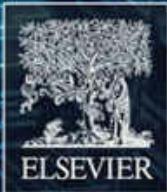


3^a EDIÇÃO
REVISTA E
ATUALIZADA

EDUARDO BEZERRA

PRINCÍPIOS
DE ANÁLISE
E PROJETO
DE SISTEMAS
COM UML

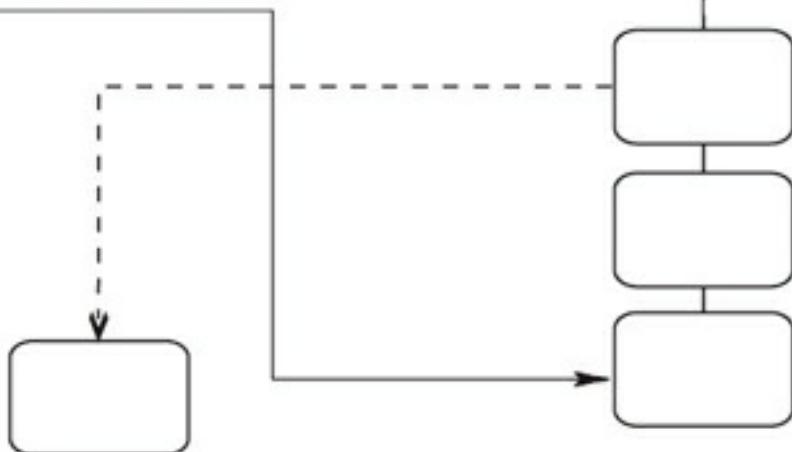


3^a EDIÇÃO
REVISTA E
ATUALIZADA

EDUARDO BEZERRA

PRINCÍPIOS
DE ANÁLISE
E PROJETO
DE SISTEMAS
COM UML

Eduardo Bezerra



PRINCÍPIOS DE ANÁLISE E PROJETO DE SISTEMAS COM UML

Consultoria Editorial

Lorenzo Ridolfi
Gerente Sênior Accenture

Sérgio Colcher
*Professor do Departamento
de Informática da PUC-Rio*

3^a edição totalmente revista e atualizada



Todos os direitos reservados e protegidos pela Lei nº 9.610, de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida, sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Copidesque: Gabriel Pereira
Editoração Eletrônica: Mojo Design
Produção digital: Freitas Bastos

Elsevier Editora Ltda.
Conhecimento sem Fronteiras
Rua Sete de Setembro, 111 – 16º andar
20050-006 – Rio de Janeiro – RJ
Rua Quintana, 753 – 8º andar
04569-011 – Brooklin – São Paulo – SP

Serviço de Atendimento ao Cliente
0800 026 53 40
atendimento1@elsevier.com

ISBN: 978-85-352-2626-3
ISBN (versão digital): 978-85-352-2627-0

Nota: Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação à nossa Central de Atendimento, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.

CIP-BRASIL. CATALOGAÇÃO-NA-FONTE
SINDICATO NACIONAL DOS EDITORES DE LIVROS, RJ

B469p
3. ed.

Bezerra, Eduardo, 1972-

Princípios de análise e projeto de sistemas com UML / Eduardo Bezerra. - [3. ed.] - Rio de Janeiro : Elsevier, 2015.
416 p. : il. ; 24 cm.

ISBN 978-85-352-2626-3

1. Métodos orientados a objetos (Computação). 2. UML (Computação). 3. Análise de sistemas. 4. Projeto de sistemas. I. Título.

14-18048

CDD: 005.117

CDU: 004.414.2

Agradecimentos

Já se passaram 12 anos desde o lançamento da primeira edição deste livro. Durante todo esse tempo, diversas pessoas me ajudaram a esclarecer meu entendimento sobre os assuntos de que trato neste livro. A todas elas, devo meus sinceros agradecimentos. Começo por agradecer aos diversos leitores das duas primeiras edições, que contribuíram com críticas e sugestões para o melhoramento da mesma. Agradeço também a meus alunos, nas diversas instituições de ensino pelas quais passei. Certamente a tarefa de professar é uma das melhores maneiras de aprender. Devo agradecimentos também a todos os meus colegas professores com os quais troquei ideias e ensinamentos sobre o problema da modelagem de sistemas de software: Ronaldo Goldschmidt, Carmem de Queiróz, Jorge Soares, Ismael Humberto, Leandro Chernicharo, Ricardo Choren, dentre outros. Obrigado também à equipe editorial da Elsevier, por toda a paciência e profissionalismo durante o tempo em que trabalhamos na produção desta edição. Finalmente, e não menos importante, agradeço a toda a minha família, pelo carinho e incentivo. Em especial, agradeço a meus irmãos, José, Edmar, Emanuel e Helton, à minha querida esposa Aline, ao meu filho, Felipe, e ao Janu.

Prefácio

1 Visão geral

- 1.1 Modelagem de sistemas de software
- 1.2 O paradigma da orientação a objetos
 - 1.2.1 Classes e objetos
 - 1.2.2 Operação, mensagem e estado
 - 1.2.3 O papel da abstração na orientação a objetos
- 1.3 Evolução histórica da modelagem de sistemas
- 1.4 A Linguagem de Modelagem Unificada (UML)
 - 1.4.1 Visões de um sistema
 - 1.4.2 Diagramas da UML

2 O processo de desenvolvimento de software

- 2.1 Atividades típicas de um processo de desenvolvimento
 - 2.1.1 Levantamento de requisitos
 - 2.1.2 Análise
 - 2.1.3 Projeto (desenho)
 - 2.1.4 Implementação
 - 2.1.5 Testes
 - 2.1.6 Implantação
- 2.2 O componente humano (participantes do processo)
 - 2.2.1 Gerentes de projeto
 - 2.2.2 Analistas
 - 2.2.3 Projetistas
 - 2.2.4 Arquitetos de software
 - 2.2.5 Programadores
 - 2.2.6 Especialistas do domínio
 - 2.2.7 Avaliadores de qualidade
- 2.3 Modelos de ciclo de vida
 - 2.3.1 O modelo de ciclo de vida em cascata
 - 2.3.2 O modelo de ciclo de vida iterativo e incremental
- 2.4 Utilização da UML no processo iterativo e incremental
- 2.5 Prototipagem
- 2.6 Ferramentas CASE

3 Mecanismos gerais

- 3.1 Estereótipos
- 3.2 Notas explicativas
- 3.3 Etiquetas valoradas (*tagged values*)
- 3.4 Restrições
- 3.5 Pacotes
- 3.6 OCL

4 Modelagem de casos de uso

- 4.1 Modelo de casos de uso
 - 4.1.1 Casos de uso
 - 4.1.2 Atores
 - 4.1.3 Relacionamentos
- 4.2 Diagrama de casos de uso
- 4.3 Identificação dos elementos do MCU
 - 4.3.1 Identificação de atores
 - 4.3.2 Identificação de casos de uso
- 4.4 Construção do modelo de casos de uso
 - 4.4.1 Construção do diagrama de casos de uso
 - 4.4.2 Documentação dos atores
 - 4.4.3 Documentação dos casos de uso
- 4.5 Documentação suplementar ao MCU
 - 4.5.1 Regras do negócio
 - 4.5.2 Requisitos de desempenho
 - 4.5.3 Requisitos de interface gráfica
- 4.6 O MCU em um processo de desenvolvimento iterativo
 - 4.6.1 O MCU nas atividades de análise e projeto
 - 4.6.2 O MCU e outras atividades do desenvolvimento
- 4.7 Estudo de caso
 - 4.7.1 Descrição da situação
 - 4.7.2 Regras do negócio
 - 4.7.3 Documentação do MCU

5 Modelagem de classes de análise

- 5.1 Estágios do modelo de classes
- 5.2 Diagrama de classes
 - 5.2.1 Classes
 - 5.2.2 Associações
 - 5.2.3 Generalizações e especializações
- 5.3 Diagrama de objetos
- 5.4 Técnicas para identificação de classes

- 5.4.1 Análise textual de Abbott
- 5.4.2 Análise dos casos de uso
- 5.4.3 Técnicas baseadas em responsabilidades
- 5.4.4 Padrões de análise
- 5.4.5 Outras técnicas de identificação
- 5.4.6 Discussão
- 5.5 Construção do modelo de classes
 - 5.5.1 Definição de propriedades
 - 5.5.2 Definição de associações
 - 5.5.3 Organização da documentação
- 5.6 Modelo de classes no processo de desenvolvimento
- 5.7 Estudo de caso
 - 5.7.1 Análise do caso de uso: Fornecer Grade de Disponibilidades
 - 5.7.2 Análise do caso de uso: Realizar Inscrição
 - 5.7.3 Análise do caso de uso: Lançar Avaliações
 - 5.7.4 Análise das regras do negócio
 - 5.7.5 Documentação das responsabilidades
 - 5.7.6 Glossário de conceitos

6 Passagem da análise para o projeto

- 6.1 Detalhamento dos aspectos dinâmicos
- 6.2 Refinamento dos aspectos estáticos e estruturais
- 6.3 Projeto da arquitetura
- 6.4 Persistência de objetos
- 6.5 Projeto de interface gráfica com o usuário
- 6.6 Projeto de algoritmos
- 6.7 Padrões de software

7 Modelagem de interações

- 7.1 Elementos da modelagem de interações
 - 7.1.1 Mensagens
 - 7.1.2 Atores
 - 7.1.3 Objetos
 - 7.1.4 Classes
 - 7.1.5 Coleções de objetos
- 7.2 Diagrama de sequência
 - 7.2.1 Linhas de vida
 - 7.2.2 Mensagens
 - 7.2.3 Ocorrências de execução
 - 7.2.4 Criação e destruição de objetos
- 7.3 Diagrama de comunicação

- 7.4 Modularização de interações
 - 7.4.1 Quadros
 - 7.4.2 Diagrama de visão geral da interação
- 7.5 Construção do modelo de interações
 - 7.5.1 Responsabilidades e mensagens
 - 7.5.2 Coesão e acoplamento
 - 7.5.3 Encapsulamento
 - 7.5.4 Procedimento de construção do modelo de interações
- 7.6 Modelo de interações em um processo iterativo
- 7.7 Estudo de caso
 - 7.7.1 Operações de sistema
 - 7.7.2 Observações gerais
 - 7.7.3 Modelos de interações
 - 7.7.4 Visão geral das interações em um caso de uso

8 Modelagem de classes de projeto

- 8.1 Reúso: padrões, frameworks, bibliotecas, componentes
- 8.2 Especificação de atributos
 - 8.2.1 Notação da UML para atributos
- 8.3 Especificação de operações
 - 8.3.1 Notação da UML para operações
 - 8.3.2 Dicas práticas
 - 8.3.3 Projeto por contrato
 - 8.3.4 Operações de criação e destruição de objetos
 - 8.3.5 Seletores e modificadores
 - 8.3.6 Outras operações típicas
- 8.4 Especificação de associações
 - 8.4.1 O conceito de dependência
 - 8.4.2 Transformação de associações em dependências
 - 8.4.3 Navegabilidade de associações
 - 8.4.4 Implementação de associações
- 8.5 Herança
 - 8.5.1 Tipos de herança
 - 8.5.2 Classes abstratas
 - 8.5.3 Operações polimórficas
 - 8.5.4 Interfaces
 - 8.5.5 Acoplamentos concreto e abstrato
 - 8.5.6 Reúso por delegação
 - 8.5.7 Classificação dinâmica
- 8.6 Padrões de projeto
 - 8.6.1 Composite

- 8.6.2 Observer
- 8.6.3 Strategy
- 8.6.4 Factory Method
- 8.6.5 Mediator
- 8.6.6 Façade
- 8.7 Modelo de classes de projeto em um processo iterativo
- 8.8 Estudo de caso

9 Modelagem de estados

- 9.1 Diagrama de transição de estado
 - 9.1.1 Estados
 - 9.1.2 Transições
 - 9.1.3 Eventos
 - 9.1.4 Condição de guarda
 - 9.1.5 Ações
 - 9.1.6 Atividades
 - 9.1.7 Ponto de junção
 - 9.1.8 Cláusulas entry, exit e do
 - 9.1.9 Transições internas
 - 9.1.10 Exemplo
 - 9.1.11 Estados aninhados
 - 9.1.12 Estados concorrentes
- 9.2 Identificação dos elementos de um diagrama de estados
- 9.3 Construção de diagramas de transições de estados
- 9.4 Modelagem de estados no processo de desenvolvimento
- 9.5 Estudo de caso

10 Modelagem de atividades

- 10.1 Diagrama de atividade
 - 10.1.1 Fluxo de controle sequencial
 - 10.1.2 Fluxo de controle paralelo
 - 10.1.2.1 Raias de natação
- 10.2 Diagrama de atividade no processo de desenvolvimento iterativo
 - 10.2.1 Modelagem dos processos do negócio
 - 10.2.2 Modelagem da lógica de um caso de uso
 - 10.2.3 Modelagem da lógica de uma operação complexa
- 10.3 Estudo de caso

11 Arquitetura do sistema

- 11.1 Arquitetura lógica
 - 11.1.1 Conceito de camada de software

- 11.1.2 Camadas típicas de um sistema de informação
- 11.1.3 O padrão MVC e sua relação com a arquitetura lógica
- 11.2 Arquitetura física
 - 11.2.1 Alocação de camadas lógicas aos nós de processamento
 - 11.2.2 Alocação de componentes aos nós de processamento
 - 11.2.3 Padrões e tecnologias para distribuição de objetos
- 11.3 Projeto da arquitetura no processo de desenvolvimento

12 Mapeamento de objetos para o modelo relacional

- 12.1 Projeto de banco de dados
 - 12.1.1 Conceitos do modelo de dados relacional
 - 12.1.2 Mapeamento de objetos para o modelo relacional
 - 12.1.3 Classes e seus atributos
 - 12.1.4 Associações
 - 12.1.5 Agregações e composições
 - 12.1.6 Associações reflexivas
 - 12.1.7 Associações ternárias
 - 12.1.8 Classes associativas
 - 12.1.9 Generalização
- 12.2 Construção da camada de persistência
 - 12.2.1 Acesso direto ao banco de dados
 - 12.2.2 Uso de um SGBDOO ou de um SGBDOR
 - 12.2.3 Padrão DAO
 - 12.2.4 Frameworks ORM

Referências

S eja bem-vindo à terceira edição de *Princípios de Análise e Projeto de Sistemas com UML*. Este livro é uma introdução aos conceitos fundamentais necessários para se realizar a análise e o projeto de sistemas de software orientados a objetos com o uso da Linguagem de Modelagem Unificada (UML). Desde o lançamento da 1^a edição desta obra, já existiam bons livros disponíveis aqui no Brasil discutindo a modelagem de sistemas orientados a objetos com UML. No entanto, uma razão que me levou a escrever esta obra foi o fato de alguns desses livros darem uma ênfase maior à descrição da UML em si.

De fato, a UML define uma notação padrão que pode ser utilizada por desenvolvedores de software orientado a objetos. Sem dúvida o domínio dessa notação é importante para qualquer desenvolvedor que queira aproveitar todas as capacidades que a UML fornece. Mas, igualmente importante, em especial para iniciantes no desenvolvimento de software, é o entendimento de como aplicar a notação da UML na modelagem. É esse enfoque que procurei dar neste livro. Em vista disso, esta obra *não* fornece uma referência completa sobre a notação definida pela UML. Em vez disso, ela descreve uma parte dessa notação e também como realizar a análise e o projeto de sistemas orientados a objetos através de parte da notação mais utilizada.

Durante todo o livro, exemplos são utilizados para demonstrar a aplicação da UML em situações práticas de modelagem. Ao fim de cada capítulo são fornecidos exercícios para testar o conteúdo apreendido pelo leitor. Além disso, um estudo de caso, o Sistema de Controle Acadêmico (SCA), é desenvolvido para os principais tópicos abordados com o objetivo de exemplificar a aplicação dos procedimentos e dicas de modelagem que são apresentadas em cada capítulo.

Público-alvo

Este livro é destinado a estudantes de cursos técnicos, de graduação ou pósgraduação em informática, computação, sistemas de informação ou engenharia de software que devem cursar uma ou mais disciplinas de análise e projeto orientados a objetos. Esta obra também pode ser utilizada como guia por estudantes no desenvolvimento de seus projetos finais de curso. Profissionais que desenvolvem sistemas segundo outros paradigmas (que não o orientado a objetos) também podem encontrar neste livro uma boa iniciação aos conceitos da orientação a objetos e da sua aplicação à modelagem de sistemas de software. Em todos os casos, o livro pode servir como uma fonte de referência e dicas práticas sobre a aplicação da UML e de outras técnicas no desenvolvimento de um sistema de software orientado a objetos.

O conhecimento de alguma linguagem de programação orientada a objetos (p. ex., Java, C#, C++ etc.) é desejável (mas não obrigatório) para o bom entendimento dos assuntos tratados neste livro. Mais especificamente, este livro fornece diversos exemplos de trechos de código-fonte em linguagem Java. Entretanto, esses exemplos devem ser facilmente entendidos por profissionais familiarizados com outras linguagens orientadas a objetos.

Organização dos capítulos

O Capítulo 1 apresenta uma breve introdução à utilização do paradigma da orientação a objetos e da

UML. O objetivo deste capítulo é fornecer uma visão geral sobre a análise e o projeto de sistemas de software sob o ponto de vista de orientação a objetos. Os principais conceitos do paradigma da orientação a objetos são introduzidos neste capítulo.

O [Capítulo 2](#) descreve as principais atividades constituintes de um processo de desenvolvimento de software. Também descrevemos os principais profissionais envolvidos nesse processo, juntamente com suas respectivas atribuições. O processo de desenvolvimento em cascata é apresentado com o objetivo de motivar o surgimento do processo incremental e evolutivo. Em seguida, este último é também descrito e apresentado como a forma atual de se desenvolver sistemas orientados a objetos. Na maioria dos capítulos seguintes são feitas alusões à utilização da UML em um processo de desenvolvimento incremental e evolutivo.

O [Capítulo 3](#), o menor deste livro, é apenas uma apresentação dos mecanismos de uso geral da UML. Essa apresentação se faz necessária em virtude de esses mecanismos serem utilizáveis em diversos diagramas da UML. Nos capítulos posteriores, fazemos uso e estendemos os conceitos introdutórios apresentados neste capítulo.

No [Capítulo 4](#), apresentamos o modelo de casos de uso e os diversos elementos do diagrama de casos de uso da UML. Além disso, são fornecidas diversas dicas práticas que podem ser utilizadas na construção desse modelo. Esse capítulo também enfatiza o modelo de casos de uso como um ponto central de um processo de desenvolvimento que utilize a UML como linguagem de modelagem.

O [Capítulo 5](#) descreve a construção do modelo de classes de análise de um sistema de software orientado a objetos (SSOO). Os principais elementos de notação definidos pela UML para a construção do diagrama de classes são descritos. Também é apresentado o conceito de responsabilidade de um objeto. Descrevemos, além disso, diversas técnicas úteis na identificação das classes iniciais de um SSOO, como a análise textual de Abbot, a análise de casos de uso e o uso de padrões de análise. Nesta 3^a edição, adiciono a este capítulo uma pequena introdução aos padrões táticos do DDD (*Domain Driven Design*) no contexto de identificação de classes do domínio.

O [Capítulo 6](#) serve como uma apresentação do conteúdo dos capítulos que o seguem. A partir desse capítulo, a descrição das atividades de projeto começa a tomar o lugar da descrição das atividades de análise.

A modelagem de interações entre objetos em um SSOO é discutida no [Capítulo 7](#). Nesse capítulo, apresento a ideia de que as construções do modelo de classe e do modelo de interações são interdependentes: a construção de um modelo fornece informações para a construção do outro e vice-versa. Seguindo a filosofia das edições anteriores, não me preocupei em apresentar todos os elementos de notação, mas apenas os que, na minha visão, são os mais importantes e relevantes em situações práticas de modelagem. Nessa 3^a edição, estendi o conteúdo desse capítulo com a descrição de boas práticas e princípios de projeto relevantes para a construção correta do modelo de interações.

O [Capítulo 8](#) retoma a discussão sobre o modelo de classes, agora com um enfoque nas características de modelagem referentes à fase de projeto. Conceitos fundamentais ao projeto de um SSOO são apresentados: classe abstrata, interface, polimorfismo, tipos de acoplamento, projeto por contrato etc. Na 2^a edição, apresentei uma pequena introdução a um assunto um tanto avançado, mas cada vez mais sedimentado no desenvolvimento de um SSOO: padrões de projeto. Essa descrição sobre padrões de projeto continua nesse capítulo, mas é aprofundada em outras partes do livro.

O [Capítulo 9](#) descreve a sintaxe, a semântica e a construção dos diagramas de transições de estados.

O [Capítulo 10](#) finaliza a apresentação dos diagramas da UML relacionados à parte

comportamental do sistema. Esse capítulo descreve os diagramas de atividades.

O [Capítulo 11](#) faz uma introdução aos conceitos relacionados à arquitetura de um sistema de SSOO. Termos como subsistema, componente e camada são descritos. Outros diagramas da UML são apresentados: o de componentes, o de pacotes e o de implantação. Nesta 3^a edição, esse capítulo foi estendido com um maior detalhamento acerca das camadas tipicamente encontradas na arquitetura em um sistema de informação.

Finalmente, o [Capítulo 12](#) descreve alternativas de representação de objetos em um mecanismo de armazenamento persistente como um sistema de gerência de bancos de dados relacional. É feita também uma introdução a questões relacionadas à implementação de uma camada de persistência em um SSOO.

