemplo acima há duas classes com nome Controlador, mas não há perigo das duas serem confundidas, já que pertencem a pacotes (espaços de nomes) distintos.

4.8.1 Visibilidade de Classes, Atributos e Operações

As Seções 4.1.4 e 4.4.2 apresentaram os conceitos de visibilidade pública (‘‘+’’), privada (‘‘-’’) e protegida (‘‘#’’), que podem ser utilizadas em atributos e operações das classes. Com o conceito de pacotes, surge a necessidade de se ter mais um nível de visibilidade: a visibilidade de pacote (‘‘∼’’).

As classes, atributos e métodos declarados com visibilidade “de pacote” são acessíveis por todas as classes dentro de um mesmo módulo. Em Java, a visibilidade “de pacote” é indicada pela ausência de todos os outros indicador de visibilidade: `public`, `private` ou `protected`.

Em UML é possível representar graficamente as quatro visibilidades apresentadas. A Figura 4.42 mostra um exemplo dessa representação em uma classe. Como pode ser visto, cada método ou atributo pode ser precedido por um símbolo que indica a sua visibilidade: ‘+’→ visibilidade pública; ‘-’→ visibilidade privada; ‘#’→ visibilidade protegida; e ‘∼’→ visibilidade “de pacote”.

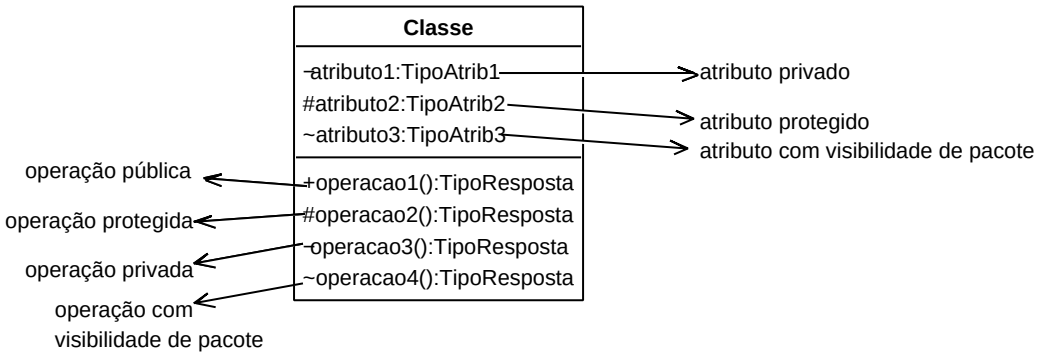


Figura 4.42: Visibilidade de atributos e operações em UML

Em UML, para que uma classe contida num pacote possa ser utilizado por classes de outros pacotes, deve ser definida com visibilidade pública. Se isso não for feito, a classe só será visível para as classes dentro do mesmo pacote, ficando invisíveis até mesmo para as classes de seus subpacotes. Por exemplo, se a classe Publicacao da Figura 4.41 não for declarada como pública, ela só será visível para as classes Livro, Periodico e Controlador do pacote `sistemaBibliotecas` (ou simplesmente sistemaBibliotecas.Controlador). A Figura 4.43 mostra a hierarquia de pacotes apresentada na Figura 4.41 do ponto de vista de um observador externo, assumindo que as únicas classes públicas são sistemaBibliotecas.Controlador e sistemaBibliotecas.reservas.Controlador.

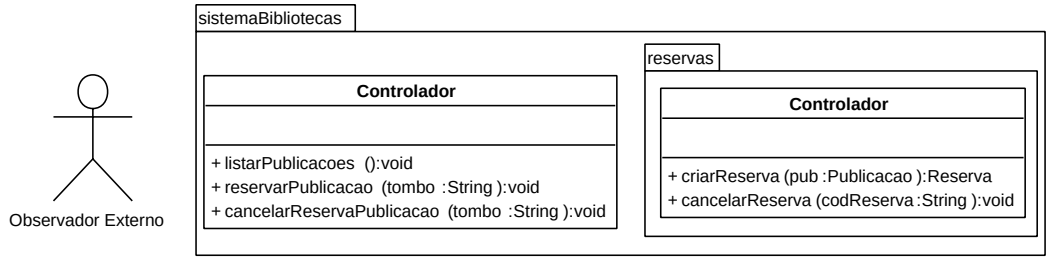


Figura 4.43: Exemplo de Visibilidade de Pacote em UML

4.9 Relacionamento de Delegação

4.9.1 Sistemas baseados em Delegação

No modelo de objetos, o mecanismo de herança permite compartilhamento de comportamento (do inglês: *behaviour sharing*), isto é, reutilização de operações, entre classes derivadas e classes bases. O mecanismo de herança não é aplicado a objetos individualmente; entretanto, existe um mecanismo chamado de **delegação**, parecido com o de herança, que também permite compartilhamento de comportamento, mas que opera diretamente entre objetos, não entre classes.

Nas linguagens de programação baseadas no mecanismo de delegação, tais como Self [47] e Actor [30], objetos são vistos como **protótipos** ou **exemplares**. Os protótipos têm o papel de **delegadores** ao delegarem seu comportamento para protótipos relacionados chamados de **delegados**. Sistemas baseados em delegação, em geral, não têm como foco principal a estruturação baseada em classes.

O relacionamento de **delega-para** entre um delegador e um delegado pode ser estabelecido dinamicamente enquanto que o relacionamento de herança de linguagens baseadas em classes, tais como C++ e Java, é estabelecido e estaticamente fixado quando a classe é criada. Todos os objetos de uma classe têm uma mesma estrutura e compartilham um comportamento comum. Como conseqüência, alterações efetuadas nos métodos e estrutura de uma classe, podem ser automaticamente passadas para todas as suas instâncias.

Este mecanismo implícito de propagação de alterações permite que sistemas computacionais possam ser modificados com base em grupos/classes. Portanto, linguagens baseadas em classes usam um compartilhamento de comportamento estático e baseado em grupos, constratando com as linguagens baseadas em delegação que proporcionam um compartilhamento de comportamento dinâmico e basedo em protótipos (objetos).

4.9.2 Delegação versus Herança

Em sistemas baseados em delegação, as distinções entre classes e objetos são removidas. Na verdade, a noção de classes é eliminada, ficando apenas a noção de objetos, no caso, chamado de protótipos. O desenvolvedor inicialmente identifica um conjunto de protótipos, e depois descobre similaridades e/ou diferenças com outros objetos. Dentro dessa nova concepção, um objeto antigo pode tornar-se um protótipo para um objeto recém-descoberto. A idéia é começar com casos particulares, e depois generalizá-los ou especializá-los.

Por exemplo, suponha que um desenvolvedor inicialmente identifique um objeto retangular, e, em seguida, ele identifique um objeto quadrado. Nesse caso, podemos dizer que o quadrado "se parece" com um retângulo. Suponha ainda que posteriormente o desenvolvedor identifique um outro quadrado cujo tamanho seja maior do que o primeiro quadrado.

Num sistema baseado em protótipos, podemos definir o retângulo como sendo o protótipo para a criação do primeiro quadrado. Similarmente, o primeiro quadrado pode ser definido como o protótipo de criação para o segundo quadrado. Vale ressaltar, que nessa nova concepção, a noção de classe, a partir da qual um objeto é criado, não existe.

Seja ret1 o protótipo retângulo, qua1 o protótipo primeiro quadrado e qua2 o protótipo segundo quadrado (Figura 4.44). O protótipo ret1 tem operações, tais como, mover(), rotacionar(), e reajustarTamanho(), e também operações de inspeção de atributos, como por exemplo, getCentro(), getPerimetro() e getRazao(), que retornam valores relativos aos atributos centro, lado e altura.

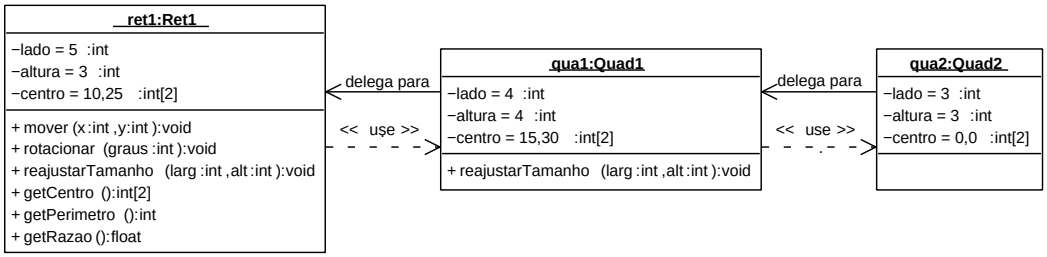


Figura 4.44: Exemplo de Protótipos Geométricos

O protótipo qua1 é similar ao protótipo ret1; a única diferença são os valores dos atributos centro, lado e altura. O objeto qua1 tem uma associação de **delega-para** com o objeto ret1. Isto significa que os atributos e operações que não são redefinidos em qua1 serão "herdados" de ret1. No nosso exemplo, a operação reajustarTamanho() foi redefinida no protótipo qua1, significando que sua execução não será delegada para ret1. Portanto, qua1 delega a execução de mensagens, como por exemplo rotacionar(), para o seu objeto delegado ret1. No entanto, vale ressaltar que quando a operação rotacionar() é executada no protótipo ret1, essa operação utiliza o estado do objeto qua1, isto é, o receptor inicial da mensagem.

De forma similar, qua2 delega todas as suas mensagens recebidas para qua1. Por exemplo, se qua2 recebe a mensagem mover(), ela será inicialmente delegada para qua1. Desde que a mensagem não pode ser executada por qua1, ela será delegada para o objeto ret1. Entretanto, quando a

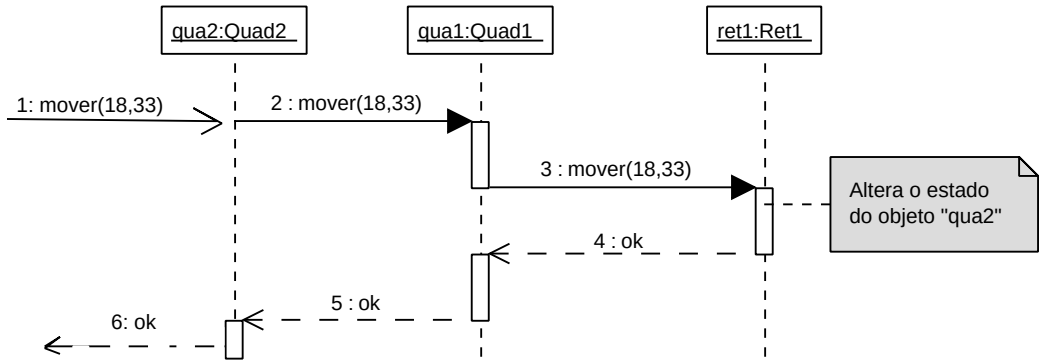


Figura 4.45: Delegação de mensagens entre protótipos

operação mover() for executada no protótipo ret1, ela acessará o estado do objeto qua2, o receptor inicial da mensagem. A Figura 4.45 mostra a sequência de delegação da mensagem mover().

O exemplo dos protótipos geométricos pode ser remodelado usando a noção de classes através da criação de uma superclasse Ret1, que possui a subclasse Quad1, que por sua vez e superclasse de Quad2. Além disso, os objetos ret1, qua1 e qua2 devem ser instanciados (Figura 4.46). Nesse modelo, quando o objeto qua1 recebe a mensagem rotacionar(), a busca pela operação se inicia pela classe Quad2. Como ela não é encontrada, a busca continua na superclasse Quad1, e persistindo a ausência, continua na superclasse Ret1, exatamente como no caso dos protótipos baseados em delegação. Quando a operação é encontrada na classe Ret1, ela é executada no estado do objeto qua2, o receptor inicial da mensagem.

As linguagens baseadas em delegação implementam implicitamente o mecanismo de delegação entre protótipos, da mesma forma que o mecanismo de herança entre classes é implicitamente implementado em linguagens basedas em classes através do relacionamento de generalização/especialização entre classes. Entretando, o mecanismo de delegação é mais genérico do que o mecanismo de herança, no sentido que o mecanismo de delegação pode simular o mecanismo de herança, mas não vice-versa. Em sistemas baseados em delegação, a lista de delegados de um protótipo pode conter zero ou muitos delegados, que podem ser alterados dinamicamente.

4.9.3 Delegação em Sistemas Basedos em Classes

Compartilhamento de comportamento pode ser obtido pelo menos de duas formas diferentes: (i) compartilhamento baseado em classes (mecanismo de herança) e (ii) compartilhamento baseado em protótipos/objetos (mecanismo de delegação), como discutido nas seções 4.9.1 e 4.9.2. Embora herança e delegação sejam geralmente definidas como soluções alternativas no projeto de sistemas orientados a objetos, delegação pode ser "simulada" em linguagens baseadas em classes de forma sf ad hoc.

Essa implementação de delegação em sistemas baseados em herança é uma forma de implementar

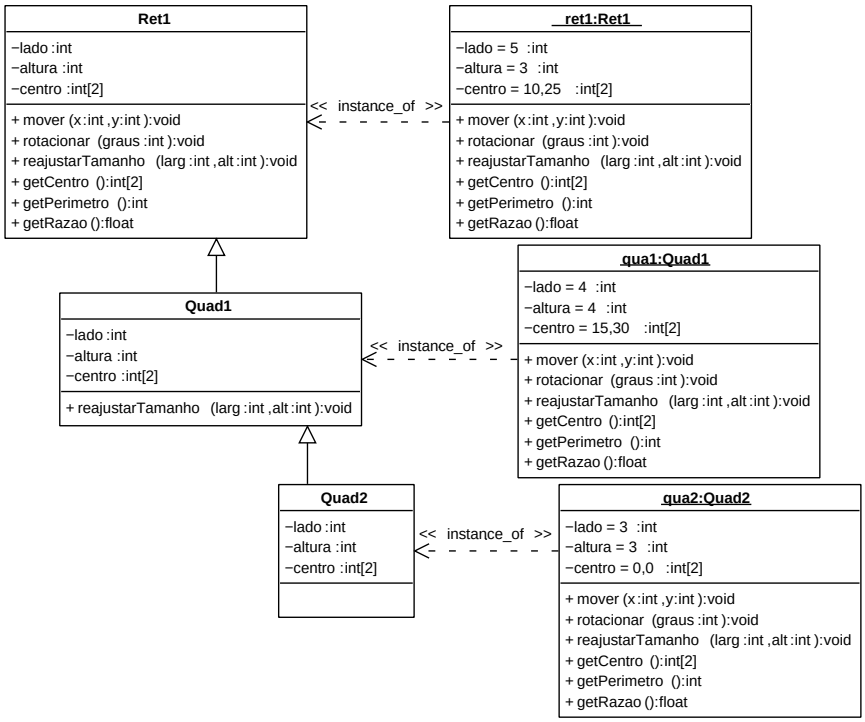


Figura 4.46: Figuras Geométricas baseadas em Classes

compartilhamento de comportamento quando um objeto precisa, por exemplo, ser capaz de alterar suas respostas para pedidos de serviços em tempo de execução. A natureza do mecanismo de herança é estática e relacionada com classes enquanto que a natureza do mecanismo de delegação é dinâmica e concentrada em objetos, o que traz certa flexibilidade para o sistema.

A principal vantagem de delegação sobre herança é que delegação facilita a mudança de implementação de operações em tempo de execução. A maioria das linguagens baseadas em classes não permitem que objetos mudem de classes pois eles estariam alterando os seus tipos em tempo de execução; entretanto, é fácil mudar o delegado de um objeto sem necessariamente isso implicar numa mudança de grupo.

Além disso, uma linguagem baseada em classes com verificação estática de tipos pode garantir que um delegado entenderá todas as mensagens delegadas pelo delegador. Portanto, delegação é compatível com verificação estática de tipos presumindo-se que todos os delegados sejam conhecidos em tempo de compilação. Em tempo de execução, existe a escolha de qual delegado será ativado. Portanto, delegação pode ser vista como uma técnica de projeto alternativa que pode ser usada em qualquer linguagem orientada a objetos baseada em classes, incluindo aquelas que são estaticamente tipadas.

4.9.4 O Padrão Delegação para Sistemas baseados em Classes

Motivação

Em sistemas orientados a objetos convencionais baseados em classes, que usam o mecanismo de herança, **todas** as instâncias de uma determinada classe devem seguir o mesmo padrão de comportamento. Uma vez que um objeto é criado, ele está permanentemente ligado ao comportamento estabelcido para o grupo ao qual ele pertence.

Muitas entidades do mundo real exibem diferentes tipos de comportamentos em diferentes estágios de vida, em particular, depedendo se a entidade está operando corretamente ou não. É desejável encontrar meios através dos quais objetos que representam tais entidades possam realizar tais mudanças de comportamento de forma explícita documentada no projeto do sistema.

Outra restrição de modelar um sistema apenas baseado no conceito de classe, é que muitas vezes não é claro ou mesmo não é conhecido no início da concepção do sistema, todos os diferentes grupos existentes no domínio do problema para que todas as classes sejam criadas inicialmente. O desenvolvedor gostaria de reutilizar comportamentos já implementados por outros protótipos sem ter a obrigação de garantir que o relacionamento de generalização/especialização é verdadeiro.

Solução

Como ilustrado na Figura 4.47, delegação pode ser implementada numa linguagem baseada em classes através da inclusão do receptor original (isto é, o objeto delegador) como um parâmetro extra para cada mensagem delegada para o objeto delegado.

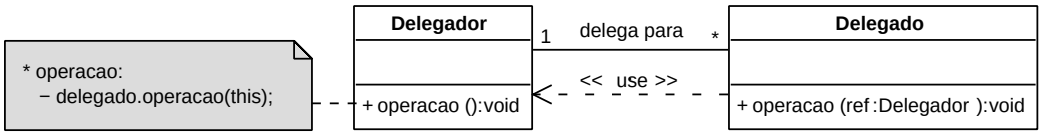


Figura 4.47: Implementação de Delegação em Linguagens Baseadas em Classes

Consequências

A implementação do padrão de delegação numa linguagem baseda em classe requer, portanto, um esforço extra por parte do desenvolvedor pois: (i) o objeto delegador e os seus objetos delegados devem ser estaticamente definidos (isto é, não existiria criação dinâmica de delegados a princípio), e (ii) um objeto delegado ao receber uma mensagem delegada, utiliza o parâmetro extra com a referência do receptor inicial da mensagem para realizar operações no estado do delegador.

O padrão acima pode ser recursivamente aplicado, no sentido de que um objeto delegado pode assumir o papel de delegador e ter sua própria lista de delegados, e assim por diante. Em linguagens baseadas em delegação, essa noção mais geral de vários nívies de delegadores é explorada, conforme explicado na seção 4.9.1. O leitor deve se lembrar, entretanto, que a referência a ser repassada para a cadeia de delegados armazena a referência para o receptor original da mensagem, isto é, o delegador inicial. De forma geral, existe um delegador inicial que recebe a mensagem inicialmente e

um conjunto de delegadores intermediários que repassam a execução da mensagem até que o código da operação seja encontrado.

Linguagens basedas em delegação implementam este nível extra de indireção de forma transparente. Portanto, o programador pode facilmente implementar mudanças de comportamento de entidades do mundo real através da delegação de operações para os diferentes delegados representando os diferentes comportamentos que a entidade externa pode ter. Entretanto, não existe garantia de que os diversos delegados representando os diferentes **estados lógicos** de uma entidade externa sejam aderentes a uma mesma especificação (geralmente não existe verificação estática de tipos). Em outras palavras, linguagens baseadas em delegação enfatizam flexibilidade, isto é, dão suporte para mudanças em tempo de execução baseando-se em verificação dinâmica de tipos.

4.9.5 Uma Aplicação do Padrão Delegação

Problema

O estudo dos aspectos dinâmicos de entidades no domínio do problema nos leva a avaliar o comportamento de um objeto através do tempo. Quando nós consideramos o comportamento de entidades no mundo real, nós observamos que elas geralmente tem um **ciclo de vida**. Desde que todas as instâncias de uma classe devem seguir as mesmas regras de comportamento, quando nós abstraimos um grupo de entidades similares no domínio do problema em objetos correspondentes no domínio da solução, nós também abstraimos o seu comportamento comum. Muitas vezes entidades no mundo real exibem diferentes fases de comportamento durante o seu ciclo de vida. Por exemplo, uma borboleta começa a vida como uma caterpilar, depois muda para uma crisálida e finalmente se transforam numa borboleta adulta.

O termo **estado lógico** é aplicado para cada fase de comportamento observável da borboleta, isto é, caterpilar, crisálida e borboleta adulta. Num dado momento, diferentes instâncias de uma classe podem estar em estados lógicos diferentes. O estado lógico de uma instância em particular é chamado de bf estado lógico corrente. Entretanto, a implementação do comportamento é diferente nesses três estados lógicos. Por exemplo, uma operção como mover() precisa ser implementada de pelo menos duas formas distintas quando a borboleta é ainda caterpilar e depois quando ela se transforma em adulta.

Em geral, os estados lógicos são informações obtidas na fase de análise, e linguagens de programação baseadas em classes na maioria das vezes forçam que a informação sobre estados lógicos fique escondida na implementação dos métodos de um objeto. É desejável encontrar uma forma explícita de representar estados lógicos ao invés de adotar uma abordagem de representação implícita através do uso de comandos condicionais.

Em algumas situações, o comportamento de um objeto depende do seu estado e esse comportamente precisa ser modificado em tempo de execução, quando esse estado interno muda. Normalmente, esse problema é resolvido através de métodos que usam comandos condicionais para decidir qual comportamento deve ser adotado, dependendo do estado do objeto. Essa abordagem não é recomendada, porém, porque código pode ser repetido em vários métodos (tanto código relativo

às condições quanto ao comportamento). Além disso, um número grande de condições complexas torna o código dos métodos muito difícil de entender e manter.

Exemplo Motivador

Para ilustrar nossas idéias considere o problema típico de um sistema de controle de bibliotecas bem simples:

> Uma biblioteca tem um conjunto de usuários. Suponha que existe esses usuários possam reservar publicações, de tal modo que uma publicação reservada não possa ser reservada por outro usuário. Além disso, o usuário pode a qualquer cancelar a sua reserva a qualquer momento. Porém, se a publicação estiver bloqueada por algum professor, isso implica que a sua utilização é exclusivamente para consulta dentro da biblioteca, não sendo possível efetuar reservas.

Primeira Solução

Uma solução possível é representar os estados lógicos de um objeto Publicacao dentro dos atributos da própria classe. Mudanças nos valor desses atributos são detectados por comandos condicionais.

```
1  class Publicacao {
2     double status; // 0:disponivel; 1:reservada, 2:bloqueada
3
4     public reservar(){
5     ...
6     if (status = 0) {
7        //realizar a reserva
8     }
9     else {
10       // imprimir uma mensagem de aviso: operação indisponível
11    }
12
13    public cancelarReserva(){
14       if (status == 1)
15          status = 0;
16       }
17    }
18 }
```

Segunda Solução

Uma solução alternativa é elimar os comandos condicionais inseridos na implementação das operações da classe PUBLICACAO (Figura 4.48). A criação de tipos PublicacaoDisponivel, PublicacaoReservada e PublicacaoBloqueada substitui o atributo flagNormal que guarda uma informação de tipo na proposta da primeira implementação.

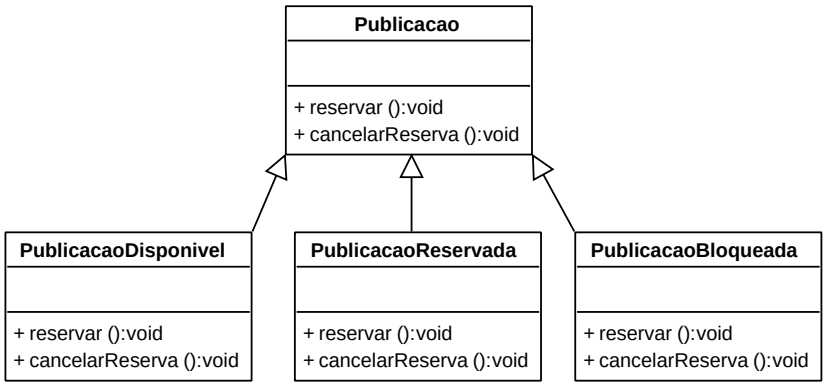


Figura 4.48: Representação Explícita dos Estados da Publicação

Nessa solução, as operações `reservar()` e `cancelarReserva()` estão redefinidas nas classes PUBLICACAODISPONIVEL, PUBLICACAORESERVADA e PublicacaoBloqueada. Essa solução captura o relacionamento entre as diferentes informações sobre uma publicação, mas apresenta pelo menos duas limitações:

1. Linguagens orientadas a objetos da escola escandinava, como Java e C++, não permitem que objetos mudem de classes. Isto implica que quando um objeto publicacao muda de estado, por exemplo, de PUBLICACAODISPONIVEL para PUBLICACAORESERVADA, um novo objeto deve ser criado e o antigo objeto PUBLICACAODISPONIVEL deve ser destruído. Entretanto, essa operação de destruição e criação para sistemas complexos pode ser bem custosa pois as identidades desses objetos não são mantidas ao longo dos seus ciclos de vida. Este problema é crucial se outros objetos do sistema mantém referências para objetos antigos.

   Uma possível solução é criar uma tabela contendo referências para todos os objetos mutavéis. Outra solução é manter uma lista de dependências dos objetos que referenciam objetos mutáveis. Portanto, quando a identidade do objeto muda, é possível avisar todos os objetos que devem ser atualizados. Essa solução está documentada no padrão *Observer* [19].

2. As classes PublicacaoDisponivel, PublicacaoReservada e PublicacaoBloqueada não são realmente subtipos de publicação, mas sim estados conceituais separados pertencentes a uma mesma abstração. Na verdade, os estados lógicoa de uma publicação podem ser classificados como disponível, reservada, e bloqueada, e não a publicação propriamente dita.

Terceira Solução: o padrão de projeto *state*

O padrão de projeto *state* permite que um objeto altere seu comportamento quando seu estado interno muda. Ele usa o mecanismo de delegação e o relacionamento de agregação para

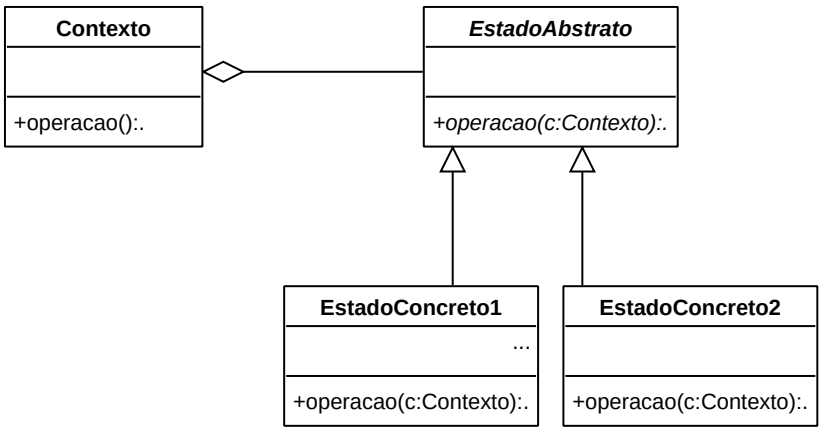


Figura 4.49: Estrutura do Padrão de Projeto *State*

lidar com situações em que um objeto precisa se comportar de maneiras diferentes em diferentes circunstâncias.

Sendo assim, a terceira solução consiste em criar classes que encapsulam os diferentes estados lógicos do objeto conta e os comportamentos associados a esses estados. Dessa forma, a complexidade do objeto é quebrada em um conjunto de objetos menores e mais coesos. A Figura 4.49 mostra a estrutura geral do padrão State.

Na Figura 4.49, a classe Contexto define objetos cujos estados são decompostos em diversos objetos diferentes definidos por subtipos da classe abstrata Estado. A classe ESTADO define uma interface unificada para encapsular o comportamento associado a um estado particular de de um objeto do tipo CONTEXTO. Um objeto do tipo Contexto mantém uma referência para uma instância de uma subclasse concreta de Estado e, para cada estado de Contexto, é definida uma subclasse EstadoConcreto de Estado que implementa o comportamento associado a esse estado.

Um objeto Contexto delega requisições dependentes de estado para o objeto EstadoConcreto atual, de acordo com o padrão delegação da seção4.9.4. A classe Contexto passa uma referência de si própio como argumento para o objeto do tipo Estado responsável por tratar a requisição. Clientes podem configurar um contexto com objetos do tipo Estado. Uma vez que isso seja feito, porém, esses clientes passam a se comunicar apenas com o objeto do tipo Contexto e não interferem mais com seu estado.

Consequências

O padrão *state* tem as seguintes consequências:

- Localiza o comportamento dependente de estado e particiona o comportamento relativo a diferentes estados. O padrão substitui uma classe Contexto complexa por uma classe Contexto muito mais simples e um conjunto de subclasses de Estado altamente coesas.

- Torna transições de estado explícitas, ao invés de espalhá-las em comandos condicionais.
- Torna o estado de um objeto compartilhável, caracterizando uma quebra de encapsulamento entre a classe Contexto e a classe Estado.

O padrão *state* aplicado na hierarquia de publicações

A Figura 4.50 apresenta a aplicação do padrão *state* no contexto da hierarquia de publicações. Perceba que do ponto de vista de quem está fora do pacote, a única classe visível é a classe Publicacao. Mesmo assim, as operações `defineEstadoDisponivel()`, `defineEstadoReservada()` e `defineEstadoBloqueada()` não são visíveis externamente ao pacote `publicacao`.