Recursos na web

Como informação suplementar à contida neste livro, é fornecido um site na própria editora Elsevier. Acesse a página da Editora (www.elsevier.com.br). Nesse endereço, o leitor pode obter informações e material relacionado ao livro. Entre os recursos que podem ser encontrados no site, estão os seguintes:

- *Soluções de alguns dos exercícios propostos no livro.* O leitor pode encontrar diversos exercícios resolvidos no material disponibilizado no site da editora.
- *Complementos ao estudo de caso apresentado no livro.* O estudo de caso que desenvolvo nesse livro é denominado *Sistema de Controle Acadêmico* (SCA). No final de alguns capítulos, forneço diversos exemplos de modelagem no contexto do SCA. Um problema que surge é como continuar e complementar esses exemplos. Uma solução que adoto a partir dessa segunda edição é utilizar a Internet como fornecedora de novos materiais acerca deste estudo de caso.
- *Outras fontes de informação.* O material disponível no site da editora contém também endereços para outras fontes interessantes sobre modelagem de sistemas de software orientados a objetos. Seguindo a natureza dinâmica da Internet, o conteúdo do site será modificado de tempos em tempos. O leitor também pode utilizar esse site para entrar em contato comigo, com o objetivo de trocar ideias sobre o livro.

Convite ao leitor

Finalmente, convido o leitor a prosseguir pelo restante desta obra. Espero que as informações contidas neste livro o ajudem de alguma forma, e que a leitura seja a mais agradável possível. Tentei dar o meu melhor para produzir um texto cuja leitura seja aprazível e didática. Entretanto, pelo fato de a produção de um livro ser uma tarefa bastante complexa, tenho consciência de que erros e inconsistências ainda se escondem por entre as linhas que o compõem. Para os que quiserem entrar em contato comigo para trocar ideias e fornecer críticas e sugestões, fiquem à vontade para enviar uma mensagem.

Eduardo Bezerra

Rio de Janeiro

ebezerra@cefet-rj.br

18 de agosto de 2014

Visão geral

Coisas simples devem ser simples, e coisas complexas devem ser possíveis.
— ALAN KAY

No decorrer da história, diversos tipos de bens serviram como base para o desenvolvimento da economia. Propriedade, mão de obra, máquinas e capital são exemplos desses bens. Atualmente, está surgindo um novo tipo de bem econômico: a informação. Nos dias de hoje, a empresa que dispõe de mais informações sobre seu processo de negócio está em vantagem em relação às suas competidoras.

Há um ditado que diz que “a necessidade é a mãe das invenções”. Em consequência do crescimento da importância da informação, surgiu a necessidade de gerenciar informações de uma forma adequada e eficiente e, a partir dela, apareceram os denominados *sistemas de informações*.

Um sistema de informações é uma combinação de pessoas, dados, processos, interfaces, redes de comunicação e tecnologia que interagem com o objetivo de dar suporte e melhorar o processo de negócio de uma organização empresarial com relação às informações que nela fluem. Considerando o caráter estratégico da informação nos dias de hoje, pode-se dizer também que os sistemas de informações têm o objetivo de prover vantagens para uma organização do ponto de vista competitivo.

O objetivo principal e final da construção de um sistema de informações é a *adição de valor* à empresa ou organização na qual esse sistema será utilizado. O termo “adição de valor” implica que a produtividade nos processos da empresa na qual o sistema de informações será utilizado deve aumentar de modo significativo, de forma que compense os recursos utilizados na construção do sistema. Para que um sistema de informações adicione valor a uma organização, tal sistema deve ser economicamente justificável.

O desenvolvimento de um sistema de informações é uma tarefa das mais complexas. Um dos seus componentes é denominado *sistema de software*. Esse componente comprehende os módulos funcionais computadorizados que interagem entre si para proporcionar ao(s) usuário(s) do sistema a automatização de diversas tarefas.

1.1 Modelagem de sistemas de software

Uma característica intrínseca de sistemas de software é a complexidade de seu desenvolvimento, que aumenta à medida que o tamanho do sistema cresce. Para se ter uma ideia da complexidade envolvida na construção de alguns sistemas, pense no tempo e nos recursos materiais necessários para se construir uma casa de cachorro. Para essa tarefa, provavelmente tudo de que se precisa é de algumas ripas de madeira, alguns pregos, uma caixa de ferramentas e certa dose de amor por seu cachorro. Depois de alguns dias, a casa estaria pronta. O que dizer da construção de uma casa para sua família? Decerto tal empreitada não seria realizada tão facilmente. O tempo e os recursos

necessários seriam uma ou duas ordens de grandeza maiores do que o necessário para a construção da casa de cachorro. O que dizer, então, da construção de um edifício? Certamente, para se construir habitações mais complexas (casas e edifícios), algum planejamento adicional é necessário. Engenheiros e arquitetos constroem plantas dos diversos elementos da habitação antes do início da construção propriamente dita. Na terminologia da construção civil, plantas hidráulicas, elétricas, de fundação etc. são projetadas e devem manter consistência entre si. Provavelmente, uma equipe de profissionais estaria envolvida na construção, e aos membros dessa equipe seriam delegadas diversas tarefas, no tempo adequado para cada uma delas.

Na construção de sistemas de software, assim como na de sistemas habitacionais, há também uma graduação de complexidade. Para a construção de sistemas de software mais complexos, é igualmente necessário um planejamento inicial. O equivalente ao projeto das plantas da engenharia civil também deve ser realizado. Essa necessidade leva ao conceito de *modelo*, tão importante no desenvolvimento de sistemas. De uma perspectiva mais ampla, um modelo pode ser visto como uma representação idealizada de um sistema a ser construído. Maquetes de edifícios e de aviões e plantas de circuitos eletrônicos são apenas alguns exemplos de modelos. São várias as razões para se utilizar modelos na construção de sistemas. Segue-se uma lista de algumas delas.

- 1. Gerenciamento da complexidade:** um dos principais motivos de utilizar modelos é que há limitações no ser humano em como lidar com a complexidade. Pode haver diversos modelos de um mesmo sistema, cada qual descrevendo uma perspectiva do sistema a ser construído. Por exemplo, um avião pode ter um modelo para representar sua parte elétrica, outro para representar sua parte aerodinâmica etc. Por meio de modelos de um sistema, os indivíduos envolvidos no seu desenvolvimento podem fazer estudos e prever comportamentos do sistema em desenvolvimento. Como cada modelo representa uma perspectiva do sistema, detalhes irrelevantes que podem dificultar o entendimento do sistema podem ser ignorados por um momento estudando-se separadamente cada um dos modelos. Além disso, modelos se baseiam no denominado *Princípio da Abstração* (ver [Seção 1.2.3](#)), segundo o qual só as características relevantes à resolução de um problema devem ser consideradas. Modelos revelam as características essenciais de um sistema; detalhes não relevantes e que só aumentariam a complexidade do problema podem ser ignorados.
- 2. Comunicação entre as pessoas envolvidas:** sem dúvida o desenvolvimento de um sistema envolve a execução de uma quantidade significativa de atividades. Essas atividades se traduzem em informações sobre o sistema em desenvolvimento. Grande parte dessas informações corresponde aos modelos criados para representar o sistema. Nesse sentido, os modelos de um sistema servem também para promover a difusão de informações relativas ao sistema entre os indivíduos envolvidos em sua construção. Além disso, diferentes expectativas em relação ao sistema geralmente surgem durante a construção dos seus modelos, já que eles servem como um ponto de referência comum.
- 3. Redução dos custos no desenvolvimento:** no desenvolvimento de sistemas, seres humanos estão invariavelmente sujeitos a cometer erros, que podem ser tanto individuais quanto de comunicação entre os membros da equipe. Certamente a correção desses erros é menos custosa quando detectada e realizada ainda no(s) modelo(s) do sistema (p. ex., é muito mais fácil corrigir uma maquete do que pôr abaixo parte de um edifício). Lembre-se: modelos de sistemas são mais baratos de construir do que sistemas. Consequentemente, erros identificados sobre modelos têm um impacto menos desastroso.

4. Previsão do comportamento futuro do sistema: o comportamento do sistema pode ser discutido mediante uma análise dos seus modelos. Os modelos servem como um “laboratório”, em que diferentes soluções para um problema relacionado à construção do sistema podem ser experimentadas.

Outra questão importante é como os modelos de sistemas de software se organizam. Em construções civis, frequentemente há profissionais para analisar as plantas da construção. A partir delas, que podem ser vistas como modelos, os profissionais tomam decisões sobre o andamento da obra. Modelos de sistemas de software não são muito diferentes dos modelos de sistemas da construção civil. Nos próximos capítulos, apresentaremos diversos modelos cujos componentes são desenhos gráficos que seguem algum padrão lógico. Esses desenhos são normalmente denominados *diagramas*. Um diagrama é uma apresentação de uma coleção de *elementos gráficos* que possuem um significado predefinido.

Talvez o fato de os modelos serem representados por diagramas possa ser explicado pelo ditado: “uma imagem vale por mil palavras”. Graças aos desenhos gráficos que modelam o sistema, os desenvolvedores têm uma representação concisa do sistema. No entanto, modelos de sistemas de software também são compostos de informações textuais. Embora um diagrama consiga expressar diversas informações de forma gráfica, em diversos momentos há a necessidade de adicionar informações na forma de texto, com o objetivo de explicar ou definir certas partes desse diagrama. Dado um modelo de uma das perspectivas de um sistema, diz-se que o seu diagrama e a informação textual associada a ele formam a *documentação* desse modelo.

A modelagem de sistemas de software consiste na utilização de notações gráficas e textuais com o objetivo de construir modelos que representam as partes essenciais de um sistema, considerando-se várias perspectivas diferentes e complementares.

1.2 O paradigma da orientação a objetos

Indispensável ao desenvolvimento atual de sistemas de software é o *paradigma da orientação a objetos*. Esta seção descreve o que esse termo significa e justifica por que a orientação a objetos é importante para a modelagem de sistemas. Pode-se começar pela definição da palavra *paradigma*. Essa palavra possui diversos significados, mas o que mais se aproxima do sentido aqui utilizado encontra-se no dicionário Aurélio Século XXI: *paradigma*. [Do gr. *parádeigma*, pelo lat. tard. *paradigma*.] S. m. Termo com o qual Thomas Kuhn designou as realizações científicas (p. ex., a dinâmica de Newton ou a química de Lavoisier) que geram modelos que, por período mais ou menos longo e de modo mais ou menos explícito, orientam o desenvolvimento posterior das pesquisas exclusivamente na busca da solução para os problemas por elas suscitados.

Para o leitor que ainda não se sentiu satisfeito com essa definição, temos aqui outra, mais concisa e apropriada ao contexto deste livro: *um paradigma é uma forma de abordar um problema*. Como exemplo, considere a famosa história da maçã caindo sobre a cabeça de Isaac Newton, citado na definição anterior.¹ Em vez de pensar que somente a maçã estava caindo sobre a Terra, Newton também considerou a hipótese de o próprio planeta também estar caindo sobre a maçã! Essa outra maneira de abordar o problema pode ser vista como um paradigma.

Pode-se dizer, então, que o termo “paradigma da orientação a objetos” é uma forma de abordar um problema. Há alguns anos, Alan Kay, um dos pais do paradigma da orientação a objetos, formulou a chamada “analogia biológica”. Por meio dela, ele imaginou um sistema de software que funcionasse como um ser vivo. Nesse sistema, cada “célula” interagiria com outras células através do envio de mensagens com o objetivo realizar um objetivo comum. Além disso, cada célula se comportaria como uma unidade autônoma.

De uma forma mais geral, Kay pensou em como construir um sistema de software a partir de agentes autônomos que interagem entre si. Ele estabeleceu então os seguintes princípios da orientação a objetos:

1. Qualquer coisa é um objeto.
2. Objetos realizam tarefas por meio da requisição de serviços a outros objetos.
3. Cada objeto pertence a uma determinada *classe*. Uma classe agrupa objetos similares.
4. A classe é um repositório para comportamento associado ao objeto.
5. Classes são organizadas em hierarquias.

Vamos ilustrar esses princípios com a seguinte história: suponha que alguém queira comprar uma pizza. Chame este alguém de João. Ele está muito ocupado em casa e resolve pedir sua pizza por telefone. João liga para a pizzaria e faz o pedido, informando ao atendente (digamos, José) seu nome, as características da pizza desejada e o seu endereço. José, que só tem a função de atendente, comunica a Maria, funcionária da pizzaria e responsável por preparar as pizzas, qual pizza deve ser feita. Quando Maria termina de fazer a pizza, José chama Antônio, o entregador. Finalmente, João recebe a pizza desejada das mãos de Antônio meia hora depois de tê-la pedido.

Pode-se observar que o objetivo de João foi atingido graças à colaboração de diversos agentes, os funcionário da pizzaria. Na terminologia do paradigma da orientação a objetos, esses objetos são denominados *objetos*. Há diversos objetos na história (1º princípio): João, Maria, José, Antônio. Todos colaboram com uma parte e o objetivo é alcançado quando todos trabalham juntos (2º princípio). Além disso, o comportamento esperado de Antônio é o mesmo esperado de qualquer entregador. Diz-se que Antônio é um objeto da classe Entregador (3º princípio). Um comportamento comum a todo entregador, não somente a Antônio, é o de entregar a mercadoria no endereço especificado (4º princípio). Finalmente, José, o atendente, é também um ser humano, mamífero, animal etc. (5º princípio).

Mas o que o paradigma da orientação a objetos tem a ver com a modelagem de sistemas? Antes da orientação a objetos, outro paradigma era utilizado na modelagem de sistemas: o paradigma estruturado. Nesse paradigma, os elementos são dados e processos. Os processos agem sobre os dados para que um objetivo seja alcançado. Por outro lado, no paradigma da orientação a objetos há um elemento, o objeto, uma unidade autônoma que contém seus próprios dados que são manipulados pelos processos definidos para o objeto e que interage com outros objetos para alcançar um objetivo. É o paradigma da orientação a objetos que os seres humanos utilizam no cotidiano para a resolução de problemas. Uma pessoa atende a mensagens (requisições) para realizar um serviço; essa mesma pessoa envia mensagens a outras para que elas realizem serviços. Por que não aplicar essa mesma forma de pensar à modelagem de sistemas?

de agentes interconectados chamados *objetos*. Cada objeto é responsável por realizar tarefas específicas. Para cumprir com algumas das tarefas sob sua responsabilidade, um objeto pode ter que interagir com outros objetos. É pela interação entre objetos que uma tarefa computacional é realizada.

Pode-se concluir que a orientação a objetos, como técnica para modelagem de sistemas, diminui a diferença semântica entre a realidade sendo modelada e os modelos construídos. Este livro descreve o importante papel da orientação a objetos na modelagem de sistemas de software atualmente. Explícita ou implicitamente, as técnicas de modelagem de sistemas aqui descritas utilizam os princípios que Alan Kay estabeleceu há mais de trinta anos. As seções a seguir continuam descrevendo os conceitos principais da orientação a objetos.

Um sistema de software orientado a objetos consiste em objetos em colaboração com o objetivo de realizar as funcionalidades desse sistema. Cada objeto é responsável por tarefas específicas. É graças à cooperação entre objetos que a computação do sistema se desenvolve.

1.2.1 Classes e objetos

O mundo real é formado de coisas. Como exemplos dessas coisas, pode-se citar um cliente, uma loja, uma venda, um pedido de compra, um fornecedor, este livro etc. Na terminologia de orientação a objetos, essas coisas do mundo real são denominadas *objetos*.

Seres humanos costumam agrupar os objetos. Provavelmente realizam esse processo mental de agrupamento para tentar gerenciar a complexidade de entender as coisas do mundo real. De fato, é bem mais fácil entender a ideia *cavalo* do que entender todos os cavalos que existem. Na terminologia da orientação a objetos, cada ideia é denominada *classe de objetos*, ou simplesmente *classe*. Uma classe é uma descrição dos atributos e serviços comuns a um grupo de objetos. Sendo assim, pode-se entender uma classe como sendo um *molde* a partir do qual objetos são construídos. Ainda sobre terminologia, diz-se que um objeto é uma *instância* de uma classe.

Por exemplo, quando se pensa em um cavalo, logo vem à mente um animal de quatro patas, cauda, crina etc. Pode ser que algum dia você veja dois cavalos, um mais baixo que o outro, um com cauda maior que o outro, ou mesmo, por um acaso infeliz, um cavalo com menos patas que o outro. No entanto, você ainda terá certeza de estar diante de dois cavalos. Isso porque a *ideia* (classe) cavalo está formada na mente dos seres humanos, independentemente das pequenas diferenças que possa haver entre os *exemplares* (objetos) da ideia cavalo.

É importante notar que uma classe é uma *abstração* das características de um grupo de coisas do mundo real. Na maioria das vezes, as coisas do mundo real são muito complexas para que *todas* as suas características sejam representadas em uma classe. Além disso, para fins de modelagem de um sistema, somente um subconjunto de características pode ser relevante. Portanto, uma classe representa uma abstração das características *relevantes* do mundo real.

Finalmente, é preciso atentar para o fato de que alguns textos sobre orientação a objetos (inclusive este livro!) utilizam os termos *classe* e *objeto* de maneira equivalente para denotar uma classe de objetos.

1.2.2 Operação, mensagem e estado

Dá-se o nome de **operação** a alguma ação que um objeto sabe realizar quando solicitado. De uma forma geral, um objeto possui diversas operações. Objetos não executam suas operações aleatoriamente. Para que uma operação em um objeto seja executada, deve haver um estímulo enviado a esse objeto. Se esse objeto for visto como uma entidade ativa que representa uma abstração de algo do mundo real, então faz sentido dizer que ele pode responder aos estímulos que lhe são enviados (assim como faz sentido dizer que seres vivos reagem aos estímulos que recebem). Seja qual for a origem do estímulo, quando ele ocorre diz-se que o objeto em questão está recebendo uma **mensagem** requisitando que ele realize alguma operação. Quando se diz na terminologia de orientação a objetos que *objetos de um sistema estão trocando mensagens* significa que esses objetos estão enviando mensagens uns aos outros com o objetivo de realizar alguma tarefa dentro do sistema no qual eles estão inseridos.

Por definição, o **estado** de um objeto corresponde ao conjunto de valores de seus atributos em um dado momento. Uma mensagem enviada a um objeto tem o potencial de mudar o estado desse objeto. Por exemplo, o recebimento de uma mensagem em um objeto da classe ContaBancaria solicitando o saque de determinada quantia possivelmente irá alterar o valor do atributo desse objeto relativo ao saldo da conta, o que é o mesmo que dizer que esse objeto está mudando de estado. Por outro lado, uma mensagem enviada para solicitar o valor atual do saldo não altera o estado desse objeto.

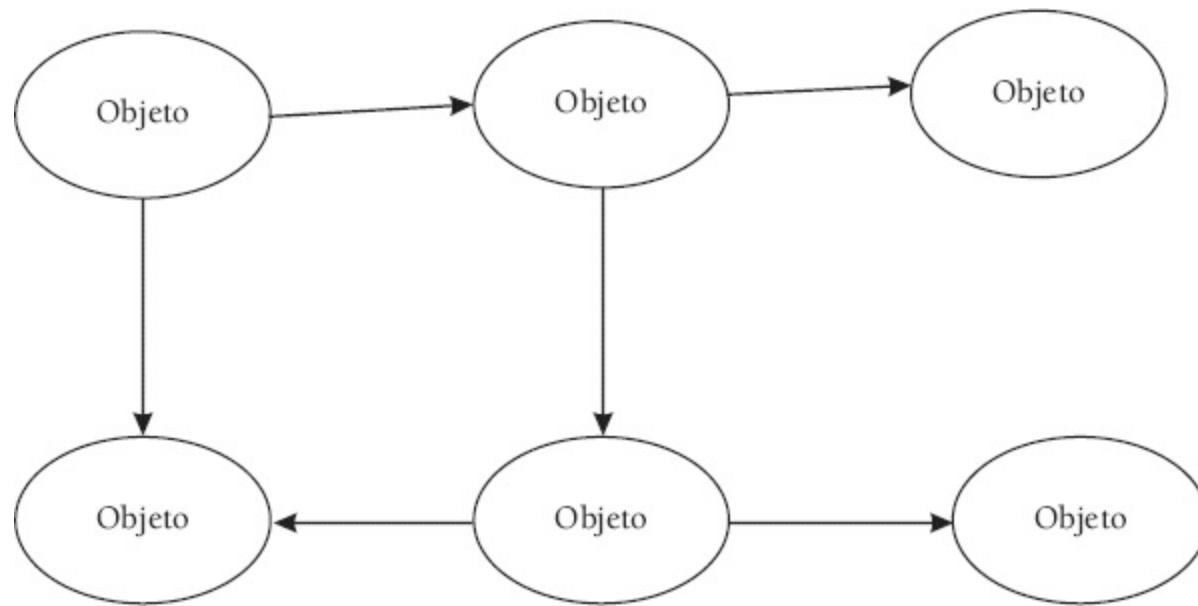


Figura 1-1: Objetos interagem através do envio de mensagens.

1.2.3 O papel da abstração na orientação a objetos

Nesta seção, apresentamos os principais conceitos do paradigma da orientação a objetos. Discutimos também o argumento de que todos esses conceitos são, na verdade, a aplicação de um único conceito mais básico, o *princípio da abstração*.

Primeiramente vamos descrever o conceito de abstração. A abstração é um processo mental pelo qual nós, seres humanos, nos atemos aos aspectos mais importantes (relevantes) de alguma coisa, ao mesmo tempo em que ignoramos os menos importantes. Esse processo mental nos permite gerenciar a complexidade de um objeto, enquanto concentrarmos nossa atenção nas características essenciais do mesmo. Note que uma abstração de algo é dependente da perspectiva (contexto) sobre a qual uma

coisa é analisada: o que é importante em um contexto pode não ser importante em outro. Nas próximas seções, descrevemos alguns conceitos fundamentais da orientação a objetos e estabelecemos sua correlação com o conceito de abstração.

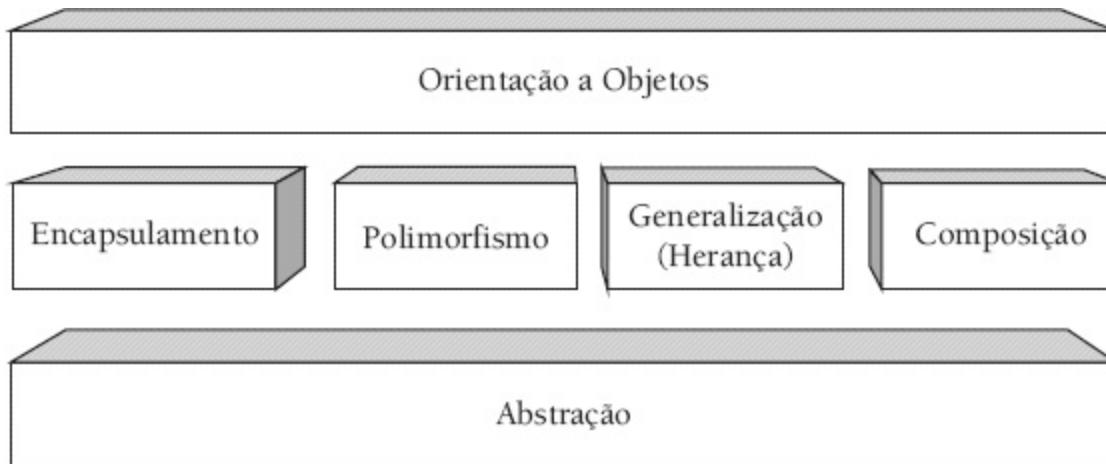


Figura 1-2: Princípios da orientação a objetos podem ser vistos como aplicações de um princípio mais básico, o da abstração.

1.2.3.1 Encapsulamento

Objetos possuem *comportamento*. O termo comportamento diz respeito a operações realizadas por um objeto, a medida que ele recebe mensagens. O mecanismo de *encapsulamento* é uma forma de restringir o acesso ao comportamento interno de um objeto. Um objeto que precise da colaboração de outro objeto para realizar alguma operação simplesmente envia uma mensagem a este último. Segundo o mecanismo do encapsulamento, o *método* que o objeto requisitado usa para realizar a operação não é conhecido dos objetos requisitantes. Em outras palavras, o objeto remetente da mensagem não precisa conhecer a forma pela qual a operação requisitada é realizada; tudo o que importa a esse objeto remetente é obter a operação realizada, não importando como.

Certamente o remetente da mensagem precisa conhecer quais operações o receptor sabe realizar ou que informações este objeto receptor pode fornecer. Para tanto, a *classe* de um objeto descreve o seu comportamento. Na terminologia da orientação a objetos, diz-se que um objeto possui uma *interface* (ver [Figura 1-3](#)). Em termos bastante simples, a interface de um objeto corresponde ao que ele conhece e ao que sabe fazer, sem, no entanto, descrever *como* conhece ou faz. Se visualizarmos um objeto como um provedor de serviços, a interface de um objeto define os serviços que ele pode fornecer. Consequentemente, a interface de um objeto também define as mensagens que ele está apto a receber e a responder. Um serviço definido na interface de um objeto pode ter várias formas de *implementação*. Mas, pelo encapsulamento, a implementação de um serviço requisitado não importa ou não precisa ser conhecida pelo objeto requisitante.

Porque existe o princípio do encapsulamento, a única coisa que um objeto precisa saber para pedir a colaboração de outro objeto é conhecer a interface deste último. Nada mais. Isso contribui para a autonomia dos objetos, pois cada um envia mensagens aos outros para realizar certas operações, sem se preocupar em *como* se realizaram as operações.

Note também que, de acordo com o encapsulamento, a implementação de uma operação pode ser trocada sem que o objeto requisitante da mesma precise ser alterado. Não posso deixar de enfatizar a importância desse aspecto no desenvolvimento de softwares. Se a implementação de uma operação pode ser substituída por outra sem alterações nas regiões do sistema que precisam dessa operação, há menos possibilidades de propagações de mudanças. No desenvolvimento de softwares modernos,

nos quais os sistemas se tornam cada vez mais complexos, é fundamental manter as partes de um sistema tão independentes quanto possível. Daí a importância do mecanismo do encapsulamento no desenvolvimento de softwares orientados a objetos.

E qual a relação entre os conceitos de encapsulamento e abstração? Podemos dizer que o encapsulamento é uma aplicação do conceito de abstração. A aplicação da abstração, neste caso, está em esconder os detalhes de funcionamento interno de um objeto. Voltando ao contexto de desenvolvimento de softwares, note a consequência disso sobre a produtividade do desenvolvimento. Se pudermos utilizar os serviços de um objeto sem precisarmos entender seus detalhes de funcionamento, é claro que a produtividade do desenvolvimento aumenta.

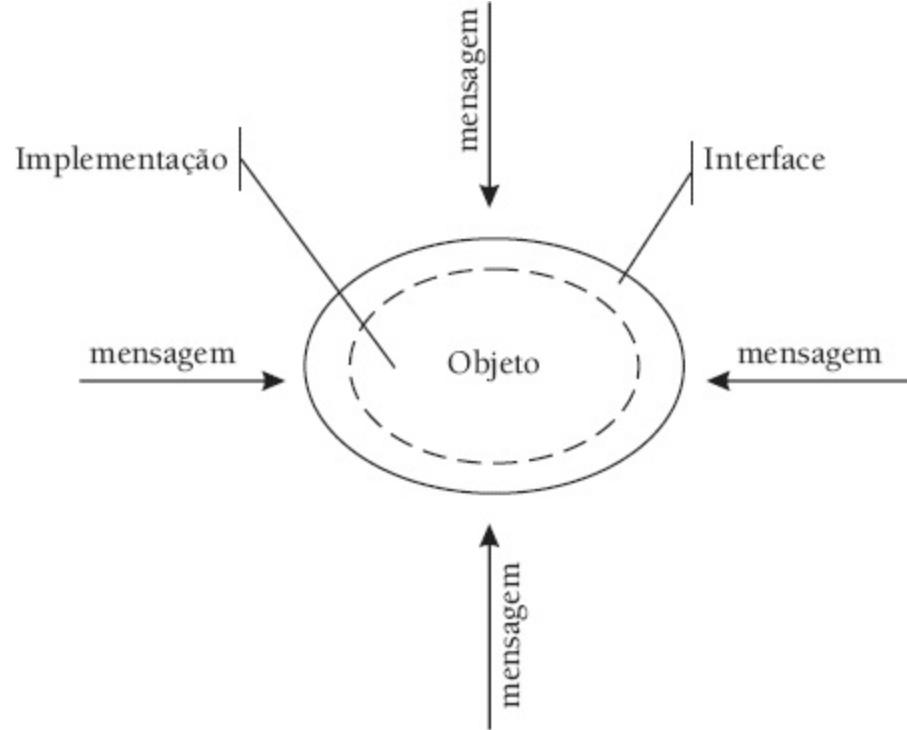


Figura 1-3: Princípio do encapsulamento: visto externamente, o objeto é a sua interface.

1.2.3.2 Polimorfismo

O polimorfismo indica a capacidade de abstrair várias implementações diferentes em uma única interface. Há algum tempo, o controle remoto de meu televisor quebrou. (Era realmente enfadonho ter de levantar para desligar o aparelho ou trocar de canal.) Um tempo depois, comprei um videocassete do mesmo fabricante de meu televisor. Para minha surpresa, o controle remoto do videocassete também funcionava para o televisor. Esse é um exemplo de aplicação do *princípio do polimorfismo* na vida real. Nesse caso, dois objetos do mundo real, o televisor e o aparelho de videocassete, respondem à mesma mensagem enviada.

E no que diz respeito à orientação a objetos, qual é a importância e quais são as consequências do polimorfismo? Nesse contexto, o polimorfismo se refere à capacidade de duas ou mais classes de objetos responderem à mesma mensagem, cada qual de seu próprio modo. O exemplo clássico do polimorfismo em desenvolvimento de software é o das formas geométricas. Pense em uma coleção de formas geométricas que contenha círculos, retângulos e outras formas específicas. Pelo princípio do polimorfismo, quando uma região de código precisa desenhar os elementos daquela coleção, essa região não deve precisar conhecer os tipos específicos de figuras existentes; basta que cada elemento da coleção receba uma mensagem solicitando que desenhe a si próprio. Note que isso simplifica a

região de código cliente (ou seja, a região de código que solicitou o desenho das figuras). Isso porque essa região de código não precisa conhecer o tipo de cada figura. Ao mesmo tempo, essa região de código não precisa ser alterada quando, por exemplo, uma classe correspondente a um novo tipo de forma geométrica (uma reta, por exemplo) tiver que ser adicionada. Esse novo tipo deve responder à mesma mensagem (solicitação) para desenhar a si próprio, muito embora implemente a operação a seu modo.

Note mais uma vez que, assim como no caso do encapsulamento, a abstração também é aplicada para obter o polimorfismo: um objeto pode enviar a *mesma* mensagem para objetos semelhantes, mas que implementam a sua interface de formas diferentes. O que está se abstraindo aqui são as diferentes maneiras pelas quais os objetos receptores respondem à mesma mensagem.

1.2.3.3 Generalização (Herança)

A generalização é outra forma de abstração utilizada na orientação a objetos. A [Seção 1.2.1](#) declara que as características e o comportamento comuns a um conjunto de objetos podem ser abstraídos em uma classe. Dessa forma, uma classe descreve as características e o comportamento comuns de um grupo de objetos semelhantes. A generalização pode ser vista como um nível de abstração acima da encontrada entre classes e objetos. Na generalização, classes semelhantes são agrupadas em uma hierarquia (ver [Figura 1-4](#)). Cada nível dessa hierarquia pode ser visto como um nível de abstração. Cada classe em um nível da hierarquia herda as características e o comportamento das classes às quais está associada nos níveis acima dela. Além disso, essa classe pode definir características e comportamento particulares. Dessa forma, novas classes podem ser criadas a partir do reúso da definição de outras preexistentes.

O mecanismo de generalização facilita o compartilhamento de comportamento comum entre um conjunto de classes semelhantes. Além disso, as diferenças ou variações entre classes em relação àquilo que é comum entre elas podem ser organizadas de forma mais clara.

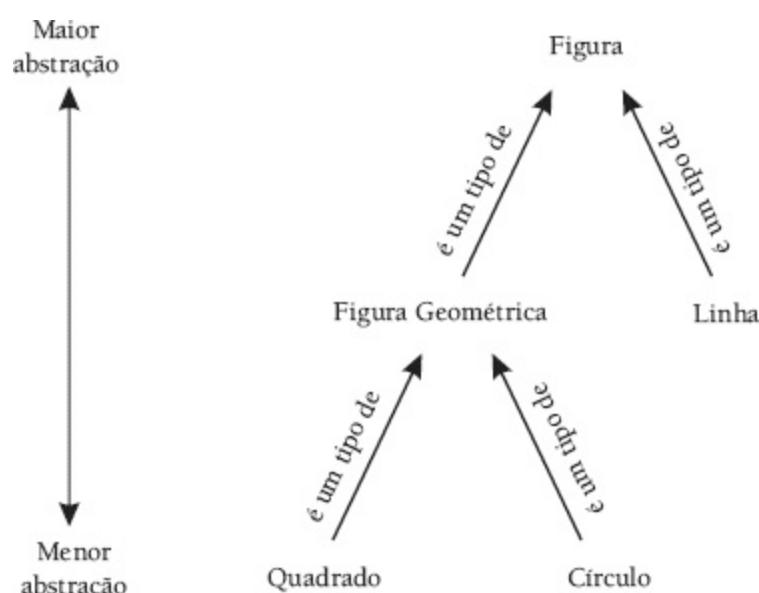


Figura 1-4: Princípio da generalização: classes podem ser organizadas em hierarquias.

1.2.3.4 Composição

É natural para nós, seres humanos, pensar em coisas do mundo real como objetos compostos de outros objetos. Por exemplo, um livro é composto de páginas; páginas possuem parágrafos;

parágrafos possuem frases e assim por diante. Outro exemplo: uma televisão contém um painel de controle, uma tela. Da mesma forma que o livro e a televisão podem ser vistos como objetos por si próprios, seus respectivos componentes também podem ser vistos como objetos. De uma forma geral objetos podem ser *compostos* de outros objetos; esse é o princípio da composição. A composição permite que criemos objetos a partir da reunião de outros objetos.

Repare a semelhança entre os princípios de generalização e composição. Ambos propiciam formas de definir novos conceitos (i.e., classes) a partir de conceitos preexistentes. Entretanto, há situações adequadas para aplicação de um princípio ou outro, conforme descrito mais adiante neste livro. O correto entendimento da aplicação desses dois princípios é uma competência importante para o profissional que realiza atividades de modelagem e desenvolvimento de software orientados a objetos.

1.3 Evolução histórica da modelagem de sistemas

A *Lei de Moore* é bastante conhecida na área da computação. Embora seja conhecida como lei, foi apenas uma declaração feita em 1965 pelo engenheiro Gordon Moore, cofundador da Intel™. A Lei de Moore estabelece que “a densidade de um transistor dobra em um período entre 18 e 24 meses”. Isso significa que se um processador pode ser construído hoje com capacidade de processamento P, em 24 meses é possível construir um processador com capacidade 2P. Em 48 meses, pode-se construir um com capacidade 4P, e assim por diante. Ou seja, a Lei de Moore implica uma taxa de crescimento *exponencial* na capacidade de processamento dos computadores.²

O que a Lei de Moore tem a ver com a modelagem de sistemas? Bom, provavelmente o ditado “a necessidade é a mãe das invenções” também se aplica a esse caso. O rápido crescimento da capacidade computacional das máquinas resultou na demanda por sistemas de software cada vez mais complexos, que tirassem proveito dessa nova capacidade. Por sua vez, o surgimento desses sistemas mais complexos resultou na necessidade de reavaliação da forma de desenvolver sistemas. Consequentemente, desde o aparecimento do primeiro computador até os dias de hoje, as técnicas para a construção de sistemas computacionais evoluíram de forma impressionante, principalmente no que diz respeito à modelagem de sistemas.

A seguir, temos um breve resumo histórico da evolução das técnicas de desenvolvimento com o objetivo de esclarecer como se chegou às técnicas atualmente utilizadas.³

- **Décadas de 1950/60:** os sistemas de software eram bastante simples. O desenvolvimento desses sistemas era feito de forma *ad hoc*.⁴ Os sistemas eram significativamente mais simples e, consequentemente, as técnicas de modelagem também: era a época dos *fluxogramas* e dos *diagramas de módulos*.
- **Década de 1970:** nessa época, computadores mais avançados e acessíveis começaram a surgir. Houve uma grande expansão do mercado computacional. Sistemas mais complexos nasciam. Por conseguinte, modelos mais robustos foram propostos. Nesse período surge a *programação estruturada*, baseada nos trabalhos de David Parnas. No final dessa década aparecem também a *análise* e o *projeto estruturado*, consolidados pelos trabalhos de Tom DeMarco. Além de Tom, os autores Larry Constantine e Edward Yourdon foram grandes colaboradores nessas técnicas de modelagem.
- **Década de 1980:** nessa fase, os computadores se tornaram ainda mais avançados e baratos. Surge a necessidade por interfaces homem-máquina mais sofisticadas, o que originou a

produção de sistemas de softwares mais complexos. A *análise estruturada* se consolidou na primeira metade desta década com os trabalhos de Edward Yourdon, Peter Coad, Tom DeMarco, James Martin e Chris Gane. Em 1989, Edward Yourdon lança o clássico *Análise estruturada moderna*, livro que se tornou uma referência no assunto.

- **Início da década de 1990:** esse é o período em que surge um novo paradigma de modelagem a *Análise Orientada a Objetos*, como resposta a dificuldades encontradas na aplicação da Análise Estruturada a certos domínios de aplicação. Grandes colaboradores no desenvolvimento do paradigma orientado a objetos são Sally Shlaer, Stephen Mellor, Rebecca Wirfs-Brock, James Rumbaugh, Grady Booch e Ivar Jacobson.
- **Fim da década de 1990:** o paradigma da orientação a objetos atinge sua maturidade. Os conceitos de padrões de projeto, frameworks, componentes e qualidade começam a ganhar espaço. Surge a *Linguagem de Modelagem Unificada (UML)*.
- **Década de 2000:** o reúso por meio de padrões de projeto e de frameworks se consolida. As denominadas metodologias ágeis começam a ganhar espaço. Técnicas de testes automatizados e refatoração começam a se disseminar na comunidade de desenvolvimento orientado a objetos. Grandes influenciadores dessa fase são Rebecca Wirfs-Brock, Martin Fowler e Eric Evans.

Um estudo mais detalhado da primeira metade da década de 1990 pode mostrar o surgimento de várias propostas de técnicas para modelagem de sistemas segundo o paradigma da orientação a objetos. A [Tabela 1-1](#) lista algumas das técnicas existentes durante esse período; nota-se uma grande proliferação de propostas para modelagem orientada a objetos. Nesse período, era comum o fato de duas técnicas possuírem diferentes notações gráficas para modelar uma mesma perspectiva de um sistema. Ao mesmo tempo, cada técnica tinha seus pontos fortes e fracos em relação à notação que utilizava. A essa altura percebeu-se a necessidade de uma notação de modelagem que viesse a se tornar um padrão para a modelagem de sistemas orientados a objetos; essa notação deveria ser aceita e utilizada amplamente pela indústria e pelos ambientes acadêmicos. Surgiram, então, alguns esforços para essa padronização, o que resultou na definição da UML (*Unified Modeling Language*) em 1996 como a melhor candidata para ser a linguagem “unificadora” de notações, diagramas e formas de representação existentes em diferentes técnicas de modelagem. Descrevemos mais detalhadamente essa proposta na próxima seção.

Tabela 1-1: Principais propostas de técnicas de modelagem orientada a objetos durante a década de 1990

Ano	Autor (Técnica)
1990	Shaler & Mellor
1991	Coad & Yourdon (OOAD – Object-Oriented Analysis and Design)
1993	Grady Booch (Booch Method)
1993	Ivar Jacobson (OOSE – Object-Oriented Software Engineering)
1995	James Rumbaugh <i>et al.</i> (OMT – Object Modeling Technique)
1996	Wirfs-Brock (Responsibility Driven Design)
1996	(Fusion)

1.4 A Linguagem de Modelagem Unificada (UML)

A construção da UML teve muitos contribuintes, mas os principais atores no processo foram Grady Booch, James Rumbaugh e Ivar Jacobson. Esses três pesquisadores costumam ser chamados de “os três amigos”. No processo de definição inicial da UML, esses pesquisadores buscaram aproveitar o melhor das características das notações preexistentes, principalmente das técnicas que haviam proposto anteriormente (essas técnicas eram conhecidas pelos nomes *Booch Method*, *OMT* e *OOSE*). Consequentemente, a notação definida para a UML é uma união de diversas notações preexistentes, com alguns elementos removidos e outros adicionados com o objetivo de torná-la mais expressiva.

Finalmente, em 1997, a UML foi aprovada como padrão pelo OMG.⁵ Desde então, a UML tem tido grande aceitação pela comunidade de desenvolvedores de sistemas. A sua definição ainda está em desenvolvimento e conta com diversos colaboradores da área comercial.⁶ Desde o seu surgimento, várias atualizações foram feitas para torná-la mais clara e útil. Atualmente, a especificação do padrão UML está na versão 2.5 (OMG, 2013).

A UML é uma *linguagem visual* para modelar sistemas orientados a objetos. Isso quer dizer que a UML é uma linguagem que define elementos gráficos (visuais) que podem ser utilizados na modelagem de sistemas. Esses elementos permitem representar os conceitos do paradigma da orientação a objetos. Por meio dos elementos gráficos definidos nesta linguagem pode-se construir diagramas que representam diversas perspectivas de um sistema.

Cada elemento gráfico da UML possui uma *sintaxe* e uma *semântica*. A sintaxe de um elemento corresponde à forma predeterminada de desenhar o elemento. A semântica define o que significa o elemento e com que objetivo ele deve ser utilizado. Além disso, conforme descrito mais adiante, tanto a sintaxe quanto a semântica dos elementos da UML são *extensíveis*. Essa extensibilidade permite que a UML seja adaptada às características específicas de cada projeto de desenvolvimento.

Pode-se fazer uma analogia da UML com uma *caixa de ferramentas*. Um pedreiro usa sua caixa de ferramentas para realizar suas tarefas. Da mesma forma, a UML pode ser vista como uma caixa de ferramentas utilizada pelos desenvolvedores de sistemas para realizar a construção de modelos.

A UML é *independente tanto de linguagens de programação quanto de processos de desenvolvimento*. Isso quer dizer que ela pode ser utilizada para a modelagem de sistemas, não importando que linguagem de programação será utilizada na implementação do sistema, nem a forma (processo) de desenvolvimento adotada. Esse é um fator importante para a utilização da UML, pois diferentes sistemas de software requerem abordagens diversas de desenvolvimento.

A definição completa da UML está contida na *Especificação da Linguagem de Modelagem Unificada da OMG*. Essa especificação pode ser obtida gratuitamente no site da OMG (www.uml.org). Embora essa documentação seja bastante completa, ela está longe de fornecer uma leitura fácil, pois é direcionada a pesquisadores ou àqueles que trabalham com ferramentas de suporte (ferramentas CASE; ver Seção 2.6) ao desenvolvimento de sistemas.