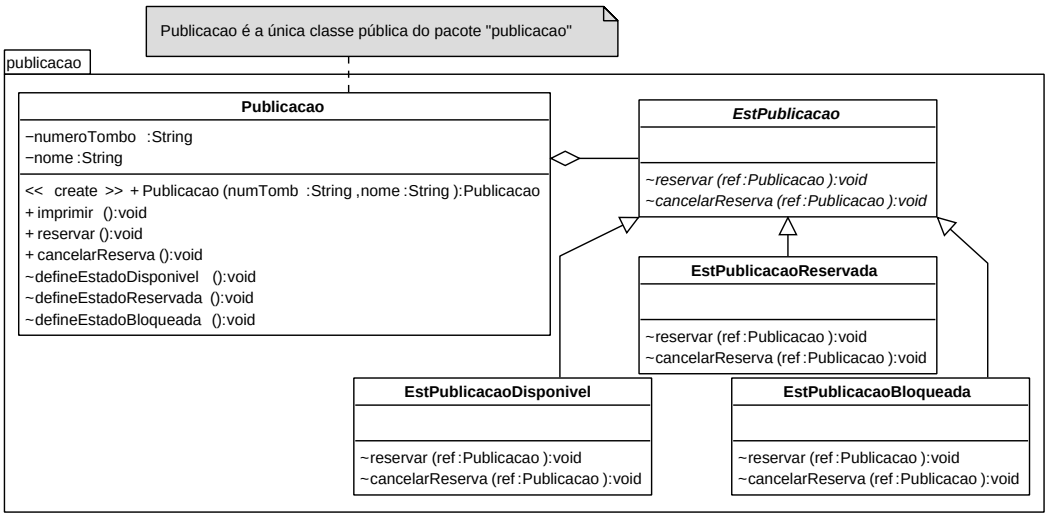


Figura 4.50: Padrão *state* Aplicado na Hierarquia de Publicações

A seguir, é apresentada a implementação em Java do modelo apresentado na Figura 4.50.

```
1 //Publicacao.java
2 package publicacao;
3 public class Publicacao {
4     //Constantes
5     private static EstPublicacaoDisponivel estDisponivel = new
          EstPublicacaoDisponivel();
6     private static EstPublicacaoReservada estReservada = new EstPublicacaoReservada()
          ;
7     private static EstPublicacaoBloqueada estBloqueada = new EstPublicacaoBloqueada()
          ;
8     //Atributos
9     ...
```

```
10    //Associacoes
11    private EstPublicacao estado;
12
13    //Operações
14    ...
15    public reservar(){estado.reservar(this)}
16    public cancelarReserva(){estado.cancelarReserva()}
17
18 class abstract EstPublicacao extends Conta {
19    public debitar(float qtde){
20       //  realizar o saque
21       if (saldo = 0) {
22          // transição de ContaNormal para ContaAnormal,
23          // elimando o objeto do tipo
24          // ContaNormal e criando um objeto ContaAnormal
25       }
26    }
27 }
28
29 //EstPublicacao.java
30 package publicacao;
31 import publicacao.Publicacao;
32 public abstract class EstPublicacao {
33    //Operações
34    public Publicacao publicacao;
35    abstract void reservar(Publicacao ref);
36    abstract void cancelarReserva(Publicacao ref);
37 }
38
39
40 //EstPublicacaoDisponivel.java
41 package publicacao;
42 public class EstPublicacaoDisponivel extends publicacao.EstPublicacao {
43    //Operações
44    void reservar(Publicacao ref) {ref.defineEstadoReservada();}
45    void cancelarReserva(Publicacao ref) {System.out.println(``A publicação já está
          disponível'');}
```

```
46 }
47
48 //EstPublicacaoReservada.java
49 package publicacao;
50 public class EstPublicacaoReservada extends publicacao.EstPublicacao {
51    //Operações
52    void reservar(Publicacao ref) {System.out.println(``A publicação já está
       disponível'');}
53    void cancelarReserva(Publicacao ref) {ref.defineEstadoDisponivel();}
54 }
55
56 //EstPublicacaoBloqueada.java
57 package publicacao;
58 public class EstPublicacaoBloqueada extends publicacao.EstPublicacao {
59    //Operações
60    void reservar(Publicacao ref) {System.out.println(``A publicação não pode ser
       reservada'');}
61    void cancelarReserva(Publicacao ref) {``A publicação não pode ser reservada''}
62 }
```

4.10 Metaclasses

Metainformação é um termo genérico aplicado a qualquer dado que descreve outro dado. Em particular, na maioria das linguagens de programação orientadas a objetos, classes são vistas como "fábricas" que criam e inicializam instâncias (i.e. objetos). As classes não são objetos porque elas próprias não são instâncias de classes. Entretanto, em algumas linguagens orientadas a objetos, tal como Smalltalk-76 [31], existem dois tipos de elementos geradores no modelo de objetos: (i) metaclasses que geram classes e (ii) classes que geram instâncias terminais (Figura 4.51). Portanto, todas as entidades do modelo de objetos são objetos, inclusive as classes. Com isto uma uniformização do modelo de objetos é obtida.

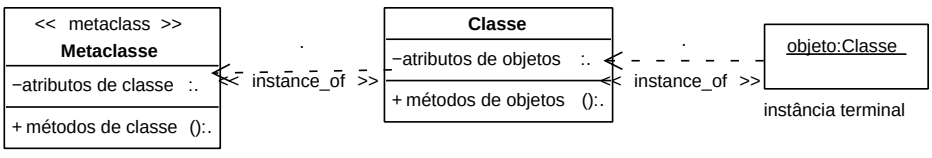


Figura 4.51: O Conceito de Metaclasse

Mais especificamente, metaclasse é uma classe que descreve outra classe, isto é, ela é uma classe cujas instâncias são classes. Em outras palavras, uma metaclasse guarda a metainformação de uma classe. Existem pelo menos dois benefícios da representação de classes como objetos. O primeiro é que as informações globais relativas a todos os objetos de uma classe podem ser armazenadas nos atributos de classe (seguindo a terminologia de Smalltalk). Os métodos associados com a metaclasse (chamados de métodos de classe) podem ser usados para recuperar e atualizar os valores dos atributos de classe. A segunda vantagem é o seu uso para criação/iniciação de novas instâncias da classe. Em Smalltalk-80, por exemplo, existe uma metaclasse distinta para cada classe do sistema e a hierarquia de metaclasses é construída de forma paralela à hierarquia de classes. Algumas linguagens orientadas a objetos, como C++ e Java, não dão apoio direto para a abordagem "classes como instâncias de metaclasses", como Smalltalk. No entanto, C++ e Java dão apoio para as noções de atributos de classes e métodos de classes através de uma "fusão" do conceito de metaclasse com o de classe (Figura 4.52).

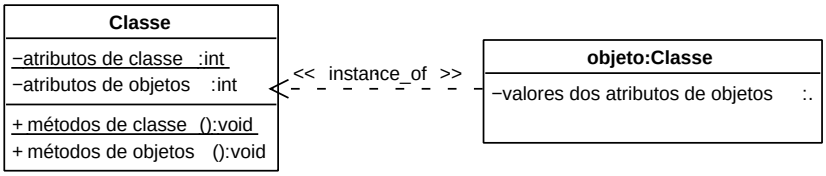


Figura 4.52: "Fusão" de Metaclasse com Classe

De forma mais genérica, podemos dizer que o paradigma de objetos dá suporte a pelo menos três níveis de abstrações: (i) abstração de dados para comunicação de objetos; (ii) super-abstração (herança) para o compartilhamento de comportamento e gerência de objetos; e (iii) meta-abstração (metaclasse) como base para a auto-representação do sistema.

Em UML, metaclasses são representadas através do estereótipo $\ll metaclass \gg$ (Figura 4.53), especificando uma classe (no caso do exemplo, MetaLivro) cujos objetos são classes (classe Quarto). UML também permite a especificação de métodos e atributos de classe. Estes aparecem sublinhados em um diagrama de classes comum, conforme apresentado na Figura 4.54.

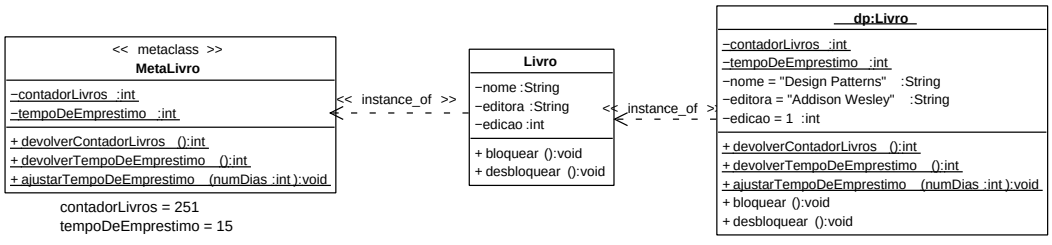


Figura 4.53: Metaclasses

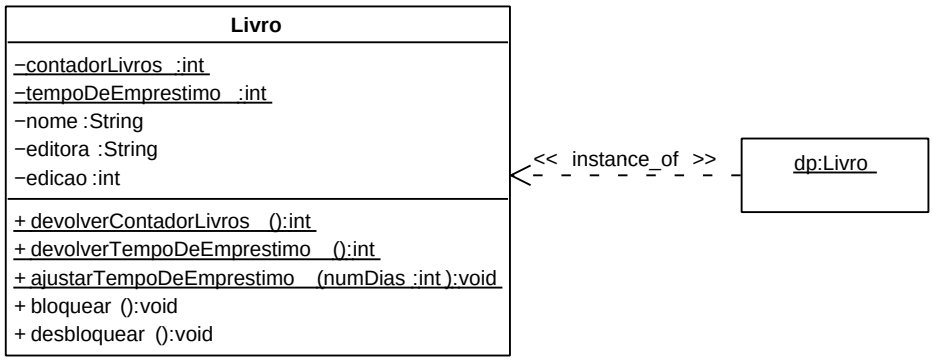


Figura 4.54: Atributos e métodos de classe.

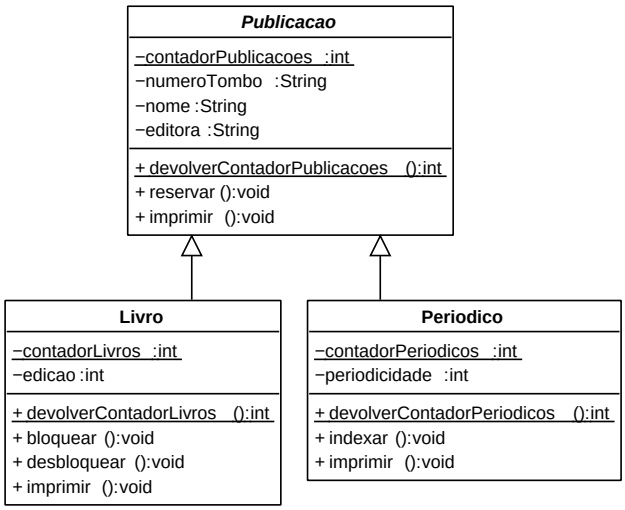


Figura 4.55: Solução com Atributos e Métodos Estáticos

4.10.1 Exemplo de utilização de metaclasses

Vamos imaginar uma situação onde se deseja controlar o número de publicações de uma biblioteca. Além disso, deseja-se saber o número específico dos tipos instanciados: número de livros e número de periódicos.

Primeira Solução: atributos e operações estáticas na classe Livro

A Figura 4.55 apresenta a primeira solução possível para esse problema. Perceba que os atributos `contadorDePublicacoes`, `contadorDeLivros` e `contadorDePeriodicos` pertencem às classes Publicacao, Livro e Periodico, enquanto `numeroTombo`, `nome`, `editora`, edicao e `periodicidade` pertencem aos objetos dessas classes.

O código a seguir apresenta uma implementação em Java para a hierarquia de publicações

apresentada na Figura 4.55. Perceba que os atributos e métodos de classe podem ser acessados diretamente a partir da classe (Linhas 11,25,28,29,44,47 e 48), sem necessitar instanciar qualquer objeto. O acesso desses atributos e operações através do nome da classe (e não de um objeto) é considerada uma boa prática de programação, pois auxilia o desenvolvedor a diferenciar facilmente os elementos da classe e os elementos do objeto.

```
1  //Arquivo Publicacao.java
2  public abstract class Publicacao { // classe abstrata
3      //Atributos da classe
4      private static int contadorPublicacoes;
5      //Atributos do objeto
6      private String numeroTombo;
7      private String nome;
8      private String editora;
9
10     //Operações da classe
11     public static int devolverContadorPublicacoes(){return Publicacao.
           contadorPublicacoes;}
12     //Operações do objeto
13     public void reservar() {...}
14     public void imprimir(){...}
15 }
16
17 //Arquivo Livro.java
18 public class Livro {
19     //Atributos da classe
20     private static int contadorLivros;
21     //Atributos do objeto
22     private int edicao;
23
24     //Operações da classe
25     public static int devolverContadorLivros(){return Livro.contadorLivros;}
26     //Operações do objeto
27     public Livro(){
28         Livro.contadorLivros++; // ou simplesmente ''contadorLivros++''
29         Publicacao.contadorPublicacoes++; // ou simplesmente contadorPublicacoes++''
30     }
31     public void bloquear() {...}
32     public void desbloquear() {...}
```

```
33    public void imprimir(){...}
34 }
35
36 //Arquivo Periodico.java
37 public class Livro {
38    //Atributos da classe
39    private static int contadorPeriodicos;
40    //Atributos do objeto
41    private int periodicidade;
42
43    //Operações da classe
44    public static int devolverContadorPeriodicos(){return Periodico.
          contadorPeriodicos;}
45    //Operações do objeto
46    public Periodico(){
47       Periodico.contadorPeriodico++; // ou simplesmente ''contadorPeriodicos++''
48       Publicacao.contadorPublicacoes++; // ou simplesmente contadorPublicacoes++''
49    }
50    public void indexar() {...}
51    public void imprimir(){...}
52 }
```

Segunda Solução: O padrão de projeto *factory method*

O padrão de projeto *factory method* lida com o problema da instanciação de objetos de maneira controlada. Utilizando atributos e métodos estáticos, o padrão define uma operação única para instanciação de onjetos, que não é o construtor da classe. Dessa forma, o padrão reduz inclusive o acoplamento entre as classes do sistema, uma vez que quem requisita a instanciação do objeto não conhece qual a classe exata da hierarquia que foi instanciada; facilitando a evolução do sistema.

A Figura 4.56 apresenta o padrão *factory method* aplicado para instanciar publicações da biblioteca. Perceba que as classes Publicacao, Livro e Periodico não são instanciáveis diretamente por classes externas ao pacote `publicacao`. Enquanto a classe Publicacao é uma classe abstrata, os construtores padrões das classes Livro e Periodico possuem visibilidade de pacote. Sendo assim, o padrão *factory method* define um único ponto de acesso para se instanciar objetos de Livro e Periodico, que são as operações `createLivro()` e `createPeriodico()`, respectivamente.

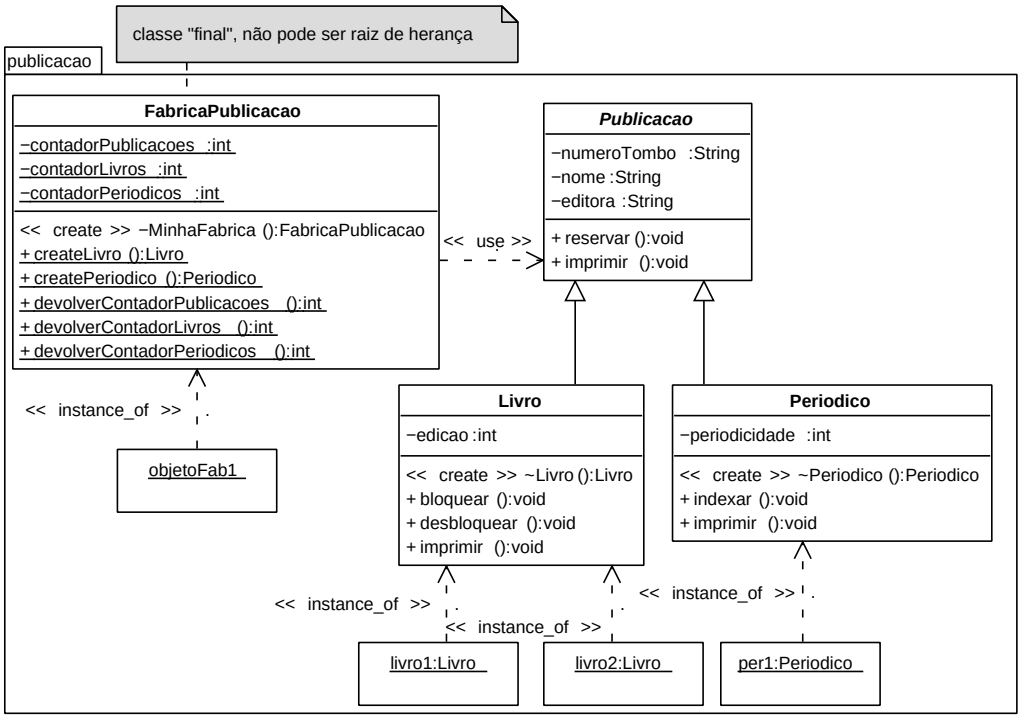


Figura 4.56: O Padrão de Projeto *Factory Method*

4.11 Resumo

Este capítulo apresentou alguns conceitos adicionais sobre o paradigma de objetos. Vimos que um objeto pode ser alocado em memória de duas formas diferentes: (i) alocação estática, realizada no momento em que o programa ou módulo é carregado na memória; e (ii) alocação dinâmica, realizada em tempo de execução. Cada uma dessas estratégias de alocação possui vantagens e desvantagens. Por ser feita no momento em que o software é carregado, os objetos alocados estaticamente são acessados de forma mais eficiente, uma vez que ficam armazenados na pilha. Porém, a alocação estática aumenta o acoplamento entre tipo e implementação, o que restringe fortemente a utilização de técnicas avançadas de programação OO, tais como o polimorfismo de inclusão, que será visto na Seção 4.5.5. Além disso, a alocação estática só pode ser utilizada quando o programador já sabe de antemão as classes e a quantidade exata dos objetos do sistema.

Enquanto isso, a alocação dinâmica prioriza a flexibilidade e a expressividade da programação, perdendo as vantagens de desempenho alcançadas com a alocação estática. Os objetos alocados dinamicamente são armazenados no *heap*, como conseqüência, o processo de alocação e desalocação de objetos se torna mais caro. Além disso, o programador deve se preocupar explicitamente com o tempo de vida do objeto, isto é, enquanto na alocação estática o objeto é desalocado automaticamente no final da execução do bloco de instruções, na alocação dinâmica o programador deve indicar explicitamente o momento da desalocação. Uma exceção a essa regra são as linguagens que implementam algum mecanismo de coleta automática de lixo (do inglês *garbage collection*).

Também foram apresentadas duas manieras de se implementar o conceito de tipos em uma linguagem de programação: (i) tipagem estática, definida explicitamente pelo programador, durante a codificação do sistema; e (ii) tipagem dinâmica, definida dinamicamente, podendo ser alterado no decorrer da execução. Dependendo do mecanismo de tipagem utilizado pela linguagem de programação, deve ser utilizado um processo diferente para verificar a consistência entre os tipos das variáveis declaradas.

Nas linguagens que implementam a tipagem estática, como por exemplo C, C++, Java e C#, o tipo de cada objeto é definido no momento da declaração da variável que o representa. Essa definição antecipada de tipos possibilita uma melhor verificação de compatibilidade em tempo de compilação, uma vez que os tipos não são alterados implicitamente, de acordo com o conteúdo das variáveis. Nas linguagens que implementam a tipagem dinâmica, como por exemplo Smalltalk-80, Objective-C, Perl e Python, as variáveis declaradas não estão associadas a um tipo particular. Nessas linguagens, o tipo das variáveis é associado no momento da atribuição de valores. Dessa forma, o tipo pode ser alterado implicitamente no decorrer da execução e a verificação da compatibilidade de tipo só pode ser feita em tempo de execução.

Em relação ao relacionamento de generalização/especialização, foram apresentados dois tipos de herança, que são implementados por linguagens de programação populares: (i) herança de comportamento, que apresenta a semântica padrão de generalização/especialização, inclusive com a relação de equivalência de tipos entre as superclasses e subclasses; e (ii) herança de implementação, um artifício para reutilizar código da superclasse, quando a subclasse não "é um tipo da" superclasse. Na herança de implementação, os atributos e operações públicos da superclasse são herdados pela subclasse com visibilidade privada. Apesar de não recomendado, algumas linguagens de programação possibilitam a sua utilização. Uma alternativa para esse tipo de reutilização de código é através da utilização de agregação/decomposição.

Além dos tipos de herança, este capítulo discutiu em detalhes alguns problemas relacionados à heraça múltipla, assim como as principais abordagens de resolução, que são implementadas pelas linguagens de programação. Também foi discutida uma alternativa à herança múltipla de classes. Utilizando o relacionamento de agregação/decomposição, é possível usufruir das vantagens de reutilização de código alcançada com a herança múltipla, sem comprometer a legibilidade do modelo e do código.

Finalmente, foi discutido um pouco mais sobre a implementação de pacotes UML, apresentando alguns exemplos em Java.

4.12 Exercícios

1. Um editor de desenhos geométricos possui uma tela para desenhar. Essa tela pode ser constituída de muitas figuras. Figuras podem ser círculos, linhas, polígonos e grupos. Um grupo consiste de muitas figuras e um polígono é composto por um conjunto de linhas. Quando um cliente pede para que uma tela se desenhe, a tela, por sua vez, pede para cada uma das figuras associadas que se desenhe. Da mesma forma, um grupo pede que todos os seus com-

ponentes se desenhem. Crie uma hierarquia de generalização/especialização que classifique essas diferentes figuras geométricas e identifique o comportamento de cada tipo abstrato de dados que você criar, bem como os seus respectivos atributos.

2. Defina uma classe Robô com as seguintes propriedades. A localização do robô é dada por 2 coordenadas inteiras: `x` para sua posição lest-oeste e `y` para a sua posição norte-sul (`x` aumenta quando o robô move para o leste e `y` aumenta quando o robô move para o norte). O robô pode se virar para qualquer posição definida pelo tipo enumerado: Direção = {norte, sul, leste, oeste}. A interface pública do robô deve conter uma operação para a sua movimentação, e para o ajuste da sua orientação (dobrar à esquerda ou à direita num ângulo de 90 graus).

3. Classifique e crie uma hierarquia de generalização/especialização para os diferentes tipos de itens encontrados na biblioteca de uma universidade. Sugestão: livros, periódicos, teses, etc. Defina atributos (o máximo que você souber) para cada uma das classes identificadas.

4. Explique a diferença entre o acesso público, privado, protegido e a visibilidade de pacote de atributos e métodos de uma classe.

5. Uma abstração pode ser vista através de múltiplas perspectivas. Por exemplo, uma pessoa pode ser classificada de acordo com o seu sexo como masculino ou feminino e pode também ser classificada de acordo com a sua idade como criança, jovem e adulto. Baseado na Figura 4.57

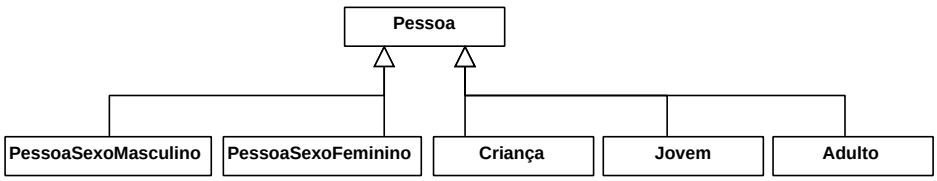


Figura 4.57: Especialização da Classe Pessoa em Duas Dimensões

   (a) Apresente uma solução de modelagem que combine essas duas perspectivas usando herança múltipla. Note que um objeto pode ser instanciado a partir de apenas uma única classe (um objeto não pode ser instância de 2 classes). Por exemplo, o objeto `José` é do sexo masculino e adulto, mas ele não pode ser instanciado a partir de ambas as classes `SexoMasculino` e `Adulto` ao mesmo tempo.

   (b) Discuta duas desvantagens da sua solução proposta no ítem anterior.

   (c) Proponha uma solução alternativa mais flexível que supere as desvantagens identificadas no ítem anterior. (Dica: pense na hierarquia de agregação/decomposição.)

6. Faça uma comparação entre tipos abstratos de dados, classes e interfaces de Java.

7. C++ e Java usam mecanismos diferentes para desalocar objetos que não são mais necessários. C++ usa desalocação automática de objetos declarados estaticamente, mas os objetos criados dinamicamente devem ser desalocados explicitamente pelo programador. Java usa coleta de lixo automática. Pesquise o que isso significa e compare as duas abordagens.

8. Java, Eiffel e Smalltalk-80 têm diferentes regras associadas ao acesso de atributos internos de uma classe por clientes fora da classe. Em Smalltalk-80, os atributos são invisíveis fora da classe e, portanto, não podem ser acessados fora dela. Em Eiffel, um atributo pode tornar-se visível fora da sua classe, mas ele pode ser somente lido e nunca modificado. Finalmente, Java e C++ permitem que atributos sejam lidos e escritos desde que eles sejam declarados como `public`. Compare essas três abordagens. Qual delas na sua opinião é a mais razoável?

9. Discuta herança de comportamento versus herança de implementação (como apresentado na Seção 4.4), dando exemplos de cada uma delas. Em sua opinião, qual das duas abordagens deveria ser incentivada? Justifique sua resposta.

10. Considerando o trecho de código em C++ apresentado a seguir, o que aconteceria se você tentasse compilar esse programa? Haveria erro de compilação? Caso exista, quais seriam os erros? E se o atributo `codigo` de `Publicacao` fosse privado?

```
1  class Publicacao {
2     protected:
3        int codigo;
4     private:
5        String nome;
6        int status;
7     public:
8        void reservar(){...}
9        int obterNome(){...}
10 }
11
12 class Livro : public Publicacao {
13    private:
14       String isbn;
15       Publicacao *pub = new Publicacao;
16    public:
17       void operacao1() {
18          if (codigo == 0) //AQUI (i)
19             cout << isbn; // AQUI (ii)
20          if ((pub.codigo) == 0) // AQUI (iii)
21             cout << (pub.nome); // AQUI (iv)
22             cout << (pub.obterNome()); // AQUI (v)
23       }
24 }
```

11. Dadas as seguintes declarações de classes em Java:

```
1 public class T {
2    public void op(){//implementação1
3    }
4 }
5
6 public class S extends T {
7    public void op(){//implementação2
8    }
9 }
10
11 public class ProgramaPrincipal{
12    public static void main(String[] args){
13       T x = new T();
14       S y = new S();
15       T p = new T();
16    }
17 }
```

Considere as invocações de operações apresentadas a seguir. Para cada uma das invocações indicadas de (i) a (v), determine qual implementação (1 ou 2) será executada. Justifique as suas respostas.

```
1 p.op();  // (i)
2 p = y;   // o apontador ''p'' recebe o endereço da variável ''y''
3 p.op();  // (ii)
4 x.op();  // (iii)
5 y.op();  // (iv)
6 x = y;   // o apontador ''x'' recebe o endereço da variável ''y''
7 x.op();  // (v)
```

12. Em Ada, pode-se escrever uma função genérica que aceita parâmetros de diversos tipos. Compare esta forma de polimorfismo com o polimorfismo de inclusão encontrado em linguagens orientadas a objetos.

13. Modele uma fila polimórfica de veículos (ônibus, caminhões, carros e motos). Essa fila pode ser usada, por exemplo, em um sistema de pedágio. Utilize os conceitos de hierarquia de generalização/especialização, agregação/decomposição, classe abstrata e classe concreta.

14. Em C++, um programador pode definir funções paramétricas que aceitam parâmetros de tipos diferentes. Compare essa abordagem com a redefinição de operações no polimorfismo de inclusão encontrada em todas as linguagens orientadas a objetos.

15. O diagrama de classes apresentado na Figura 4.58 não representa fielmente um *buffer* que passa por estados sucessivos de transformação. O mesmo *buffer* é inicialmente vazio, depois pode ficar parcialmente cheio e possivelmente, pode ficar cheio. Modifique o diagrama de tal forma que ele reflita melhor os diferentes estados de transformação pelos quais o objeto passa.

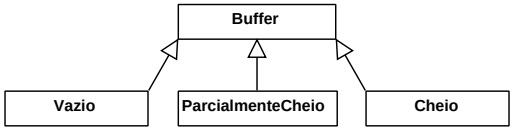


Figura 4.58: Hierarquia de Classes de um Buffer

16. Dada a Figura 4.59, que apresenta o diagrama de classes de um sistema de controle de consultas médicas:

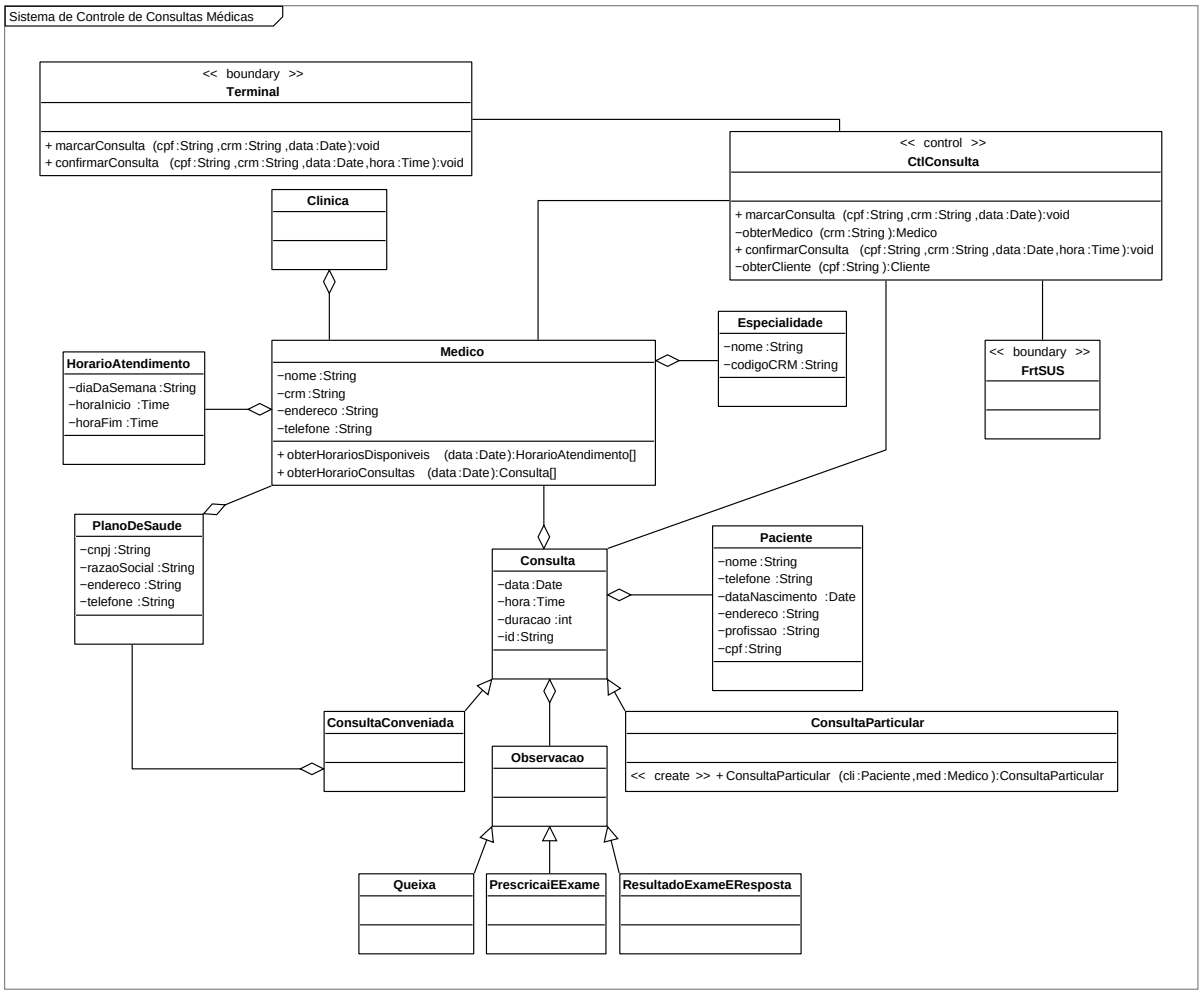


Figura 4.59: Sistema de Controle de Consultas

(a) Como esse diagrama de classes poderia ser evoluído usando o conceito de interface UML? Suponha o caso onde o sistema deve ser integrado a vários planos de saúde diferentes. Justifique sua resposta.

(b) Defina o conceito de classe abstrata e mostre como diagrama de classes do sistema de controle de consultas pode explorar esse conceito.