1.4.1 Visões de um sistema

O desenvolvimento de um sistema de software complexo demanda que seus desenvolvedores tenham a possibilidade de examinar e estudar esse sistema a partir de perspectivas diversas. Os autores da UML sugerem que um sistema pode ser descrito por cinco visões interdependentes desse sistema (BOOCH *et al.*, 2006). Cada visão enfatiza aspectos variados do sistema. As visões propostas são as seguintes:

- *Visão de Casos de Uso*: descreve o sistema de um ponto de vista externo como um conjunto de interações entre o sistema e os agentes externos ao sistema. Esta visão é criada em um estágio inicial e direciona o desenvolvimento das outras visões do sistema.
- *Visão de Projeto*: enfatiza as características do sistema que dão suporte, tanto estrutural quanto comportamental, às funcionalidades externamente visíveis do sistema.
- *Visão de Implementação*: abrange o gerenciamento de versões do sistema, construídas pelo agrupamento de módulos (componentes) e subsistemas.
- *Visão de Implantação*: corresponde à distribuição física do sistema em seus subsistemas e à conexão entre essas partes.
- *Visão de Processo*: esta visão enfatiza as características de concorrência (parallelismo), sincronização e desempenho do sistema.

Dependendo das características e da complexidade do sistema, nem todas as visões precisam ser construídas. Por exemplo, se o sistema tiver de ser instalado em um ambiente computacional de processador único, não há necessidade da visão de implantação. Outro exemplo: se o sistema for constituído de um único processo, a visão de processo é irrelevante. De forma geral, dependendo do sistema, as visões podem ser ordenadas por grau de relevância.

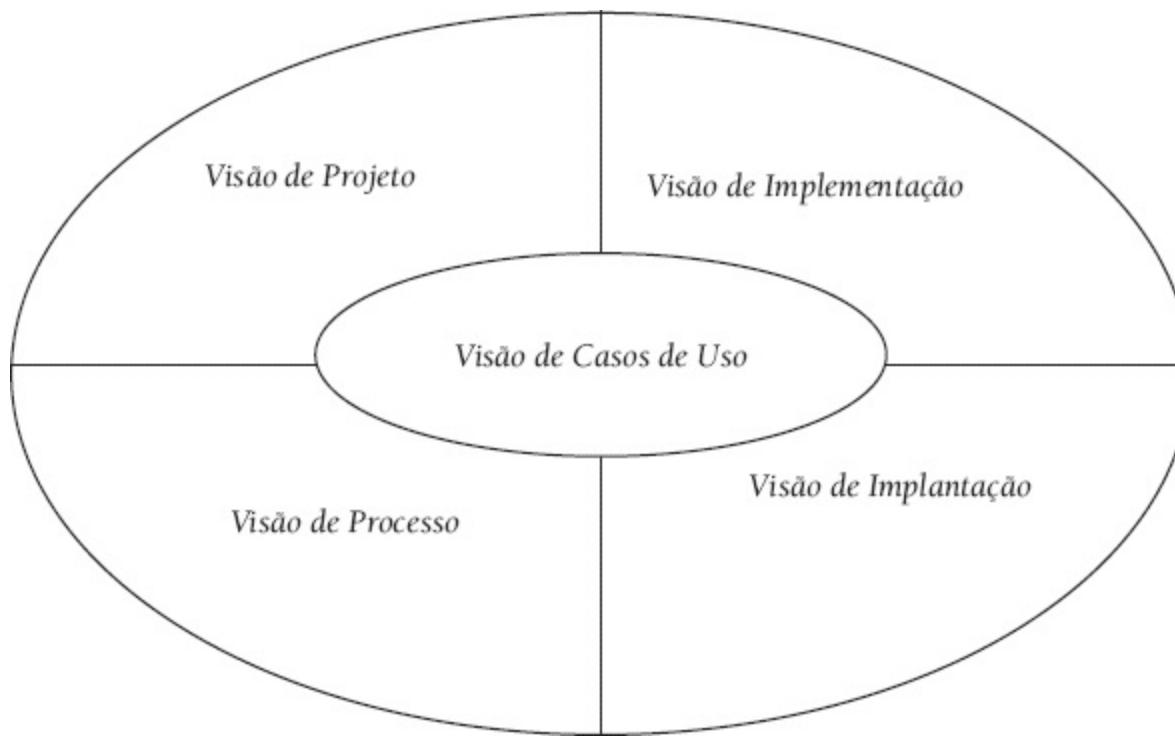


Figura 1-5: Visões (perspectivas) de um sistema de software.

1.4.2 Diagramas da UML

Um processo de desenvolvimento que utilize a UML como linguagem de suporte à modelagem envolve a criação de diversos documentos. Esses documentos podem ser textuais ou gráficos. Na terminologia da UML, eles são denominados *artefatos de software*, ou simplesmente *artefatos*. São os artefatos que compõem as visões do sistema.

Os artefatos gráficos produzidos durante o desenvolvimento de um sistema de software orientado a objetos (SSOO) podem ser definidos pela utilização dos diagramas da UML. Os 13 diagramas da UML 2.0 são listados na [Figura 1-6](#). Nela, os retângulos com os cantos retos representam

agrupamentos (tipos) de diagramas da UML. Já os retângulos com os cantos boleados representam os diagramas propriamente ditos.

O iniciante na modelagem de sistemas pode muito bem perguntar: “Para que um número tão grande de diagramas para modelar um sistema? Será que um ou dois tipos de diagramas já não seriam suficientes?” Para justificar a existência de vários diagramas, vamos utilizar novamente uma analogia. Carrinhos em miniatura são cópias fiéis de carros de verdade. Cada carrinho é um modelo físico tridimensional que pode ser analisado sob diversas perspectivas (por cima, por baixo, por dentro etc.). O mesmo não ocorre com os diagramas, que são desenhos bidimensionais.



Figura 1-6: Diagramas definidos pela UML.

Para compensar essa dimensão a menos, utilizam-se diversos diagramas para construir modelos de várias perspectivas do sistema. Cada diagrama da UML fornece uma perspectiva parcial do sistema sendo modelado, consistente com as demais perspectivas. No decorrer deste livro, descreveremos os diagramas da UML e sua utilização na construção de modelos que fornecem as várias perspectivas existentes em sistemas de software.

Por outro lado, é importante notar que, na maioria dos casos práticos de modelagem, apenas uma pequena fração dos diagramas que a UML fornece é realmente necessária. De fato, uma das críticas que a versão 2.0 recebeu foi relativa ao aumento de diagramas propostos. Na versão anterior à 2.0, havia dez tipos de diagramas. A versão 2.0 aumentou essa quantidade para 13. Por fim a versão atual (2.5) acrescentou mais três, o que elevou o total para 16 tipos de diagramas. Em minha opinião, à exceção do diagrama de visão geral da interação, todos os demais diagramas definidos a partir da versão 2.0 não contribuem significativamente para facilitar a modelagem de sistemas, particularmente para o caso de sistemas de informação, que é o escopo principal deste livro. Portanto, o leitor não deve criar a expectativa de encontrar nos próximos capítulos deste livro descrições detalhadas de todos os diagramas da versão atual da UML. Em vez disso, apresentamos os elementos dessa linguagem que, no nosso entendimento, são os mais utilizados nas práticas de modelagem de sistemas orientados a objetos.

semântica correspondente para representar visualmente uma ou mais perspectivas de um sistema.

► EXERCÍCIOS

1-1: Considere o mapa de uma cidade que mostra rodovias e prédios e que esconde a cor dos prédios. Um mapa pode ser considerado um modelo? Por quê? Discuta as características desse mapa com relação ao princípio da abstração.

1-2: Identifique paralelos entre as seguintes características de uma célula e os conceitos da orientação a objetos descritos neste capítulo.

- a. Mensagens são enviadas de uma célula a outra por meio de *receptores químicos*.
- b. Células têm uma fronteira (a *membrana celular*). Cada célula tem um comportamento interno que não é visível por fora.
- c. A células podem se reagrupar para resolver problemas ou realizar uma função.

1-3: Considere os seguintes itens: elevador, maçã, a Terra, este livro, você mesmo, o Cristo Redentor. Será que esses itens são objetos, de acordo com os princípios estabelecidos por Alan Kay?

1-4: O termo *modelagem* é bastante amplo e popular. As áreas da Matemática, Filosofia, Psiquiatria e Química, por exemplo, também utilizam esse termo. Discuta com um profissional de alguma dessas áreas se há qualquer correlação entre a noção de modelagem por ele utilizada e a noção utilizada no contexto deste capítulo.

1-5: Explique e relate os termos objeto, classe, generalização e mensagem. Dê exemplos de cada um desses conceitos.

-
- 1. Talvez tal história não seja verídica, mas ilustra bem o conceito que quero passar.
 - 2. Desde que foi declarada, a Lei de Moore vem se verificando com uma precisão impressionante. No entanto, alguns especialistas, incluindo o próprio Moore, acreditam que a capacidade de dobrar a capacidade de processamento dos processadores a cada 18 meses não seja mais possível por volta de 2017.
 - 3. É importante notar que a divisão de períodos aqui apresentada é meramente didática. Na realidade, não há uma divisão clara das diversas propostas de modelagem. Por exemplo, propostas iniciais de modelagem orientada a objetos podem ser encontradas já em meados da década de 1970.
 - 4. Este termo significa “direto ao assunto” ou “direto ao que interessa”. Talvez o uso desse termo denote a abordagem dessa primeira fase do desenvolvimento de sistemas, na qual não havia um planejamento inicial. O código-fonte do programa a ser construído era o próprio modelo.
 - 5. Sigla para Object Management Group. O OMG é um consórcio internacional de empresas que define e ratifica padrões na área da orientação a objetos.
 - 6. Algumas das empresas que participam da definição da UML são: Digital, HP, IBM, Oracle, Microsoft, Unisys, IntelliCorp, i-Logix e Rational.

O processo de desenvolvimento de software

Quanto mais livros você leu (ou escreveu), mais aulas assistiu (ou lecionou), mais linguagens de programação aprendeu (ou projetou), mais software OO examinou (ou produziu), mais documentos de requisitos tentou decifrar (ou tornou decifrável), mais padrões de projeto aprendeu (ou catalogou), mais reuniões assistiu (ou conduziu), mais colegas de trabalho talentosos teve (ou contratou), mais projetos ajudou (ou gerenciou), tanto mais você estará equipado para lidar com um novo desenvolvimento.

– BERTRAND MEYER

O desenvolvimento de software é uma atividade complexa. Essa complexidade corresponde à sobreposição das complexidades relativas ao desenvolvimento dos seus diversos componentes: software, hardware, procedimentos etc. Isso se reflete no alto número de projetos de software que não chegam ao fim, ou que extrapolam recursos de tempo e dinheiro alocados.

Para dar uma ideia da complexidade no desenvolvimento de sistemas de software, são listados a seguir alguns dados levantados no *Chaos Report*, um estudo clássico feito pelo *Standish Group* sobre projetos de desenvolvimento (CHAOS, 1994).

- Porcentagem de projetos que terminam dentro do prazo estimado: 10%.
- Porcentagem de projetos que são descontinuados antes de chegar ao fim: 25%.
- Porcentagem de projetos acima do custo esperado: 60%.
- Atraso médio nos projetos: um ano.

Tentativas de lidar com essa complexidade e de minimizar os problemas envolvidos no desenvolvimento de software envolvem a definição de *processos de desenvolvimento de software*. Um processo de desenvolvimento de software (chamado simplesmente de *processo de desenvolvimento* ou *metodologia de desenvolvimento*) compreende todas as atividades necessárias para definir, desenvolver, testar e manter um produto de software. Alguns objetivos de um processo de desenvolvimento são: definir *quais* as atividades a serem executadas ao longo do projeto; *quando, como e por quem* tais atividades serão executadas; prover pontos de controle para verificar o andamento do desenvolvimento; padronizar o modelo de desenvolvimento de softwares em uma organização. Exemplos de processos de desenvolvimento propostos são o ICONIX, o RUP (*Rational Unified Process*), o EUP (*Enterprise Unified Process*), XP (*Extreme Programming*) e o OPEN (*Object-oriented Process, Environment and Notation*).

2.1 Atividades típicas de um processo de desenvolvimento

Um processo de desenvolvimento classifica em atividades as tarefas realizadas durante a construção de um sistema de software. Há vários processos de desenvolvimento propostos. Por outro lado, é um consenso na comunidade de desenvolvimento de software o fato de que não existe um melhor

processo, aquele que melhor se aplica a todas as situações de desenvolvimento.

Cada processo tem suas particularidades em relação ao modo de arranjar e encadear as atividades de desenvolvimento. Entretanto, podem-se distinguir atividades que, com uma ou outra modificação, são comuns à maioria dos processos existentes. Nesta seção, descrevemos essas atividades, posto que duas dessas atividades, a análise e o projeto, fazem parte do assunto principal desse livro.

2.1.1 Levantamento de requisitos

A atividade de *levantamento de requisitos* (também conhecida como *elicitação de requisitos*) corresponde à etapa de compreensão do problema aplicada ao desenvolvimento de software. O principal objetivo do levantamento de requisitos é que usuários e desenvolvedores tenham a mesma visão do problema a ser resolvido. Nessa etapa, os desenvolvedores, juntamente com os clientes, tentam levantar e definir as necessidades dos futuros usuários do sistema a ser desenvolvido.¹ Essas necessidades são geralmente denominadas *requisitos*.

Formalmente, um requisito é uma condição ou capacidade que deve ser alcançada ou possuída por um sistema ou um de seus componentes para satisfazer um contrato, padrão, especificação ou outros documentos formalmente impostos (MACIASZEK, 2000). Normalmente os requisitos de um sistema são identificados a partir de um *domínio*. Denomina-se domínio a área de conhecimento ou atividade específica caracterizada por um conjunto de conceitos e de terminologia compreendidos por especialista nessa área. No contexto do desenvolvimento de software, um domínio corresponde à parte do mundo real que é *relevante*, no sentido de que algumas informações e processos desse domínio precisam ser incluídos no sistema em desenvolvimento. O domínio também é chamado de *domínio do problema* ou *domínio do negócio*.²

Durante o levantamento de requisitos, a equipe de desenvolvimento tenta entender o domínio que deve ser automatizado pelo sistema de software. O levantamento de requisitos compreende também um estudo exploratório das necessidades dos usuários e da situação do sistema atual (caso exista um). Há várias técnicas utilizadas para isso, como, por exemplo: leitura de obras de referência e livros-texto, observação do ambiente do usuário, realização de entrevistas com os usuários, entrevistas com *especialistas do domínio*³ (ver a Seção 2.2.6), reutilização de análises anteriores, comparação com sistemas preexistentes do mesmo domínio do negócio.

O produto do levantamento de requisitos é o *documento de requisitos*, que declara os diversos tipos de requisitos do sistema. É normal esse documento ser escrito em uma notação informal (em linguagem natural). As principais seções de um documento de requisitos são:

1. *Requisitos funcionais*: definem as funcionalidades do sistema. Alguns exemplos de requisitos funcionais são os seguintes:
 - a. “O sistema deve permitir que cada professor realize o lançamento de notas das turmas nas quais lecionou.”
 - b. “O sistema deve permitir que um aluno realize a sua matrícula nas disciplinas oferecidas em um semestre letivo.”
 - c. “Os coordenadores de escola devem poder obter o número de aprovações, reprovações e trancamentos em cada disciplina oferecida em um determinado período.”
2. *Requisitos não funcionais*: declaram as características de *qualidade* que o sistema deve possuir e que estão relacionadas às suas funcionalidades. Alguns tipos de requisitos não funcionais são os seguintes:
 - a. Confiabilidade: corresponde a medidas quantitativas da confiabilidade do sistema, tais

como tempo médio entre falhas, recuperação de falhas ou quantidade de erros por milhares de linhas de código-fonte.

- b. Desempenho: requisitos que definem os tempos de resposta esperados para as funcionalidades do sistema.
- c. Portabilidade: restrições quanto às plataformas de hardware e software nas quais o sistema será implantado e quanto ao grau de facilidade para transportar o sistema para outras plataformas.
- d. Segurança: limitações sobre a segurança do sistema em relação a acessos não autorizados.
- e. Usabilidade: requisitos que se relacionam ou afetam a usabilidade do sistema. Exemplos incluem requisitos sobre a facilidade de uso e a necessidade ou não de treinamento dos usuários.

3. *Requisitos normativos*: declaração de restrições impostas sobre o desenvolvimento do sistema. Restrições definem, por exemplo, a adequação a custos e prazos, a plataforma tecnológica, aspectos legais (licenciamento), limitações sobre a interface com o usuário, componentes de hardware e software a serem adquiridos, eventuais necessidades de comunicação do novo sistema com sistemas legados etc. Restrições também podem corresponder a *regras do negócio*, restrições ou políticas de funcionamento específicas do domínio do problema que influenciarão de um modo ou de outro no desenvolvimento do software. Regras do negócio são descritas na [Seção 4.5.1](#).

Uma das formas de se medir a qualidade de um sistema de software é pela sua utilidade. E um sistema será útil para seus usuários se *atender aos requisitos* definidos e se esses requisitos refletirem as necessidades dos usuários. Portanto, os requisitos devem ser expressos de uma maneira tal que possam ser verificados e comunicados a leitores técnicos e não técnicos. A equipe técnica (leitores técnicos) deve entender o documento de requisitos de forma que possa obter as soluções técnicas apropriadas. Clientes (leitores não técnicos) devem entender esse documento para que possam priorizar o desenvolvimento dos requisitos, conforme as necessidades da organização em que trabalham.

Um ponto importante a respeito do documento de requisitos é que ele não deve conter informações sobre as *soluções técnicas* que serão adotadas para desenvolver o sistema. O enfoque prioritário do levantamento de requisitos é responder claramente à questão: “O que o usuário necessita do novo sistema?”. Lembre-se sempre: novos sistemas serão avaliados pelo seu grau de conformidade aos requisitos, não importa quão complexa a solução tecnológica tenha sido. Requisitos definem o problema a ser resolvido pelo sistema de software; eles não descrevem o software que resolve o problema.

O levantamento de requisitos é a etapa mais importante em termos de retorno em investimentos feitos para o projeto de desenvolvimento. Muitos sistemas foram abandonados ou nem chegaram a ser utilizados, porque os membros da equipe não dispensaram tempo suficiente para compreender as necessidades do cliente em relação ao novo sistema. De fato, um sistema de informações é normalmente utilizado para automatizar processos de negócio de uma organização. Portanto, esses processos devem ser compreendidos antes da construção do sistema de informações. Em um estudo baseado em 6.700 sistemas feito em 1997, Carper Jones mostrou que os custos resultantes da má realização dessa etapa de entendimento podem ser duzentas vezes maiores que o realmente necessário (JONES, 1997).

O documento de requisitos serve como um termo de consenso entre a equipe técnica

(desenvolvedores) e o cliente. Esse documento constitui a base para as atividades subsequentes do desenvolvimento do sistema e fornece um ponto de referência para qualquer validação futura do software construído. O envolvimento do cliente desde o início do processo de desenvolvimento ajuda a assegurar que o produto desenvolvido realmente atenda às necessidades identificadas. Além disso, o documento de requisitos estabelece o *escopo do sistema* (isto é, o que faz parte ou não do sistema). O escopo de um sistema muitas vezes muda durante o seu desenvolvimento. Dessa forma, se o escopo muda, tanto clientes quanto desenvolvedores têm um parâmetro para decidir em que medida os recursos de tempo e financeiros devem mudar. Contudo, o planejamento inicial do projeto deve se basear no escopo inicial.

Outro ponto importante sobre os requisitos é sua característica de *volatilidade*. Um requisito volátil é aquele que pode sofrer modificações durante o desenvolvimento do sistema.⁴ Nos primórdios da modelagem de sistemas, era comum a prática de “congelar” os requisitos levantados antes de se iniciar a construção do sistema. Isto é, os requisitos considerados eram os mesmos do início ao fim do projeto de desenvolvimento. Atualmente, a volatilidade dos requisitos é um fato com o qual a equipe de desenvolvimento de sistemas tem de conviver e consequentemente o congelamento de requisitos é impraticável. Isso porque, nos dias atuais, as organizações precisam se adaptar a mudanças cada vez mais rápidas. Durante o período em que o sistema está em desenvolvimento, diversos aspectos podem ser alterados: as tecnologias utilizadas, as expectativas dos usuários, as regras do negócio etc. E isso para mencionar apenas algumas possibilidades.

Isso parece se contrapor ao fato de que o documento de requisitos deve definir de forma clara quais são os requisitos do sistema. Na realidade, conforme mencionado anteriormente, o documento de requisitos serve como um *consenso inicial*. O ponto principal do levantamento de requisitos é compreender o sistema o máximo possível antes de começar a construí-lo. A regra é definir completamente qualquer requisito já conhecido, mesmo os mais simples. À medida que novos requisitos sejam detectados (ou que requisitos preexistentes mudem), os desenvolvedores devem verificar cuidadosamente o impacto das mudanças resultantes no escopo do sistema. Dessa forma, os clientes podem decidir se tais mudanças devem ser consideradas no desenvolvimento, uma vez que influenciam o cronograma e os recursos financeiros alocados.

A menos que o sistema a ser desenvolvido seja bastante simples e estático (características raras nos sistemas atuais), é praticamente impossível pensar em todos os detalhes a princípio. Além disso, quando o sistema entrar em produção e os usuários começarem a utilizá-lo, eles próprios descobrirão requisitos nos quais não tinham pensado inicialmente. Em resumo, os requisitos de um sistema complexo inevitavelmente mudarão durante o seu desenvolvimento.

No desenvolvimento de sistemas de software, muitas vezes a existência de requisitos voláteis corresponde mais à regra do que à exceção.

Uma característica desejável em um documento de requisitos é ter os seus requisitos ordenados pelos usuários em função do seu grau de *prioridade*. O grau de prioridade de um requisito para os usuários normalmente é estabelecido em função da *adição de valor* que o desenvolvimento desse requisito no sistema trouxer aos usuários. Saber o grau de prioridade de um requisito permite que a equipe de desenvolvimento (mais particularmente o gerente do projeto) decida em que momento cada um deve ser considerado durante o desenvolvimento. As prioridades atribuídas aos requisitos permitirão ao gerente de projeto tomar decisões a respeito do momento no qual cada requisito deve

ser considerado durante o desenvolvimento do sistema.

2.1.2 Análise

De acordo com o professor Wilson de Pádua, **as fases de levantamento de requisitos e de análise de requisitos recebem o nome de *engenharia de requisitos*** (PÁDUA, 2003). Na seção anterior, descrevemos a fase de levantamento de requisitos. Nesta seção, oferecemos uma visão geral da fase de análise de requisitos, também conhecida como *fase de análise*. Formalmente, o termo *análise* corresponde a “quebrar” um sistema em seus componentes e estudar como eles interagem entre si com o objetivo de entender como esse sistema funciona. No contexto dos sistemas de software, esta é a etapa na qual os *analistas* (ver a [Seção 2.2.2](#)) realizam um estudo detalhado dos requisitos levantados na atividade anterior. A partir desse estudo, são construídos *modelos* para representar o sistema a ser construído.

Assim como no levantamento de requisitos, a análise de requisitos não leva em conta o ambiente tecnológico a ser utilizado. Nesta atividade, o foco de interesse é tentar construir uma estratégia de solução sem se preocupar com a maneira *como* essa estratégia será realizada. A razão desta prática é tentar obter a melhor solução para o problema sem se preocupar com os detalhes da tecnologia a ser utilizada. Em outras palavras, é necessário saber *o que* o sistema proposto deve fazer para, então, definir *como* esse sistema irá fazê-lo.

O termo “paralisia da análise” é conhecido no desenvolvimento de sistemas de software. Esse termo denota a situação em que há uma estagnação da fase de análise: os analistas passam muito tempo construindo os modelos do sistema. Embora essa situação certamente exista, na prática raramente podemos encontrá-la. O que costuma ocorrer na prática é exatamente o contrário: equipes de desenvolvimento que passam para a construção da solução sem antes terem definido completamente o problema. Portanto, os modelos construídos na fase de análise devem ser cuidadosamente *validados* e *verificados* pela validação e verificação dos modelos, respectivamente.

O objetivo da *validação* é assegurar que as necessidades do cliente estão sendo atendidas pelo sistema: será que o software correto está sendo construído? Com a validação, os analistas querem se assegurar de que a especificação que construíram do software é correta, consistente, completa, realista e sem ambiguidades. Nessa atividade, os analistas apresentam os modelos criados para representar o sistema aos futuros usuários para que esses modelos sejam validados. Quando um usuário oferece sua validação, quer dizer que entendeu o modelo construído e que, segundo esse entendimento, ele reflete suas necessidades com relação ao sistema a ser desenvolvido. Se um modelo não é bem definido, é possível que tanto usuários quanto desenvolvedores tenham diferentes interpretações acerca do sistema a ser desenvolvido. Essas diferentes interpretações se originam das ambiguidades e contradições em relação ao que usuários e desenvolvedores entendem dos requisitos. Um erro na etapa de levantamento de requisitos, se identificado tarde, implica a construção de um sistema que não corresponde às expectativas do usuário. Nesses casos, o impacto nos custos e prazos do projeto de desenvolvimento pode ser devastador. Portanto, um fator decisivo para o sucesso de um sistema é o envolvimento de especialistas do domínio (ver a [Seção 2.2.6](#)) durante o desenvolvimento. Por isso, a atividade de validação dos requisitos por parte dos usuários é tão importante.

Já a *verificação* tem o objetivo de analisar se os modelos construídos estão em conformidade com os requisitos definidos: será que o software está sendo construído corretamente? Na verificação dos modelos, são analisadas a exatidão de cada modelo em separado e a consistência entre eles.

Diferente da validação (que é uma atividade de análise), a verificação é uma etapa típica da fase de projeto (ver a [Seção 2.1.3](#)).

Em um processo de desenvolvimento orientado a objetos, um dos resultados da análise é o modelo de objetos que representa as classes componentes do sistema. Além disso, a análise também resulta em um modelo funcional do sistema de software a ser desenvolvido.

De acordo com Blaha e Humbaugh (2006), a fase de análise pode ser subdividida em duas subfases: *análise do domínio* (ou *análise do negócio*) e *análise da aplicação*. Descrevemos essas subfases nos próximos parágrafos.

Na análise do domínio, um primeiro objetivo é identificar e modelar os objetos do mundo real que, de alguma forma, serão processados pela aplicação em desenvolvimento. Por exemplo, um aluno é um objeto do mundo real que um sistema de controle acadêmico deve processar. Assim, aluno é um objeto do domínio. Uma característica da análise de domínio é que os objetos identificados fazem sentido para os especialistas por corresponderem a conceitos com os quais esses profissionais lidam diariamente em seu trabalho. Por isso, na análise de domínio, os analistas devem interagir com os especialistas do domínio. Outros exemplos de objetos do domínio “instituição de ensino” são professor, disciplina, turma, sala de aula etc. Outro objetivo da análise do domínio é identificar as *regras do negócio* e os *processos do negócio* realizados pela organização. A análise do domínio é também conhecida como *modelagem do negócio* ou *modelagem dos processos do negócio*.

A análise do domínio normalmente é seguida pela análise da aplicação. A análise da aplicação tem como objetivo identificar objetos de análise que normalmente não fazem sentido para os especialistas do domínio, mas que são necessários para suprir as funcionalidades do sistema em questão. Esses objetos têm a ver com os aspectos computacionais de alto nível da aplicação. Por exemplo, uma *tela de inscrição em disciplinas* é um objeto componente de uma aplicação de controle de uma instituição de ensino. De forma geral, qualquer objeto necessário para que o sistema em desenvolvimento possa se comunicar com seu ambiente é um objeto da aplicação. Objetos da aplicação somente têm sentido no contexto de um sistema de software, diferentemente dos objetos de domínio, que normalmente representam conceitos do domínio do problema. Note que análise da aplicação envolve apenas a *identificação* desses objetos, e *não* o detalhamento deles. Ou seja, não é escopo da análise da aplicação definir a forma como esses objetos serão implementados; isso é escopo da fase de projeto (ver [Seção 2.1.3](#)).

Sendo assim, a análise do domínio identifica objetos do domínio, e a análise da aplicação identifica objetos da aplicação. Mas por que essa separação da análise em duas subfases? Uma razão para isso é o reúso. Na medida em que modelamos o domínio separadamente dos aspectos específicos da aplicação, os componentes resultantes dessa modelagem têm maior potencial de reusabilidade no desenvolvimento de aplicação dentro do mesmo domínio de problema.

Nas últimas décadas, diversas ferramentas de modelagem foram propostas para auxiliar a realização das atividades de análise. De forma geral, cada ferramenta é útil para modelar determinado aspecto de um sistema. É comum utilizarem-se diversas ferramentas para capturar aspectos diferentes de certo problema. As principais ferramentas da UML para realizar análise são o *diagrama de casos de uso* e o *diagrama de classes* (para a modelagem de casos de uso e de classes, respectivamente). Outros diagramas da UML também utilizados na análise são: diagrama de interação ([Capítulo 7](#)), diagrama de estados ([Capítulo 9](#)) e diagrama de atividades ([Capítulo 10](#)).

2.1.3 Projeto (desenho)

O foco principal da análise são os aspectos lógicos e independentes de implementação de um sistema (i.e., os requisitos desse sistema). Na fase de projeto, determina-se “como” o sistema funcionará para atender aos requisitos, de acordo com os recursos tecnológicos existentes (a fase de projeto considera os aspectos físicos e dependentes de implementação). Aos modelos construídos na fase de análise são adicionadas as denominadas “restrições de tecnologia”. Exemplos de aspectos a serem considerados na fase de projeto: arquitetura física do sistema, padrão de interface gráfica, algoritmos específicos, o gerenciador de banco de dados a ser utilizado etc.

A fase de projeto produz uma descrição *computacional* do que o software deve fazer de uma maneira coerente com a descrição feita na análise. Em alguns casos, certas restrições da tecnologia a ser utilizada já foram amarradas no levantamento de requisitos. Por exemplo, pode ser que o sistema deva ser implantado em uma organização em que existe um banco de dados corporativo, que centraliza o armazenamento das informações de diversos sistemas. Em outros casos, essas restrições devem ser especificadas. Mas, em todas as situações, as atividades da fase de projeto do sistema são direcionadas pelos modelos construídos na fase de análise e pelo planejamento do desenvolvimento do sistema.

A fase de projeto consiste em duas atividades principais: *projeto da arquitetura* (também conhecido como *projeto de alto nível*) e *projeto detalhado* (também conhecido como *projeto de baixo nível*). Essas duas atividades são descritas a seguir.

Durante o processo de desenvolvimento de um sistema de software orientado a objetos, o *projeto da arquitetura* consiste em distribuir as classes de objetos relacionadas do sistema em *subsistemas* e seus *componentes*. Consiste também em distribuir esses componentes *fisicamente* pelos recursos de hardware disponíveis. Os diagramas da UML normalmente utilizados nessa fase do projeto são os diagramas de implementação ([Capítulo 11](#)). O projeto da arquitetura é normalmente realizado pelos chamados *arquitetos de software* (consulte a [Seção 2.2.4](#)).

No *projeto detalhado*, são modeladas as colaborações entre os objetos de cada módulo com o objetivo de realizar suas funcionalidades. Também são realizados o projeto da interface com o usuário e o projeto de banco de dados, bem como são considerados aspectos de concorrência e distribuição do sistema. Além disso, aspectos de mapeamento dos modelos de análise para artefatos de software são também considerados na fase de projeto. O projeto dos algoritmos a serem utilizados no sistema também é uma atividade do projeto detalhado. Os diagramas da UML que podem ser utilizados nesta fase de projeto são: diagrama de classes, diagrama de casos de uso, diagrama de interação, diagrama de estados e diagrama de atividades.

Embora a análise e o projeto sejam descritos separadamente neste livro, é importante notar que, durante o desenvolvimento, não há uma distinção assim tão clara entre essas duas fases. As atividades dessas duas fases frequentemente se misturam, principalmente no desenvolvimento de sistemas orientados a objetos (ver [Capítulo 6](#)).

2.1.4 Implementação

Na fase de implementação, o sistema é *codificado*, ou seja, ocorre a tradução da descrição computacional obtida na fase de projeto em código executável mediante o uso de uma ou mais linguagens de programação.

Em um processo de desenvolvimento orientado a objetos, a implementação envolve a criação do código-fonte correspondente às classes de objetos do sistema utilizando linguagens de programação como C#, C++, Java, PHP, Phyton e Ruby etc. Além da codificação desde o início, a implementação

pode também reutilizar componentes de software, bibliotecas de classes e frameworks para agilizar a atividade.

2.1.5 Testes

Diversas atividades de teste são realizadas para verificação do sistema construído, levando-se em conta a especificação feita nas fases de análise e de projeto. Um possível produto dessa fase são os *relatórios de testes*, que apresentam informações sobre erros detectados no software.

Idealmente, testes devem ser realizados de forma hierárquica. Isso quer dizer que partes pequenas do software podem ser testadas em separado e, a seguir, unidas a outras partes, para formar um componente maior. Este, por sua vez, também pode ser testado separadamente, e assim por diante, até a realização de testes que englobem o sistema como um todo. Essa estratégia hierárquica de realização de testes deu origem a uma diversidade de terminologias para denominar cada nível hierárquico de atividades de testes. A nomenclatura não é um consenso (e, por vezes, é contraditória), mas, dentre os termos encontrados nessa terminologia, temos: testes de unidades, testes de integração, testes de sistemas e testes de aceitação. Os próximos parágrafos apresentam uma visão geral desses tipos de testes.

Testes de unidades são realizados sobre elementos do código-fonte do sistema. No contexto do desenvolvimento orientado a objetos, esses elementos correspondem a classes ou mesmo a métodos de uma classe. Uma tendência nos últimos anos é a *automatização* desse tipo de teste. No contexto da orientação a objetos, isso significa construir métodos de teste cuja finalidade é realizar chamadas a métodos de uma classe e, a seguir, averiguar se o valor retornado está em conformidade com o esperado. Testes de unidades normalmente são realizados pelos próprios desenvolvedores. Diversos estudos na engenharia de software que constataram que o custo para a correção de um defeito em um software aumenta na medida em que se distancia (no tempo) do momento em que esse defeito foi introduzido. Nesse contexto, a aplicação dos testes de unidades se justifica por evitar que os momentos entre a codificação de um método e a realização de testes sobre ele fiquem muito distantes.

Os testes de integração são realizados após os testes de unidades. No contexto de um sistema orientado a objetos, o elemento de uma atividade de teste de integração pode ser uma *operação de sistema*. Conforme descrito no [Capítulo 7](#), uma operação de sistema é um serviço fornecido pelo sistema e que pode ser obtido por meio da chamada dessa operação. Do ponto de vista de implementação, uma operação de sistema é o resultado da interação de diversos objetos (cujas classes passaram previamente por testes de unidades). Nos testes de integração, quando uma operação de sistema é testada, também estão sendo testadas indiretamente as interfaces de cada objeto participante, assim como as interações entre eles. Portanto, a finalidade dos testes de integração é detectar possíveis falhas na comunicação entre objetos participantes de uma interação ou na interface de algum deles.

Após a atividade de testes de integração, devem ser realizados os *testes de sistema*. O objetivo desse tipo de teste é verificar se o sistema construído está em conformidade com os requisitos levantados para ele. Por consequência, testes de sistema são baseados na *especificação* do sistema, e não em sua implementação. Se o sistema está sendo desenvolvido de acordo com algum processo incremental, então um momento adequado para a realização dos testes de sistema é o fim de cada iteração incremental, quando uma nova versão do sistema está para ser liberada para uso. Em um teste de sistema, o elemento a ser testado corresponde a um ou mais casos de uso. Testes de sistema devem ser executados em um ambiente o mais similar possível ao de produção. A razão para isso é

fazer com que esses testes reflitam as condições reais de funcionamento do sistema quando ele tiver sido implantado no ambiente de produção.

Testes de unidades, de integração e de sistemas são realizados pela equipe técnica (i.e., pelos próprios desenvolvedores ou por uma equipe independente de testadores). Por outro lado, testes de aceitação são realizados pelos usuários da aplicação. De forma semelhante aos testes de sistema, os de aceitação têm como meta verificar a aderência do sistema desenvolvido aos requisitos previamente definidos.

2.1.6 Implantação

Na fase de implantação, o sistema é empacotado, distribuído e instalado no ambiente do usuário. Os manuais do sistema são escritos, os arquivos são carregados, os dados são importados para o sistema e os usuários treinados para utilizar o sistema corretamente. Em alguns casos, nesse momento também ocorre a migração de sistemas de software e de dados preexistentes.

2.2 O componente humano (participantes do processo)

O desenvolvimento de software é uma tarefa altamente cooperativa. Tecnologias complexas demandam especialistas em áreas específicas. Uma equipe de desenvolvimento de sistemas de software pode envolver vários especialistas, como, por exemplo, profissionais de informática, que forneçam o conhecimento técnico necessário ao desenvolvimento do sistema de software, além de especialistas do domínio para o qual o sistema de software deve ser desenvolvido.

Uma equipe de desenvolvimento de software típica consiste em um gerente, analistas, projetistas, programadores, clientes e grupos de avaliação de qualidade. Esses participantes do processo de desenvolvimento são descritos a seguir. Contudo, é importante notar que a descrição dos participantes do processo tem um fim mais didático. Na prática, a mesma pessoa desempenha diferentes funções e, por outro lado, uma mesma função é normalmente desempenhada por várias pessoas.

Em virtude de seu tamanho e complexidade, o desenvolvimento de sistemas de software é um empreendimento realizado em equipe.

2.2.1 Gerentes de projeto

Como o próprio nome diz, o gerente de projetos é o profissional responsável pela gerência ou coordenação das atividades necessárias à construção do sistema. Esse profissional também é responsável por fazer o orçamento do projeto de desenvolvimento, como, por exemplo, estimar o tempo necessário para o desenvolvimento do sistema, definir qual o processo de desenvolvimento, o cronograma de execução das atividades, a mão de obra especializada, os recursos de hardware e software etc.

O acompanhamento das atividades realizadas durante o desenvolvimento do sistema também é tarefa do gerente do projeto. É sua função verificar se os diversos recursos alocados estão sendo gastos na taxa esperada e, caso contrário, tomar providências para adequação dos gastos.

Questões como identificar se o sistema é factível, escalonar a equipe de desenvolvimento e

definir qual o processo de desenvolvimento a ser utilizado também devem ser cuidadosamente estudadas pelo gerente do projeto.

2.2.2 Analistas

O analista de sistemas é o profissional que deve ter conhecimento do *domínio do negócio* e entender seus problemas para que possa definir os requisitos do sistema a ser desenvolvido. Analistas devem estar aptos a se comunicar com especialistas do domínio para obter conhecimento acerca dos problemas e das necessidades envolvidas na organização empresarial. O analista não precisa ser um especialista. Contudo, deve ter domínio suficiente do vocabulário da área de conhecimento na qual o sistema será implantado para que o especialista de domínio não precise ser interrompido a todo o momento para explicar conceitos básicos da área.

Uma característica do analista de sistemas é ser o profissional responsável por entender as necessidades dos clientes em relação ao sistema a ser desenvolvido e repassar esse entendimento aos demais desenvolvedores do sistema (ver as [Seções 2.1.1](#) e [2.1.2](#)). Neste sentido, o analista de sistemas representa uma ponte de comunicação entre duas “facções”: a dos profissionais de computação e a dos profissionais do negócio.

Para realizar suas funções, o analista deve não só entender do domínio do negócio da organização, mas também ter sólido conhecimento dos aspectos relativos à modelagem de sistemas. Neste sentido, o analista de sistemas funciona como um tradutor, mapeando informações entre duas “linguagens” diferentes: a dos especialistas do domínio e a dos profissionais técnicos da equipe de desenvolvimento. Em alguns casos, há profissionais em uma equipe de desenvolvimento para desempenhar dois papéis distintos: o de *analista de negócios* e o de *analista de sistemas*. O analista de negócios é responsável por entender o que o cliente faz, por que o faz, e determinar se as práticas atuais da organização realmente apresentam lógica. O analista do negócio tem que fazer isso a partir da consideração de diferentes (e muitas vezes conflitantes) perspectivas. Já o analista de sistemas é especializado em traduzir as necessidades do usuário em características de um produto de software.

Graças à experiência adquirida com a participação no desenvolvimento de diversos projetos, alguns analistas se tornam gerentes de projetos. Na verdade, as possibilidades de evolução na carreira de um analista são bastante grandes. Isso se deve ao fato de que, durante a fase de levantamento de requisitos de um sistema, o analista se torna quase um especialista no domínio do negócio da organização. Para algumas organizações, é bastante interessante ter à sua disposição profissionais que entendam ao mesmo tempo de técnicas de desenvolvimento de sistemas e do processo de negócio da empresa. Por essa razão, não é rara a situação em que uma organização oferece um contrato de trabalho ao analista de sistemas no final do desenvolvimento do sistema.

Uma característica importante que um analista deve ter é a capacidade de comunicação, tanto escrita quanto falada, pois ele é um agente facilitador da comunicação entre os clientes e a equipe técnica. Muitas vezes as capacidades de se comunicar agilmente e de ter um bom relacionamento interpessoal são mais importantes para o analista do que o conhecimento tecnológico.

Outra característica necessária a um analista é a ética profissional. Esse profissional está frequentemente em contato com informações sigilosas e estratégicas dentro da organização na qual está trabalhando. Os analistas têm acesso a informações como preços de custo de produtos, margens de lucro aplicadas, algoritmos proprietários etc. Certamente, pode ser desastroso para a organização se informações de caráter confidencial como essas caírem em mãos erradas. Portanto, a ética profissional do analista na manipulação dessas informações é fundamental.

2.2.3 Projetistas

O projetista de sistemas é o integrante da equipe de desenvolvimento cujas funções são: (1) avaliar as alternativas de solução (da definição) do problema resultante da análise; e (2) gerar a especificação de uma solução computacional detalhada. A tarefa do projetista de sistemas é muitas vezes chamada de *projeto físico*.⁵

Na prática, existem diversos tipos de projetistas. Pode-se falar em projetistas de interface (especializados nos padrões de uma interface gráfica, como o *Windows* ou o *MacOS*), de redes (especializados no projeto de redes de comunicação), de bancos de dados (especializados no projeto de bancos de dados) e assim por diante. O ponto comum a todos eles é que todos trabalham nos modelos resultantes da análise para adicionar os aspectos tecnológicos a tais modelos.

2.2.4 Arquitetos de software

Um profissional encontrado principalmente em grandes equipes reunidas para desenvolver sistemas complexos é o arquiteto de software. O objetivo desse profissional é elaborar a arquitetura do sistema como um todo. É ele quem toma decisões sobre quais subsistemas compõem o sistema como um todo e quais são as interfaces entre esses subsistemas.

Além de tomar decisões globais, o arquiteto também deve ser capaz de tomar decisões técnicas detalhadas (p. ex., decisões que têm influência no desempenho do sistema). Esse profissional trabalha em conjunto com o gerente de projeto para priorizar e organizar o plano de projeto.

2.2.5 Programadores

São os responsáveis pela *implementação do sistema*. É comum haver vários programadores em uma equipe de desenvolvimento. Um programador pode ser proficiente em uma ou mais linguagens de programação, além de ter conhecimento sobre bancos de dados e poder ler os modelos resultantes do trabalho do projetista.