17. Refine a hierarquia de cômodos apresentada na Figura 4.60 usando o conceito de metaclasses visto na Seção 4.10, onde os atributos e métodos de classe são separados dos atributos e métodos de objetos.

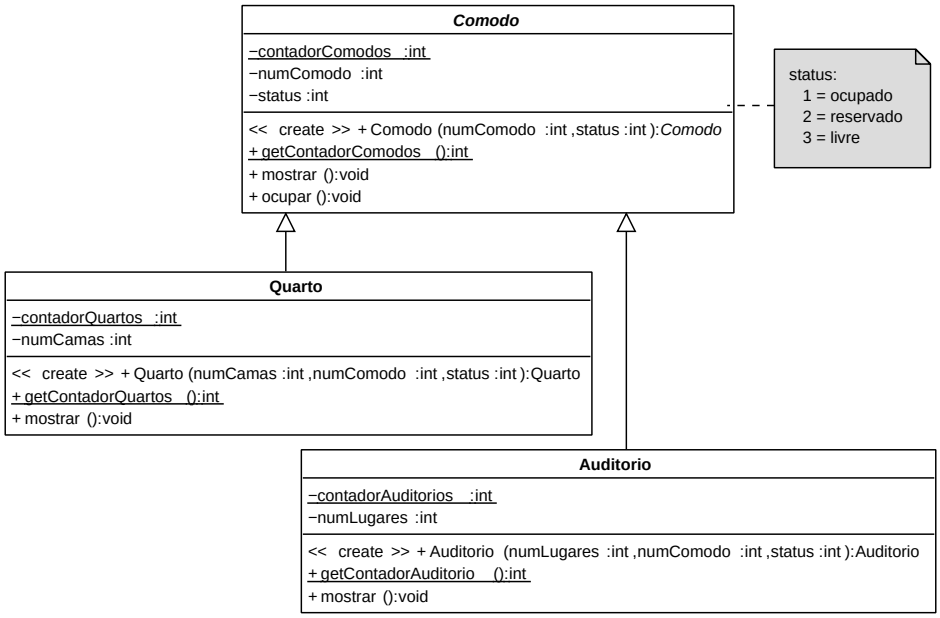


Figura 4.60: Hierarquia de Cômodos de um Hotel

Capítulo 5

Análise Orientada a objetos: Modelagem Dinâmica

No Capítulo 3 foi apresentada a modelagem estática, na qual as especificações dos casos de uso do sistema são analisadas e as informações extraídas são representadas através de um diagrama de classes de análise. Essa primeira versão do diagrama de classes inclui elementos estáticos do domínio do problema, isto é, conceitos relevantes (classes), informações associadas a esses conceitos (atributos) e as ligações entre esses conceitos (associações, agregações e relacionamentos de herança).

Neste capítulo, apresentamos a modelagem dinâmica, que identifica e modela os aspectos do sistema de software que podem mudar durante a sua execução, devido à ocorrência de eventos. Na modelagem dinâmica, o comportamento esperado do sistema é representado através de diagramas dinâmicos da UML, mais especificamente, os diagramas de atividades, de seqüência, e de colaboração, introduzidos no Capítulo 2, além dos diagramas de estados, que serão vistos na Seção 5.6. Os diagramas de atividades representam a lógica do negócio a partir das suas principais atividades e as respectivas condições de ativação. Por esse motivo, como visto na Seção 2.9.10, esses diagramas são utilizados normamente para representar os fluxos de eventos dos casos de uso. Já os diagramas de seqüência e colaboração representam as interações entre objetos dentro do sistema. Um diagrama de estados, por sua vez, fornece uma visão complementar aos diagramas dinâmicos anteriores, através de uma representação do estado do sistema ou de suas partes. Isso é feito através de um conjunto de estados previstos e das respectivas transições entre esses estados. As transições são disparadas pela ocorrência de eventos, que tanto podem ser estímulos externos fornecidos pelos atores do sistema, quanto condições internas que devem ser detectadas antes das ações serem executadas.

Além de produzir um conjunto de diagramas que representam o comportamento e o estado esperados do sistema de software, a modelagem dinâmica também acrescenta novas informações

ao diagrama de classes de análise, que continua sendo o principal artefato produzido pela análise orientada a objetos. Através do estudo do modelo de casos de uso do sistema e das mensagens trocadas entre objetos nos diagramas de seqüência e colaboração, as operações públicas das classes de análise são identificadas e conseqüentemente, as interfaces dessas classes são atualizadas. Apesar de já serem identificadas nesse momento da análise, essas operações serão refinadas, posteriormente, durante a fase de projeto do sistema.

É importante ressaltar que, assim como na modelagem estática, nenhuma solução técnica é identificada ou descrita durante a modelagem dinâmica. Essa última é apenas uma etapa da análise e, conseqüentemente, trata estritamente do domínio do problema.

5.1 Atividades da Modelagem Dinâmica

Durante a modelagem dinâmica, estamos interessados em definir o comportamento do sistema. Esse comportamento normalmente é descrito em termos dos eventos que ocorrem ao longo da execução do sistema. Chamamos de evento qualquer ocorrência que envolva o sistema com algum tipo de troca de informação. é importante ressaltar que um evento não é a informação trocada, mas o fato de alguma informação ter sido trocada. Durante a análise, o tipo de evento mais comum que encontramos é a interação entre um ator e o sistema, embora outros tipos também sejam possíveis.

Neste livro, a modelagem dinâmica consiste das atividades mostradas a seguir, que são baseadas em outras metodologias descritas na literatura[41][29][26]:

Atividade 1 Identificar eventos do sistema.

Atividade 2 Construir diagramas de seqüência para os cenários primários dos casos de uso.

Atividade 3 Construir um único diagrama de colaboração para o sistema, combinando os diagramas de seqüência obtidos na Atividade 2.

Atividade 4 Atualizar interfaces públicas das classes de análise.

Atividade 5 Construir diagramas de estados.

Atividade 6 Iterar e refinar.

Nas seções seguintes, descrevemos cada uma dessas atividades em detalhes. Para tornar a explicação mais ilustrativa, lançamos mão do mesmo estudo de caso que foi apresentado no Capítulo 3, o Sistema de Controle para Bibliotecas.

5.2 Atividade 1: Identificar Eventos do Sistema

Como visto no Capítulo 2, casos de uso representam seqüências de atividades que envolvem atores e suas interações com o sistema. Como dissemos antes, os eventos decorrentes da interação entre esses atores e o sistema são bastante relevantes para a análise OO, lembrando que atores podem ser pessoas, dispositivos ou até mesmo outros sistemas computacionais. Tendo isso em vista, assim como na modelagem estática, o modelo de casos de uso é a principal fonte de informações para a modelagem dinâmica.

Para identificar os eventos relevantes, realizamos uma nova análise textual das especificações dos casos de uso, destacando os pontos nos quais ocorrem trocas de informação. Esses pontos são facilmente identificáveis pela presença de ações, que normalmente são identificadas a partir de verbos ou verbos substantivados (Figura 5.1). A informação que deve ser capturada consiste tanto dos verbos empregados quanto dos contextos nos quais eles aparecem.

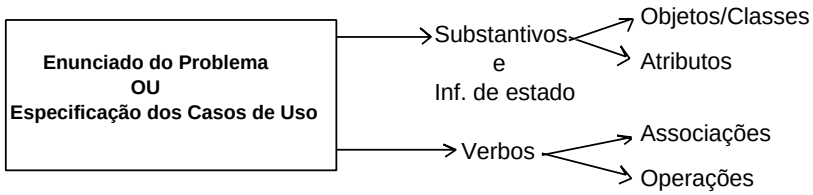


Figura 5.1: Extração de Informações a partir do Enunciado do Problema

5.2.1 Análise Textual do Caso de Uso Emprestar Exemplar

A seguir, apresentamos novamente a especificação do fluxo de eventos do caso de uso Emprestar Exemplar, destacando com sublinhado os eventos encontrados.

Fluxo Básico :

1. O cliente solicita empréstimo de um exemplar de alguma publicação (livro, periódico, tese ou manual), fornecendo o seu número de registro e o número de tombo da publicação desejada.
2. A atendente solicita o empréstimo ao sistema, fornecendo o código do cliente e o tombo da publicação
3. O sistema valida o cliente e verifica o seu status no sistema de cadastro ("Normal" ou "Suspenso") através de seu número de registro. (<< *include* >> Validar Usuário)
4. O sistema verifica se existe algum exemplar disponível da publicação desejada.
5. Se o status do cliente for "Normal" e algum exemplar da publicação estiver disponível
   5.1. O sistema registra um novo empréstimo;

5.2. O sistema verifica o período do empréstimo, que depende do tipo de usuário - 7 dias para alunos e 15 para professores

5.3. O sistema atualiza seu banco de dados com a informação de que o exemplar não irá se encontrar na biblioteca até completar o período.

Fluxo Alternativo 1 :

No passo 5, se o usuário estiver suspenso, o sistema o informa de sua proibição de retirar exemplares e o empréstimo não é realizado.

Fluxo Alternativo 2 :

No passo 5, se todas as cópias da publicação estiverem emprestadas ou reservadas, o sistema informa à atendente que não será possível realizar o empréstimo.

Entre os trechos destacados acima, alguns não dão uma idéia imediata de troca de informação, mas isso é fruto da maneira como o caso de uso foi especificado. Conseqüentemente, é preciso atenção ao realizar a análise textual da especificação à procura de eventos, já que nem sempre eventos são tão fáceis de identificar quanto conceitos. Por exemplo, a frase "O empréstimo é realizado" não parece com um evento, já que não indica a ocorrência de troca de informações entre o sistema e algum ator. Entretanto, uma análise mais cuidadosa mostra que isso não procede, já que o atendente (ou quem quer que seja responsável pela operação do sistema) precisa ser informada sobre o fato de o empréstimo ter sido realizado com sucesso ou não. Conseqüentemente, essa frase precisaria ser reescrita para que passasse a descrever um evento: "O sistema informa ao atendente que o empréstimo foi realizado".

Pelo exemplo acima, é fácil perceber que nem toda ocorrência de verbo corresponde a um evento, embora eventos sempre estejam associados a verbos, ou verbos substantivados. Por exemplo, a frase "A publicação não se encontrará na biblioteca" não pode ser vista como um evento. Ao invés disso, diz respeito à necessidade do sistema manter algum registro da situação de uma publicação emprestada. Por outro lado, dependendo da forma como o caso de uso foi especificado, a expressão "gerenciamento de usuário", por exemplo, apesar de não conter um verbo, pode significar funcionalidades relativas à inclusão, exclusão e manutenção dos dados do usuário. Nesse contexto, a palavra "gerenciamento" desempenha o papel do verbo "gerenciar"e deve ser analisado como tal.

Além disso, é importante entender a relação dos atores com o sistema. Por exemplo, no fluxo de eventos apresentado, clientes nunca interagem diretamente com o sistema, já que quem opera este último é o atendente. Conseqüentemente, eventos não precisam ser especificados com clientes no papel de atores. Deve-se prestar atenção, porém, para diferenciar as situações em que clientes são atores, como na oração: *"O* cliente *solicita empréstimo de publicação"*, daquelas em que o cliente é, na verdade, uma informação que o sistema manipula, como por exemplo, a frase: *"O atendente verifica o status do* cliente*"*. Neste último caso, "cliente" é, na verdade, uma instância da classe Usuário ou alguma de suas subclasses (Capítulo 3) e é responsável por conter os dados do cliente que são relevantes para o sistema.

5.2.2 Eventos Identificados

Selecionando e reescrevendo os trechos destacados no fluxo de eventos e levando em consideração as observações feitas acima, identificamos os seguintes eventos:

1. O cliente solicita empréstimo de um exemplar de alguma publicação;
2. O cliente fornece o seu número de registro e o número de tombo da publicação;
3. A atendente solicita o empréstimo ao sistema;
4. A atendente fornece o número de registro do cliente e o número de tombo da publicação;
5. O sistema valida o cliente e verifica o seu status;
6. O sistema verifica se existe algum exemplar disponível da publicação desejada;
7. O sistema registra um novo empréstimo;
8. O sistema verifica o período do empréstimo;
9. O sistema atualiza seu banco de dados;
10. O sistema o informa de sua proibição de retirar exemplares;
11. O sistema informa à atendente que não será possível realizar o empréstimo.

Note que cada um dos eventos acima indica explicitamente se algum ator ou o próprio sistema é o responsável por disparar o evento. Por exemplo, no evento 3: *"A atendente solicita o empréstimo ao sistema"*, fica claro que o atendente inicia a interação e, conseqüentemente, "dispara" o evento. Essa informação torna mais fácil determinar, em etapas seguintes da análise, quais eventos dão origem a operações e quais são simplesmente respostas produzidas pelas chamadas dessas operações. Além disso, cada evento também explicita o tipo de informação que é trocada quando ele ocorre, embora sem entrar nos detalhes de sua representação. Por exemplo, no evento 6: *"O sistema verifica se existe algum exemplar disponível da publicação desejada"*, a informação que o sistema obtém tem que ser suficiente para que ela seja capaz de conhecer se há ou não exemplares disponíveis. Essa informação pode ser um valor booleano ou a quantidade de exemplares disponíveis. A decisão final quanto a isso só será tomada na fase de projeto.

Na lista de eventos apresentada acima, nem todos os eventos indicam interações diretas entre o atendente e o sistema, embora todos sejam conseqüência dessas interações. Por exemplo, o evento 7: *"O sistema registra um novo empréstimo"* não envolve qualquer ator. Além disso, é fácil perceber que há uma certa redundância de informação. Por exemplo, quando o atendente solicita o empréstimo de uma publicação para um usuário cadastrado, pressupõe-se que informações relativas ao usuário e à publicação serão fornecidas. Apesar disso, foi usado um evento diferente para fornecer esses dados. Isso é de se esperar da especificação de um sistema real e não caracteriza um erro.

Nas atividades seguintes da modelagem dinâmica essa informação será organizada de modo que as redundâncias sejam eliminadas.

Uma vez que todos os eventos tenham sido identificados, eles são associados aos objetos envolvidos. Um objeto pode ser responsável por gerar eventos ou reconhecer eventos gerados por outros. É fácil perceber que, no caso de uso Emprestar Publicação, há apenas dois objetos interagindo: o atendente e o sistema. Podemos usar, porém, as diretrizes descritas na Seção 3.10.1, e dividir o sistema de acordo com os três tipos de classes de análise apresentados: fronteira, controle e entidade. Deste modo, podemos descobrir mais informações sobre o funcionamento esperado do sistema. A seção seguinte descreve como isso pode ser feito.

5.3 Atividade 2: Construir Diagrama de seqüência para o Caso de Uso

No Capítulo 2, Seção 2.9.11, introduzimos o conceito de diagrama de seqüência de sistema. Um diagrama de seqüência de sistema representa graficamente as interações entre o sistema e os atores relacionados com a realização de um certo caso de uso. Interações são explicitamente representadas através de mensagens entre atores distintos e entre atores e o sistema. A ordenação temporal das mensagens é representada pela posição em que elas aparecem no diagrama.

5.3.1 Diagrama de Seqüência de Sistema

Para construir um diagrama de seqüência de sistema, é necessário identificar, a partir da descrição de um caso de uso, todas as interações relevantes para a sua realização. É possível perceber, a partir desse fato, que existe uma relação direta entre diagramas de seqüência de sistema e os eventos identificados na Seção 5.2.2. Por exemplo, a Figura 5.2 mostra o diagrama de seqüência de sistema para o caso de uso Emprestar Publicação, onde as mensagens que fluem entre o ator Atendente e o sistema são os eventos identificados naquela seção.

No exemplo acima, o atendente só confirma a realização do empréstimo se o usuário estiver em dia com a biblioteca ([status = “Normal”]) ou ([status $\neq$ “Suspenso”]) e se houver algum exemplar da publicação solicitada disponível para empréstimo (# exemplaresDisponiveis > 0). Normalmente, recomenda-se que condições em diagramas de seqüência sejam usadas raramente, já que podem introduzir uma complexidade desnecessária e dificultar o entendimento desses diagramas.

O diagrama de seqüência de sistema é um artefato útil para a análise. A partir dele, percebe-se claramente quais eventos correspondem a serviços que o sistema deve oferecer e quais se referem a respostas que ele deve fornecer, mediante requisições disparadas por atores. Além disso, a ordenação dos eventos é explicitada e tanto mensagens redundantes quanto novos eventos tornam-se imediatamente identificáveis. No diagrama da Figura 5.2, assumimos que, uma vez que o registro do usuário e o número de tombo da publicação desejada sejam fornecidos, o sistema já informa o

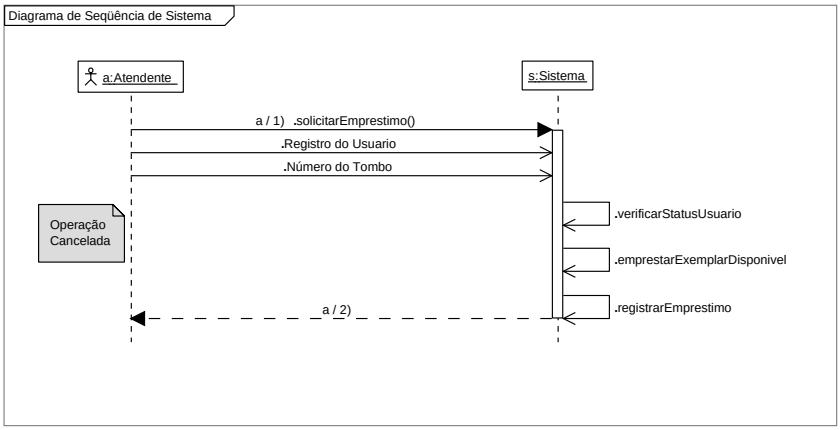


Figura 5.2: Diagrama de seqüência de Sistema

status do usuário e o número de exemplares disponíveis da publicação, ao invés de esperar que essas informações sejam pedidas explicitamente. Além disso, um novo evento: *"Confirma empréstimo"*, foi incluído no diagrama, correspondente ao momento em que a bibliotecária informa ao sistema que o empréstimo deve ser registrado.

Notação UML para Condições e Iterações em Diagramas de Seqüência

Um diagrama de seqüência pode incluir condições a fim de modelar interações mais complexas. Nesse tipo de diagrama, uma condição indica quando uma mensagem é ou não enviada. Condições são indicadas entre colchetes, junto com o rótulo de uma mensagem. Na Figura 5.2, duas condições são utilizadas [status = "Suspenso" ou # exemplaresDisponiveis = 0] e [status = "Normal" e # exemplaresDisponiveis > 0]. Estas estão associadas aos possíveis retornos da mensagem 3: *"fornece número de tombo da publicação"*, enviada pelo atendente.

Também é possível representar iterações em diagramas de seqüência. A notação é a mesma que é empregada para condições, exceto pelo fato de iterações serem precedidas pelo símbolo "*". Uma iteração em um diagrama de seqüência indica que uma mensagem é enviada várias vezes para múltiplos receptores. Normalmente, esses receptores correspondem aos elementos de uma coleção, por exemplo: (i) todas as publicações de um empréstimo; ou (ii) todos os usuários cadastrados.

5.3.2 Diagrama de Seqüência Refinado

Apesar dos benefícios trazidos pelo emprego de diagramas de seqüência de sistema, sua aplicabilidade é limitada, já que o sistema é tratado como uma "caixa preta". Conseqüentemente, não é feita relação entre os eventos representados e as classes de análise descobertas na modelagem estática.

No Capítulo 3, mais precisamente na Seção 3.10.1, foi apresentada uma categorização para

classes de análise sugerida pelo processo RUP. Essa categorização visa, principalmente, aumentar o entendimento do analista sobre o sistema que está sendo modelado. De acordo com ela, as classes de análise do sistema são divididas entre três categorias: fronteira, controle e entidade. Essas categorias definem uma estruturação em camadas para o sistema, o que limita a maneira como instâncias das classes de análise interagem. Classes de fronteira interagem apenas com atores e classes de controle. Já as classes de entidade interagem com outras classes de entidade agregadas a ela e também com as classes de controle, que implementam a lógica do negócio. Essas classes de controle podem interagir com qualquer classe do sistema, seja ela de fronteira, de entidade, ou até mesmo outra classe de controle.

Utilizando essa distinção e obedecendo as restrições de comunicação entre as categorias das classes de análise, pode-se enriquecer o diagrama de seqüência da Figura 5.2 com informações adicionais. Para tanto, devemos substituir o objeto do tipo Sistema por instâncias das classes de análise que o compõem. Revisando a Figura 3.9 do Capítulo 3, percebemos que o sistema tem apenas uma classe de fronteira e apenas uma classe de controle. Conseqüentemente, as duas classes foram selecionadas para serem incluídas no diagrama de seqüência refinado. Além disso, é necessário averiguar quais classes de entidade serão usadas no diagrama. Essa atividade pode ser executada a partir da análise da lista de eventos identificada na atividade anterior. Revendo a lista de eventos identificados para o caso de uso Emprestar Publicações, percebe-se que, inicialmente, apenas as classes Publicação, Usuário e Empréstimo são relevantes. O diagrama de seqüência obtido após realizar as alterações descritas é apresentado na Figura 5.3.

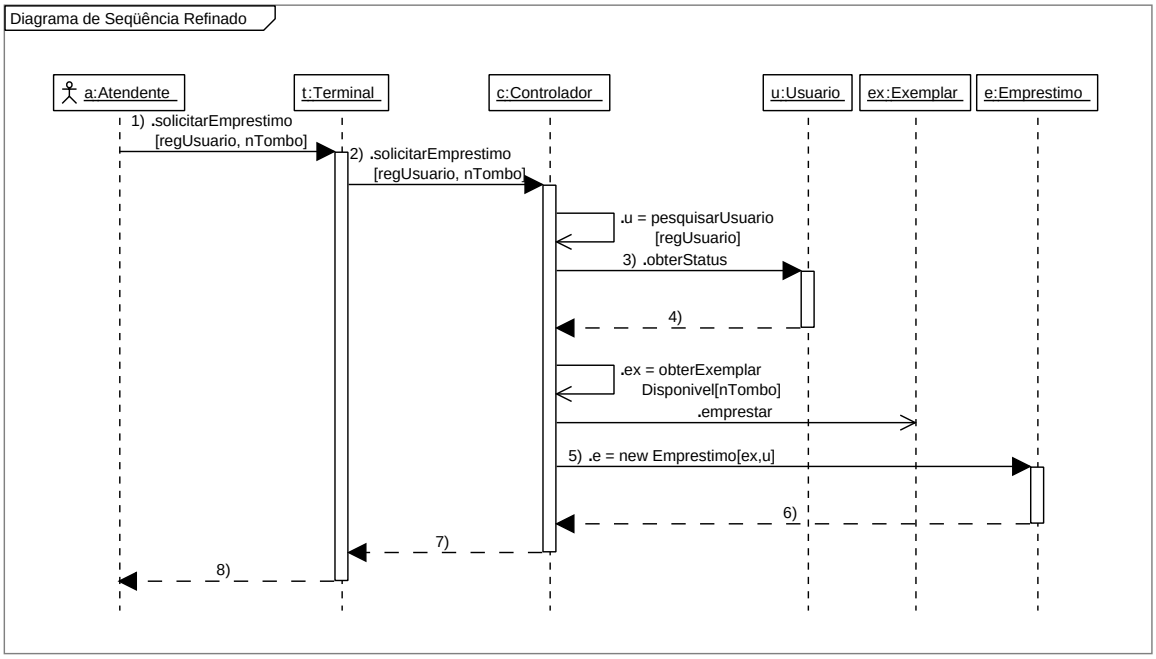


Figura 5.3: Diagrama de seqüência Refinado

Fica claro no diagrama de seqüência da Figura 5.3 que o número de eventos aumentou consideravelmente, em relação ao diagrama da Figura 5.2. Além de todos os eventos presentes na versão

simplificada, o novo diagrama adicionou novos eventos internos ao sistema. Esses eventos surgem naturalmente, à medida que o diagrama é construído. Por exemplo, para poder informar o status do usuário, o sistema precisa primeiro procurar pelo objeto do tipo Usuário correspondente para, em seguida, obter o status desse usuário e apresentá-lo na tela do terminal. Similarmente, para registrar um novo empréstimo, um objeto do tipo Empréstimo precisa ser criado e armazenado. Os novos eventos identificados nesta atividade serão usadas na atividade 4, quando são adicionadas operações às interfaces públicas das classes de análise. Vale ressaltar, porém, que o analista não deve perder muito tempo tentando identificar todos os eventos possíveis, já que estes poderão ser identificados em iterações subseqüentes, à medida que seu entendimento sobre o sistema crescer.

O diagrama de seqüência da Figura 5.3 apresenta elementos que, até então, não haviam aparecido. Por exemplo, quando o terminal envia para o controlador a mensagem 4: *"solicitaEmprestimo"*, ele também passa dois parâmetros ("regUsuario e nTombo") correspondentes ao número de registro do usuário e ao número de tombo, respectivamente. Dessa forma, o diagrama torna explícito o tipo de informação transferida em decorrência desse evento. Além disso, quando o controlador envia para si mesmo a mensagem 5: *"pesquisaUsuário"*, além de passar como parâmetro o número de registro "regUsuario", também atribui o retorno a uma variável "u" do tipo Usuário, correspondente ao usuário encontrado. A nomeação dos objetos é um artifício para tornar explícitas as relações entre os objetos no diagrama. Note que o mesmo objeto do tipo Usuário que é referenciado por "u" (u:Usuário) é o que recebe a mensagem 6: *"obtemStatus"*, enviada do controlador. Dessa forma, fica explícito quando o destino das mensagens não é qualquer objeto, mas aquele que foi obtido como resultado da pesquisa.

5.4 Atividade 3: Construção de um Diagrama de Colaboração para o Sistema

Uma vez que os eventos tenham sido identificados para todos os casos de uso de um sistema, cria-se uma representação de como esses eventos fluem de um objeto para outro. Para tanto, utiliza-se um diagrama de colaboração em que os eventos não são numerados. Esses diagramas também são conhecidos como diagramas de fluxo de eventos. A Figura 5.4 mostra o diagrama de fluxo de eventos relativo ao caso de uso Emprestar Publicação.

Na Figura 5.4 a ordem em que os eventos são trocados pelos objetos não é explicitada. Conseqüentemente, em um diagrama de fluxo de eventos, podem ser incluídos os eventos identificados para todos os casos de uso do sistema, o que não seria possível, por exemplo, em um diagrama de seqüência. Desse modo, consegue-se desfrutar de uma visão geral das trocas de informações entre objetos no sistema. É importante ressaltar, porém, que esses diagramas tendem a tornar-se muito complexos, à medida que o número de casos de uso analisados aumenta.

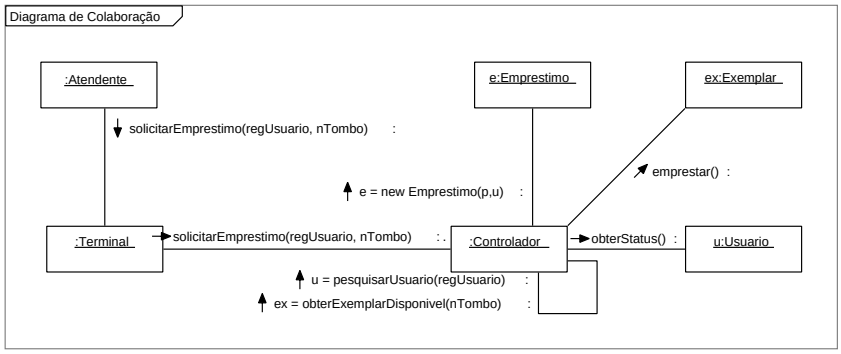


Figura 5.4: Diagrama de Colaboração

5.5 Atividade 4: Atualizar interfaces públicas das classes de análise.

Nas atividades 1 e 2 da modelagem dinâmica, foram identificados diversos eventos relacionados à execução do caso de uso Emprestar Publicação. Esses eventos representam tanto o comportamento externamente observável do sistema, isto é, aquele que é conseqüência de interações entre atores e o sistema; quanto o comportamento decorrente das interações entre seus elementos internos.

Os diagramas de seqüência apresentados na Seção 5.3 determinam, ainda que implicitamente, a maneira como os eventos identificados anteriormente se relacionam com as classes de análise encontradas durante a modelagem estática. Na atividade atual, deve-se estudar esses diagramas de seqüência, em especial o diagrama refinado, mostrado na Figura 5.3, a fim de atualizar as interfaces públicas das classes de análise com as **operações**, que ofereçam os serviços que essas classes devem oferecer e que, durante o projeto do sistema, serão materializados em métodos.

Pode-se começar separando os eventos identificados até o momento, associando-os às classes dos objetos presentes no diagrama de seqüência da Figura 5.3 e, para cada classe, indicando se o objeto foi responsável pela ocorrência do evento ou se apenas o recebeu. A Tabela 5.1 apresenta essa informação.

Uma vez que os eventos produzidos e recebidos pelas classes de análise tenham sido identificados, pode-se precisar atualizar o diagrama de classes de análise com essa informação. Os eventos recebidos por uma classe se transformam em operações de sua interface pública. Já os eventos produzidos por um objeto podem indicar duas coisas: (i) uma resposta a um evento recebido, como por exemplo, o evento 6(b): *"devolve status"* da classe Usuário; ou (ii) o fato de que uma classe usa algum serviço oferecido por outra, como por exemplo, o evento 6: *"obtem status"*, que é produzido pela classe Controlador). No primeiro caso, os eventos servem para indicar o tipo de informação que é devolvida por uma operação. No segundo, eles oferecem informações úteis para a implementação dos métodos, que acontecerá na fase de projeto.

Operações são especificadas como métodos das classes de análise. O tipo de cada parâmetro

Tabela 5.1: Eventos relevantes para as Classes identificadas.

Classe	Eventos Produzidos	Eventos Recebidos
Usuário	"devolve status"	"obterStatus()"
Emprestimo		"new Emprestimo(ex, u)"
Controlador	"pesquisarUsuario(regUsuario)" "obterStatus()" "obterExemplarDisponivel (nTombo)" e="new Empréstimo(p,u)" "registrarEmprestimo(e) "Não é possível realizar o empréstimo" "Informa status do usuário e # de exemplares disponíveis"	"solicitarEmprestimo (regUsuario, nTombo)" u="pesquisarUsuario(regUsuario)" "devolve status" "obterExemplarDisponivel (nTombo)" "registrarEmprestimo(e)"
Terminal	 "solicitarEmprestimo (regUsuario, nTombo)"	"solicitaEmprestimo()" "Registro do Usuário" "Número do Tombo" "Não é possível realizar o empréstimo" "Informa status do usuário e # de exemplares disponíveis"

corresponde ao tipo de informação que é transferida quando o evento ocorre, como por exemplo, o parâmetro "regUsuario", passado no evento 4 da Figura 5.3. Já o tipo de retorno de cada operação depende do evento que é produzido como resposta para a mensagem que inicia a sua execução. A Figura 5.5 apresenta uma versão modificada do diagrama de classes de análise do sistema de controle para bibliotecas, no qual operações foram adicionadas às classes. Na figura, o evento "new Emprestimo(p, u)", que representa uma chamada ao construtor da classe Emprestimo, não foi incluído. Esse tipo de evento normalmente não precisa ser documentado durante a análise.