Na verdade, a maioria das equipes de desenvolvimento possui analistas que realizam alguma programação e programadores que realizam alguma análise. No entanto, para fins didáticos, podemos enumerar algumas diferenças entre as atividades desempenhadas por esses profissionais. Em primeiro lugar, o analista de sistemas está envolvido em todas as etapas do desenvolvimento, diferentemente do programador, que participa unicamente das fases finais (implementação e testes). Outra diferença é que analistas de sistemas devem entender tanto de tecnologia de informação quanto do processo de negócio; programadores tendem a se preocupar somente com os aspectos tecnológicos do desenvolvimento.

É comum bons programadores serem “promovidos” a analistas de sistemas. Essa é uma prática recorrente nas empresas de desenvolvimento de software, baseada na falsa lógica de que bons programadores serão bons analistas de sistemas. A verdade é que não se pode ter certeza de que um bom programador será um bom analista. Além disso, um programador não tão talentoso pode se tornar um ótimo analista. De fato, uma crescente percentagem de analistas sendo formados tem uma formação anterior diferente de computação. Por outro lado, um analista de sistemas deve ter algum conhecimento de programação para que produza especificações técnicas de um processo de negócio a serem passadas a um programador.

2.2.6 Especialistas do domínio

Outro componente da equipe de desenvolvimento é o *especialista do domínio*, também conhecido como *especialista do negócio*. Esse componente é o indivíduo, ou um grupo deles, que possui conhecimento acerca da área ou do negócio em que o sistema em desenvolvimento estará inserido. Um termo mais amplo que especialista de domínio é *cliente*. Podem-se distinguir dois tipos de clientes: o *cliente usuário* e o *cliente contratante*. O cliente usuário é o indivíduo que efetivamente utilizará o sistema. Ele normalmente é um especialista do domínio. É com esse tipo de cliente que o analista de sistemas interage para levantar os requisitos do sistema. O cliente contratante é o indivíduo que solicita o desenvolvimento do sistema. Ou seja, é quem encomenda e patrocina os custos de desenvolvimento e manutenção.

Em pequenas organizações, o cliente usuário e o cliente proprietário são a mesma pessoa. Já em grandes organizações, o cliente proprietário faz parte da gerência e é responsável por tomadas de decisões estratégicas dentro da empresa, enquanto o cliente usuário é o responsável pela realização de processos operacionais.

Há casos em que o produto de software não é encomendado por um cliente. Em vez disso, o produto é desenvolvido para posteriormente ser comercializado. Esses produtos são normalmente direcionados para o mercado de massa. Exemplos de produtos de software nessa categoria são: processadores de texto, editores gráficos, jogos eletrônicos etc. Nesses casos, a equipe de desenvolvimento trabalha com o pessoal de marketing como se estes fossem seus clientes reais.

Em qualquer caso, a participação do cliente usuário no processo de desenvolvimento é de suma importância. O distanciamento de usuários durante esse período se manifesta, sobretudo, em projetos que excedem o seu orçamento, estão atrasados ou que não correspondem às reais necessidades. Mesmo que o analista entenda exatamente o que o usuário necessitava inicialmente, se o sistema construído não contemplar as necessidades atuais desse cliente, ainda será um sistema inútil. A única maneira de se ter um usuário satisfeito com o sistema de informações é torná-lo um legítimo participante no desenvolvimento do sistema.

Não importa o processo de desenvolvimento utilizado; o envolvimento do especialista do domínio no desenvolvimento de um sistema de software é de fundamental importância.

2.2.7 Avaliadores de qualidade

O desempenho e a confiabilidade são exemplos de características que devem ser encontradas em um sistema de software de boa qualidade. Avaliadores de qualidade asseguram a adequação do processo de desenvolvimento e do produto de software sendo desenvolvido aos padrões de qualidade estabelecidos pela organização.

2.3 Modelos de ciclo de vida

O desenvolvimento de um sistema envolve diversas fases, descritas na [Seção 2.1](#). A um encadeamento específico dessas fases para a construção do sistema dá-se o nome de Modelo de Ciclo de Vida. Há diversos Modelos de Ciclo de Vida. A diferença entre um e outro está na maneira como as várias fases são encadeadas. Nesta seção, dois modelos de ciclo de vida de sistema são descritos: o *modelo em cascata* e o *modelo iterativo e incremental*.

2.3.1 O modelo de ciclo de vida em cascata

Este ciclo de vida também é chamado de *clássico* ou *linear* e se caracteriza por possuir uma tendência na progressão sequencial entre uma fase e a seguinte. Eventualmente, pode haver uma retroalimentação de uma fase para a anterior, mas, de um ponto de vista macro, as fases seguem sequencialmente (ver [Figura 2-1](#)).

Há diversos problemas associados a esse tipo de ciclo de vida, todos provenientes de sua característica principal: a sequencialidade das fases. A seguir são descritos alguns desses problemas:

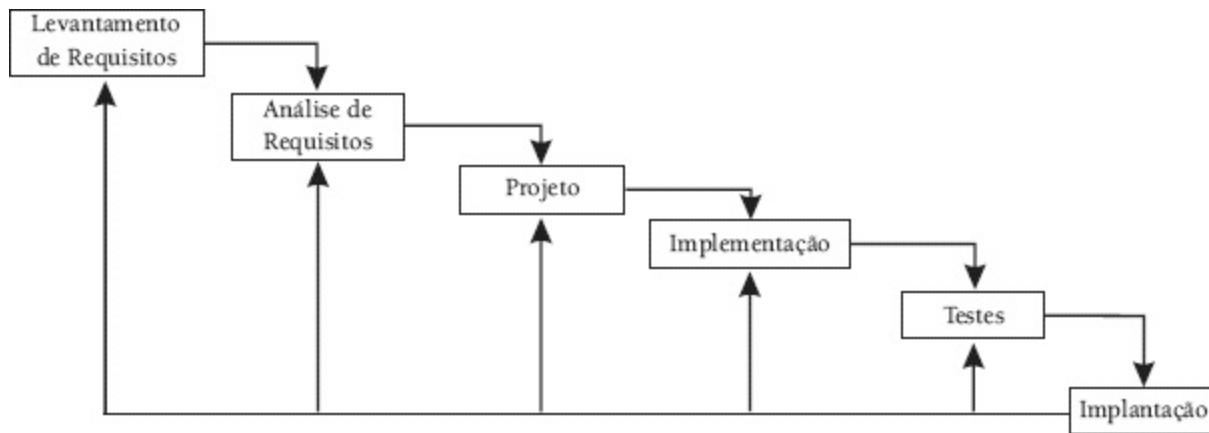


Figura 2-1: A abordagem de desenvolvimento de software em cascata.

1. Projetos de desenvolvimento reais raramente seguem o fluxo sequencial que esse modelo propõe. Tipicamente, algumas atividades de desenvolvimento podem ser realizadas em paralelo.
2. A abordagem clássica presume que é possível declarar detalhadamente todos os requisitos ant do início das demais fases do desenvolvimento (veja a discussão sobre requisitos voláteis na [Seção 2.1.1](#)). Nessa hipótese é possível que recursos sejam desperdiçados na construção de um requisito incorreto. Tal falha pode se propagar por todas as fases do processo, sendo detectada apenas quando o usuário começar a utilizar o sistema.
3. Uma *versão de produção*⁶ do sistema não estará pronta até que o ciclo do projeto de desenvolvimento chegue ao final. Como as fases são realizadas sequencialmente, a implantação do sistema pode ficar muito distanciada no tempo da fase inicial em que o sistema foi “encomendado”. Sistemas muito complexos podem levar meses ou até anos para serem desenvolvidos. Nesses tempos de grande concorrência entre as empresas, é difícil pensar que o usuário espere pacientemente até que o sistema todo esteja pronto para ser utilizado. Mesmo se isso acontecer, pode haver o risco de que o sistema já não corresponda mais às reais necessidades de seus usuários, em virtude de os requisitos terem mudado durante o tempo de desenvolvimento.

Apesar de todos os seus problemas, o modelo de ciclo de vida em cascata foi utilizado durante muitos anos (juntamente com o paradigma de modelagem estruturada). Atualmente, devido à complexidade cada vez maior dos sistemas, esse modelo de ciclo de vida é pouco utilizado. Os modelos de ciclo de vida mais utilizados para o desenvolvimento de sistemas complexos hoje em dia são os que usam a abordagem incremental e iterativa, descrita a seguir.

2.3.2 O modelo de ciclo de vida iterativo e incremental

O modelo de ciclo de vida incremental e iterativo foi proposto como uma resposta aos problemas encontrados no modelo em cascata. Um processo de desenvolvimento utilizando essa abordagem divide o desenvolvimento de um produto de software em ciclos. Em cada ciclo dessa etapa podem ser identificadas as fases de análise, projeto, implementação e testes. Essa característica contrasta com a abordagem clássica, na qual as fases de análise, projeto, implementação e testes são realizadas uma única vez.

Cada um dos ciclos considera um subconjunto de requisitos. Os requisitos são desenvolvidos uma vez que sejam alocados a um ciclo de desenvolvimento. No próximo ciclo, outro subconjunto dos requisitos é considerado para ser desenvolvido, o que produz um novo incremento do sistema que contém extensões e refinamentos sobre o incremento anterior. **Assim, o desenvolvimento evolui em versões, ao longo da construção incremental e iterativa de novas funcionalidades, até que o sistema completo esteja construído.** Note que apenas uma parte dos requisitos é considerada em cada ciclo de desenvolvimento. Na verdade, um modelo de ciclo de vida iterativo e incremental pode ser visto como uma generalização da abordagem em cascata: o software é desenvolvido em incrementos, que por sua vez seguem o formato em cascata (ver [Figura 2-2](#)).

A abordagem incremental e iterativa somente é possível se existir um mecanismo para dividir os requisitos do sistema em partes, para que cada parte seja alocada a um ciclo de desenvolvimento. Essa alocação é realizada em função do grau de importância atribuído a cada requisito. Os fatores considerados nessa divisão são a *prioridade* (importância do requisito para o cliente; ver [Seção 2.1.1](#)) e o *risco* de cada requisito. É função do gerente de projeto alocar os requisitos aos ciclos de desenvolvimento. Na [Seção 4.6](#), é descrita uma maneira indireta de alocar os requisitos aos ciclos de desenvolvimento, por meio dos *casos de uso* do sistema.

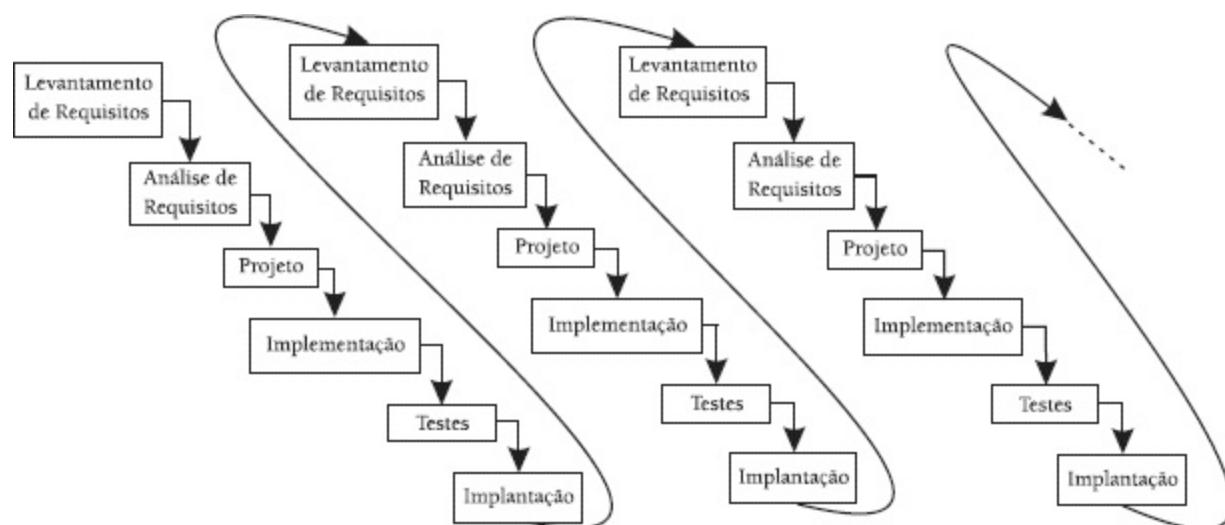


Figura 2-2: No processo incremental e iterativo, cada iteração é uma “minicascata”.

No modelo de ciclo de vida incremental e iterativo, um sistema de software é desenvolvido em vários passos similares (portanto, *iterativo*). Em cada passo, o sistema é estendido com mais funcionalidades (logo, *incremental*).

A abordagem incremental incentiva a participação do usuário nas atividades de desenvolvimento do sistema, o que diminui em muito a probabilidade de interpretações erradas em relação aos

requisitos levantados.

Vários autores consideram uma desvantagem da abordagem incremental e iterativa o usuário se entusiasmar excessivamente com a primeira versão do sistema e pensar que ela já corresponde ao sistema finalizado. De qualquer forma, o fato de essa abordagem incentivar a participação do usuário no processo de desenvolvimento de longe compensa qualquer falsa expectativa que ele possa criar sobre o sistema.

Outra vantagem dessa abordagem é que os *riscos* do projeto podem ser mais bem gerenciados. Um risco de desenvolvimento é a possibilidade de ocorrência de algum evento que cause prejuízo ao processo de desenvolvimento, juntamente com as consequências desse prejuízo. O prejuízo pode incorrer na alteração de diversos parâmetros do desenvolvimento, como custos do projeto, cronograma, qualidade do produto, satisfação do cliente etc. Exemplos de riscos inerentes ao desenvolvimento de software:

- O projeto pode não satisfazer aos requisitos do usuário.
- A verba do projeto pode acabar.
- O produto de software pode não ser adaptável, manutenível ou extensível.
- O produto de software pode ser entregue ao usuário tarde demais.

Um consenso geral em relação ao desenvolvimento de software é que os riscos de projeto não podem ser eliminados por completo. Portanto, todo processo de desenvolvimento deve levar em conta a probabilidade de ocorrência de riscos. Na abordagem incremental, os requisitos mais arriscados são os primeiros a serem considerados. Visto que cada ciclo de desenvolvimento gera um incremento do sistema que é liberado para o usuário, inconsistências entre os requisitos considerados no ciclo e sua implementação se tornam evidentes mais cedo no desenvolvimento. Se as inconsistências identificadas não são tão graves, tanto melhor: elas são removidas e uma nova versão do sistema é entregue ao usuário. Por outro lado, se as inconsistências descobertas são graves e têm um impacto grande no desenvolvimento do sistema, pelo menos a sua identificação torna possível reagir a elas mais cedo sem muitas consequências graves para o projeto.

Os primeiros requisitos a serem considerados devem ser selecionados com base nos riscos que fornecem. Os requisitos mais arriscados devem ser considerados o mais cedo possível.

Para entender o motivo de o conjunto de requisitos mais arriscados ser considerado o mais cedo possível, vamos lembrar de uma frase do consultor Tom Gilb: “Se você não atacar os riscos [do projeto]ativamente, então eles irãoativamente atacar você” (GILB e FINZ, 1988). Ou seja, quanto mais inicialmente a equipe de desenvolvimento considerar os requisitos mais arriscados, menor é a probabilidade de ocorrerem prejuízos devido a esses requisitos.

Uma desvantagem do desenvolvimento incremental e iterativo é que a tarefa do gerente do projeto fica bem mais difícil. Gerenciar um processo de desenvolvimento em que as fases de análise, projeto e implementação, testes e implantação ocorrem em paralelo é de fato consideravelmente mais complicado do que gerenciar o desenvolvimento de um sistema que utilize a abordagem clássica.

2.3.2.1 Organização geral de um processo incremental e iterativo

O ciclo de vida de processo incremental e iterativo pode ser estudado segundo duas dimensões: **dimensão temporal** e **dimensão de atividades** (ou de fluxos de trabalho). A [Figura 2-3](#) ilustra essas duas dimensões.

Na dimensão temporal, o processo está estruturado em fases. Em cada uma dessas fases há uma ou mais *iterações*. **Cada iteração tem uma duração preestabelecida (de duas a seis semanas).** Ao final de cada uma delas é produzido um *incremento*, ou seja, uma parte do sistema final. Um incremento pode ser liberado para os usuários, ou pode ser somente interno.

A dimensão de atividades compreende aquelas realizadas durante a iteração de uma fase: levantamento de requisitos, análise de requisitos, projeto, implementação, testes e implantação (as mesmas atividades descritas na [Seção 2.1](#)). Essas atividades são apresentadas verticalmente na [Figura 2-3](#).

Em cada uma das fases diferentes artefatos de software são produzidos, ou artefatos começados em uma fase anterior são estendidos com novos detalhes. Cada fase é concluída com um *marco* (mostrado na parte superior da [Figura 2-3](#)). **Um marco é um ponto do desenvolvimento no qual decisões sobre o projeto são tomadas e importantes objetivos são alcançados.** Os marcos são úteis para o gerente de projeto estimar os gastos e o andamento do cronograma de desenvolvimento.

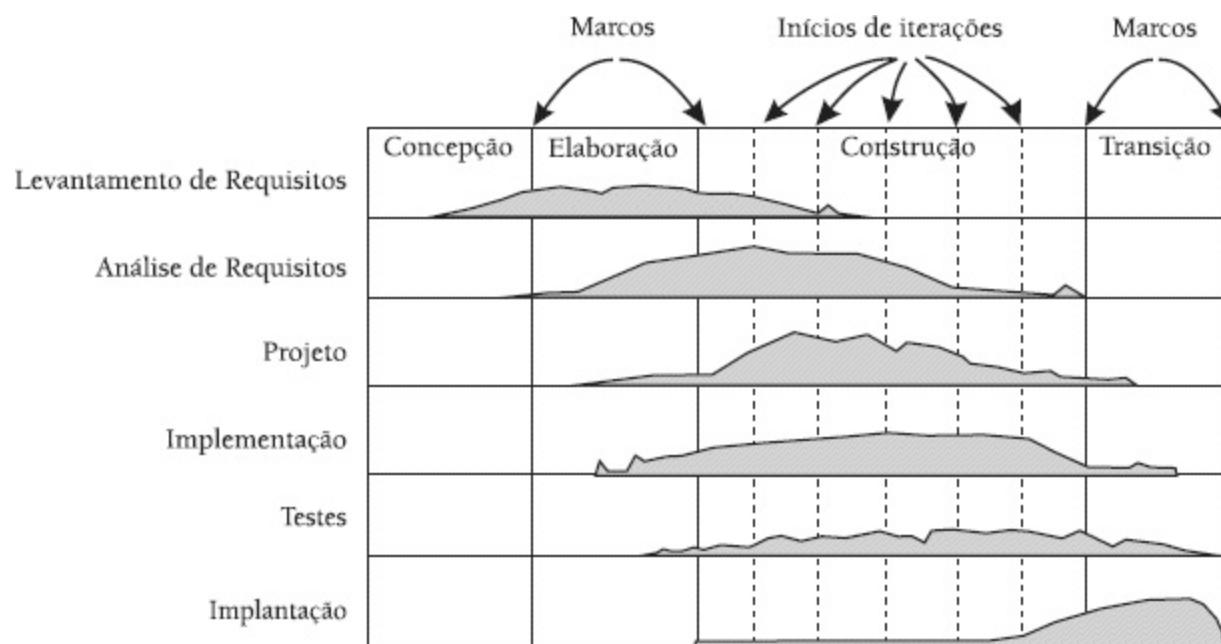


Figura 2-3: Estrutura geral de um processo de desenvolvimento incremental e iterativo.

As fases do processo unificado delimitadas pelos marcos são as seguintes: concepção, elaboração, construção e transição. Essas fases são descritas a seguir:

- Na *concepção*, a ideia geral e o escopo do desenvolvimento são desenvolvidos. Um planejamento de alto nível do desenvolvimento é realizado. São determinados os marcos que separam as fases.
- Na fase de *elaboração*, é alcançado um entendimento inicial sobre como o sistema será construído. O planejamento do projeto de desenvolvimento é completado. Nessa fase, o domínio do negócio é analisado. Os requisitos do sistema são ordenados considerando-se prioridade e risco. Nessa fase, também são planejadas as iterações da próxima fase, a de construção. Isso envolve definir a duração de cada iteração e o que será desenvolvido em cada iteração.

- Na *construção*, as atividades de análise e projeto aumentam em comparação com as demais. Esta é a fase na qual ocorrem mais iterações incrementais. No final dessa fase, decide-se se o produto de software pode ser entregue aos usuários sem que o projeto seja exposto a altos riscos. Se este for o caso, tem início a construção do manual do usuário e a descrição dos incrementos realizados no sistema.
- Na *transição*, os usuários são treinados para utilizar o sistema. Questões de instalação e configuração do sistema também são tratadas. Ao final desta fase, a aceitação do usuário e os gastos são avaliados. Uma vez que o sistema é entregue aos usuários, provavelmente surgem novas questões que demandam a construção de novas versões do mesmo. Se este for o caso, um novo ciclo de desenvolvimento pode ser iniciado.

Em cada iteração, uma proporção maior ou menor de cada uma dessas atividades é realizada, dependendo da fase em que se encontra o desenvolvimento. Por exemplo, pela Figura 2-3 percebemos que, na fase de transição, a atividade de implantação é a predominante. Por outro lado, na fase de construção, as atividades de análise, projeto e implementação são as predominantes. Normalmente a fase de construção é a que possui mais iterações. No entanto, as demais fases também podem conter iterações, dependendo da complexidade do sistema.