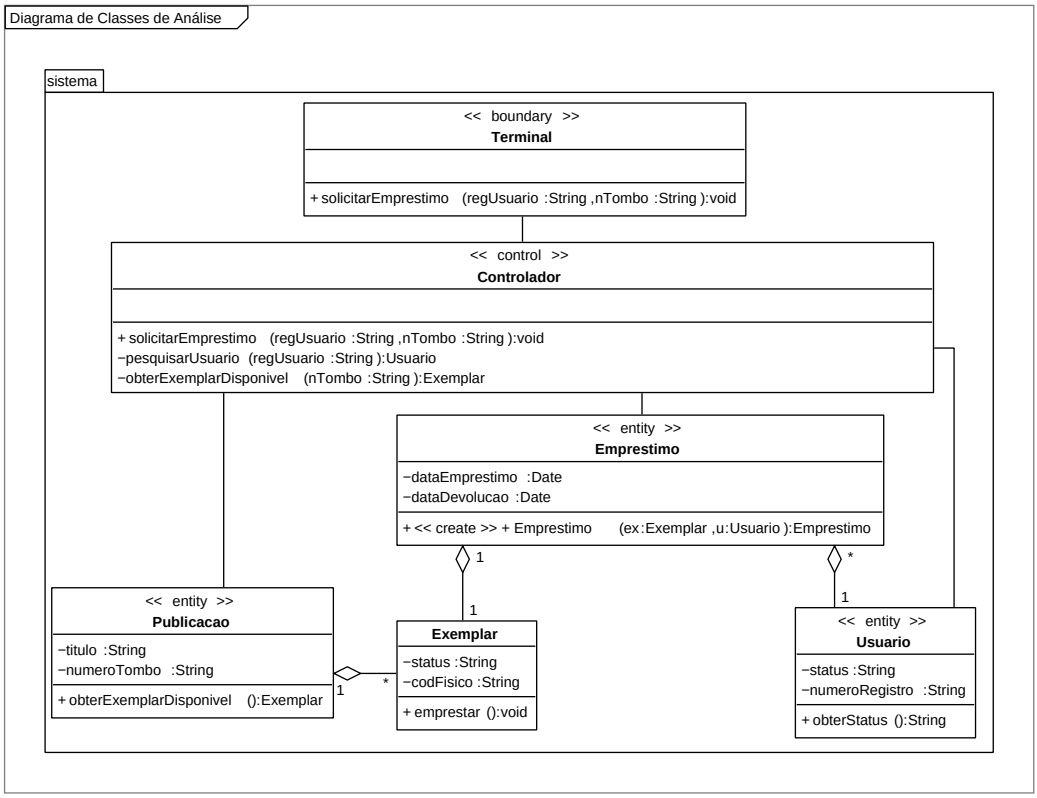


Figura 5.5: Diagrama de Classes de Análise com Operações

Note que, na Figura 5.5, alguns tipos de parâmetros e tipos de retorno não correspondem a tipos normalmente encontrados em linguagens de programação. Por exemplo, o tipo de retorno da operação obterNumeroExemplaresDisponiveis() é quantidade. Não usamos tipos existentes em linguagens de programação, como inteiro ou real, porque a escolha dos tipos de dados que serão usados para modelar os conceitos do domínio do problema é feita apenas durante o estágio de projeto.

A classe de análise Terminal corresponde à interface entre a bibliotecária e o sistema. Embora a tenhamos representado de forma análoga às outras classes de análise, durante o projeto essa classe sofrerá um tratamento diferente. Suas operações provavelmente não serão mapeadas para métodos em classes de projeto. Ao invés disso, serão transformadas em comandos e informações exibidas através de uma interface gráfica. Durante a análise, porém, não há mal nenhum em representar a classe Terminal da mesma forma que as outras classes.

Vale ressaltar que é possível complementar o diagrama de classes de análise especificando as semânticas das operações. Podemos utilizar as técnicas descritas no Capítulo 4, como (i) pré- e pós-condições e (ii) especificações algébricas. A última atividade da modelagem dinâmica, que é mostrada na seção seguinte, abordará a criação de diagramas de estados de UML. Esses diagramas podem ser utilizados como forma complementar às técnicas de especificação semântica de métodos citadas anteriormente.

5.6 Atividade 5: Construir Diagramas de Estados

Esta seção, mostra como máquinas de estados podem ser usadas para especificar o comportamento de sistemas e classes. Utilizamos a abordagem de UML, que é baseada no conceito de *statecharts*, criado por David Harel [23]. De um modo geral, diagramas de estados ilustram os possíveis estados que um objeto pode assumir e as respectivas condições de alternância entre esses estados. Além disso, esses diagramas mostram a maneira como o objeto reage aos eventos, sejam eles externos ou internos. A Figura 5.6 apresenta um diagrama de estados muito simples.

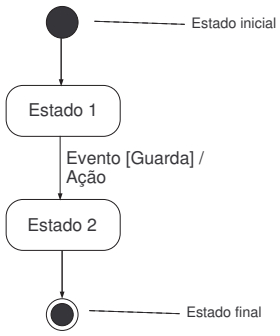


Figura 5.6: Um Diagrama de Estados UML

Os elementos básicos de um diagrama de estados são estados e transições. Em UML, estados são representados por retângulos com os cantos arredondados, enquanto transições são as setas que ligam os estados. Os estados inicial e final são representados através de símbolos especiais (Figura 5.6). Transições têm rótulos que consistem de três partes, todas opcionais: *Evento [Condição] / Ação*. Uma transição exige que um certo evento ocorra e que a sua condição seja verdadeira. A ausência de condições é equivalente a uma condição que é sempre verdadeira. Quando uma transição ocorre, uma ação pode ser executada, por exemplo, a invocação de um método.

Diagramas de estados normalmente são usados para modelar: (i) a seqüência legal de eventos externos que são reconhecidos e tratados por um sistema, no contexto da execução de um caso de uso, (ii) o conjunto de todos os estados e transições do sistema, ao longo da execução de todos os casos de uso que o especificam, e (iii) o comportamento de classes complexas que dependem de algum tipo de estado. Em todos os casos, é recomendável que a modelagem tenha foco nos eventos externamente observáveis produzidos pela entidade especificada. No passo 5 da modelagem

dinâmica, damos ênfase ao terceiro uso de diagramas estados, ou seja, estamos interessados em modelar o comportamento das instâncias das classes de análise.

O estado de um objeto tem tanto características ativas quanto passivas. O estado passivo de um objeto consiste dos valores de todos os seus atributos. Já o estado ativo indica o estado atual do objeto em um contexto específico, relacionado à lógica do negócio. Um objeto passa de um estado ativo para outro apenas se um evento relevante e pré-definido tiver ocorrido.

Para definir o comportamento de uma classe de análise, começamos procurando, na especificação do sistema, todos os estados ativos relevantes. Por exemplo, observando a descrição do nosso estudo de caso, apresentada no Capítulo 3, percebemos que um usuário (classe Usuário) pode (1) estar apto a pegar livros emprestados sem estar de posse de nenhum livro; (2) ter pego alguns livros e ainda ter tempo para devolvê-los; (3) estar de posse de livros que já deveriam ter sido devolvidos; e (4) estar suspenso. Poderia-se também levar em consideração dois estados adicionais identificáveis a partir da descrição do problema: (i) cadastrado; e (ii) não-cadastrado. Esses estados ativos não são relevantes, porém, já que todos os outros estados assumem que o usuário já foi cadastrado. Além disso, o sistema não sabe da existência um usuário não-cadastrado.

Depois disso é necessário identificar as transições entre estados. Transições são facilmente identificáveis a partir dos eventos que ocorrem durante a operação do sistema. Por exemplo, quando o evento 10 da Figura 5.3: *"registraEmprestimo(e)"* ocorre, sabe-se que ou (i) o usuário passa do estado 1 do parágrafo anterior para o estado 2, ou (ii) se mantém no estado 2. O segundo caso pode ou não corresponder a uma transição, dependendo de como o comportamento da classe é modelado. Neste exemplo, note que o evento responsável pela transição deve atualizar explicitamente o estado do objeto do tipo Usuário, uma vez que a instância dessa classe não percebe a ocorrência do evento inicial. Isso é um indicativo de que cuidado adicional deve ser tomado quando se especifica o comportamento de uma classe através de diagramas de estados. A Figura 5.7 apresenta o diagrama de estados que modela o comportamento da classe Usuário. Os eventos que aparecem no diagrama foram retirados das especificações de diversos casos de uso, não apenas de Emprestar Publicação.

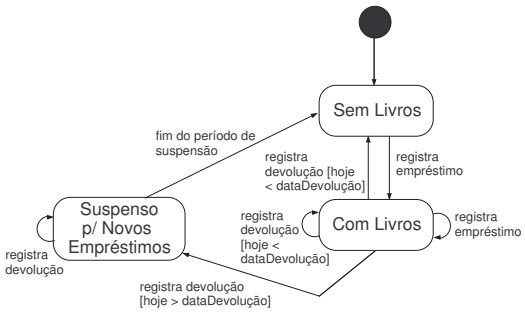


Figura 5.7: Diagrama de Estados para a Classe Usuário

é importante ter cuidado para que diagramas de estados não sejam super-utilizados. Especificar comportamentos muito simples envolve esforço e não trás benefícios significativos, já que esses comportamentos normalmente são fáceis de entender mesmo sem a ajuda dos diagramas. Além disso, identificar as transições de um diagrama de estados pode ser um trabalho muito dispendioso.

Capítulo 6

Estudo de Caso: Sistema de Caixa Automático

Este capítulo apresenta um estudo de caso completo compreendendo a modelagem estática e a modelagem dinâmica da análise orientada a objetos.

6.1 Enunciado do Problema

O sistema de caixa automático permite que clientes realizem saques e verifiquem seus saldos, de acordo com as seguintes regras de negócios:

1. Quando uma conta é criada no banco, o seu saldo é maior que zero.
2. Um cliente pode possuir várias contas no banco.
3. O cliente acessa uma conta através do terminal de um caixa eletrônico do seu banco.
4. Antes de executar qualquer operação da conta, o cliente deve fornecer o número da sua conta e a senha referente à mesma.
5. Para a realização do saque, o cliente utiliza um terminal para solicitar um valor numérico de dinheiro.
6. O cliente pode sacar qualquer quantia do caixa, desde que a mesma seja igual ou inferior ao saldo disponível. Vale a política do banco de que uma conta não aceita uma operação de saque quando a conta está com o saldo zerado. O dinheiro é liberado no dispensador de notas do caixa e debitado do saldo da conta.

7. Além de possuir o dinheiro disponível na conta, em uma operação de saque, a quantidade de dinheiro disponível no caixa deve ser maior ou igual à quantia solicitada.

8. Se o saldo de uma conta é zerado durante uma operação de saque, a conta deve se tornar inativa.

9. Os clientes que vão operar o caixa eletrônico devem estar devidamente cadastrados no banco e suas contas estão ativas.

10. Cada conta tem associado um número e uma senha. Além disso, cada conta é associada a um cliente do banco, que possui informações como nome, RG, CPF, etc.

11. As informações adicionais sobre as contas e seus clientes estão armazenadas em um Cadastro de Contas do Banco que interage com o Sistema de Caixa Automático.

12. Qualquer cliente cadastrado no banco pode efetuar depósitos em uma conta, quer a conta esteja ativa, quer ela esteja inativa.

13. Caso a conta esteja inativa e após o depósito seu saldo fique maior que zero, a conte deve ser reativada.

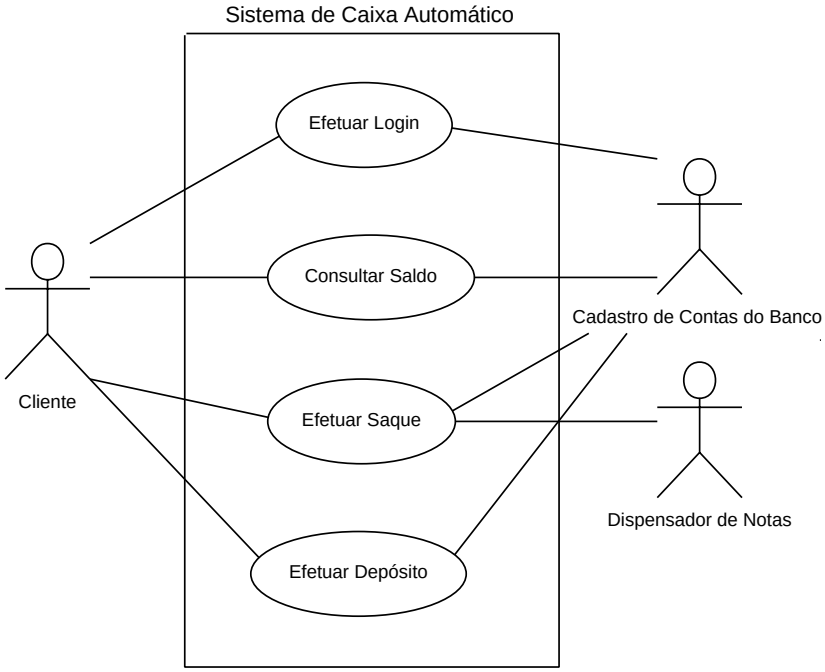


Figura 6.1: Sistema de Caixa Automático

6.2 Descrição dos Casos de Uso

Nesta seção são apresentados os casos de uso correspondentes ao enunciado de problema apresentado na Seção 6.1. A Figura 6.1 mostra o diagrama de casos de uso do sistema de caixa automático. Nesse diagrama são apresentados quatro casos de uso: (i) Efetuar Login; (ii) Consultar Saldo; (iii) Efetuar Saque; e (iv) Efetuar Depósito. Esses casos de uso serão estudados em detalhes no restante deste capítulo.

6.2.1 Caso de Uso Efetuar Login

Breve Descrição : Em um caixa eletrônico, o cliente fornece o número de sua conta e senha e, caso estes estejam corretos, o acesso ao sistema é liberado.

Atores : Cliente, Cadastro de Contas do Banco.

Pré-condição : O cliente deve ter uma conta no banco e a senha informada deve ser igual à senha da conta.

Pós-condição : Estado da conta inalterado e cliente autorizado a usar o sistema.

Requisitos Especiais : Criptografia dos dados de acesso.

Fluxo Básico :

1. O cliente solicita a opção de "Efetuar Login" no sistema.
2. O sistema pede que o cliente informe o número da conta.
3. O cliente fornece o número da conta.
4. O sistema pede que o cliente informe a sua senha.
5. O cliente fornece a senha.
6. O sistema verifica se a conta é válida e se a senha está correta, através do Cadastro de Contas do Banco. Em caso positivo, o sistema atualiza o estado do caixa eletrônico com as informações de login.
7. O sistema exibe no terminal o menu de opções que o cliente pode acessar.

Fluxo Alternativo 1 :

No passo 6 do Fluxo Básico, se a conta fornecida não existir ou se a senha estiver errada, o sistema informa que alguma das informações fornecidas está incorreta e que não é possível autenticar o cliente. Em seguida, volta ao passo 2 do Fluxo Básico.

Fluxo Alternativo 2 :

Nos passos 3 e 5 do Fluxo Básico, o cliente pode cancelar a operação.

6.2.2 Diagrama de Atividades do Caso de Uso Efetuar Login

A Figura 6.2 mostra o diagrama de atividades referente aos fluxos do caso de uso Efetuar Login.

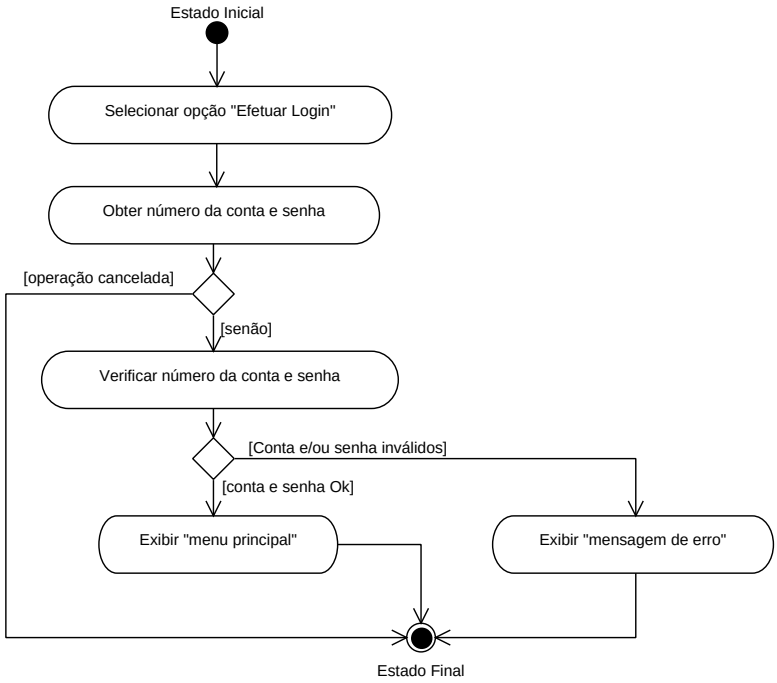


Figura 6.2: Diagrama de atividades para o caso de uso Efetuar Login

6.2.3 Diagrama de Seqüencia do Caso de Uso Efetuar Login

A Figura 6.3 mostra o diagrama de seqüência referente aos fluxos do caso de uso Efetuar Login.

6.2.4 Caso de Uso Consultar Saldo

Breve Descrição : O cliente, já autenticado, escolhe a opção "Consultar Saldo" e o sistema apresenta o seu saldo.

Atores : Cliente, Cadastro de Contas do Banco.

Pré-condição : A conta deve estar ativa e o cliente já deve ter sido autenticado junto ao sistema, através do caso de uso Efetuar Login.

Pós-condição : Estado da conta inalterado.

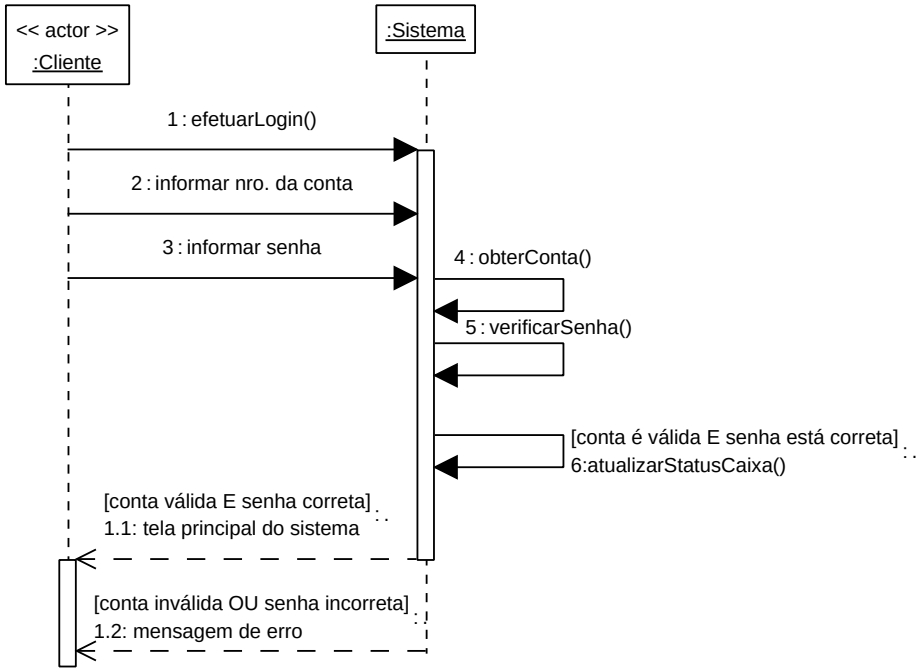


Figura 6.3: Diagrama de seqüência do caso de uso Efetuar Login

Requisitos Especiais : nenhum.

Fluxo Básico :

1. O cliente escolhe no menu principal do terminal a opção "Consultar Saldo".
2. O sistema verifica se o login foi efetuado
3. O sistema verifica se a conta está ativa, através do Cadastro de Contas do Banco.
4. O sistema obtém o saldo da conta do cliente e o imprime.

Fluxo Alternativo 1 :

No passo 2 do Fluxo Básico, se o login não foi efetuado, o sistema informa isso ao cliente.

Fluxo Alternativo 2 :

No passo 3 do Fluxo Básico, se a conta não estiver ativa, o sistema informa isso ao cliente e avisa que a consulta não pôde ser realizada.

6.2.5 Diagrama de atividades do caso de uso Consultar Saldo

A Figura 6.4 mostra o diagrama de atividades referente aos fluxos do caso de uso Consultar Saldo.

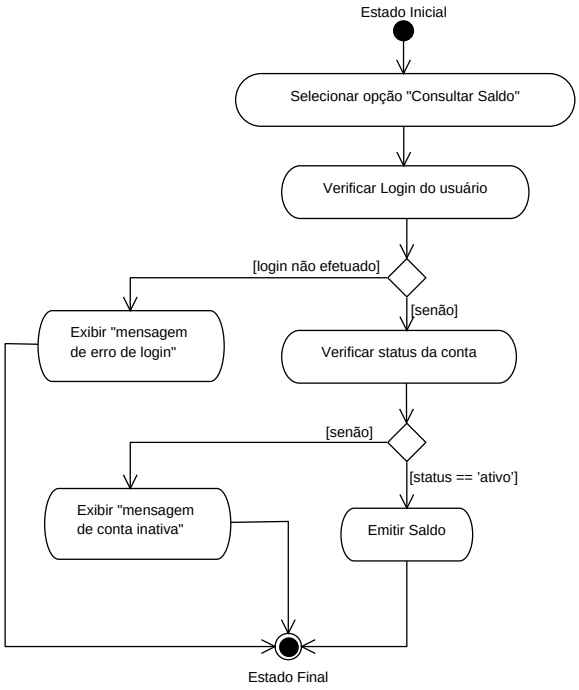


Figura 6.4: Diagrama de atividades para o caso de uso Consultar Saldo

6.2.6 Diagrama de Seqüencia do Caso de Uso Consultar Saldo

A Figura 6.5 mostra o diagrama de seqüência referente aos fluxos do caso de uso Consultar Saldo.

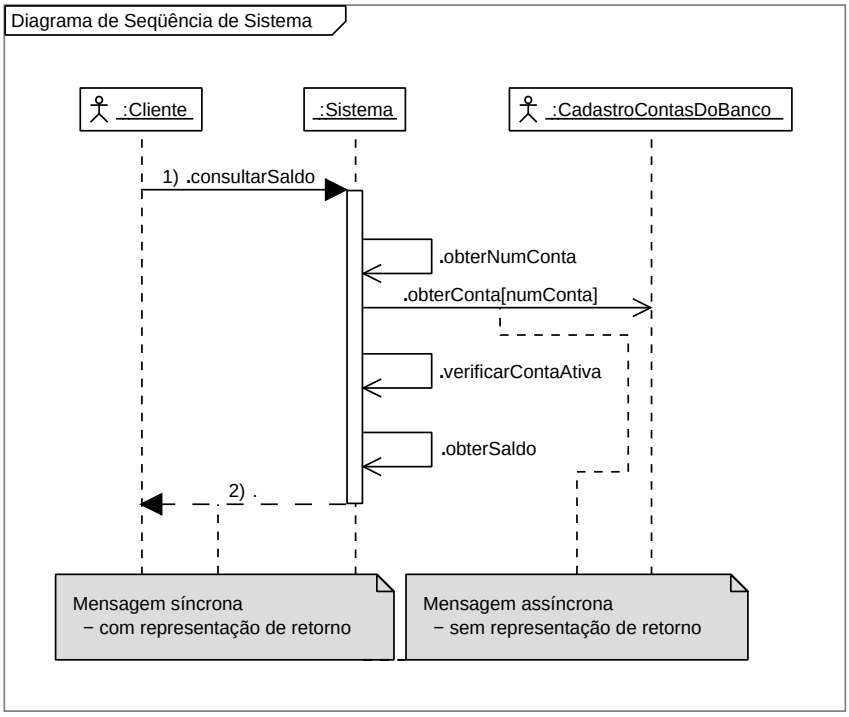


Figura 6.5: Diagrama de seqüência do caso de uso Consultar Saldo

6.2.7 Caso de Uso Efetuar Saque

Breve Descrição : O cliente, já autenticado, escolhe a opção "Efetuar Saque", informa a quantia desejada e, caso o saldo da conta seja suficiente e o caixa tenha o dinheiro necessário, a quantia é liberada.

Atores : Cliente, Cadastro de Contas do Banco, Dispensador de Notas.

Pré-condição : O cliente deve estar logado no sistema, através do caso de uso Efetuar Login. Além disso, a conta deve estar ativa e o valor a debitar deve ser maior que zero e não pode ser superior ao saldo da conta nem à quantidade de dinheiro disponível no caixa.

Pós-condição : O valor a ser sacado é subtraído do saldo da conta e do total disponível no caixa eletrônico e a quantia solicitada é fornecida ao cliente.

Requisitos Especiais : nenhum.

Fluxo Básico :

1. O cliente escolhe no menu principal do terminal a opção "Efetuar Saque".
2. O sistema verifica se o login foi efetuado.
3. O sistema verifica se a conta está ativa, através do Cadastro de Contas do Banco.
4. O sistema solicita que o cliente informe a quantia desejada.

5. O cliente informa a quantia desejada.
6. O sistema verifica se o saldo da conta é suficiente para realizar a transação e, em caso afirmativo, se há dinheiro em quantidade suficiente no caixa.
7. O sistema subtrai o valor solicitado do saldo da conta do cliente e do valor disponível no caixa e libera a quantia solicitada, através do dispensador de notas.

Fluxo Alternativo 1 :

No passo 2 do Fluxo Básico, se o login não tiver sido efetuado, o sistema informa isso ao cliente.

Fluxo Alternativo 2 :

No passo 3 do Fluxo Básico, se a conta não estiver ativa, o sistema avisa isso ao cliente e informa que o saque não pôde ser realizado.

Fluxo Alternativo 3 :

No passo 6 do Fluxo Básico, se o valor solicitado for menor que zero ou superior ao saldo da conta ou à quantidade de dinheiro disponível no caixa, o sistema informa que não é possível realizar o saque e o porquê. Em seguida, volta ao passo 4 do Fluxo Básico.

Fluxo Alternativo 4 :

Após o passo 7 do Fluxo Básico, se o saldo da conta for menor ou igual a zero, a conta deve ser desativada.

Fluxo Alternativo 5 :

No passo 5 do Fluxo Básico, o cliente pode cancelar a operação.

6.2.8 Diagrama de Atividades do Caso de Uso Efetuar Saque

A Figura 6.6 mostra o diagrama de atividades referente aos fluxos do caso de uso Efetuar Saque.

6.2.9 Diagrama de Seqüência do Caso de Uso Efetuar Saque

A Figura 6.7 mostra o diagrama de seqüência referente aos fluxos do caso de uso Efetuar Saque.

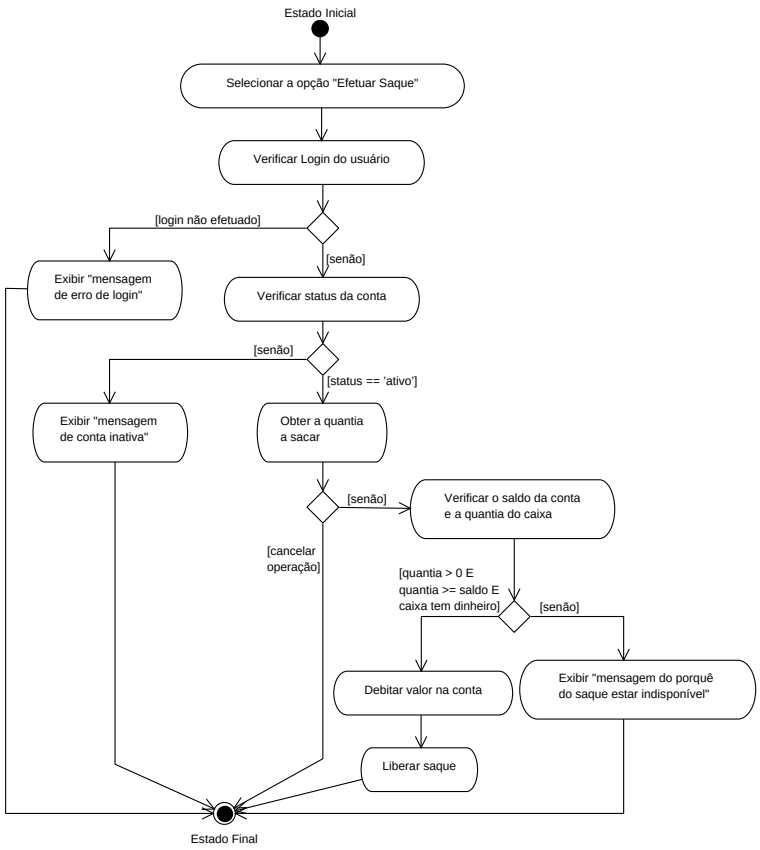


Figura 6.6: Diagrama de atividades para o caso de uso Efetuar Saque

6.2.10 Caso de Uso Efetuar Depósito

Breve Descrição : O cliente, escolhe a opção "Efetuar Depósito", informa a conta destino e a quantia desejada e, caso a quantia seja maior que zero, o sistema adiciona esse valor ao saldo da conta indicada.

Atores : Cliente, Cadastro de Contas do Banco, Dispensador de Notas.

Pré-condição : A conta destino do depósito deve ser válida e o valor a depositar deve ser maior que zero.

Pós-condição : O valor a ser depositado é adicionado ao saldo da conta.

Requisitos Especiais : nenhum.

Fluxo Básico :

1. O cliente escolhe no menu principal do terminal a opção "Efetuar Depósito".
2. O sistema solicita que o cliente informe a conta destino do depósito.