O principal representante da abordagem de desenvolvimento incremental e iterativa é o denominado *Processo Unificado Racional* (*Rational Unified Process, RUP*). Esse processo de desenvolvimento é patenteado pela empresa Rational, na qual trabalham os “três amigos”, Jacobson, Booch e Rumbaugh (em 2002, a IBM comprou a Rational). A descrição feita nesta seção é uma versão simplificada do Processo Unificado. Maiores detalhes sobre o Processo Unificado podem ser obtidos em RUP (2002).

2.4 Utilização da UML no processo iterativo e incremental

Na Seção 1.4, a UML é descrita como uma linguagem de modelagem independente do processo de desenvolvimento. Ou seja, vários processos de desenvolvimento podem utilizar a UML como ferramenta para construção dos modelos de um sistema de software orientado a objetos.

Em um modelo de ciclo de vida iterativo e incremental, os artefatos de software construídos com uso da UML evoluem à medida que as iterações do processo são realizadas. A cada iteração novos detalhes são adicionados a esses artefatos. Além disso, a construção de cada artefato não é isolada. Em vez disso, a construção de um artefato fornece informações para adicionar detalhes a outros.

Os próximos capítulos deste livro descrevem os diagramas da UML e a construção de modelos, considerando-se a utilização de um processo iterativo e incremental. Para cada modelo descrito, é apresentada a sua inserção no contexto do processo de desenvolvimento como um todo.

2.5 Prototipagem

Uma técnica que serve de complemento à análise de requisitos é a construção de protótipos. No contexto do desenvolvimento de softwares, **um protótipo é um esboço de alguma parte do sistema**. A construção de protótipos é comumente chamada de *prototipagem*.

Protótipos podem ser construídos para telas de entrada, telas de saída, subsistemas ou mesmo

para o sistema como um todo. A construção de protótipos utiliza as denominadas linguagens de programação visual. Exemplos são o *Delphi*, o *PowerBuilder*, o *Visual Basic* e o *Front Page* (para construção de interface WEB), que, na verdade, são ambientes com facilidades para a construção da interface gráfica (telas, formulários etc.). Além disso, muitos sistemas de gerência de bancos de dados também fornecem ferramentas para a construção de telas de entrada e saída de dados. Note, entretanto, que protótipos não precisam necessariamente ser construídos para aspectos da interface gráfica com o usuário. Em vez disso, qualquer aspecto que precise ser mais bem entendido é um alvo em potencial de prototipagem pela equipe de desenvolvimento.

Na prototipagem, após o levantamento de requisitos, um protótipo do sistema é construído para ser usado na validação, quando o protótipo é revisto por um ou mais usuários que fazem críticas a respeito de uma ou outra característica. O protótipo é então corrigido ou refinado com base nessas observações. Esse processo de revisão e refinamento continua até o protótipo ser aceito pelos usuários. Portanto, a técnica de prototipagem tem o objetivo de assegurar que os requisitos do sistema foram realmente bem entendidos. O resultado da validação pelo protótipo pode ser usado para refinar os modelos do sistema. Após a aceitação, o protótipo (ou parte dele) pode ser descartado ou utilizado como uma versão inicial do sistema.

Embora a técnica de prototipagem seja opcional, ela costuma ser aplicada em projetos de desenvolvimento de software, especialmente quando há dificuldades no entendimento dos requisitos do sistema, ou há requisitos arriscados que precisam ser mais bem entendidos. A ideia é que um protótipo é mais concreto para fins de validação do que modelos representados por diagramas bidimensionais. Isso incentiva a participação ativa do usuário na validação. Consequentemente, a tarefa de validação se torna menos suscetível a erros. No entanto, alguns desenvolvedores usam essa técnica como substituta à construção de modelos do sistema. Tenha em mente que a prototipagem é uma técnica *complementar* à construção dos modelos do sistema. Os modelos do sistema devem ser construídos para guiar as demais fases do projeto de desenvolvimento de software. O ideal é que os erros detectados na validação do protótipo sejam utilizados para modificar e refinar os modelos do sistema.

2.6 Ferramentas CASE

Um processo de desenvolvimento de software é muito complexo. Várias pessoas com diferentes especialidades estão envolvidas nesse processo altamente cooperativo. Essa atividade pode ser facilitada pelo uso de ferramentas que auxiliam na construção de modelos do sistema, na integração do trabalho de cada membro da equipe, no gerenciamento do andamento do desenvolvimento etc.

Existem sistemas de software que são utilizados para dar suporte ao ciclo de vida de desenvolvimento de um sistema. Dois tipos de software com esse objetivo são as *ferramentas CASE* e os ambientes de desenvolvimento. Uma discussão detalhada a respeito desses sistemas de apoio ao desenvolvimento está fora do escopo deste livro. Descrevemos sucintamente esses dois tipos de software a seguir.

O termo CASE é uma sigla em inglês para *Computer Aided Software Engineering (Engenharia de Software Auxiliada por Computador)*. A utilização dessa sigla já se consolidou no Brasil. Existem diversas ferramentas CASE disponíveis no mercado. Não está no escopo deste livro detalhar as funcionalidades dessas ferramentas. No entanto, são descritas sucintamente a seguir algumas de suas características:

- Criação de diagramas e manutenção da consistência entre os mesmos. É comum a quase toda as ferramentas CASE a possibilidade de produzir a perspectiva gráfica dos modelos de software. Com relação à manutenção desses diagramas, um padrão que vem ganhando importância nos últimos anos é o XMI (*XML Metadata Interchange*). O XMI é um padrão baseado em XML para exportar modelos definidos em UML. Esse padrão é importante, pois permite a interoperabilidade entre diversas ferramentas CASE: um diagrama produzido em uma ferramenta A pode ser importado para uma ferramenta B de outro fabricante, considerando que os fabricantes das ferramentas A e B dão suporte ao padrão XMI.
- Manutenção da consistência entre os modelos gerados: outra funcionalidade frequentemente encontrada em ferramentas CASE é a que permite verificar a validade de um conjunto de modelos e a consistência entre os mesmos.
- Engenharia *Round-Trip* (*Round-Trip Engineering*): denomina-se engenharia *round-trip* à capacidade de uma ferramenta CASE interagir com o código-fonte do sistema em desenvolvimento. Há dois tipos de engenharia *round-trip*: *direta* e *reversa*. A engenharia direta corresponde à possibilidade de geração de código-fonte a partir de diagramas produzidos com a ferramenta CASE em questão. A engenharia reversa corresponde ao processo inverso, ou seja, a geração de diagramas a partir de código-fonte preexistente.
- Rastreamento de requisitos: uma praticidade importante em um processo de desenvolvimento é a possibilidade de rastreamento de um requisito. Rastrear um requisito significa poder localizar os artefatos de software gerados como consequência da existência daquele requisito. Essa funcionalidade é importante quando um requisito é alterado; nesse caso, todos os artefatos correspondentes devem ser revisados.

Além das ferramentas CASE, um segundo tipo de software de suporte ao desenvolvimento são os ambientes de desenvolvimento integrado (*Integrated Development Environment, IDE*). Esses ambientes possibilitam a codificação (implementação) do sistema, além de fornecer diversas facilidades, algumas delas listadas a seguir:

- Depuração de código-fonte: capacidade dos ambientes de desenvolvimento que permite ao programador encontrar erros de lógica em partes de um programa.
- Verificação de erros em tempo de execução: é comum nos ambientes de desenvolvimento modernos a facilidade de compilação do programa em paralelo à escrita do mesmo. Com este recurso, o programador é notificado a respeito de um erro em alguma linha de código logo após ter passado para a construção da próxima instrução.
- Refatoração: corresponde a alguma técnica de alteração do código-fonte de uma aplicação de tal forma que não altere o comportamento da mesma, mas, em vez disso, melhore a qualidade de sua implementação e consequentemente torne mais fácil a manutenção. É comum em ambientes de desenvolvimento integrado modernos a existência de facilidades para que o programador realize refatorações no código-fonte de sua aplicação.

Além das ferramentas CASE e dos ambientes de desenvolvimento integrado, outras ferramentas são importantes em um processo de desenvolvimento. A seguir, listamos alguns exemplos de facilidades encontradas em ferramentas de suporte ao desenvolvimento atuais:

- Relatórios de cobertura de testes: ferramentas que geram relatórios informando sobre partes de um programa que não foram testadas.
- Gerenciamento de versões: ferramentas que permitem gerenciar as diversas versões dos artefatos de software gerados durante o ciclo de vida de um sistema.
- Suporte à definição de testes automatizados: ferramentas que realizam testes automáticos no sistema.
- Monitoração e averiguação do desempenho: averiguar o tempo de execução de módulos de um sistema, assim como o tráfego de dados em sistemas em rede.
- Tarefas de gerenciamento: ferramentas que fornecem ao gerente de projetos (ver [Seção 2.2.1](#)) funcionalidades para suporte a algumas de suas atribuições: desenvolvimento de cronogramas de tarefas, alocações de recursos de mão de obra, monitoração do progresso das atividades e dos gastos etc.

► EXERCÍCIOS

2-1: O que os seguintes termos significam. Como eles se relacionam uns com os outros?

- Análise e Projeto
- Análise e Projeto Orientados a Objetos
- UML

2-2: Em 1957, um matemático chamado George Polya descreveu um conjunto de passos genéricos para a resolução de um problema (POLYA, 1957). Estes passos são descritos sucintamente a seguir:

- a. Compreensão do problema
- b. Construção de uma estratégia para resolver o problema
- c. Execução da estratégia
- d. Revisão da solução encontrada

Reflita sobre a seguinte afirmação: o processo de desenvolvimento de um sistema de software pode ser visto como um processo de resolução de um problema.

2-3: Uma teoria da Física relativamente nova é a chamada *Teoria do Caos*. Entre outras afirmações surpreendentes, essa teoria afirma que *uma borboleta voando sobre o Oceano Pacífico pode causar uma tempestade no Oceano Atlântico*. Ou seja, eventos aparentemente irrelevantes podem levar a consequências realmente significativas. Discuta com um analista de sistemas as consequências de pequenas falhas na fase de levantamento em relação a fases posteriores do desenvolvimento de um sistema de software.

2-4: Com base em sua experiência, tente escrever um documento de requisitos para um *sistema de controle acadêmico*. Esse sistema deve controlar as inscrições de alunos em disciplinas, a distribuição das turmas, salas, professores etc. Deve permitir também o controle de notas atribuídas aos alunos em diversas disciplinas. Você pode se basear na forma de funcionamento da sua própria faculdade.

2-5: Com base em sua experiência, tente escrever um documento de requisitos para um sistema de software do seu cotidiano (p. ex., um sistema para automatizar algum processo na empresa em que trabalha, aproveitando o conhecimento do domínio do negócio que você tiver). Durante a elaboração desse documento, resista o máximo possível à tentação de considerar detalhes técnicos e de implementação.

-
1. Em outras literaturas, o leitor pode encontrar os termos *análise de requisitos* ou *projeto lógico* para denotar essa fase de definição e compreensão do problema.
 2. Neste livro o termo *domínio* também é utilizado como sinônimo para domínio do negócio.
 3. Um especialista do domínio é uma pessoa que tem familiaridade com o domínio do negócio, mas não necessariamente com o desenvolvimento de sistemas de software. Frequentemente, esses especialistas são os futuros usuários do sistema de software em desenvolvimento.
 4. A característica de volatilidade também pode ser aplicada à declaração de requisitos como um todo para denotar que, durante o desenvolvimento de um produto de software, novos requisitos podem aparecer ou requisitos preexistentes podem não ser mais necessários.
 5. Note que o termo *projeto* tem diferentes interpretações no contexto do desenvolvimento de sistemas de software, podendo significar o conjunto de atividades para o desenvolvimento de um sistema de software. No entanto, *projeto* também pode significar uma das atividades desse desenvolvimento. No decorrer deste livro, o termo *projeto de desenvolvimento* é utilizado para denotar o primeiro significado. O termo *projeto* é utilizado para denotar o segundo significado.
 6. Uma versão de produção (em contraposição ao termo “versão de desenvolvimento”) de um software é uma versão que pode ser utilizada pelo usuário.

Mecanismos gerais

Podemos apenas ver uma curta distância à frente, mas podemos ver que há muito lá a ser feito.
– ALAN TURING

AUML consiste em três grandes componentes: blocos de construção básicos, regras que restringem como esses blocos podem ser associados e mecanismos de uso geral. Este capítulo descreve os mecanismos de uso geral, pelo fato de eles poderem ser utilizados na maioria dos diagramas da UML que são apresentados nos capítulos seguintes. A descrição aqui apresentada é apenas introdutória. Detalhes e exemplos sobre os mecanismos de uso geral encontram-se nos demais capítulos do livro, quando for necessário usar esses mecanismos.

3.1 Estereótipos

Um estereótipo é um dos mecanismos de uso geral da UML, que é utilizado para estender o significado de determinado elemento em um diagrama. A UML predefine diversos estereótipos. Alguns deles aparecem nos capítulos seguintes deste livro. A UML também permite que o usuário defina os estereótipos a serem utilizados em determinada situação de modelagem. Ou seja, a própria equipe de desenvolvimento pode definir os estereótipos que serão utilizados em situações específicas. Dessa forma, podemos ter estereótipos de dois tipos: predefinidos ou definidos pela equipe de desenvolvimento.

Os estereótipos definidos pela própria equipe de desenvolvimento devem ser documentados de tal forma que a sua semântica seja entendida sem ambiguidades por toda a equipe. Além disso, um estereótipo definido pelo usuário deve ser utilizado de forma consistente na modelagem de todo o sistema. Ou seja, não é correto utilizar um mesmo estereótipo para denotar diferentes significados.

Outra classificação que pode ser aplicada aos estereótipos (tanto os predefinidos quanto os definidos pela equipe de desenvolvimento) é quanto à sua forma: *estereótipos gráficos* (ou ícones) e *textuais*.

Um estereótipo gráfico é representado por um ícone que lembre o significado do conceito ao qual ele está associado. Por exemplo: a [Figura 3-1](#) ilustra quatro estereótipos gráficos. Os dois mais à esquerda (Ator e Componente) são predefinidos na UML. Já os dois elementos gráficos mais à direita (Servidor HTTP e Portal de Segurança) são exemplos de estereótipos definidos pela equipe de desenvolvimento. Note que o ícone escolhido para esses estereótipos é coerente com o significado do conceito que eles representam.

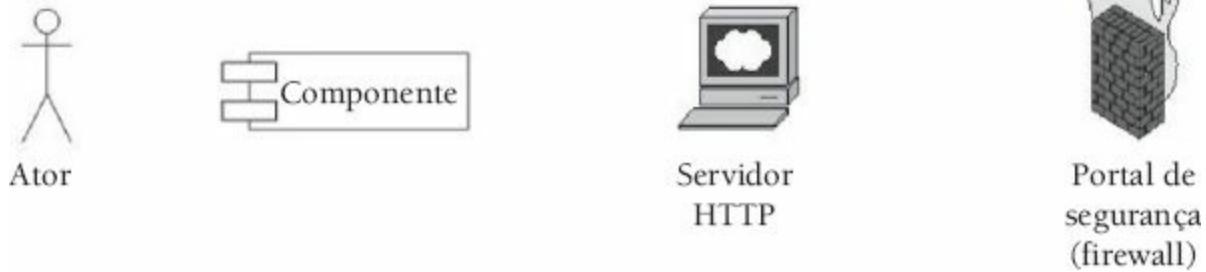


Figura 3-1: Exemplos de estereótipos gráficos.

Um estereótipo de rótulo é representado por um nome delimitado pelos símbolos << e >> (esses símbolos são chamados de *aspas francesas*), e posicionado próximo ao símbolo. Os estereótipos <<document>>, <<interface>>, <<control>> e <<entity>> são exemplos de estereótipos textuais predefinidos da UML. Nesse sentido, os estereótipos permitem estender a UML e adaptar o seu uso em diversos casos.

3.2 Notas explicativas

Notas explicativas são utilizadas para definir uma informação que comenta ou esclarece alguma parte de um diagrama. Podem ser descritas em texto livre; também podem corresponder a uma expressão formal utilizando a linguagem de restrição de objetos da UML, a OCL (ver [Seção 3.4](#)).

Graficamente, as notas são representadas por um retângulo com uma “orelha”. O conteúdo da nota é inserido no interior do retângulo e este é ligado ao elemento que se quer esclarecer ou comentar por meio de uma linha tracejada. A [Figura 3-2](#) ilustra a utilização de notas explicativas. Como a figura sugere, as notas devem ser posicionadas próximo aos elementos que elas descrevem.

É importante notar que, ao contrário dos estereótipos, as notas textuais não modificam nem estendem o significado do elemento ao qual estão associadas. Elas servem somente para explicar algum elemento do modelo sem modificar sua estrutura ou semântica.

Notas explicativas devem ser utilizadas com cuidado. Embora ajudem a explicar certos elementos de um diagrama, sua utilização em excesso torna os modelos gráficos “carregados” visualmente. Além disso, com a evolução dos modelos durante o desenvolvimento do sistema, algumas notas podem deixar de fazer sentido, gerando, assim, a sobrecarga de atualização das mesmas. A ideia é utilizar notas explicativas quando realmente for necessário.

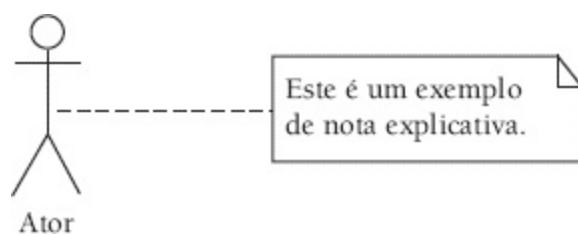


Figura 3-2: Exemplo de nota explicativa.

3.3 Etiquetas valoradas (*tagged values*)

Os elementos gráficos de um diagrama da UML possuem propriedades predefinidas. Por exemplo,

uma classe tem três propriedades predefinidas: um nome, uma lista de atributos e uma lista de operações. Além das propriedades predefinidas, podem-se também definir outras propriedades para determinados elementos de um diagrama através do mecanismo de *etiquetas valoradas*. Na UML 2.0, uma etiqueta valorada somente pode ser utilizada como um atributo definido sobre um estereótipo (que pode ser predefinido ou definido pelo usuário). Dessa forma, um determinado elemento de um modelo deve primeiramente ser estendido por um estereótipo e só depois por uma etiqueta valorada.

Tabela 3-1: Alternativas para definição de etiquetas (tags) na UML

```
{ tag = valor }
{ tag1 = valor1, tag2 = valor2 ... }
{ tag }
```

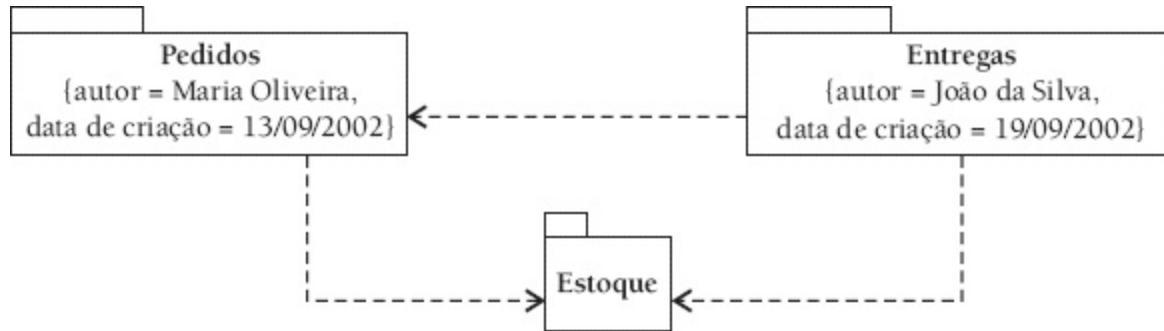


Figura 3-3: Utilização de etiquetas.

Uma etiqueta pode ser definida utilizando-se uma das três forma alternativas apresentadas na [Tabela 3-1](#). As chaves fazem parte da sintaxe. Por exemplo, uma etiqueta pode ser utilizada para informar o autor e a data de criação de um determinado diagrama ou elemento de um diagrama. A [Figura 3-3](#) ilustra essa situação.

3.4 Restrições

A todo elemento da UML está associada alguma semântica. Isso quer dizer que cada elemento gráfico dessa linguagem possui um significado bem definido que, uma vez entendido, fica implícito na utilização do elemento em algum diagrama. As restrições permitem estender ou alterar a semântica natural de um elemento gráfico. Esse mecanismo geral especifica restrições sobre um ou mais valores de um ou mais elementos de um modelo. Restrições podem ser especificadas tanto formal quanto informalmente. A especificação formal de restrições se dá pela OCL (ver [Seção 3.6](#)). Além de poderem ser especificadas com o uso da OCL, restrições também podem ser definidas informalmente pelo texto livre (linguagem natural). Assim como para as etiquetas, uma restrição, seja ela formal ou informal, também deve ser delimitada por chaves. Essas restrições devem aparecer dentro de *notas explicativas* (ver [Seção 3.2](#)).

3.5 Pacotes

Um pacote é um mecanismo de *agrupamento* definido pela UML. Esse mecanismo pode ser utilizado para agrupar elementos semanticamente relacionados. A notação para um pacote é a de uma pasta com uma aba, conforme mostra a [Figura 3-3](#), e pacotes têm nomes.

As ligações entre pacotes na [Figura 3-3](#) são relacionamentos de dependência. Um pacote P_1 depende de outro P_2 se algum elemento contido em P_1 depende de algum elemento contido em P_2 . O significado específico dessa dependência pode ser definido pela própria equipe de desenvolvimento com o uso de *estereótipos*.

Um pacote constitui um mecanismo de agrupamento genérico. Sendo assim, ele pode ser utilizado para agrupar quaisquer outros elementos, inclusive outros pacotes. Na [Seção 4.4.1](#) e na [Seção 11.1](#) apresentamos mais detalhes sobre pacotes, nos contextos do *modelo de casos de uso* e da *arquitetura lógica do sistema*, respectivamente.