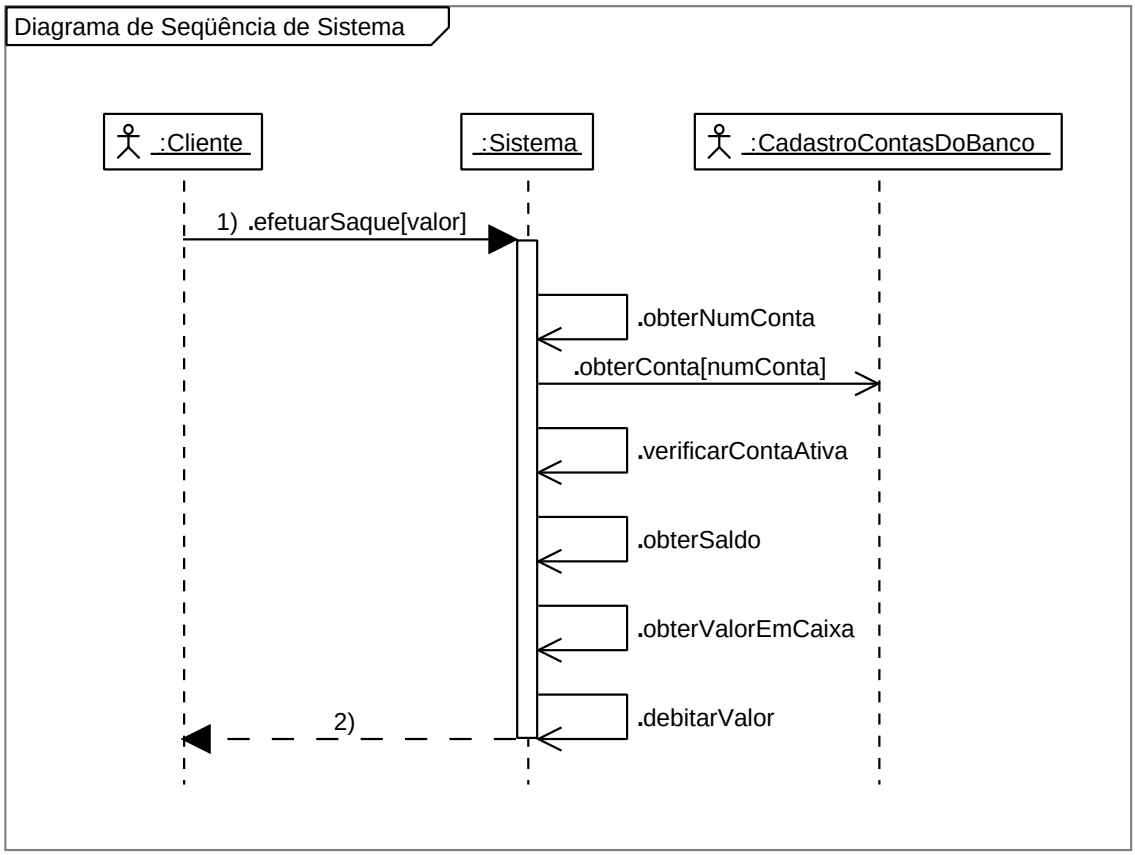


Figura 6.7: Diagrama de seqüência do caso de uso Efetuar Saque

3. O cliente informa a conta destino do depósito.
4. O sistema solicita que o cliente informe a quantia desejada.
5. O cliente informa a quantia desejada.
6. O sistema adiciona o valor depositado ao saldo da conta.
7. O sistema verifica se a conta deve ser reativada (saldo ¿ 0 E conta inativa). Em caso positivo, o sistema altera o estado da conta para ativo

Fluxo Alternativo 1 :

No passo 3 do Fluxo Básico, se a conta for inválida, o sistema informa isso ao cliente. Em seguida, volta ao passo 2 do Fluxo Básico.

Fluxo Alternativo 2 :

No passo 5 do Fluxo Básico, se a quantia informada pelo cliente for menor que zero, o sistema deve informar isso ao cliente, explicando o porquê. Em seguida, volta ao passo 4.

Fluxo Alternativo 3 :

Nos passos 3 e 5 do Fluxo Básico, o cliente pode cancelar a operação.

6.2.11 Diagrama de Atividades do Caso de Uso Efetuar Depósito

A Figura 6.8 mostra o diagrama de atividades referente aos fluxos do caso de uso Efetuar Depósito.

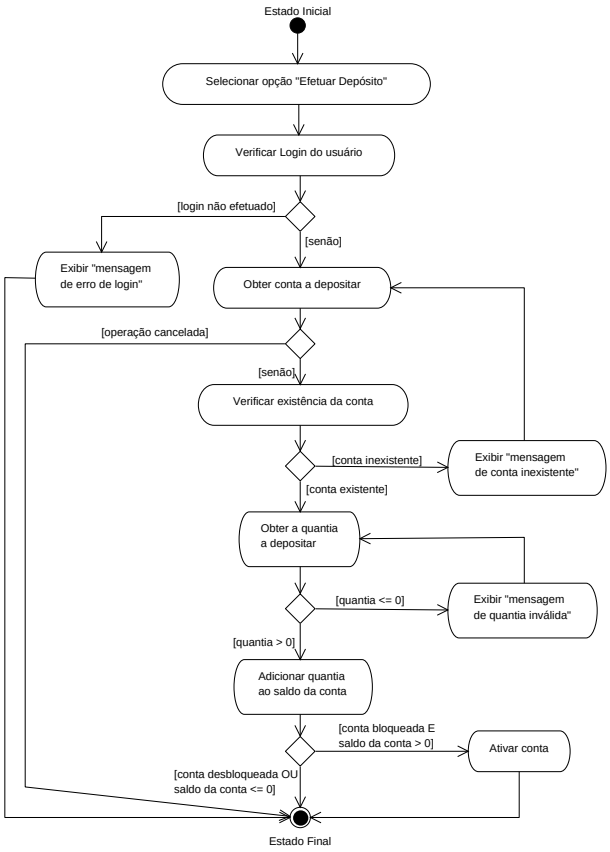


Figura 6.8: Diagrama de atividades para o caso de uso Efetuar Depósito

6.2.12 Diagrama de Seqüencia do Caso de Uso Efetuar Depósito

A Figura 6.9 mostra o diagrama de seqüência referente aos fluxos do caso de uso Efetuar Depósito.

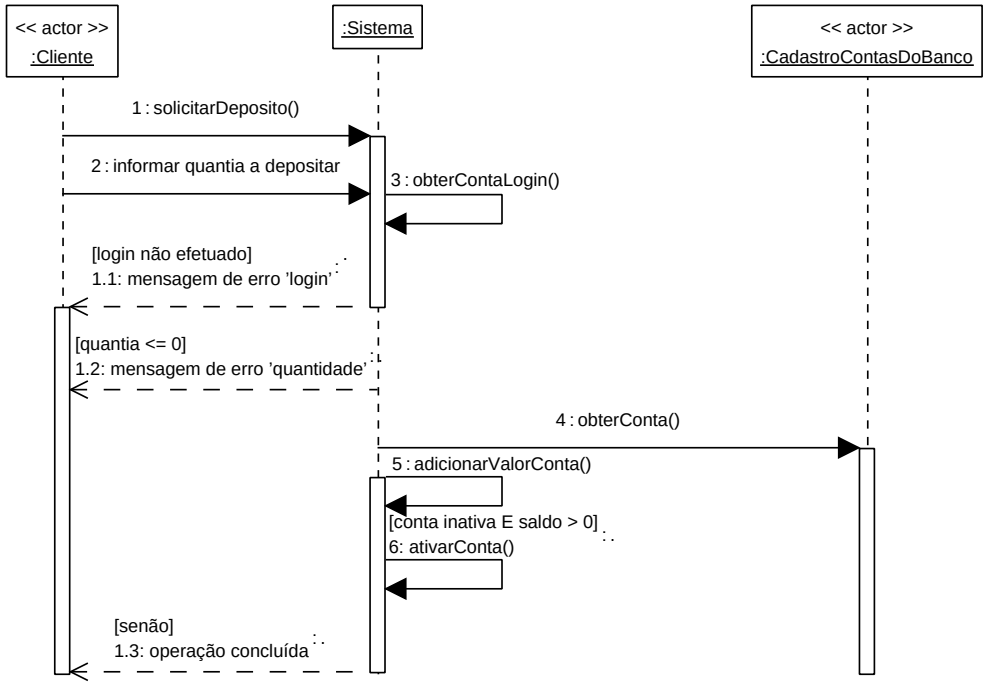


Figura 6.9: Diagrama de seqüência do caso de uso Efetuar Depósito

6.3 Modelagem Estática

Nesta seção, focamos na modelagem estática do sistema do caixa automático para a identificação das classes de análise, seus atributos e os relacionamentos entre elas. Para identificar os conceitos relevantes para o domínio do problema, será utilizada a técnica de análise textual, descrita no Capítulo 3. Sendo assim, serão destacados inicialmente os substantivos das especificações dos quatro casos de uso descritos na Seção 6.2: Efetuar Login, Consultar Saldo, Efetuar Saque e Efetuar Depósito. O artefato final desta fase será a versão inicial do diagrama de classes de análise. A Figura 6.10 mostra uma versão simplificada da seqüência de atividades a serem seguidas para a modelagem desse diagrama.

Neste capítulo é sugerida a seqüência de atividades apresentada no Capítulo 3 (Seção 3.4). A Figura 6.10 sintetiza os principais passos a serem seguidos.

6.3.1 Atividade 1: Identificar as Classes de Análise

A seguir é iniciada a identificação dos substantivos a partir da especificação dos casos de uso do sistema. Inicialmente será analisado o caso de uso Efetuar Login. Em seguida será feito o mesmo com os casos de uso Consultar Saldo, Efetuar Saque e Efetuar Depósito, destacando basicamente as diferenças entre eles.

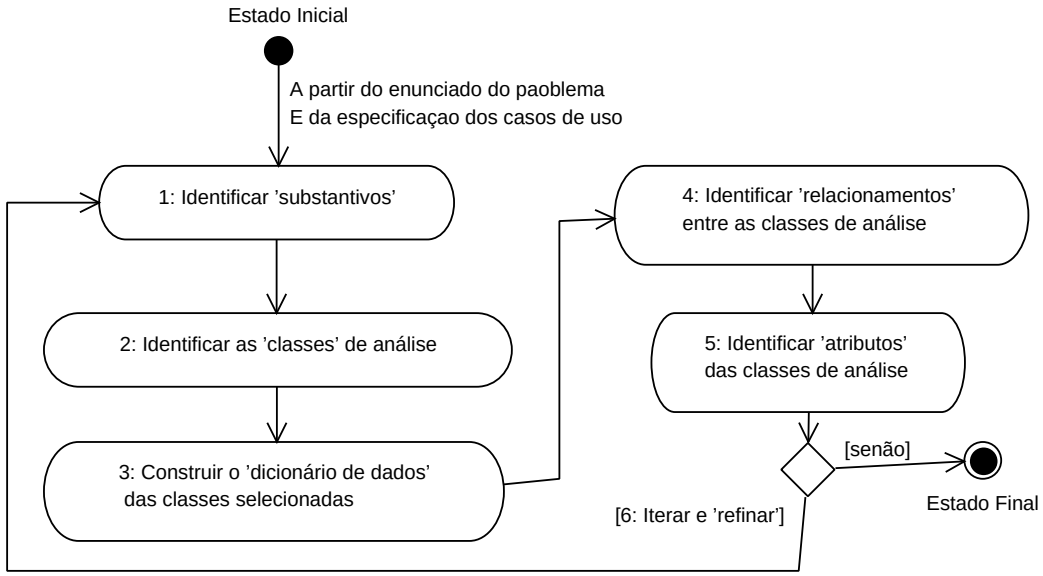


Figura 6.10: Atividades simplificadas da análise textual

Caso de Uso Efetuar Login

Breve Descrição : Em um caixa eletrônico, o cliente fornece o número de sua conta e senha e, caso estes estejam corretos, o acesso ao sistema é liberado.

Atores : Cliente, Cadastro de Contas do Banco.

Pré-condição : O cliente deve ter uma conta no banco e a senha informada deve ser igual à senha da conta.

Pós-condição : Estado da conta inalterado e cliente autorizado a usar o sistema.

Requisitos Especiais : Criptografia dos dados de acesso.

Fluxo Básico :

1. O cliente solicita a opção de "Efetuar Login" no sistema.
2. O sistema pede que o cliente informe o número da conta.
3. O cliente fornece o número da conta.
4. O sistema pede que o cliente informe a sua senha.
5. O cliente fornece a senha.
6. O sistema verifica se a conta é válida e se a senha está correta, através do Cadastro de Contas do Banco. Em caso positivo, o sistema atualiza o estado do caixa eletrônico com as informações de login.

7. O sistema exibe no terminal o menu de opções que o cliente pode acessar.

Fluxo Alternativo 1 :

No passo 6 do Fluxo Básico, se a conta fornecida não existir ou se a senha estiver errada, o sistema informa que alguma das informações fornecidas está incorreta e que não é possível autenticar o cliente. Em seguida, volta ao passo 2 do Fluxo Básico.

Fluxo Alternativo 2 :

Nos passos 3 e 5 do Fluxo Básico, o cliente pode cancelar a operação.

Caso de Uso Consultar Saldo

Breve Descrição : O cliente, já autenticado, escolhe a opção "Consultar Saldo" e o sistema apresenta o seu saldo.

Atores : Cliente, Cadastro de Contas do Banco.

Pré-condição : A conta deve estar ativa e o cliente já deve ter sido autenticado junto ao sistema, através do caso de uso Efetuar Login.

Pós-condição : Estado da conta inalterado.

Requisitos Especiais : nenhum.

Fluxo Básico :

1. O cliente escolhe no menu principal do terminal a opção "Consultar Saldo".
2. O sistema verifica se o login foi efetuado
3. O sistema verifica se a conta está ativa, através do Cadastro de Contas do Banco.
4. O sistema obtém o saldo da conta do cliente e o imprime.

Fluxo Alternativo 1 :

No passo 2 do Fluxo Básico, se o login não foi efetuado, o sistema informa isso ao cliente.

Fluxo Alternativo 2 :

No passo 3 do Fluxo Básico, se a conta não estiver ativa, o sistema informa isso ao cliente e avisa que a consulta não pôde ser realizada.

Caso de Uso Efetuar Saque

Breve Descrição : O cliente, já autenticado, escolhe a opção "Efetuar Saque", informa a quantia desejada e, caso o saldo da conta seja suficiente e o caixa tenha o dinheiro necessário, a quantia é liberada.

Atores : Cliente, Cadastro de Contas do Banco, Dispensador de Notas.

Pré-condição : O cliente deve estar logado no sistema, através do caso de uso Efetuar Login. Além disso, a conta deve estar ativa e o valor a debitar deve ser maior que zero e não pode ser superior ao saldo da conta nem à quantidade de dinheiro disponível no caixa.

Pós-condição : O valor a ser sacado é subtraído do saldo da conta e do total disponível no caixa eletrônico e a quantia solicitada é fornecida ao cliente.

Requisitos Especiais : nenhum.

Fluxo Básico :

1. O cliente escolhe no menu principal do terminal a opção "Efetuar Saque".
2. O sistema verifica se o login foi efetuado.
3. O sistema verifica se a conta está ativa, através do Cadastro de Contas do Banco.
4. O sistema solicita que o cliente informe a quantia desejada.
5. O cliente informa a quantia desejada.
6. O sistema verifica se o saldo da conta é suficiente para realizar a transação e, em caso afirmativo, se há dinheiro em quantidade suficiente no caixa.
7. O sistema subtrai o valor solicitado do saldo da conta do cliente e do valor disponível no caixa e libera a quantia solicitada, através do dispensador de notas.

Fluxo Alternativo 1 :

No passo 2 do Fluxo Básico, se o login não tiver sido efetuado, o sistema informa isso ao cliente.

Fluxo Alternativo 2 :

No passo 3 do Fluxo Básico, se a conta não estiver ativa, o sistema avisa isso ao cliente e informa que o saque não pôde ser realizado.

Fluxo Alternativo 3 :

No passo 6 do Fluxo Básico, se o valor solicitado for menor que zero ou superior ao saldo da conta ou à quantidade de dinheiro disponível no caixa, o sistema informa que não é possível realizar o saque e o porquê. Em seguida, volta ao passo 4 do Fluxo Básico.

Fluxo Alternativo 4 :

Após o passo 7 do Fluxo Básico, se o saldo da conta for menor ou igual a zero, a conta deve ser desativada.

Fluxo Alternativo 5 :

No passo 5 do Fluxo Básico, o cliente pode cancelar a operação.

Caso de Uso Efetuar Depósito

Breve Descrição : O cliente, escolhe a opção "Efetuar Depósito", informa a conta destino e a quantia desejada e, caso a quantia seja maior que zero, o sistema adiciona esse valor ao saldo da conta indicada.

Atores : Cliente, Cadastro de Contas do Banco, Dispensador de Notas.

Pré-condição : A conta destino do depósito deve ser válida e o valor a depositar deve ser maior que zero.

Pós-condição : O valor a ser depositado é adicionado ao saldo da conta.

Requisitos Especiais : nenhum.

Fluxo Básico :

1. O cliente escolhe no menu principal do terminal a opção "Efetuar Depósito".
2. O sistema solicita que o cliente informe a conta destino do depósito.
3. O cliente informa a conta destino do depósito.
4. O sistema solicita que o cliente informe a quantia desejada.
5. O cliente informa a quantia desejada.
6. O sistema adiciona o valor depositado ao saldo da conta.
7. O sistema verifica se a conta deve ser reativada (saldo ¿ 0 E conta inativa). Em caso positivo, o sistema altera o estado da conta para ativo

Fluxo Alternativo 1 :

No passo 3 do Fluxo Básico, se a conta for inválida, o sistema informa isso ao cliente. Em seguida, volta ao passo 2 do Fluxo Básico.

Fluxo Alternativo 2 :

No passo 5 do Fluxo Básico, se a quantia informada pelo cliente for menor que zero, o sistema deve informar isso ao cliente, explicando o porquê. Em seguida, volta ao passo 4.

Fluxo Alternativo 3 :

Nos passos 3 e 5 do Fluxo Básico, o cliente pode cancelar a operação.

6.3.2 Atividade 1.1: Extrair as Classes Candidatas

Efetuar Login	Consulta Saldo	Efetua Saque	Efetuar Depósito
Caixa eletrônico	Saldo	Caixa	Valor a depositar
Cliente	Terminal	Dinheiro	Valor a ser depositado
Número da conta	Saldo da conta	Quantia	Valor depositado
Senha	Login	Dispensador de Notas	Estado da conta
Acesso	Consulta	Valor a debitar	Quantia informada pelo cliente
Sistema		Quantia de dinheiro disponível em caixa	Conta destino do depósito
Cadastro de Contas do Banco		Valor a ser sacado	
Opção		Quantia solicitada	
Menu Principal		Quantia desejada	
Conta		Transação	
Banco			
Estado da conta			
Criptografia			
Operação			
Estado do Caixa eletrônico			

6.3.3 Atividade 1.2: Refinar Classes Candidatas

Após fazer uma lista inicial de possíveis classes candidatas, procura-se identificar as classes redundantes, classes irrelevantes, classes vagas, atributos, operações e papéis.

Classes Redundantes

- Valor a ser depositado, Valor depositado e Quantia informada pelo cliente: equivalentes a Valor a depositar.

- Caixa: idêntica à classe Caixa eletrônico.
- Valor a ser sacado, Quantia desejada, Quantia solicitada, Quantia: equivalentes a Valor a debitar.
- Saldo da conta, Saldo da conta do cliente: equivalentes a Saldo.
- Operação e Opção: equivalente a Transação.
- Conta destino do depósito: equivalente a Número da conta.

Classes Irrelevantes

- Transação: o enunciado do problema não exige que informações sobre as transações realizadas sejam armazenadas.
- Número da conta: atributo da classe Conta.
- Senha: atributo da classe Conta.
- Estado do caixa: termo genérico para os atributos da classe Caixa eletrônico.
- Quantidade de dinheiro disponível no caixa: atributo da classe Caixa eletrônico.
- Estado da conta: termo genérico para os atributos da classe Conta.
- Saldo: atributo da classe Conta.

Classes Vagas

- Acesso
- Menu Principal
- Criptografia
- Login
- Consulta
- Valor a debitar
- Valor a depositar

6.3.4 Atividade 1.3: Revisar Lista de Classes Candidatas

- CaixaEletrônico
- Cliente
- Sistema
- Cadastro de Contas do Banco
- Conta
- Banco
- Terminal
- Dispensador de Notas

Na lista de classes acima, observa-se que Cliente, Dispensador de Notas e Cadastro de Contas do Banco são três elementos que não compõem o sistema pois são atores que interagem com o sistema. Apesar disso, o enunciado do problema deixa claro a necessidade de se ter uma classe que represente os dados do cliente. Esse fato fica mais claro na seguinte afirmação: *"Cada conta tem associado um número e uma senha. Além disso, cada conta é associada a um cliente do banco, que possui informações como: nome, RG, CPF, etc."* Por motivos didáticos, a classe que representa os dados de um cliente do banco será chamada DadosCliente, a fim de não confundir com o ator Cliente.

6.3.5 Atividade 2: Construir o Dicionário de Dados

- **Classe** Terminal**:** classe através da qual são transmitidos os valores de entrada e saída a serem utilizados pelo caixa eletrônico. Esta classe encapsula toda a interface com o usuário, assim, o tipo de terminal a ser implementado por um Sistema de Caixa Automático pode ser facilmente modificado através da substituição desta classe.
- **Classe** DadosCliente**:** classe cujos objetos encapsulam os dados pessoais dos clientes do banco.
- **Classe** Conta**:** classe cujos objetos são as contas mantidas pelos clientes do banco.
- **Classe** CaixaEletronico**:** classe que representa o estado do caixa automático.
- **Classe** Banco**:** classe que representa o estado do banco ao qual está vinculada a conta e os clientes.
- **Classe** Sistema: o Sistema de Caixa Automático pode ser visto como uma classe que representa o sistema e engloba todas as classes pertencentes a este. Esta é uma classe conceitual e não pertence ao sistema propriamente dito.

6.3.6 Atividade 3: Identificar os Relacionamentos entre as Classes

Além das trocas de mensagens já descritas, deve-se definir a associação existente entre as classes Terminal, CaixaEletrônico, Conta, Banco e DadosCliente com a classe Sistema. A classe Sistema representa o sistema como um todo e, desta forma, todas as outras classes podem ser consideradas partes dela. A Figura 6.11 mostra o diagrama de classes inicial do Sistema de Caixa Automático. Para simplificar a representação do modelo, a classe Sistema pode ser substituída por um pacote que contenha todas as classes que compõem o sistema. Essa simplificação pode ser vista na Figura 6.12.

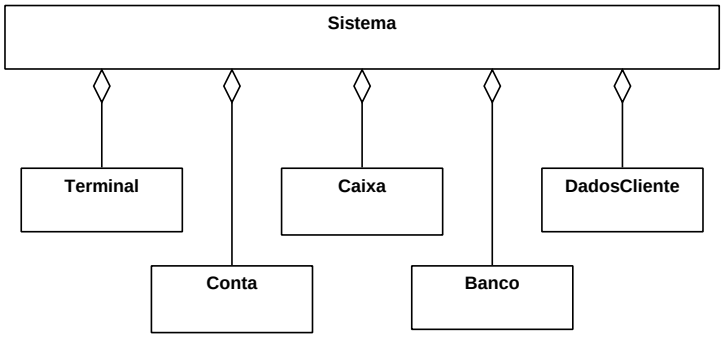


Figura 6.11: Diagrama inicial de classes com agregações.

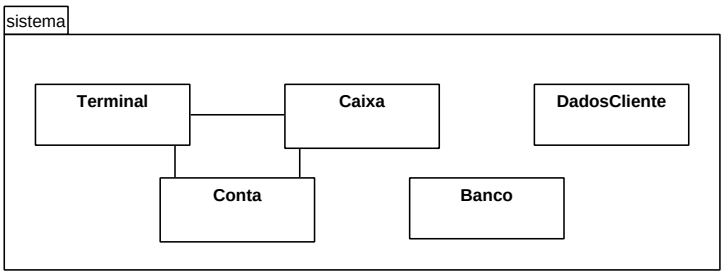


Figura 6.12: Diagrama inicial de classes com pacote.

As classes identificadas até o momento devem ser analisadas com o intuito de identificar os relacionamentos de agregação/decomposição e de generalização/especialização entre elas. Inicialmente, para identificar as agregações , deve-se analisar a descrição do sistema e as especificações dos casos de uso, identificando o modo como as entidades se relacionam. Por exemplo, a frase *"Quando uma conta é criada no banco, o seu saldo deve ser maior que zero"*, evidencia o fato de um Banco possuir uma ou mais Contas. De forma análoga, analisando a frase: *"O cliente acessa uma conta através do terminal de um caixa eletrônico do banco"*, identifica-se dois relacionamentos de agregação: (i) a classe Caixa eletrônico agrega o Terminal; e (ii) a classe Banco agrega a classe CaixaEletronico. Por fim, a frase: *"cada conta tem associado um número e uma senha. Além disso, cada conta é associada a um cliente do banco, que possui informações como: nome, RG, CPF, etc."* evidencia mais duas agregações, uma vez que a classe DadosCliente pode possuir

várias Contas e deve pertencer a um Banco.

Após a identificação das agregações, o próximo passo é identificar os possíveis relacionamentos de generalização/especialização entre as classes. Para isso, é necessário analisar a existência (ou possibilidade de criação) de tipos comuns entre as classes, de modo a generalizar nas classes base, as características compartilhadas pelas classes derivadas. O enunciado do problema e as especificações dos casos de uso também podem auxiliar essa atividade. No contexto específico do sistema do caixa automático, não foram identificados relacionamentos de generalização/especialização.

A Figura 6.13 mostra as classes do sistema com os relacionamentos identificados.

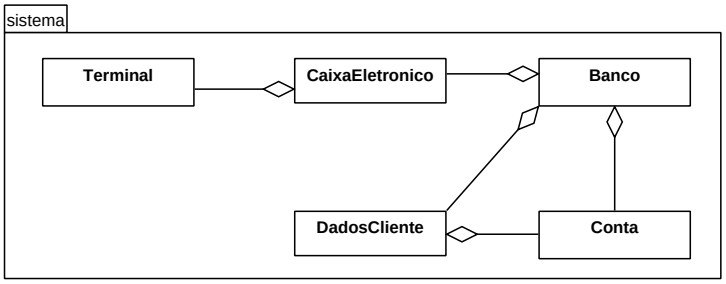


Figura 6.13: Identificação dos relacionamentos entre as classes.

6.3.7 Atividade 4: Identificação de Atributos

Para descobrir os atributos das classes de análise, retornamos à lista de classes irrelevantes apresentada na Seção 6.3.1. Diversas dessas classes candidatas foram eliminadas por representarem atributos de classes de análise e não classes de análise propriamente ditas. A Tabela 6.3.7 apresenta esses atributos, juntamente com as classes às quais eles pertencem.

Além dessa análise das especificações dos casos de uso, pode-se utilizar o enunciado do problema para descobrir outros atributos das classes de análise. O parágrafo seguinte levanta novas informações que devem ser guardadas por objetos do tipo DadosCliente:

- Cada conta tem associados um número e uma senha. Além disso, cada conta é associada a um cliente do banco, que possui informações como: nome, RG, CPF, etc..

A Figura 6.14 apresenta a versão revisada do diagrama de classes de análise, já levando em consideração os atributos identificados neste passo. Incluímos também neste diagrama a informação referente ao número de instâncias de cada uma das classes em tempo de execução (Seção 1.3.3).

Tabela 6.1: Atributos Identificados das Classes de Entidade

Classes de Entidade	Atributos
CaixaEletronico	Quantidade de dinheiro disponível Status do Login Número da Conta logada
Conta	Senha Número Saldo Status
Banco	Código Nome

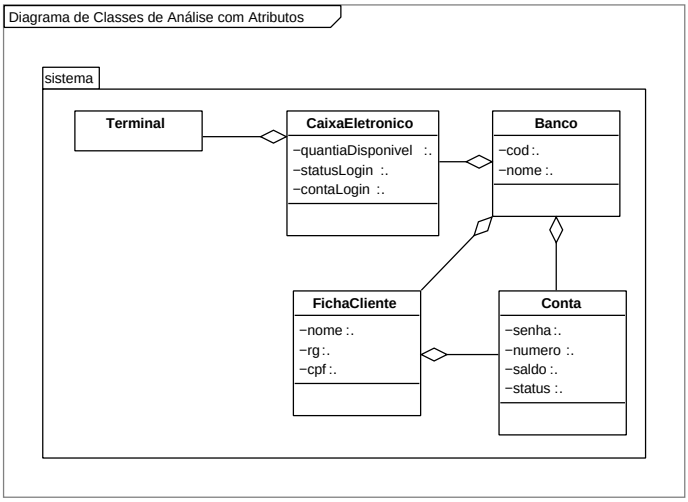


Figura 6.14: Diagrama inicial de classes de análise com atributos.

6.3.8 Atividade 5 (Iteração 2): Iterar e Refinar

Por se tratar de um processo de desenvolvimento iterativo, o sistema é construído gradativamente, a partir de refinamentos sucessivos dos modelos produzidos. Essa construção gradual do sistema proporciona uma distribuição da sua complexidade, o que além de permitir mudanças tardias dos requisitos, aumenta a qualidade final do software produzido.

Nesta seção, serão mostradas as atividades onde houve algum refinamento do diagrama inicial de classes produzido na primeira iteração. Esse diagrama é mostrado na Figura 6.14.

Atividade 1 (Iteração 2): Refinamento das Classes de Análise

Como toda a interação entre o sistema e os atores precisa ser intermediada por classes de fronteira, percebe-se que, além do Terminal já identificado, duas novas classe de fronteira devem ser introduzidas entre as candidatas. Essas novas classes, FronteiraCadastroContas e FronteiraDispensadorNotas, são responsáveis por mediar a interação entre o sistema e os atores Cadastro de Contas do Banco e Dispensador de Notas, respectivamente.

A lista seguinte apresenta as classes de análise do sistema do caixa automático e seus respectivos estereótipos. Além das novas classes de fronteira, uma nova classe foi incluída na lista inicial de classes: a classe ControladorCaixa. Esta é uma classe de controle que representa as regras de negócio da aplicação e segue o padrão de análise Controlador (Seção 3.11).

- CaixaEletronico $<< entity >>$
- Sistema
- Conta $<< entity >>$
- Banco $<< entity >>$
- Terminal $<< boundary >>$
- DadosCliente $<< entity >>$
- FronteiraCadastroContas $<< boundary >>$
- FronteiraDispensadorNotas $<< boundary >>$
- ControladorCaixa $<< control >>$

Atores: Cliente, Cadastro de Contas do Banco e Dispensador de Notas.

Atividade 2 (Iteração 2): Refinamento do Dicionário de Dados

O novo dicionário de dados é mostrado a seguir:

- **Classe** Terminal**:** classe de fronteira através da qual são transmitidos os valores de entrada e saída a serem utilizados pelo caixa eletrônico. Esta classe encapsula toda a interface com o usuário, assim, o tipo de terminal a ser implementado por um Sistema de Caixa Automático pode ser facilmente modificado através da substituição desta classe.
- **Classe** ControladorCaixa**:** classe de controle que encapsula as políticas definidas pela especificação do sistema e gerencia as interações entre as classes de fronteira e entidade. Os objetos da classe ControladorCaixa possuem operações para efetuar login no sistema, consultar o saldo de uma conta, efetuar um saque e efetuar um depósito.

- **Classe** DadosCliente**:** classe de entidade cujos objetos encapsulam os dados pessoais dos clientes do banco.
- **Classe** Conta**:** classe de entidade cujos objetos são as contas mantidas pelos clientes do banco.
- **Classe** CaixaEletronico**:** classe de entidade que representa o estado do caixa automático.
- **Classe** Banco**:** classe de entidade que representa o estado do banco ao qual está vinculada a conta e os clientes.
- **Classe** Sistema: o Sistema de Caixa Automático pode ser visto como uma classe que representa o sistema e engloba todas as classes pertencentes a este. Esta é uma classe conceitual e não pertence ao sistema propriamente dito.
- **Classe** FronteiraCadastroContas**:** classe de fronteira que media as interações entre o sistema e o ator Cadastro de Contas do Banco.
- **Classe** FronteiraDispensadorNotas**:** classe de fronteira que media as interações entre o sistema e o ator Dispensador de Notas.

Atividade 3 (Iteração 2): Refinamento dos Relacionamentos entre as Classes

Após a estruturação das classes seguindo o modelo MVC, apresentado na Seção 3.10.1 do Capítulo 3, pode-se utilizar as diretrizes fornecidas no Capítulo 3 (Seção 3.7.1) para a identificação de associações. Assim, deve-se averiguar quais as associações existentes entre as classes do diagrama inicial de classes da iteração anterior, mostrado na Figura 6.14, e as novas classes identificadas nesta iteração. A ligação entre Terminal e ControladorCaixa pode ser compreendida, uma vez que o terminal que faz a interface com o cliente precisa estar ligado à parte do caixa responsável por suas regras de negócios, para que operações como saque e consulta de saldo de fato sejam efetuadas.

Voltando ao enunciado do problema, as frases:

- Um cliente pode possuir várias contas no banco.
- O cliente acessa uma conta através do caixa eletrônico, bastando usar um terminal fornecendo o número de conta e a senha referente à mesma.

justificam esta associação.

A associação entre ControladorCaixa e Conta pode ser explicada, uma vez que na lógica de negócio do caixa eletrônico, ele deve acessar informações sobre a conta, necessitando assim interagir com ela. Além disso, ControladorCaixa e a classe de entidade CaixaEletronico também devem permanecer associadas, já que a segunda representa o estado do caixa automático e esse estado não pode ser acessado sem levar em consideração as regras de negócio do sistema. Outra maneira prática de identificar as associações relevantes entre as classes é simular a execução dos passos dos fluxos dos casos de uso. Essa simulação deve ser abstrata o suficiente para se ater apenas à lógica do

negócio, sem entrar em detalhes de implementação. Por exemplo, numa operação de saque, através da interface do terminal, o cliente simplesmente solicita a quantia desejada. A classe de controle ControladorCaixa deve conhecer a classe de fronteira do sistema do sistema de cadastro de contas do banco: FronteiraCadastroContas. Essa associação é necessária, uma vez que os dados da conta devem ser obtidos antes de serem manipulados. Mas para visualizar essa manipulação, o controlador também deve conhecer a classe de entidade Conta, a fim de conseguir verificar se a conta obtida do cliente está ativa, bem como confirmar se de fato o saldo disponível permite a efetivação do saque e se o caixa tem a quantia solicitada. Desta forma, é a classe de controle que intermedia a comunicação entre o terminal e as contas e gerencia o estado do caixa automático (associação com a classe CaixaEletronico). A classe Terminal simplesmente entra com o valor do saque desejado e, caso a classe ControladorCaixa verifique que é possível realizar o saque e aciona o dispensador de notas, através da associação entre as classes ControladorCaixa e FronteiraDispensadorNotas. Caso contrário, o terminal repassa para o usuário uma mensagem vinda da classe controladora avisando que o saque não pôde ser concluído.

Seguindo as restrições do modelo MVC, todas as ligações entre classes de fronteira e classes de entidade devem ser removidas do diagrama, já que essas classes devem ser sempre ligadas através de uma classe de controle (Seção 3.10.1). Sendo assim, a agregação entre as classes CarxaEletronico e Terminal deve ser removida. Dessa forma, a comunicação necessária entre essas classes será intermediada pela classe ControladorCaixa

O diagrama da Figura 6.15 apresenta o diagrama de classes final da fase de análise, sem a definição das operações das classes.

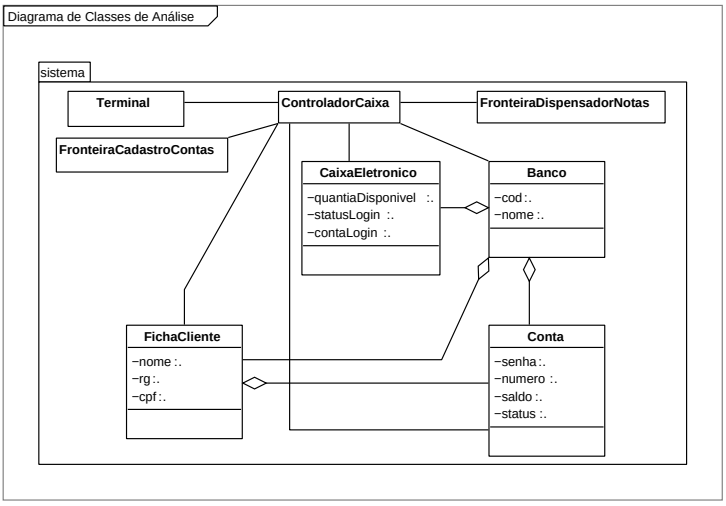


Figura 6.15: Diagrama de classes final de análise (sem operações)

6.4 Modelagem Dinâmica

Depois de modelar os aspectos estáticos do sistema, pode-se iniciar a modelagem dinâmica, relativa ao comportamento das classes de análise em tempo de execução. Realizamos uma nova análise das especificações dos quatro casos de uso especificados: Efetuar Login, Consultar Saldo, Efetuar Saque e Efetuar Depósito. Deve-se procurar por eventos que descrevam o comportamento do sistema. Seguindo as diretrizes apresentadas no Capítulo 5 (Seção 5.1), deve-se dar uma atenção especial à ocorrência de verbos e aos contextos nos quais esses verbos são empregados. A informação extraída é então utilizada para definir as operações das classes de análise pertencentes à sua interface pública. A Figura 6.16 sistematiza os passos a serem seguidos através de um diagrama de atividades UML.

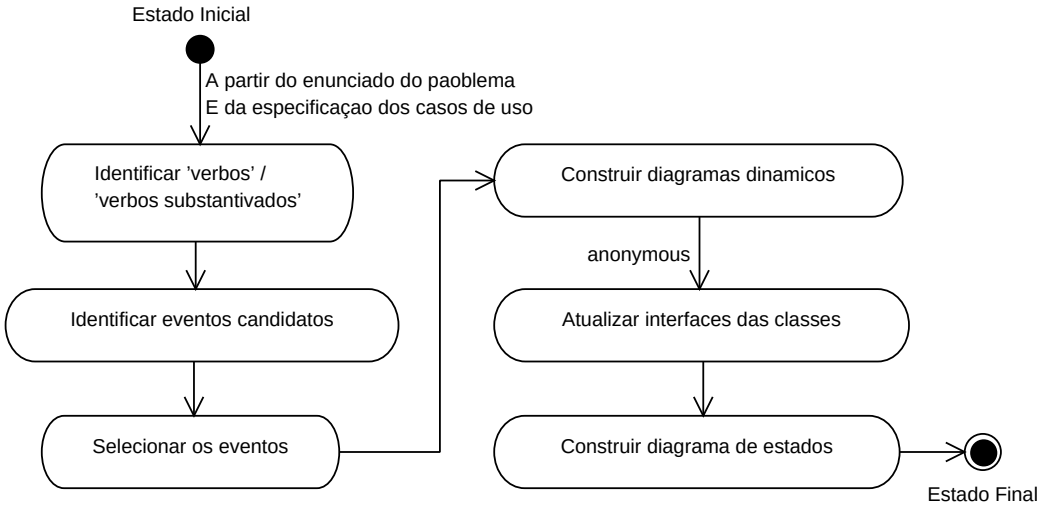


Figura 6.16: Atividades da Modelagem Dinâmica

6.4.1 Passo 1: Identificar Eventos

A seguir, são apresentados os fluxos de eventos dos casos de uso do sistema de caixa automático, destacando com sublinhado os trechos que possam corresponder a eventos.

Caso de Uso Efetuar Login

Fluxo Básico :

1. O cliente solicita a opção de "Efetuar Login" no sistema.
2. O sistema pede que o cliente informe o número da conta.
3. O cliente fornece o número da conta.

4. O sistema pede que o cliente informe a sua senha.
5. O cliente fornece a senha.
6. O sistema verifica se a conta é válida e se a senha está correta, através do Cadastro de Contas do Banco. Em caso positivo, o sistema atualiza o estado do caixa eletrônico com as informações de login.
7. O sistema exibe no terminal o menu de opções que o cliente pode acessar.

Fluxo Alternativo 1 :

No passo 6 do Fluxo Básico, se a conta fornecida não existir ou se a senha estiver errada, o sistema informa que alguma das informações fornecidas está incorreta e que não é possível autenticar o cliente. Em seguida, volta ao passo 2 do Fluxo Básico.

Fluxo Alternativo 2 :

Nos passos 3 e 5 do Fluxo Básico, o cliente pode cancelar a operação.

Caso de Uso Consultar Saldo

Fluxo Básico :

1. O cliente escolhe no menu principal do terminal a opção "Consultar Saldo".
2. O sistema verifica se o login foi efetuado.
3. O sistema verifica se a conta está ativa, através do Cadastro de Contas do Banco.
4. O sistema obtém o saldo da conta do cliente e o imprime.

Fluxo Alternativo 1 :

No passo 2 do Fluxo Básico, se o login não foi efetuado, o sistema informa isso ao cliente.

Fluxo Alternativo 2 :

No passo 3 do Fluxo Básico, se a conta não estiver ativa, o sistema informa isso ao cliente e avisa que a consulta não pôde ser realizada.

Caso de Uso Efetuar Saque

Fluxo Básico :

1. O cliente escolhe no menu principal do terminal a opção "Efetuar Saque".
2. O sistema verifica se o login foi efetuado.
3. O sistema verifica se a conta está ativa, através do Cadastro de Contas do Banco.
4. O sistema solicita que o cliente informe a quantia desejada.

5. O cliente informa a quantia desejada.
6. O sistema verifica se o saldo da conta é suficiente para realizar a transação e, em caso afirmativo, se há dinheiro em quantidade suficiente no caixa.
7. O sistema subtrai o valor solicitado do saldo da conta do cliente e do valor disponível no caixa e libera a quantia solicitada, através do dispensador de notas.

Fluxo Alternativo 1 :

No passo 2 do Fluxo Básico, se o login não tiver sido efetuado, o sistema informa isso ao cliente.

Fluxo Alternativo 2 :

No passo 3 do Fluxo Básico, se a conta não estiver ativa, o sistema avisa isso ao cliente e informa que o saque não pôde ser realizado.

Fluxo Alternativo 3 :

No passo 6 do Fluxo Básico, se o valor solicitado for menor que zero ou superior ao saldo da conta ou à quantidade de dinheiro disponível no caixa, o sistema informa que não é possível realizar o saque e o porquê. Em seguida, volta ao passo 4 do Fluxo Básico.

Fluxo Alternativo 4 :

Após o passo 7 do Fluxo Básico, se o saldo da conta for menor ou igual a zero, a conta deve
ser desativada.

Fluxo Alternativo 5 :

No passo 5 do Fluxo Básico, o cliente pode cancelar a operação.

Caso de Uso Efetuar Depósito

Fluxo Básico :

1. O cliente escolhe no menu principal do terminal a opção "Efetuar Depósito".
2. O sistema solicita que o cliente informe a conta destino do depósito.
3. O cliente informa a conta destino do depósito.
4. O sistema solicita que o cliente informe a quantia desejada.
5. O cliente informa a quantia desejada.
6. O sistema adiciona o valor depositado ao saldo da conta.
7. O sistema verifica se a conta deve ser reativada ($saldo > 0$ E conta inativa). Em caso positivo, o sistema altera o estado da conta para ativo.

Fluxo Alternativo 1 :

No passo 3 do Fluxo Básico, se a conta for inválida, o sistema informa isso ao cliente. Em seguida, volta ao passo 2 do Fluxo Básico.

Fluxo Alternativo 2 :

No passo 5 do Fluxo Básico, se a quantia informada pelo cliente for menor que zero, o sistema deve informar isso ao cliente, explicando o porquê. Em seguida, volta ao passo 4.

Fluxo Alternativo 3 :

Nos passos 3 e 5 do Fluxo Básico, o cliente pode cancelar a operação.

Atividade 1.1: Identificar Eventos Candidatos

A seguir, é apresentada a lista de eventos candidatos identificados a partir das especificações dos casos de uso. Ao lado de cada evento candidato, é indicado o ator que o inicia.

- Escolher opção “Efetuar Login” (cliente)
- Solicitar o número da conta (sistema)
- Fornecer o número da conta (cliente)
- Solicitar a senha (sistema)
- Fornecer a senha (cliente)
- Verificar se a conta é válida (sistema)
- Verificar se a senha está correta (sistema)
- Atualizar o estado do caixa eletrônico com as informações de login (sistema)
- Exibir no terminal o menu de opções (sistema)
- Informar que o login ou a senha está inválido (sistema)
- cancelar a operação de efetuar login (cliente)
- Escolher a opção “Consultar Saldo”. (cliente)
- Verificar se o login foi efetuado (sistema)
- Verificar se a conta está ativa. (sistema)
- Obter o saldo da conta. (sistema)
- Imprimir o saldo da conta. (sistema)
- Informar ao cliente que a conta não está ativa. (sistema)

- Informar ao cliente que o login não foi efetuado (sistema)
- Informar ao cliente que a consulta não pôde ser realizada. (sistema)
- Informar ao cliente que alguma das informações fornecidas é incorreta. (sistema)
- Informar ao cliente que a consulta não pôde ser realizada. (sistema)
- Cancelar a operação de consultar saldo (cliente)
- Escolher a opção “Efetuar Saque”. (cliente)
- Solicita que o cliente informe a quantia desejada. (sistema)
- Informar a quantia desejada. (cliente)
- Verificar se o cliente tem saldo suficiente para realizar a transação. (sistema)
- Verificar se há dinheiro em quantidade suficiente no caixa. (sistema)
- Subtrair o valor solicitado do saldo da conta do cliente. (sistema)
- Liberar a quantia solicitada. (sistema)
- Informar que não é possível realizar o saque porque o saldo da conta do cliente não é suficiente. (sistema)
- Informar que não é possível realizar o saque porque não há dinheiro suficiente no caixa. (sistema)
- Desativar a conta (sistema)
- Cancelar a operação de efetuar saque (cliente)
- Escolher a opção “Efetuar Depósito” (cliente)
- Solicitar que o cliente informe o número da conta do depósito (sistema)
- Informar o número da conta do depósito (cliente)
- Solicitar a quantia desejada para depósito (sistema)
- Informar a quantia desejada para depósito (cliente)
- Adicionar o valor depositado ao saldo da conta (sistema)
- Verificar se a conta deve ser reativada (sistema)
- Reativar a conta (sistema)
- Informar que a quantia desejada é inválida e que deve ser maior que zero (sistema)
- Cancelar a operação de efetuar depósito (cliente)

Note que praticamente cada passo dos fluxos de eventos produziu um evento. Isso é esperado, tendo em vista que um caso de uso modela interações entre atores e o sistema.

Passo 1.2: Selecionar os Eventos Candidatos

Depois de identificar os eventos, precisamos associá-los às classes de análise. Como os eventos encontrados até o momento enxergam o sistema como uma caixa preta, precisamos descobrir os eventos internos ao sistema que cada um desses eventos externos produz. Para tanto, usamos a divisão das classes de análise entre fronteira, controle e entidade, e do padrão de interação que essas classes seguem (Seção 3.10.1).

Primeiramente, é necessário identificar os eventos relevantes para o sistema. Pode-se iniciar analisando os eventos encontrados até o momento, à procura de eventos que não possam ser realizados apenas por classes de fronteira.

Os seguintes eventos dizem respeito exclusivamente à classe Terminal:

- Escolher opção "Efetuar Login" (cliente)
- Solicitar o número da conta (sistema)
- Fornecer o número da conta (cliente)
- Solicitar a senha (sistema)
- Fornecer a senha (cliente)
- Exibir no terminal o menu de opções (sistema)
- Informar que o login ou a senha está inválido (sistema)
- cancelar a operação de efetuar login (cliente)
- Escolher a opção "Consultar Saldo".
- Imprimir o saldo da conta.
- Informar o cliente que a conta não está ativa.
- Informar ao cliente que o login não foi efetuado.
- Informar ao cliente que a consulta não pôde ser realizada.
- Informar ao cliente que alguma das informações fornecidas é incorreta.
- Informar ao cliente que a consulta não pôde ser realizada.
- Cancelar a operação de consultar saldo.
- Escolher a opção "Efetuar Saque".
- Solicitar que o cliente informe a quantia desejada.
- Informar a quantia desejada.

- Informar que não é possível realizar o saque porque o saldo não é suficiente.
- Informar que não é possível realizar o saque porque não há dinheiro suficiente no caixa.
- Cancelar a operação de efetuar saque.
- Escolher a opção “Efetuar Depósito” (cliente)
- Solicitar que o cliente informe o número da conta do depósito (sistema)
- Informar o número da conta do depósito (cliente)
- Solicitar a quantia desejada para depósito (sistema)
- Informar a quantia desejada para depósito (cliente)
- Informar que a quantia desejada é inválida e que deve ser maior que zero (sistema)
- Cancelar a operação de efetuar depósito (cliente)

O seguinte evento diz respeito exclusivamente à classe FronteiraDispensadorNotas:

- Liberar a quantia solicitada.

O elemento comum a todos esses eventos é o fato de se referirem exclusivamente à interação entre o cliente e o sistema. Ou seja, são eventos relativos à captação de informações fornecidas pelo cliente (“informar a quantia desejada”, “escolher a opção Consulta Saldo”) e à produção de resultados externamente observáveis resultantes de computações internas previamente realizadas (“ liberar a quantia desejada”, “informar ao cliente que a consulta não pôde ser realizada.”).

Após separar os eventos de entrada e saída, sobraram os seguintes:

- Verificar se a conta é válida.
- Verificar se a senha está correta.
- Atualizar o estado do caixa eletrônico com as informações de login.
- Verificar se o login foi efetuado.
- Verificar se a conta está ativa.
- Obter o saldo da conta.
- Verificar se o cliente tem saldo suficiente para realizar a transação.
- Verificar se há dinheiro em quantidade suficiente no caixa.
- Subtrair o valor solicitado do saldo da conta do cliente.

- Desativar a conta.
- Adicionar o valor depositado ao saldo da conta.
- Verificar se a conta deve ser reativada.
- Reativar a conta.

Agora que já foram identificados os eventos que devem ser refinados, é necessário determinar quais eventos internos produzem o comportamento desejado. Deve-se levar em consideração o fato do sistema estar modelado de acordo com as classes de análise identificadas durante a modelagem estática. Tendo isso em vista, deve-se refinar os eventos identificados e, a partir destes, criar novos eventos relevantes ou eliminar eventos irrelevantes. Por exemplo, pode-se decompor o evento "obter o saldo da conta" nos seguintes passos:

1. obter o objeto do tipo Conta correspondente à conta desejada a partir do FronteiraCadastroContas.
2. obter o saldo a partir da Conta.
3. mostrar o saldo obtido no Terminal.

Note que, nos passos acima, são utilizados os nomes das classes de análise para referenciar os conceitos do domínio do problema. Isso é aceitável neste caso, já que está-se falando do funcionamento interno do sistema e não do seu comportamento externamente observável. Depois de refinar todos os eventos relevantes, pode-se construir os diagramas de seqüência que representam graficamente o funcionamento interno do sistema.

6.4.2 Diagramas de Seqüência

Após identificar os eventos que são relevantes para o sistema, se faz necessário refinar os diagramas de seqüência de sistema (Figuras 6.3, 6.5, 6.7 e 6.9) para que fiquem consistentes com esses eventos. Eventos externos, produzidos ou recebidos por atores, e eventos internos, aqueles que fluem entre as instâncias das classes de análise, são incluídos nos diagramas. Além disso, o objeto Sistema deve ser substituído por instâncias das classes de análise. Os diagramas de seqüência resultantes para os casos de uso Efetuar Login, Consultar Saldo, Efetuar Saque e Efetuar Depósito são apresentados respectivamente, nas Figuras 6.17, 6.18, 6.19 e 6.20. Em relação ao caso de uso Efetuar Saques, os atores Cadastro de Contas do Banco e Dispensador de Notas foram omitidos, a fim de manter a legibilidade da Figura 6.19.

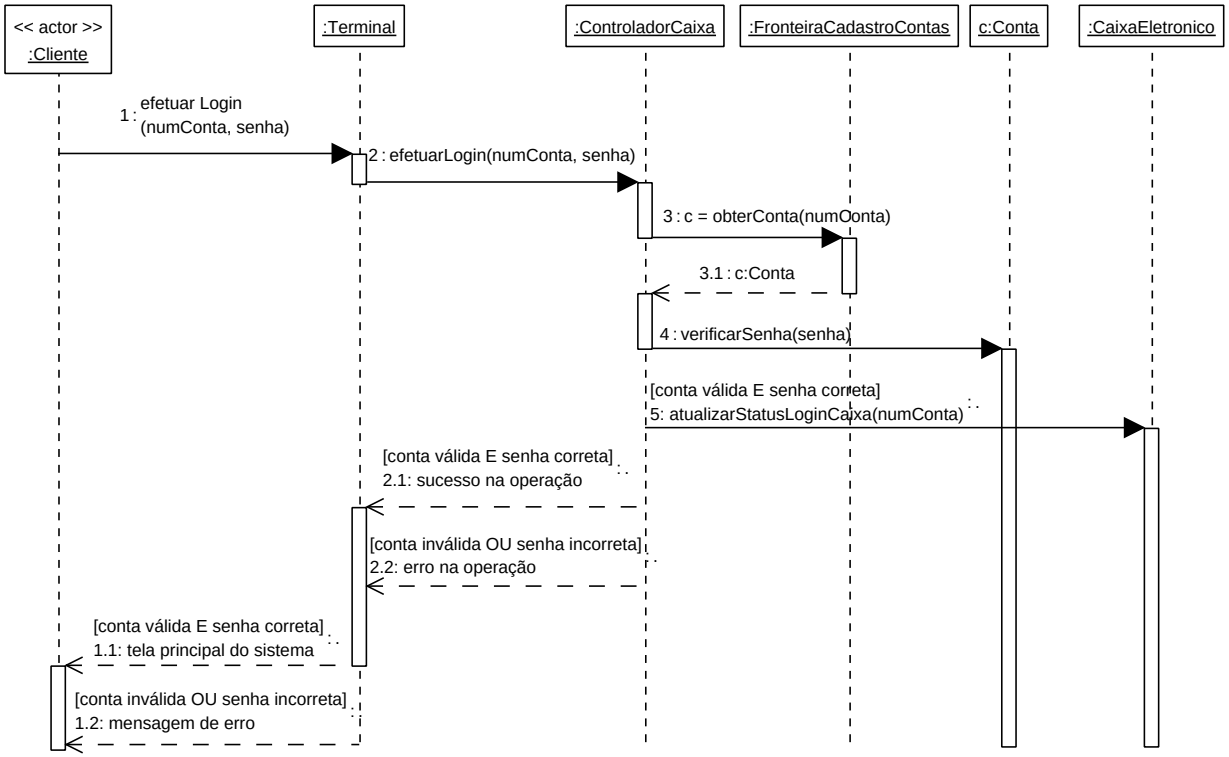


Figura 6.17: Diagrama de seqüência Efetuar Login

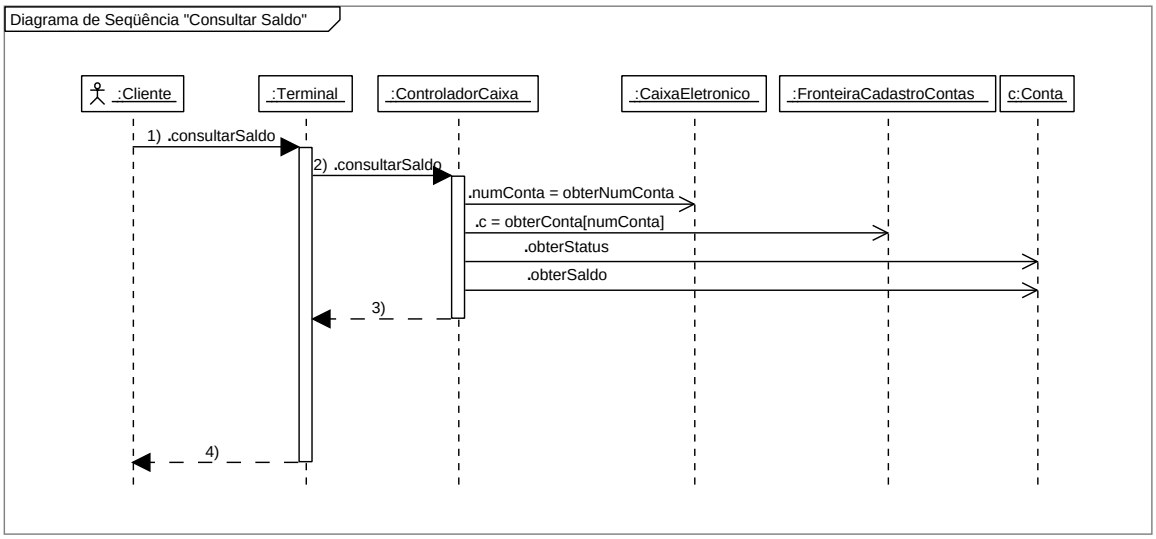


Figura 6.18: Diagrama de seqüência Consultar Saldo

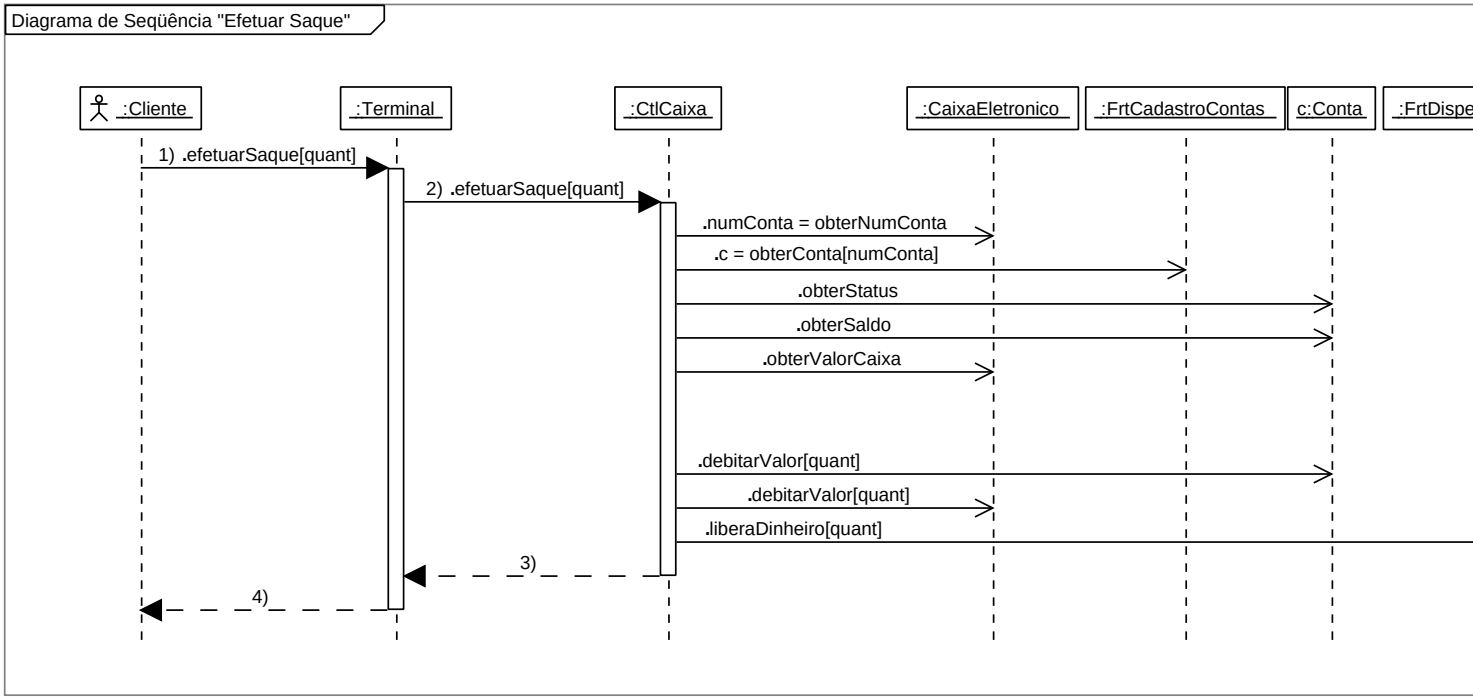


Figura 6.19: Diagrama de seqüência Efetuar Saque

6.5 Diagrama de Colaboração

A Figura 6.21 apresenta o diagrama de colaboração decorrente da união dos diagramas de seqüência das Figuras 6.17, 6.18, 6.19 e 6.20. A fim de reduzir a complexidade do diagrama, os eventos de retorno das mensagens não foram representados.

6.6 Diagrama Final de Classes da Análise

A Figura 6.22 apresenta o diagrama de classes de análise revisado e levando em consideração as operações identificadas a partir dos diagramas de seqüência das Figuras 6.17, 6.18, 6.19 e 6.20. As operações foram selecionadas usando-se as diretrizes apresentadas na Seção 5.5. Eventos recebidos por instâncias de uma classe se tornam operações de sua interface pública. As informações associadas a esses eventos correspondem aos seus parâmetros e os tipos de retorno são determinados pelos eventos produzidos em resposta à recepção de um evento, quando aplicável.

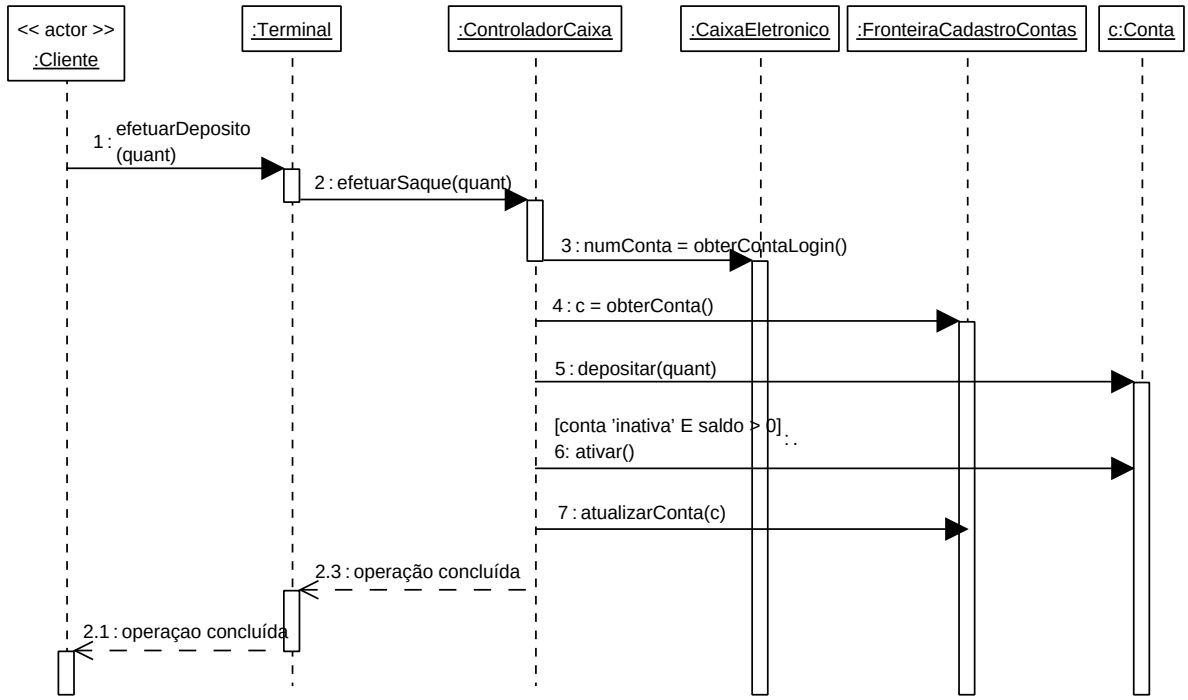


Figura 6.20: Diagrama de seqüência Efetuar Depósito

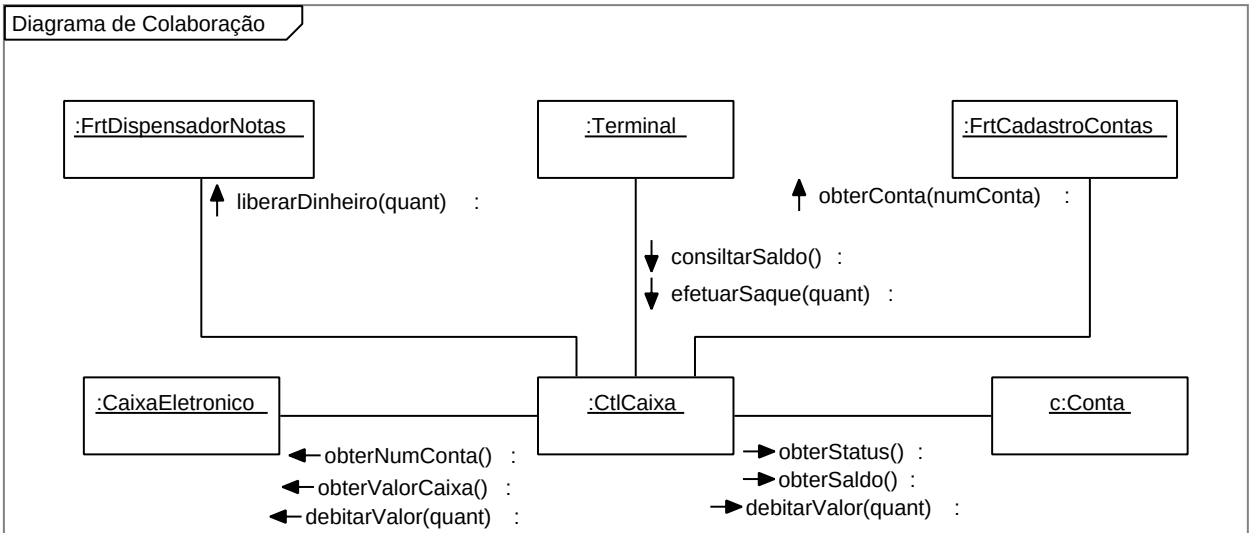


Figura 6.21: Diagrama de colaboração

6.7 Diagrama de Estados da Classe Conta

Tendo em vista a variação dos comportamentos disponíveis de acordo com o estado da conta, julgou-se interessante modelar os estados da classe Conta com as respectivas operações que podem

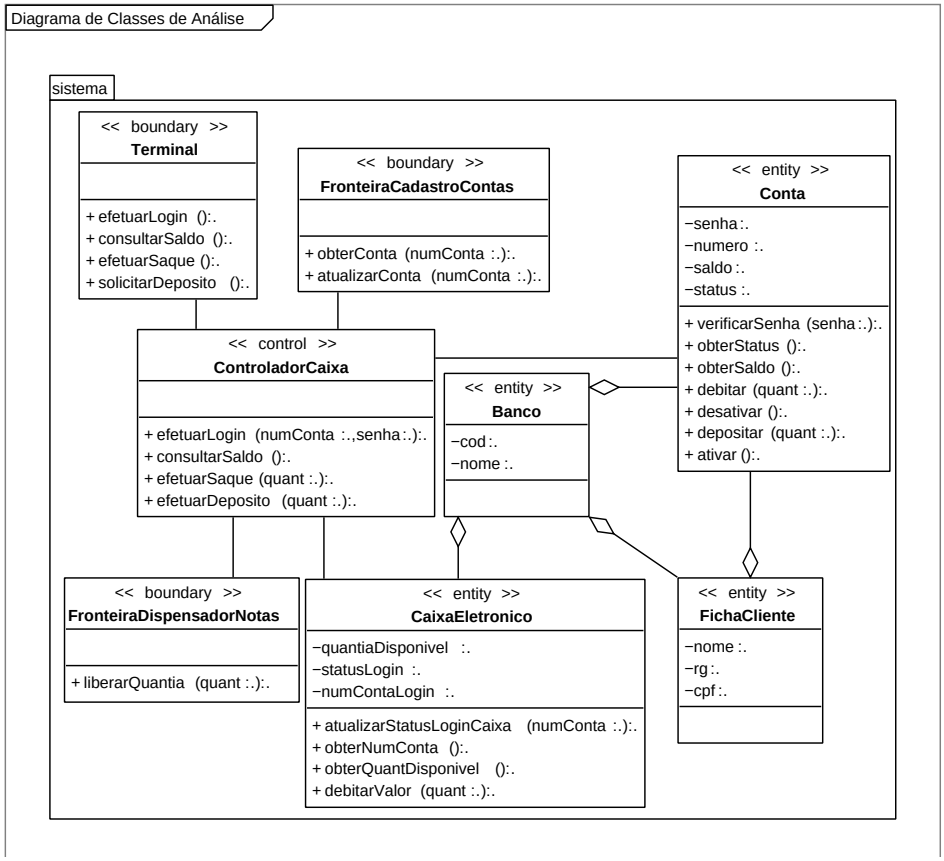


Figura 6.22: Diagrama de Classes de Análise com Operações

ser executadas em cada um desses estados. A Figura 6.23 mostra essa modelagem através de um diagrama de estados UML.

6.8 Refinamentos do Diagrama de Classes de Análise

Uma extensão realista para o sistema é o fato de um cliente poder possuir dois (ou mais) tipos distintos de contas: uma conta corrente e uma conta poupança. Neste caso, ao invés de uma única classe, Conta, correspondente a uma conta no banco, teríamos mais duas, ContaCorrente e ContaPoupanca. A Figura 6.24 apresenta um diagrama de classes com essa hierarquia expandida.

No diagrama de classes da Figura 6.22, o status de cada conta é representado por um atributo da classe Conta. É possível, porém, usar abordagens alternativas nas quais o status da conta é modelado como uma ou mais classes. Neste caso, uma classe abstrata Status encapsularia o comportamento que é comum a qualquer status, enquanto classes concretas como Ativa e Inativa encapsulariam o comportamento mais especializado. O diagrama de classes da Figura 6.25 apresenta a hierarquia

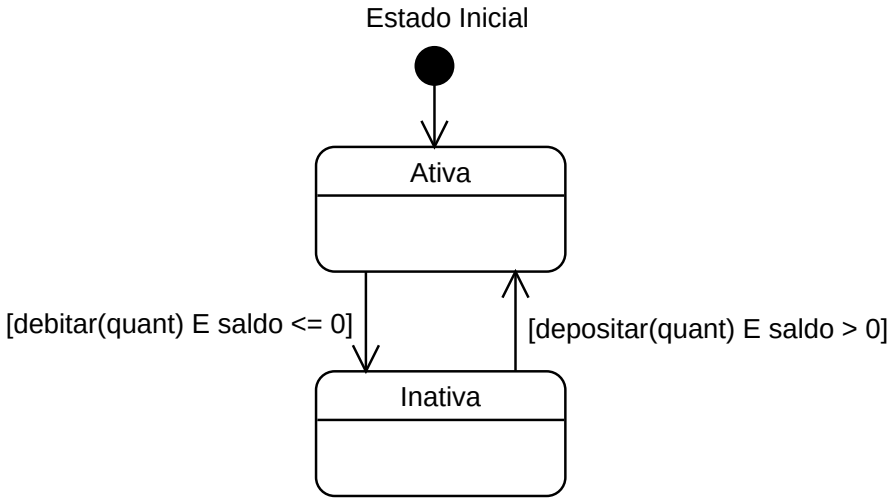


Figura 6.23: Diagrama de Estados da Classe Conta

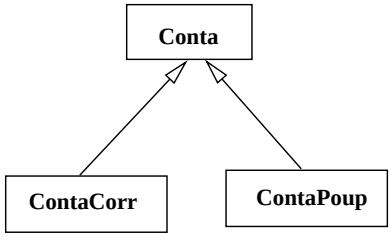


Figura 6.24: Nova hierarquia para contas.

estendida de contas, incluindo as classes que representam explicitamente o status de cada conta.

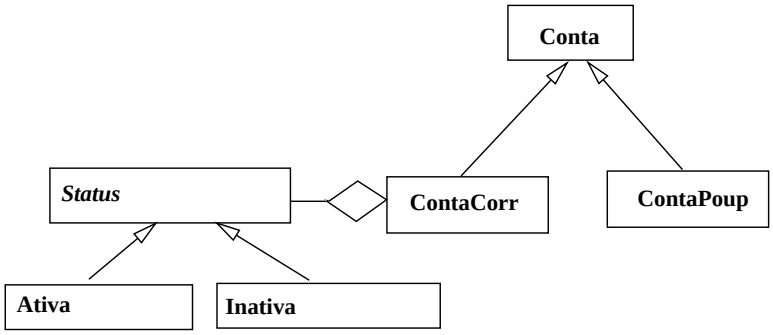


Figura 6.25: Hierarquia de contas revisada.

Estudaremos a modelagem do status de uma conta como um conjunto de classes no Capítulo 7

(Seção 7.6), que fala sobre a transição da análise para o projeto.

Capítulo 7

Transição da Análise OO para o Projeto OO

Nos Capítulos 3 e 5 foram descritos os passos da análise orientada a objetos. A análise OO foca no entendimento dos requisitos, conceitos e operações relacionados com o sistema. Ela enfatiza o que deve ser feito – quais são os processos dentro do sistema, conceitos relevantes, etc. A Figura 7.1 mostra a relação entre as classes de análise e a realização do caso de uso Efetuar Saque, apresentado no capítulo anterior. Durante a análise, um caso de uso se desdobra em diversas classes de análise distintas que correspondem a diferentes aspectos do sistema. Os resultados concretos da etapa de análise são:

(i) um diagrama de classes de análise, que descreve os conceitos relevantes do sistema; e

(ii) um conjunto de diagramas de interação, que representam os cenários aos quais o sistema deve dar suporte.

Neste capítulo, é descrita a etapa de transição da análise OO para o projeto OO. Mais especificamente, são descritos os estágios iniciais do projeto orientado a objetos e a maneira como estes se relacionam com os artefatos produzidos pela análise. Na etapa de projeto, o foco do desenvolvimento muda do domínio do problema para o domínio da solução. O projeto OO enfatiza a maneira como os requisitos serão satisfeitos pelo sistema. O projeto orientado a objetos transforma o modelo conceitual criado durante a análise em um modelo de projeto que oferece um conjunto de diretrizes para a construção do sistema de software. Os resultados concretos produzidos pelo projeto OO são:

(i) um diagrama de classes de projeto, similar ao diagrama de classes de análise, só que mais detalhado e focado no domínio da solução; e

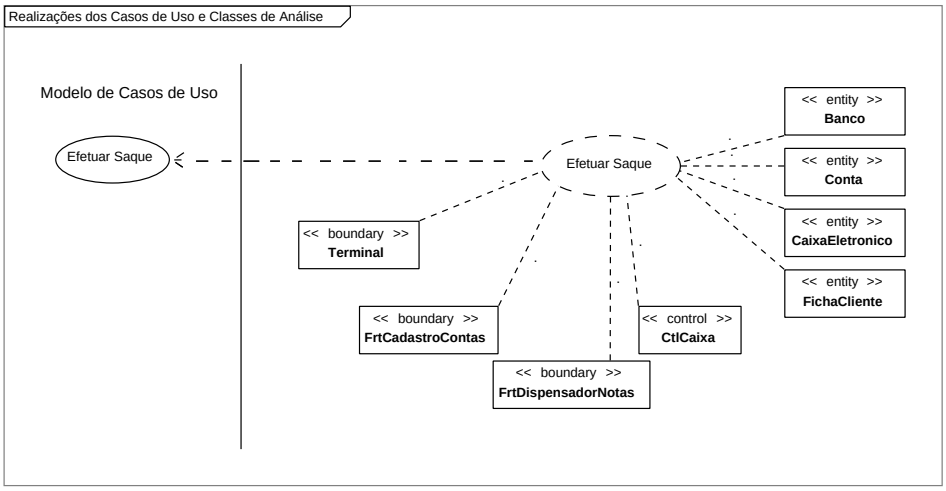


Figura 7.1: Realização do caso de uso Efetuar Saque durante a análise.

(ii) um conjunto de realizações de casos de uso, que definem a maneira como as classes de projeto colaboram a fim de oferecer o comportamento especificado pelos casos de uso.

Durante a etapa de projeto, as classes de análise, junto com o comportamento agregado a elas definido pelos diagramas de seqüência, são transformadas nas realizações dos casos de uso.

Embora as diversas metodologias existentes para projeto OO preescrevam diferentes seqüências de passos, pode-se dizer que todas dividem a fase de projeto em duas grandes etapas: (i) o projeto arquitetural; e (ii) o projeto detalhado. O projeto arquitetural, ou projeto de sistema[39], tem o objetivo de definir a estrutura geral do sistema, sua quebra em componentes, os relacionamentos entre esses componentes e a maneira como eles interagem.

O principal artefato produzido pelo projeto arquitetural é a definição da arquitetura do sistema. A arquitetura de um sistema de software reflete o primeiro conjunto de decisões que devem ser tomadas no projeto desse sistema; aquelas que são mais difíceis de mudar em etapas posteriores do desenvolvimento. Em decorrência disso, é atribuída uma importância cada vez maior à etapa de projeto arquitetural do desenvolvimento de um sistema pois ela pode ser o fator que determina a falha ou sucesso de um projeto.

Uma arquitetura de software orientada a objetos compreende um conjunto de regras, diretrizes, interfaces e convenções usadas para definir como aplicações se comunicam e interoperam umas com as outras através de técnicas orientadas a objetos [35]. A arquitetura de software está relacionada com a complexidade global do sistema e com a integração dos seus subsistemas. No desenvolvimento de uma arquitetura de software orientada a objetos, o foco está na definição da infraestrutura e da interface entre os componentes de software. O objetivo principal é o oferecimento de um ambiente confiável e de fácil manutenção onde aplicações possam cooperar e evoluir de acordo com as necessidades dos usuários. Para que esse objetivo seja alcançado, o arquiteto de software precisa definir um conjunto de abstrações que sejam efetivas no controle da complexidade do sistema, das suas mudanças, e outras necesidades. Além disso, o projetista deve preocupar-se com outras

questões, como por exemplo, particionamento do sistema em subsistemas menores, determinação do fluxo de controle interno da arquitetura, definição de camadas e protocolos de interação entre as mesmas, alocação de software/hardware, etc.

A etapa de projeto detalhado, ou projeto de classes[26], descreve as classes do sistema, seus relacionamentos e as interações entre suas instâncias em tempo de execução. No projeto detalhado, as classes de projeto (identificadas refinando-se as classes de análise) são especificadas de maneira minunciosa; atributos e estruturas de dados que devem ser mantidos em tempo de execução são definidos e operações são descritas de maneira precisa. O objetivo do projeto detalhado é produzir um modelo do sistema que possa ser implementado de maneira não-ambígua pelos programadores. Assim como foi dito para o estágio de análise, não se deve esperar acertar em todas as decisões de projeto em uma primeira tentativa. A principal vantagem de usar um processo iterativo de desenvolvimento é exatamente o fato de ser possível voltar atrás e corrigir eventuais equívocos.

Neste livro nos concentramos no projeto arquitetural. Além disso, apresentamos sucintamente um exemplo da aplicação de alguns padrões de projeto [19]. Padrões de projeto encontram-se num nível de abstração intermediário entre o projeto arquitetural e o projeto detalhado. Normalmente padrões de projeto são usados para refinar os componentes de uma arquitetura. Esse refinamento é usado como ponto de partida para as atividades do projeto detalhado. Na Seção 7.6 será utilizado o padrão State[19] para aplicar uma das extensões sugeridas na Seção 6.8 do Capítulo 6 no Sistema de Caixa Automático.

Os exemplos utilizados neste Capítulo se baseiam no Sistema de Caixa Automático descrito no Capítulo 6. O diagrama de classes de análise apresentado na Figura 6.22 é tomado como ponto de partida.

7.1 Fases do Projeto

As metodologias para projeto OO preescrevem tarefas de alto nível, semelhantes às vistas no Capítulo 3, ao falar de metodologias para análise OO. Nesta seção, será examinada de forma sucinta a organização da fase de projeto das metodologias OMT e RUP.

7.1.1 OMT

O projeto na metodologia OMT se divide em duas grandes fases: (i) o Projeto do Sistema e (ii) o Projeto de Objetos. Essas fases correspondem, respectivamente, ao projeto arquitetural e ao projeto detalhado. O projeto do sistema preocupa-se com decisões a respeito da arquitetura geral do sistema. Neste estágio, o sistema é dividido em subsistemas. Tomando o diagrama de classes de análise como guia, são realizadas as atividades mostradas na sequência de atividades a seguir:

1. Organizar o sistema em subsistemas;

2. Identificar a concorrência inerente no problema;
3. Alocar os subsistemas a processos e tarefas;
4. Escolher as estratégias para implementar banco de dados;
5. Identificar os recursos compartilhados e determinar os mecanismos de acessos a eles.

A saída desse estágio é o Documento de Projeto do Sistema que define a estrutura básica da arquitetura do sistema, bem como as decisões de estratégias de alto-nível.

O projeto de objetos preocupa-se com estruturas de dados e algoritmos necessários para implementar cada classe do modelo de objetos. Durante este estágio, os aspectos dinâmicos e funcionais são combinados e refinados. Além disso, os fluxos de controle entre os objetos são detalhados. Os passos para a realização desta fase podem ser vistos na seqüência de atividades a seguir:

1. Obter as operações para o modelo de objetos a partir dos outros modelos;
2. Projetar os algoritmos para implementar as operações;
3. Otimizar os caminhos de acesso a dados;
4. Ajustar a estrutura de classes para aumentar herança;
5. Implementar as associações;
6. Determinar arepresentação exata dos atributos;
7. Empacotar as classes e associações em módulos.

O documento resultante do estágio de projeto de objetos é chamado de Documento de Projeto, que contém um modelo de objetos, um modelo dinâmico, e um modelo funcional, todos eles com um alto nível de detalhes relacionados à implementação.

7.1.2 RUP

A etapa de projeto do RUP é composta por diversos fluxos de atividades que incluem atividades tão diversas como Projeto de Componentes de Tempo Real e Projeto de Banco de Dados. Os principais fluxos de atividades dessa etapa, porém, são a Definição da Arquitetura e o Projeto de Componentes. É interessante notar que, no RUP, a etapa de definição da arquitetura é iniciada antes da análise. Esse fato ressalta a importância crescente do projeto arquitetural nas metodologias de desenvolvimento modernas. Depois que a arquitetura é definida, a análise é iniciada e, concorrentemente, a arquitetura inicial é refinada para se adaptar à modelagem do problema.

A definição da arquitetura tão cedo no processo de desenvolvimento visa identificar, a partir das descrições dos casos de uso, um conjunto de elementos arquiteturalmente importantes que

serão usados como base para a análise. O refinamento da arquitetura realizado concorrentemente à análise OO visa tornar natural a passagem desta última para o projeto OO. As principais atividades do projeto arquitetural no RUP são apresentadas a seguir:

1. Analisar a arquitetura inicial;
2. Identificar os mecanismos de projeto;
3. Identificar os elementos de projeto;
4. Incorporar os elementos de projeto existentes;
5. Descrever a arquitetura de tempo de execução;
6. Descrever aspectos relativos a distribuição.

O principal artefato produzido pelo projeto arquitetural é um Documento de Arquitetura de Software. O nível de detalhe deste documento depende da iteração na qual o desenvolvimento se encontra. Nas iterações iniciais, ele inclue apenas diretrizes de projeto e conceitos relevantes em um alto nível de abstração. Em iterações subseqüentes, diversos elementos adicionais como divisão em camadas e tecnologias empregadas são incluídos.

No projeto de componentes, os elementos de projeto de alto nível identificados durante o projeto arquitetural são refinados até um nível de abstração que permita que programadores possam implementá-los de forma direta. O projeto de componentes compreende as seguintes atividades:

1. Projeto de classes;
2. Projeto de subsistemas;
3. Projeto de banco de dados (opcional).

Este fluxo de atividades produz como saídas (i) um diagrama de classes de projeto, (ii) um documento de projeto de subsistemas, especificando as interfaces dos subsistemas e as interações entre eles, e (iii) um modelo de dados.

7.2 Propriedades Não-Funcionais do Sistema Realizadas na Arquiteturas de Software

Nas discuções sobre arquiteturas de software, freqüentemente se faz referência às propriedades/requisitos não-funcionais do sistema que são satisfeitos pela arquitetura; em contraste, as propriedades funcionais da arquitetura são assumidas implicitamente. Uma propriedade funcional lida com algum aspecto particular da funcionalidade do sistema e está usualmente relacionada a um requisito

funcional especificado. No passado, os desenvolvedores de software concentravam-se principalmente nas propriedades funcionais dos sistemas. Entretanto, nos sistemas de software modernos, as propriedades não-funcionais estão se tornando cada vez mais importantes. Como mostrado no Capítulo 2 (Seção 2.1), uma propriedade não-funcional (ou administrativa) denota uma característica de um sistema que não é coberta pela sua descrição funcional. Uma propriedade não-funcional tipicamente está relacionada a aspectos como, por exemplo, confiabilidade, adaptabilidade, interoperabilidade, usabilidade, persistência, manutenabilidade, distribuição e segurança. As propriedades não-funcionais de uma arquitetura de software têm um grande impacto no seu desenvolvimento, manutenção e extensão. Quanto maior e mais complexo um sistema de software for e quanto maior for seu tempo de vida, maior será a importância das suas propriedades não-funcionais.

Como conseqüência, as decisões de projeto que devem ser tomadas no desenvolvimento de uma arquitetura de software geralmente são complexas e envolvem vários aspectos relacionados com as suas propriedades não-funcionais do sistema. Com o objetivo de se tomar boas decisões de projeto, é essencial que o contexto do problema seja bem definido, a fim de que seja possível identificar as melhores escolhas dentre as diversas opções de projeto possíveis. Um modo de clarificar esse contexto é aplicar a técnica de separação de interesse[1][38]. Para que os diversos aspectos envolvidos no projeto de uma arquitetura de software possam ser separados, é necessário inicialmente o estabelecimento dos limites conceituais entre eles. Cada partição obtida será responsável por resolver um subconjunto dos requisitos propostos. O arquiteto de software tem a responsabilidade de compor as diferentes partições de modo que todos os requisitos sejam conjuntamente satisfeitos.

7.3 Visões da Arquitetura de Software

Até o meio dos anos 90, devido à juventude da área de pesquisa de arquitetura de software, diversos trabalhos foram publicados nos quais representações arquiteturais falhavam em expressar as idéias de seus autores. Em 1995, Philippe Kruchten [28] publicou um artigo bastante influente em que argumentava que a causa para esse problema era a insistência dos autores em representar toda a arquitetura do sistema através de uma única abstração. Kruchten defendia a idéia de que a arquitetura de um sistema de software precisa ser representada através de quatro visões diferentes e ortogonais entre si: (i) lógica, (ii) de desenvolvimento, (iii) de processos e (iv) física. A conexão entre essas quatro visões é feita através de uma quinta visão que inclui elementos de todas elas, a visão de casos de uso. Essas cinco visões são descritas a seguir:

- **Visão lógica**: está relacionada aos requisitos funcionais do sistema. Na visão lógica, o sistema é decomposto em um conjunto de abstrações extraídas principalmente do domínio do problema, na forma de objetos ou classes que se relacionam. A visão lógica descreve tanto estrutura estática quanto estrutura dinâmica do sistema e é representada, na UML, através de diagramas de classes e de diagramas dinâmicos (seqüência, colaboração, atividades, etc.). Uma versão inicial da visão lógica da arquitetura do sistema é fornecida pelo diagrama

[1] Do Inglês *separation of concerns*.

de classes de análise e pelos diagramas de seqüência e colaboração produzidos durante a modelagem dinâmica.

- **Visão de desenvolvimento**: foca na organização do sistema em módulos de código. A visão de desenvolvimento é representada, na UML, através de diagramas de componentes contendo módulos, bibliotecas e as dependências entre esses elementos. É possível também representá-la, de forma mais simplificada, usando-se diagramas de pacotes (Seções 1.2.2, 2.9.7 e 4.8).

- **Visão de processos**: a visão de processos lida com a divisão do sistema em processos e processadores. Ela trata de aspectos relacionados à execução concorrente e distribuída do sistema e, através dela, requisitos não-funcionais como performance, tolerância a falhas e disponibilidade são realizados. A visão de processos é representada, na UML, através de diagramas dinâmicos, assim como diagramas de componentes e implantação. Esta visão também é conhecida como visão de concorrência. A visão de processos costuma ser de especial importância para a construção de sistemas com requisitos ligados a temporização e sincronização, como sistemas de controle e de tempo real.

- **Visão física**: esta visão leva em consideração principalmente os requisitos não-funcionais do sistema, em especial os que estão relacionados com sua organização física, como tolerância a falhas, escalabilidade e performance. Na visão física, também conhecida como visão de implantação, o sistema é representado como um conjunto de elementos que se comunicam e que são capazes de realizar processamento (como computadores e outros dispositios). A UML dá suporte à visão física através de diagramas de implantação.

- **Visão de casos de uso**: esta visão, também conhecida como visão de cenários, é responsável por integrar as outras quatro e descreve a funcionalidade oferecida pelo sistema aos seus atores. Esta visão apresenta informação redundante com relação às outras visões e é usada desde o início do desenvolvimento, para expressar os requisitos, até o final, quando testes de aceitação são realizados. A visão de casos de uso é representada, na UML, através de diagramas e especificações de casos de uso.

Esse modelo para a representação de arquiteturas de software foi batizado de Modelo 4+1 e se tornou praticamente um padrão na indústria[26] e na academia[5]. A Figura 7.2 aparece em quase todos os textos nos quais o modelo 4+1 é mencionado e mostra como as cinco visões se relacionam. A Tabela 7.3 mostra um resumo das visões do modelo 4+1, com os respectivos requisitos e participantes de cada uma das visões.

7.4 O Padrão Arquitetural de Camadas

No projeto OO, é necessário particionar o modelo de análise em conjuntos coesos e fracamente acoplados de classes [29]. Esses conjuntos, chamados subsistemas, incluem tanto as classes propriamente ditas quanto seus comportamentos e os relacionamentos entre elas. Eles podem estar envolvidos na realização de um mesmo conjunto de requisitos funcionais, residir dentro de um mesmo

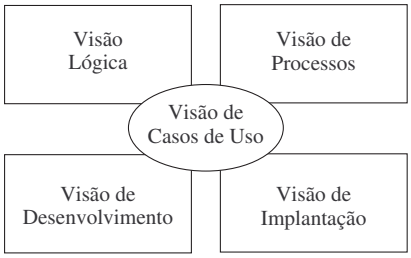


Figura 7.2: O modelo 4+1 para a representação de arquiteturas de software.

Tabela 7.1: Resumo das Visões do Modelo 4+1

Visão	Requisito Principal	Participante Principal
Visão Lógica	Funcionalidade	Usuário Final
Visão de Processo	Desempenho Escalabilidade Transferência de dados (taxa)	Integrador do Sistema
Visão de Desenvolvimento	Topologia do sistema Distribuição e instalação Comunicação	Engenheiro de Sistema
Visão Física	Gerenciamento do Software	Programador
Visão de Casos de Uso	Comportamento	Analista de sistemas Analista de testes

produto de hardware ou gerenciar um mesmo tipo de recurso[39]. Subsistemas normalmente são identificados pelos serviços que oferecem, definidos através das operações de suas interfaces. Consequentemente, uma das atividades mais importantes do projeto arquitetural é a especificação das interfaces dos subsistemas. Além disso, um subsistema pode depender de outros subsistemas, assumindo o papel de cliente, e oferecer serviços que são utilizados por outros subsistemas, assumindo o papel de servidor.

Pressman[39] apresenta uma lista de critérios de projeto com os quais subsistemas em qualquer projeto de desenvolvimento OO devem estar em conformidade:

- O subsistema deve ter uma interface bem definida através da qual ocorre toda a comunicação com o resto do sistema.
- Com exceção de um pequeno número de "classes de comunicação", as classes em um subsis-

tema devem colaborar apenas com outras classes dentro do próprio subsistema.

- O número de subsistemas deve ser pequeno, onde o significado de "pequeno" depende do projeto e deve ser julgado pelo arquiteto de software, normalmente um membro experiente do time de desenvolvimento.
- Um subsistema pode (e deve) ser particionado internamente, a fim de reduzir sua complexidade.

Quando um sistema é particionado em subsistemas, uma outra atividade ocorre concorrentemente: a divisão em camadas. Cada camada de um sistema OO contém um ou mais subsistemas e é responsável por um conjunto de funcionalidades relacionadas. Normalmente, em sistemas estruturados em camadas, subsistemas localizados em uma determinada camada n usam os serviços oferecidos pelos subsistemas da camada $n+1$ (abaixo) e oferecem serviços para os subsistemas da camada $n-1$ (acima). As camadas $n-1$ e $n+1$ não "enxergam" uma à outra diretamente[5]. Sendo assim, uma camada esconde de suas camadas superiores, os detalhes das implementações das camadas inferiores, tornando possível que a implementação de uma camada seja trocada sem que o sistema inteiro seja afetado.

Uma estruturação em camadas muito utilizada para a construção de sistemas de informação, especialmente os baseados na web, consiste em utilizar três camadas: (i) uma responsável pela interação entre o sistema e o usuário, (ii) uma outra responsável por implementar as regras de negócio da aplicação e (iii) uma terceira que lida com o armazenamento dos dados. A divisão das classes de análise entre os três tipos descritos na Seção 3.10.1 (fronteira, controle e entidade) tem suas raízes na estruturação em três camadas. Arquiteturas de software organizadas de acordo com essa divisão são conhecidas como Arquiteturas *Three-Tier* (ou "arquiteturas em três camadas"). Um exemplo de arquitetura em três camadas é apresentado na próxima seção.

7.5 Arquitetura do Sistema de Caixa Automático

Particionar o sistema do estudo de caso apresentado no Capítulo 6 em subsistemas é uma tarefa simples, tendo em vista que as classes de análise são poucas e com responsabilidades bem definidas.

As classes Terminal e FronteiraDispensadorNotas representam as classes do sistema final responsáveis pela interação entre o usuário e o sistema. Daí pode-se deduzir que, durante o projeto detalhado, essas classes se transformarão em um conjunto de classes, todas envolvidas na realização de um mesmo conjunto de requisitos. Conforme recomendado pela Seção 7.4, todas essas classes serão agrupadas em um mesmo subsistema (Interface com o Usuário).

As classes CaixaEletronico, Conta, ControladorCaixa e DadosCliente dizem respeito às regras de negócios da aplicação. ControladorCaixa representa as regras de negócios relativas às realizações dos casos de uso, como por exemplo, "transferências feitas entre contas do mesmo cliente não requerem cobrança de CPMF". Conta, por sua vez, define as regras de negócio que dizem respeito a todas as contas, por exemplo, "uma senha não pode ter menos que 5 dígitos". O mesmo para as classes

DadosCliente e CaixaEletronico. Tendo essas informações em vista, fica claro que essas classes não podem fazer parte do subsistema Interface com o Usuário, já que não lidam com a interação entre o usuário e o sistema. Conseqüentemente, um novo subsistema necessita ser definido: Regras de Negócio, do qual todas as classes relacionadas às regras de negócio da aplicação fazem parte.

A classe FronteiraCadastroContas define serviços através dos quais o sistema pode se comunicar com o Cadastro de Contas do Banco. Conforme visto no Capítulo 6, é no Cadastro de Contas do Banco que os dados sobre as contas são armazenados. Conseqüentemente, a classe FronteiraCadastroContas é o meio através do qual esses dados podem ser lidos e escritos. Tendo isso em vista e sabendo que FronteiraCadastroContas não implementa regras de negócios, pode-se definir um novo subsistema para localizar as classes responsáveis pelo acesso aos dados do sistema. Esse novo subsistema pode ser chamado de Dados. A Figura 7.3 representa graficamente o particionamento descrito. Essa figura evidencia ainda decisões tomadas em relação à distribuição física de cada um dos três módulos do sistema. Dessa forma, enquanto a Interface com o Usuário é considerado parte do módulo cliente, os demais subsistemas constituem o módulo servidor.

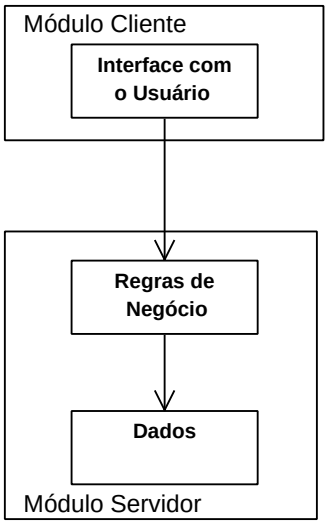


Figura 7.3: Subsistemas do Sistema de Caixa Automático – Visão Lógica.

O Sistema de Caixa Automático usa uma arquitetura em três camadas, conforme descrito na seção anterior, sendo uma delas localizada do lado cliente (Interface com o Usuário) e as demais do lado servidor (Regras de Negócio e Dados). Essa é a escolha mais natural para esse sistema, tendo em vista a sua divisão em três subsistemas representando aspectos diferentes (interface com o usuário, negócios e dados). As camadas seguem essa mesma divisão e, daqui por diante, os termos "cliente", "negócios" e "dados" serão utilizados para designar tanto as camadas correspondentes aos subsistemas de Interface com o Usuário, Regras de Negócios e Dados, respectivamente, quanto os subsistemas propriamente ditos.

Após particionar as classes de análise entre subsistemas e definir as camadas, o arquiteto deve definir as interfaces entre elas. Apesar da semelhança da nomenclatura, essas interfaces não são o

mesmo que o conceito de interface definido por Java (Seção 4.7), embora possam ser materializadas através delas, durante a implementação do sistema. Interfaces entre camadas na arquitetura estão em um nível de abstração muito mais alto do que aquelas no nível da implementação. Neste estágio, o foco está na maneira como a comunicação entre as camadas e entre os subsistemas ocorrerá, em questões como, por exemplo: (i) quais classes representarão os pontos de acesso às camadas; e (ii) quais tecnologias serão usadas para mediar a comunicação entre elas.

Em alguns casos, decisões relacionadas às tecnologias utilizadas na construção de um sistema são restrições de projeto especificadas desde a etapa de requisitos e o trabalho do arquiteto de software é simplesmente projetar sua arquitetura tendo esse fator em vista. Em outros, o arquiteto tem a liberdade para escolher e deve levar em consideração os diversos aspectos positivos e negativos de cada estilo arquitetural, tendo sempre em mente as características desejadas do sistema. No caso do Sistema de Caixa Automático, algumas restrições foram impostas desde a sua concepção. Por exemplo, a tecnologia de banco de dados empregada é aquela utilizada pelo Cadastro de Contas do Banco. O sistema deve ser capaz de se comunicar com este último e o arquiteto não tem, *a priori*, nenhuma autoridade sobre a maneira como os dados são armazenados. Por outro lado, diversos aspectos foram deixados em aberto. Por exemplo, a camada de negócios está localizada fisicamente no mesmo local que a de cliente, ou em algum outro lugar? Em outras palavras, é necessário que alguma tecnologia de distribuição seja utilizada na comunicação entre as camadas de cliente e negócios?

Serão tomadas apenas duas decisões no que concerne à interação entre as camadas. Em primeiro lugar, será considerado que a comunicação entre as camadas de cliente e negócios é distribuída. Não será feita nenhuma restrição, porém, sobre qual tecnologia deverá ser empregada. O objetivo neste ponto é apenas construir o sistema de tal maneira que seja fácil torná-lo distribuído, se isso for necessário. Para tanto, serão introduzidas duas classes adicionais, adaptadores distribuídos [2], no diagrama de classes de projeto (antigo diagrama de classes de análise). Uma delas faz parte da camada cliente (AdaptadorCliente) e a segunda é parte da camada de negócios (AdaptadorServidor). Durante o projeto detalhado, essas classes serão refinadas de acordo com a(s) tecnologia(s) de distribuição empregada(s).

A outra decisão diz respeito ao acesso aos dados. É desejável que, se os dados do Sistema de Caixa Automático forem movidos para uma nova base que utilize um sistema de gerenciamento de banco de dados diferente, o sistema possa ser adaptado a esta mudança com um mínimo de alterações. Para alcançar esse objetivo, deve-se utilizar a abordagem sugerida na Seção 4.7.3 para integração objeto-relacional. Essa abordagem é genérica o suficiente para desacoplar as camadas de negócios e dados, independentemente do tipo de banco de dados utilizado (relacional, orientado a objetos, hierárquico, etc.). Na Figura 7.4 é apresentado o mapeamento entre as classes de análise e as de projeto, tendo em vista as decisões arquiteturais tomadas. Note que essas classes de projeto serão refinadas em etapas subseqüentes dessa fase. Detalhes serão adicionados a essas classes e diversas novas classes surgirão.

Vale ressaltar que a própria organização em camadas pode mudar, devido a decisões arquiteturais tomadas *a posteriori*. Por exemplo, devido ao uso de adaptadores distribuídos, poderia ser adequado definir uma nova camada, de distribuição, responsável por encapsular toda a comunicação remota entre as camadas de cliente e de negócios. A decisão quanto a criar uma nova camada ou

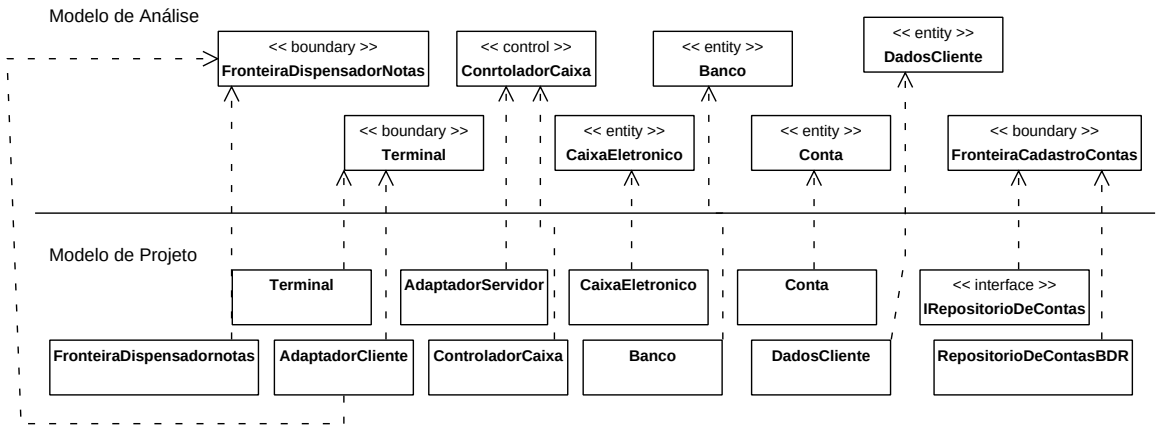


Figura 7.4: Mapeamento inicial entre as classes de análise e as classes de projeto.

não deve ser tomada levando em consideração a necessidade de modificar a implementação de um conjunto de classes responsáveis por uma mesma função. Por exemplo, é justificável criar uma camada de dados, uma vez que pode ser necessário mudar a maneira como os dados do sistema são armazenados e, idealmente, essa mudança não deve afetar as outras camadas do sistema. Por enquanto, foi resolvido deixar a arquitetura do sistema como está. A Figura 7.5 apresenta a divisão de camadas da arquitetura do sistema, levando em consideração as classes de projeto pertencentes a cada uma.

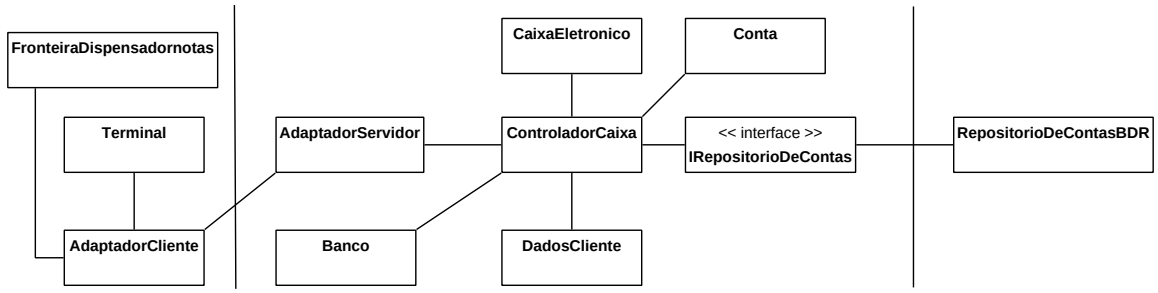


Figura 7.5: Divisão das classes de projeto entre as camadas da arquitetura.

7.6 O Padrão de Projeto *State*

O padrão de projeto *State* permite que um objeto altere seu comportamento quando seu estado interno muda. Ele usa delegação e agregação para lidar com situações em que uma classe precisa se comportar de maneiras diferentes em diferentes circunstâncias.

7.6.1 Problema

Em algumas situações, o comportamento de um objeto depende do seu estado e precisa ser modificado em tempo de execução, quando esse estado muda. Normalmente, esse problema é resolvido através de métodos que usam comandos condicionais para decidir qual comportamento deve ser adotado, dependendo do estado do objeto. Essa abordagem não é recomendada, porém, porque o código pode ser repetido em vários métodos (tanto código relativo às condições quanto ao comportamento). Além disso, um número grande de condições complexas torna o código dos métodos muito difícil de entender e manter.

7.6.2 Solução

Criar classes que encapsulam os diferentes estados do objeto e os comportamentos associados a esses estados. Dessa forma, a complexidade do objeto é quebrada em um conjunto de objetos menores e mais coesos. A Figura 7.6 mostra a estrutura geral do padrão *State*.

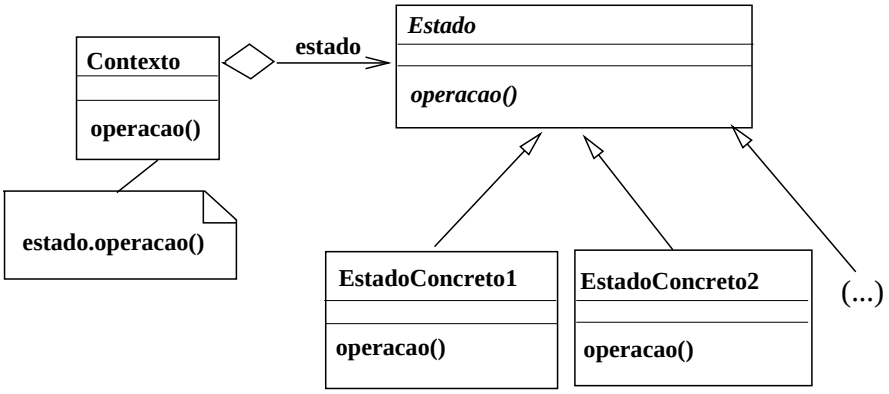


Figura 7.6: Estrutura do Padrão *State*.

Na Figura 7.6, a classe Contexto define objetos cujos estados são decompostos em diversos objetos diferentes, definidos por subtipos da classe abstrata Estado. A classe Estado define uma interface unificada para encapsular o comportamento associado a um estado particular de Contexto, que mantém uma referência para uma instância de uma subclasse concreta de Estado. Cada subclasse concreta de Estado implementa o seu respectivo comportamento associado.

Contexto delega requisições dependentes de estado para o objeto EstadoConcreto atual. Contexto pode passar a si mesmo como argumento para o objeto do tipo Estado responsável por tratar a requisição. Clientes podem configurar um contexto com objetos do tipo Estado. Uma vez que isso seja feito, porém, esses clientes passam a se comunicar apenas com o objeto do tipo Contexto e não interferem mais no seu estado diretamente.

7.6.3 Conseqüências

O padrão State tem as seguintes conseqüências:

- Localiza o comportamento dependente de estado e particiona o comportamento relativo a diferentes estados. O padrão substitui uma classe Contexto complexa por uma classe Contexto muito mais simples e um conjunto de subclasses de Estado altamente coesas.
- Torna transições de estado explícitas, ao invés de espalhá-las em comandos condicionais.
- Torna o estado de um objeto compartilhável, embora essa característica deva ser explorada somente quando esse estado consistir apenas de comportamento, sem dados (como no padrão de projeto *Flyweight* [19]).

7.6.4 Aplicação do Padrão State ao Sistema de Caixa Automático

Como exemplo, será utilizada a extensão do Sistema de Caixa Automático sugerida na Seção 6.8. Cada conta tem um *status* associado que indica se ela está ativa ou inativa. No segundo caso, saques e quaisquer outras operações que modifiquem o saldo atual da conta não podem ser realizadas. Uma abordagem alternativa de modelagem para essa situação seria criar uma variável do tipo boolean chamada ativa que teria valor false se a conta estivesse inativa e true se estivesse ativa (Figura 7.7). O problema dessa solução é que métodos da classe Conta, como efetuarSaque(), teriam que verificar o estado atual do objeto antes de executar. Esse tipo de verificação é fácil de implementar quando apenas dois estados são possíveis. Se um terceiro estado for adicionado, porém, todos os métodos dependentes do estado precisarão ser modificados para acomodar o novo comportamento. Isso nos leva a perceber que uma solução alternativa seria bastante útil. Essa solução alternativa é o padrão *State*.

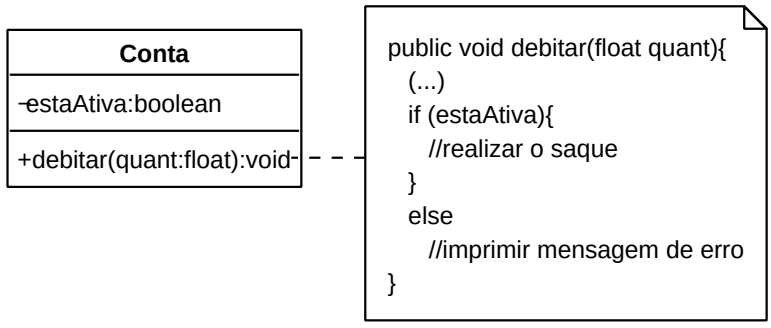


Figura 7.7: Modelagem simplificada

Para utilizar o padrão *State*, deve-se modelar o status de uma conta como um conjunto de classes separadas, cada uma implementando um estado e o comportamento associado. Desse modo

tem-se uma classe Conta que referencia objetos do tipo Estado, e duas classes correspondentes aos valores possíveis para o atributo status, Ativa e Inativa, ambas subclasses de Estado. Esta segunda opção de modelagem é apresentada na Figura 7.8.

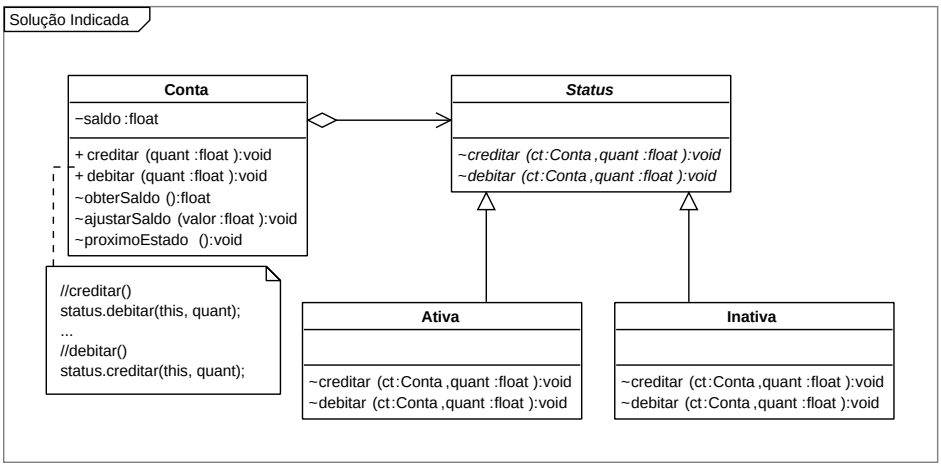


Figura 7.8: Modelagem usando o padrão State

O trecho de código seguinte apresenta uma implementação parcial para a segunda opção de modelagem.

```
// arquivo Conta.java
package conta;
public class Conta {
    double saldo = 0;
    private Status status = null;
    int numero = 0;

    public Conta(int argNum, Status argInicial, double argSld) {
        this.status = argInicial;
        this.saldo = argSld;
        this.numero = argNum;
    }
    public void debitar(double qtd) {
        this.status.debitar(this, qtd);
    }
```

```
    void proximoEstado(Status proximo) {
        this.status = proximo;
    }
}
```

A implementação do método debitar() da classe Conta chama o método debitar() do objeto correspondente ao seu status (Linha 14). Isso é feito por todas as operações que dependem do status atual da conta. Além disso, no trecho de código acima, note que o método proximoStatus() (Linhas 16 a 18) é declarado com visibilidade de pacote. Desse modo, apenas classes declaradas no mesmo pacote (conta) têm acesso a esse método.

O método proximoStatus() da classe Status, apresentado a seguir, é responsável por chamar o método proximoStatus() da classe Conta. A decisão sobre qual será o próximo status, porém, depende da implementação do padrão *State*. Em nossa implementação, as subclasses concretas de Status definem qual será o próximo status da execução. Como proximoStatus() é declarado com visibilidade protegida na classe Status, as subclasses de Status podem chamar esse método com o fim de definir o próximo estado de um objeto do tipo Conta. O trecho de código seguinte apresenta implementações para as classes Status, Inativa e Ativa.

```
// arquivo Status.java
package conta;
public abstract class Status {
     public abstract void debitar(Conta ct, double qtd);
     protected void proximoEstado(Conta ct, Status proximo) {
         ct.proximoEstado(proximo);
     }
}

// arquivo Inativa.java
package conta.status;
public class Inativa extends Status {
    public void debitar(Conta ct, double qtd) {
        System.out.println("Conta inativa. Nao e possivel realizar"
            + "o saque.");
    }
```

```
    public void creditar(Conta ct, double qtd) {
        if(qtd > 0) {
            ct.saldo = ct.saldo + qtd;
            this.proximoEstado(ct, new Ativa()); // ativa a conta ct.
        }
    }
}

// arquivo Ativa.java
package conta.status;
public class Ativa extends Status {

    public void debitar(Conta ct, double qtd) {
        if(qtd <= ct.saldo && qtd > 0) {
            ct.saldo = ct.saldo - qtd;
            if(ct.saldo == 0) { // deve desativar a conta
                this.proximoEstado(ct, new Inativa());
            }
        }
    }
   public void creditar(Conta ct, double qtd) {
        if(qtd > 0) {
            ct.saldo = ct.saldo + qtd;
        }
    }
}
```

O método debitar() é definido como abstrato na classe Status e implementado na classe Inativa (Linha 13). Essa implementação apenas emite uma mensagem de erro, informando que não se pode realizar saques em contas inativas. Uma conta torna-se inativa automaticamente quando seu saldo chega a zero, como pode ser percebido pela implementação do método debitar() da classe Ativa

(Linhas 32 e 33). Quando uma conta bloqueada recebe um depósito e seu saldo se torna maior que zero novamente, o método proximoEstado() da classe Status é invocado (Linha 20), indicando que a conta deve ser reativada.

7.7 O Padrão de Projeto *Singleton*

O padrão *singleton* busca principalmente garantir que uma classe tenha sempre uma única instância que possa ser acessada por vários clientes distintos. Além disso, o uso desse padrão faz com que a classe ofereça um ponto global de acesso a ela. Exemplos de classes implementadas por esse padrão são as classes com multiplicidade pré-definida, como por exemplo, as classes de fronteira e de controle do padrão MVC, que possuem multiplicidade "1". O conceito de multiplicidade foi apresentado no Capítulo 1 (Seção 1.3.3) e o padrão MVC foi apresentado no Capítulo 3 (Seção 3.10.1). A implementação do padrão *singleton* se baseia no ocultamento de informações, através do conceito de visibilidade visto no Capítulo 4.

7.7.1 Problema

Em algumas situações, é importante que as classes garantam a existência de apenas uma instância sua. Por exemplo, apesar de um sistema poder ter várias impressoras, deve existir apenas um spool de impressão para controlar a fila. Normalmente esse problema é resolvido utilizando variáveis globais (uma para cada instância desejada), mas além de aumentar a complexidade do sistema, essa solução é limitada no sentido de não permitir aumentar facilmente o número de instâncias possíveis. Uma solução mais adequada é tornar a classe responsável por si mesma, garantingo que haja somente uma instância. Por se tratar de um acesso centralizado, a classe pode garantir que nenhuma outra instância possa ser criada, através da interseptação de todas as requisições de criação de novas instâncias.

7.7.2 Solução

Define uma operação de instanciação que permite o acesso por parte dos clientes. Essa operação deve pertencer à classe, isto é, ser estática e é responsável por criar sua única instância da classe e passar essa referência única em cada requisição recebida.

Na Figura 7.9, a classe Singleton, que implementa o padrão *singleton*, possui um atributo de classe privado que aponta para a única instância da própria classe. Além disso, essa classe disponibiliza uma operação de classe com visibilidade pública que é responsável por controlar as requisições de instâncias da classe. Para concluir, a classe deve redefinir o construtor padrão da linguagem, mudando a visibilidade do mesmo para privada, a fim de evitar instanciações diretas da classe.

Como dito anteriormente, uma maneira utilizada para garantir a existência de uma instância

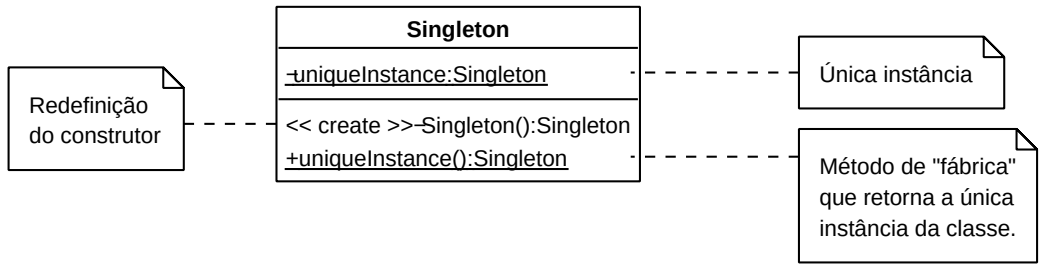


Figura 7.9: Estrutura do Padrão *Singleton*

única é ocultar as operações que instanciam a classe normalmente (os construtores), oferecendo uma operação específica com esse objetivo. Esta operação deve passar a referência de uma variável que aponte para a única instância da classe. Sendo assim, o objeto singleton deve ser instanciado apenas uma vez, antes da sua primeira utilização. Em java, por exemplo, é possível definir um atributo estático na classe Singleton que referencie um objeto do próprio tipo. De forma análoga, essa classe deve oferecer um método estático que retorne a referência da instância. Uma possível implementação em Java para a classe Singleton é mostrada a seguir:

```
    public class Singleton {
     private static Singleton instancia; // variável que referencia a única  instância da classe
     private Singleton(); // redefinição do construtor para impedir que a classe seja instanciada
     public static Singleton obterInstancia(){
        if (instancia == null) {
            instancia = new Singleton();
        }
        return instancia;
}
    }
```

Os clientes da classe Singleton a acessam exclusivamente através do método estático obterInstancia(). A variável instancia é inicializada automaticamente como null e o método estático obterInstancia() retorna o valor de instancia, inicializando essa variável uma única vez, antes da primeira requisição.

7.7.3 Conseqüências

Os principais benefícios do padrão *singleton* são [19]:

- Acesso controlado a uma instância única. Devido às classes Singleton encapsularem suas instâncias únicas, é possível ter controle estrito sobre como e quando essas referências são acessadas pelos clientes.
- Redução do espaço de nomes: O padrão Singleton apresenta vantagens em relação à utilização de variáveis globais, uma vez que evita a explosão do número dessas variáveis.
- Permite o refinamento de operações e de representação. Uma classe Singleton pode possuir subclasses. É possível configurar a aplicação com uma instância da classe que você necessita em tempo de execução, através do conceito de polimorfismo de inclusão, visto na Seção 4.5.5.
- Permite um número variável de instâncias. O padrão facilita a mudança do número de instâncias permitido pela classe. Além disso, é possível utilizar a mesma abordagem para controlar o número de instâncias que a aplicação utiliza. Em qualquer um desses casos, apenas a operação que retorna as instâncias deve ser modificada.

Referências Bibliográficas

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.

[2] Vander Alves and Paulo Borba. Distributed adapters pattern: A design pattern for object-oriented distributed applications. In *First Latin American Conference on Pattern Languages Programming, SugarLoafPLoP 2001*, October 2001. Published in UERJ Magazine: Special Issue on Software Patterns, June 2002, pages 132-142.

[3] Ken Arnold, James Gosling, and David Holmes. *The Java Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[4] Tsvi Bar-David. *Object-oriented design for C++*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.

[5] Len Bass, Paul Clements, and Rick Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[6] Gordon Blair, John Gallagher, and Javad Malik. Genericity vs. inheritance vs. delegation vs. conformance vs. .... *Journal of Object-Oriented Programming*, pages 11–17, September/October 1989.

[7] Gordon S. Blair. Basic concepts iii (types, abstract data types and polymorphism). pages 75–107, 1991.

[8] Grady Booch. *Object oriented design with applications*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1991.

[9] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[10] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture - A System of Patterns*. John-Wiley and Sons, 1996.

[11] Terry Camerlengo and Andrew Monkhouse. *SCJD Exam with J2SE 5, Second Edition*. Apress, Berkely, CA, USA, 2005.

[12] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 language definition. *SIGPLAN Not.*, 27(8):15–42, 1992.

[13] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Comput. Surv.*, 17(4):471–523, 1985.

[14] John Cheesman and John Daniels. *UML components: a simple process for specifying component-based software.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[15] Alistair Cockburn. *Writing Effective Use Cases.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[16] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming.* Number 8 in A.P.I.C. Studies in Data Processing. Academic-Press, 1972. DAH o 72:1 1.Ex.

[17] Ole-Johan Dahl. *SIMULA 67 common base language, (Norwegian Computing Center. Publication).* 1968.

[18] Martin Fowler and Kendall Scott. *UML distilled: applying the standard object modeling language.* Addison-Wesley Longman Ltd., Essex, UK, UK, 1997.

[19] Erich Gamma, Richard Helm, and Ralph Johnson. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley Professional Computing Series. Addison-Wesley, 1995. GAM e 95:1 1.Ex.

[20] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts.* John Wiley & Sons, Inc., New York, NY, USA, 1997.

[21] Adele Goldberg and David Robson. *Smalltalk-80: The Language.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1989.

[22] Samuel P. Harbison. *Modula-3.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[23] David Harel and Michal Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach.* McGraw-Hill, Inc., New York, NY, USA, 1998.

[24] Anders Hejlsberg, Scott Wiltamuth, and Peter Golde. *The C# Programming Language.* Addison-Wesley, Reading, MA, USA, 2003.

[25] Ivar Jacobson. *Object-Oriented Software Engineering: a Use Case driven Approach.* Addison–Wesley, Wokingham, England, 1995.

[26] Ivar Jacobson, Grady Booch, and James Rumbaugh. *The unified software development process.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[27] Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard. *Object-Oriented Software Engineering.* Addison-Weseley, 1994.

[28] Philippe Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[29] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design.* Prentice Hall, 1997.

[30] Henry Lieberman. Using prototypical objects to implement shared behavior in object-oriented systems. In *OOPLSA '86: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 214–223, New York, NY, USA, 1986. ACM Press.

[31] G. Masini, A. Napoli, D. Colnet, D. Leonard, and K. Tombre. *Object Oriented Languages*. Academic Press, San Diego, CA, 1991.

[32] M. D. Mcllroy. *Mass-produced software components*. Petrocelli/Charter, 1st edition, 1976.

[33] Bertrand Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[34] Bertrand Meyer. *Design by contract. In Advances in Object-Oriented Software Engineering*. Prentice Hall, Englewood Cliffs, N.J., 1991.

[35] Thomas J. Mowbray and Raphael C. Malveau. *CORBA design patterns*. John Wiley & Sons, Inc., New York, NY, USA, 1997.

[36] Glenford J. Myers. *Softwear Reliability*. John Wiley & Sons, Inc., New York, NY, USA, 1976.

[37] Glenford J. Myers. *Composite/Structured Design*. van Nostrand Reinhold Company, 1st edition, 1978.

[38] David L. Parnas. On the criteria to be used in decomposing systems into modules. pages 411–427, 2002.

[39] Roger S. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 2001.

[40] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-oriented modeling and design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1991.

[41] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen. *Object-Oriented Modeling and Design*. Prentice–Hall, Englewood Cliffs, NJ, 1991.

[42] Geri Schneider and Jason P. Winters. *Applying use cases (2nd ed.): a practical guide*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001.

[43] H.A. Simon. The architecture of complexity. In *The Sciences of the Artificial*, pages 192–229. MIT Press, Cambridge, Massachusetts, 1981.

[44] Ian Sommerville. *Software Engineering*. Addison-Wesley, 5th edition, 1995.

[45] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[46] Clemens Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[47] David Ungar and Randall B. Smith. Self: The power of simplicity. In *OOPSLA '87: Conference proceedings on Object-oriented programming systems, languages and applications*, pages 227–242, New York, NY, USA, 1987. ACM Press.

[48] Peter Wegner. Concepts and paradigms of object-oriented programming. *SIGPLAN OOPS Mess.*, 1(1):7–87, 1990.

[49] Iseult White. *Using the Booch Method: A Rational Approach*. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1994.

[50] Terry Winograd and Fernando Flores. *Understanding Computers and Cognition: A New Foundation for Design*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1987.