Em relação ao conteúdo de um pacote, há duas maneiras de representá-lo graficamente. A primeira é exibir o conteúdo dentro do pacote. A segunda é “pendurar” os elementos agrupados no ícone do pacote. A [Figura 3-4](#) ilustra essas duas formas de representação.

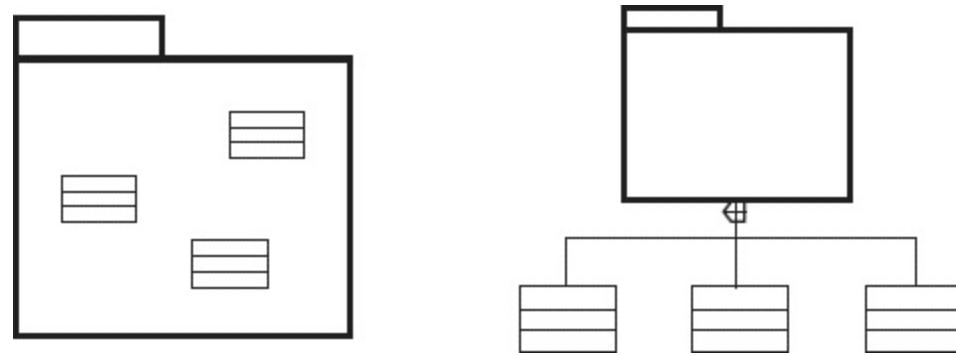


Figura 3-4: Formas de representar o conteúdo de um pacote.

Conforme mostra a [Figura 3-5](#), os pacotes podem ser agrupados dentro de outros pacotes, formando uma hierarquia de contenção.

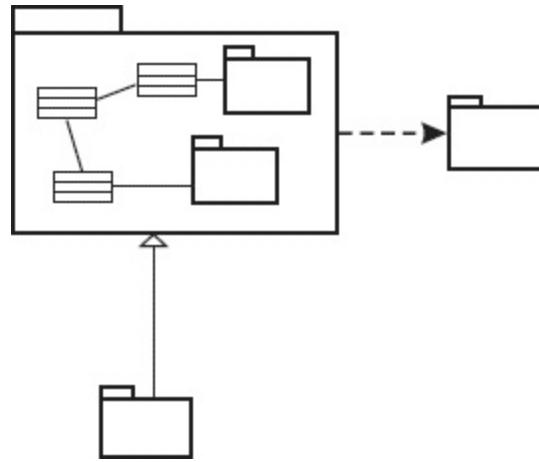


Figura 3-5: Pacotes podem conter outros pacotes.

3.6 OCL

A UML define uma linguagem formal que pode ser utilizada para especificar restrições sobre diversos elementos de um modelo. Essa linguagem se chama OCL, a *Linguagem de Restrição de UML*.

Objetos. A OCL pode ser utilizada para definir expressões de navegação entre objetos, expressões lógicas, precondições, pós-condições etc.

A maioria das declarações em OCL consiste nos seguintes elementos estruturais: *contexto*, *propriedade* e *operação*. Um contexto define o domínio no qual a declaração em OCL se aplica. Por exemplo, uma classe ou uma instância de uma classe. Em uma expressão OCL, a propriedade corresponde a algum componente do contexto. Pode ser, por exemplo, o nome de um atributo em uma classe, ou uma associação entre dois objetos. Finalmente a operação define o que deve ser aplicado sobre a propriedade. Uma operação pode envolver operadores aritméticos, operadores de conjunto e operadores de tipo. Outros operadores que podem ser utilizados em uma expressão OCL são: *and*, *or*, *implies*, *if*, *then*, *else*, *not*, *in*.

A OCL pode ser utilizada em qualquer diagrama da UML. Para uma descrição detalhada dessa linguagem recomendamos o livro *UML – Guia do Usuário* (BOOCH *et al.*, 2006). No entanto, durante as descrições dos diagramas da UML em outros capítulos deste livro, fornecemos alguns exemplos de expressões em OCL.

Modelagem de casos de uso

Não diga pouco em muitas palavras, mas sim muito em poucas.
– PITÁGORAS

O modelo de casos de uso (MCU) é uma representação das *funcionalidades* externamente observáveis do sistema e dos *elementos externos* ao sistema que interagem com ele. O MCU é um modelo de análise (ver [Seção 2.1.2](#)) que representa um refinamento dos *requisitos funcionais* (ver [Seção 2.1.1](#)) do sistema em desenvolvimento. A ferramenta da UML utilizada na modelagem de casos de uso é o *diagrama de casos de uso*.

A técnica de modelagem de casos de uso foi idealizada por um conceituado engenheiro de software sueco, Ivar Jacobson, na década de 1970, enquanto trabalhava no desenvolvimento de um sistema na empresa de telefonia Ericsson. Mais tarde, Jacobson incorporou essa técnica a um processo de desenvolvimento de software denominado *Objectory* (JACOBSON *et al.*, 1992). Posteriormente, ele se uniu a Grady Booch e a James Rumbaugh, e a notação de casos de uso foi incorporada à UML. Desde então, esse modelo vem se tornando cada vez mais popular para realizar a documentação de requisitos funcionais de uma aplicação, devido à sua notação gráfica simples e descrição em linguagem natural, o que facilita a comunicação entre a equipe técnica e os especialistas do domínio.

O MCU é importante, pois direciona diversas tarefas posteriores do processo de desenvolvimento de um sistema de software. Além disso, esse modelo força os desenvolvedores a moldar o sistema de acordo com as necessidades do usuário.

Neste capítulo, apresentamos a atividade de modelagem de casos de uso. Nossa objetivo aqui é apresentar os principais componentes de um MCU, assim como descrever dicas de modelagem úteis na criação e documentação do mesmo. Além disso, também descrevemos a documentação suplementar que normalmente deve ser criada e associada ao modelo de casos de uso. Este capítulo está organizado da seguinte forma: apresentamos os elementos constituintes de um MCU na [Seção 4.1](#). Na [Seção 4.2](#), apresentamos os elementos de notação definidos na UML para construção de diagramas de casos de uso. Na [Seção 4.3](#), apresentamos técnicas para identificação dos elementos do MCU. Na [Seção 4.4](#), descrevemos de que forma o MCU para um SSOO pode ser construído, uma vez que foram identificados seus elementos constituintes. Na [Seção 4.5](#), apresentamos detalhes acerca de artefatos de modelagem suplementares ao MCU. Na [Seção 4.6](#), apresentamos o contexto do MCU em um processo de desenvolvimento iterativo. Finalmente, na [Seção 4.7](#) iniciamos a apresentação da modelagem de nosso estudo de caso, o Sistema de Controle Acadêmico. Na maioria dos capítulos subsequentes, apresentamos uma seção que continua a modelagem iniciada nesta [Seção 4.7](#).

4.1 Modelo de casos de uso

O MCU representa os possíveis usos de um sistema da maneira como são percebidos por um observador externo a este sistema. Cada um desses usos está associado a um ou mais requisitos funcionais identificados para o sistema. A construção desse modelo envolve a definição de diversos componentes: *casos de uso*, *atores* e *relacionamentos* entre eles. Vamos descrever esses componentes separadamente nas próximas seções.

4.1.1 Casos de uso

Por definição, um caso de uso (do inglês *use case*) é a especificação de uma sequência completa de interações entre um sistema e um ou mais agentes externos a esse sistema. Um caso de uso representa um relato de uso de certa funcionalidade do sistema em questão, *sem revelar a estrutura e o comportamento internos desse sistema*. Essa última característica de um caso de uso é importante, porque sua existência implica que o MCU é um modelo com uma perspectiva *externa* do sistema. Graças ao estudo do modelo de casos de uso de um sistema, um observador sabe quais são as funcionalidades fornecidas pelo sistema em questão e quais são os resultados externos produzidos pelas mesmas. No entanto, esse observador não sabe, simplesmente com base no modelo de casos de uso, como esse sistema age internamente para produzir os resultados externamente visíveis.

É importante enfatizar o uso da palavra “completa” na definição de caso de uso que demos no parágrafo anterior. O uso dessa palavra é para enfatizar que um caso de uso não é um passo em uma funcionalidade do sistema. Ao contrário, um caso de uso é um relato detalhado de um dos usos do sistema por um agente externo. Outro exemplo para esclarecer esse ponto: *entrar no sistema* não é um caso de uso em si, visto que é de esperar que o usuário entre no sistema para alcançar outro objetivo mais útil do que só olhar a sua tela inicial. Nessa situação, este outro objetivo é que é o verdadeiro caso de uso.

Um modelo de casos de uso típico contém vários desses casos. A quantidade exata de casos de uso obviamente depende da complexidade do sistema em desenvolvimento: quanto mais complexo o sistema, maior a quantidade de casos de uso.

Um caso de uso representa uma determinada funcionalidade de um sistema conforme percebida externamente. Representa também os agentes externos que *interagem* com o sistema. Um caso de uso, entretanto, não revela a estrutura e o comportamento internos do sistema.

Cada caso de uso de um sistema se define pela descrição narrativa (textual) das interações que ocorrem entre o(s) elemento(s) externo(s) e o sistema. A UML não define uma estrutura textual a ser utilizada na descrição de um caso de uso. Consequentemente, há vários estilos de descrição propostos para definir casos de uso. A escolha de um ou de outro estilo fica a cargo da equipe de desenvolvimento, ou então pode ser uma restrição definida pelos clientes que encomendaram o sistema.

Podemos dizer que há três dimensões em que o estilo de descrição de um caso de uso pode variar. Essas dimensões são o formato, o grau de detalhamento e o grau de abstração. Como mostra a [Figura 4-1](#), essas são dimensões (escolhas) de descrição independentes entre si e, por conta disso, podem ser escolhidas de forma independente. Passamos agora a descrever mais detalhadamente cada uma dessas dimensões.

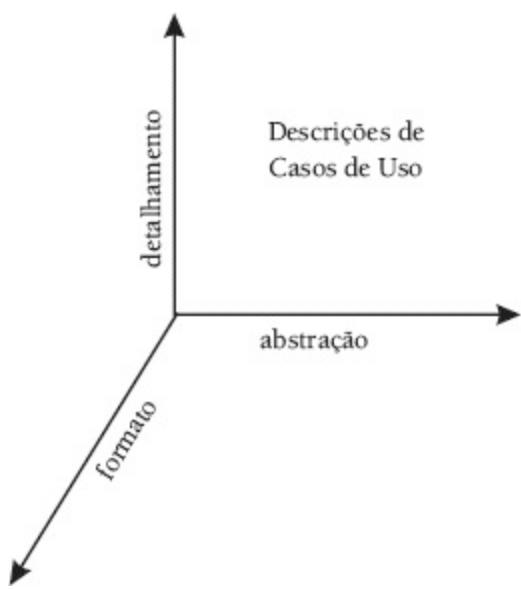


Figura 4-1: Independência entre formato, grau de abstração e detalhamento de um caso de uso.

4.1.1.1 Formato

O formato de uma descrição de caso de uso diz respeito à estrutura utilizada para organizar a sua narrativa textual. Os formatos comumente utilizados são o *contínuo*, o *numerado* e o *tabular*. Esses três formatos são apresentados a seguir. Nesta explicação, é utilizado um exemplo correspondente ao caso de uso de saque de determinada quantia em um caixa eletrônico de um sistema bancário.

O formato contínuo foi introduzido por Ivar Jacobson e seus colaboradores. Nesse formato, a narrativa acontece por texto livre. Como um exemplo desse tipo de formato, considere o caso de uso Realizar Saque em um caixa eletrônico. A descrição contínua deste caso de uso é fornecida no [Quadro 4-1](#).

Quadro 4-1: Exemplo de descrição contínua

Este caso de uso inicia quando o cliente chega ao caixa eletrônico e insere seu cartão. O sistema requisita a senha do cliente. Após o cliente fornecer sua senha e esta ser validada, o sistema exibe as opções de operações possíveis. O cliente opta por realizar um saque. Então o sistema requisita o total a ser sacado. O cliente fornece o valor da quantidade que deseja sacar. O sistema fornece a quantia desejada e imprime o recibo para o cliente. O cliente retira a quantia e o recibo, e o caso de uso termina.

No formato numerado, a narrativa é descrita por uma série de passos numerados. Considere como exemplo o mesmo caso de uso Realizar Saque. Sua descrição no formato numerado é fornecida no [Quadro 4-2](#).

Quadro 4-2: Exemplo de descrição numerada

- 1) Cliente insere seu cartão no caixa eletrônico.
- 2) Sistema apresenta solicitação de senha.
- 3) Cliente digita senha.
- 4) Sistema valida a senha e exibe menu de operações disponíveis.
- 5) Cliente indica que deseja realizar um saque.
- 6) Sistema requisita o valor da quantia a ser sacada.
- 7) Cliente fornece o valor da quantia que deseja sacar.
- 8) Sistema fornece a quantia desejada e imprime o recibo para o cliente.
- 9) Cliente retira a quantia e o recibo, e o caso de uso termina.

O formato tabular tenta prover alguma estrutura à descrição de casos de uso. Esse estilo foi proposto por Rebecca Wirfs-Brock e outros (1991). Nesse estilo, a sequência de interações entre o ator e o sistema é fragmentada em duas colunas de uma tabela. Uma das colunas apresenta as ações do ator, e a outra apresenta as reações do sistema. Essa forma de estruturação da narrativa tem o objetivo de separar as ações do ator e as reações do sistema. A leitura de um caso de uso estruturado segundo esse formato pode ser feita em zigue-zague. O caso de uso para saque utilizando o formato tabular é exibido no [Quadro 4-3](#). Note que as ações ou reações podem ser numeradas.

Quadro 4-3: Exemplo de narrativa fragmentada

Cliente	Sistema
Insere seu cartão no caixa eletrônico.	Apresenta solicitação de senha.
Digita senha.	Valida senha e exibe menu de operações disponíveis.
Solicita realização de saque.	Requisita a quantia a ser sacada.
Fornece o valor da quantia que deseja sacar.	Fornece a quantia desejada e imprime o recibo para o cliente
Retira a quantia e o recibo.	

Alguns autores também preconizam a utilização de uma narrativa semelhante ao português estruturado, na qual são apresentadas construções parecidas com as das linguagens de programação estruturada. A esse respeito, é importante notar que a descrição de um caso de uso deve ser legível para o usuário final. Isso porque o MCU é um dos modelos utilizados na fase de validação do sistema (ver [Seção 2.1.2](#)). Por conta disso, a descrição de um caso de uso deve ser a mais clara possível, o que se contrapõe diretamente à ideia de descrições de casos de uso semelhantes a códigos-fonte em linguagens de programação.

4.1.1.2 Grau de detalhamento

O grau de detalhamento a ser utilizado na descrição de um caso de uso pode variar desde o mais sucinto até a descrição com vários detalhes (*expandido*). Um caso de uso sucinto descreve as interações entre ator e sistema sem muitos detalhes. Um caso de uso expandido descreve as interações em detalhes. (Mais informações sobre o detalhamento expandido estão na [Seção 4.4](#).)

4.1.1.3 Grau de abstração

O grau de abstração de um caso de uso diz respeito à existência ou não de menção a aspectos relativos à tecnologia durante a descrição desse caso de uso. Em relação ao grau de abstração, um caso de uso pode ser *real* ou *essencial*. Um caso de uso essencial é abstrato no sentido de *não* fazer menção a aspectos relativos à tecnologia utilizada nas interações entre ator e casos de uso. Esse estilo de descrição foi introduzida por Larry Constantine e Lucy Lockwood (CONSTANTINE e LOCKWOOD, 1999).

Um exemplo de caso de uso essencial é apresentado no [Quadro 4-4](#). (Embora esse exemplo utilize o formato numerado, um caso de uso essencial pode ser descrito em qualquer um dos formatos apresentados na [Seção 4.1.1.1](#).) Note que o exemplo de caso de uso essencial é completamente

desprovido de características tecnológicas.

Quadro 4-4: Exemplo de descrição essencial (e numerada)

- 1) Cliente fornece sua identificação.
- 2) Sistema identifica o usuário.
- 3) Sistema fornece opções disponíveis para movimentação da conta.
- 4) Cliente solicita o saque de determinada quantia.
- 5) Sistema requisita o valor da quantia a ser sacada.
- 6) Cliente fornece o valor da quantia que deseja sacar.
- 5) Sistema fornece a quantia desejada.
- 6) Cliente retira dinheiro e recibo, e o caso de uso termina.

Em um caso de uso cujo grau de abstração é real, a descrição das interações cita detalhes da tecnologia a ser utilizada na interação entre o ator e o sistema. Com efeito, a descrição em um grau de abstração real se compromete com a solução de projeto (tecnologia) a ser utilizada para implementar o caso de uso. Os casos de uso apresentados como exemplos na [Seção 4.1.1.1](#) são todos reais.

É uma boa ideia utilizar a *regra prática dos cem anos* para identificar se um caso de uso contém algum detalhe de tecnologia: pergunte-se, ao ler a narrativa do caso de uso, se a mesma seria válida tanto há cem anos, quanto daqui a cem anos. Se a resposta para sua pergunta for um “sim”, então isso é um indício de que se trata de um caso de uso essencial.¹ Caso contrário, trata-se de um caso de uso real. Por exemplo, no passo 1 do caso de uso apresentado no [Quadro 4-4](#) há menção a uma tecnologia de interação específica, o cartão magnético, bastante comum nos dias de hoje, o que caracteriza tal caso de uso como real. (Certamente, a tecnologia de cartões magnéticos não existia há cem anos, e provavelmente não mais existirá daqui a cem anos.)

4.1.1.4 Cenários

Geralmente a funcionalidade de um sistema descrita por um caso de uso pode ser utilizada de diversas maneiras. Um *cenário* é a descrição de uma das maneiras pelas quais um caso de uso pode ser utilizado. Outra maneira de ver um cenário é como a descrição de um episódio de utilização de alguma funcionalidade do sistema. Um cenário também é chamado de *instância* de um caso de uso. Normalmente há diversos cenários para um mesmo caso de uso. Como exemplo, considere um caso de uso que representa a funcionalidade para realizar um pedido de compra pela Internet. A descrição de um cenário para esse caso de uso é apresentada no [Quadro 4-5](#).

Quadro 4-5: Exemplo de cenário para um caso de pedido de compra pela Internet

- O cliente seleciona um conjunto de produtos do catálogo da loja.
- Após selecionar os produtos que deseja comprar, o cliente indica o desejo de realizar o pagamento por cartão de crédito.
- O sistema informa que o último produto escolhido está fora de estoque.
- O cliente pede para que o sistema feche o pedido sem o item que está fora de estoque.
- O sistema solicita os dados do cliente para realização do pagamento.
- O cliente fornece o número do cartão, a data de expiração, além de informar o endereço de entrega do pedido.
- O sistema apresenta o valor total, a data de entrega e uma identificação do pedido para futuro rastreamento.
- O sistema também envia um correio eletrônico para o cliente como confirmação do pedido de compra.
- O sistema envia os dados do pedido para o sistema de logística da empresa.

Uma analogia válida para entender a relação entre caso de uso e cenário é a de um labirinto. Em um labirinto, temos geralmente diversas maneiras para chegar a uma determinada saída a partir de uma determinada entrada. De forma análoga, podemos comparar o labirinto ao caso de uso. Por outro lado, podemos comparar um cenário a cada uma das possíveis maneiras de atravessar o “caso de uso”.

Uma coleção de cenários para um caso de uso pode ser utilizada posteriormente na fase de testes (ver [Seção 2.1.5](#)), quando o caso de uso estiver sendo testado para verificar a existência de erros na implementação do sistema. Outro uso importante dos cenários está no esclarecimento e no entendimento dos casos de uso dos quais eles são instanciados. É comum durante a construção de um cenário novos detalhes do caso de uso serem identificados, ou mesmo que isso aconteça durante esse processo. Por exemplo, no cenário apresentado no [Quadro 4-5](#), o que acontece se o cliente sair do sistema antes de completar a realização de seu pedido? E se o cartão de crédito do cliente não for aceito? O sistema de logística é um ator do sistema?

Um conceito associado ao de cenário é o de *realização de um caso de uso*. Denominamos realização de um caso de uso ao conjunto de diagramas e artefatos textuais que especificam como esse caso de uso é executado internamente ao sistema em desenvolvimento. Em um sistema de software orientado a objetos, a realização de um caso de uso normalmente apresenta colaborações entre objetos de determinadas classes. Essas colaborações têm o objetivo de produzir o resultado externamente visível do caso de uso. Normalmente, um modelador especifica diversas colaborações para um caso de uso, um para cada cenário considerado relevante. O detalhamento da realização de cenários de um caso de uso *não* faz parte da modelagem de casos de uso. Deixamos o tratamento desse assunto para o [Capítulo 7](#), no qual se faz uma descrição da utilização de cenários para modelar as colaborações (interações) entre objetos na realização de um determinado caso de uso.

Um cenário é uma utilização específica de um caso de uso pelo ator envolvido; pode ser visto como uma *instância* de um caso de uso.

4.1.2 Atores

Na terminologia da UML, qualquer elemento *externo* ao sistema que *interage* com o mesmo é, por definição, denominado *ator*. O termo “externo” nessa definição indica que atores *não* fazem parte do sistema. Já “*interage*” significa que um ator troca informações com o sistema (envia informações para o sistema processar, ou recebe informações processadas provenientes do sistema). Atores representam a forma pela qual um sistema percebe o seu ambiente.

Atores de um sistema podem ser agrupados em diversas categorias. As categorias de atores, junto com exemplos de cada uma, são listadas a seguir:

1. *Cargos* (p. ex., Empregado, Cliente, Gerente, Almoxarife, Vendedor etc.).
2. *Organizações ou divisões de uma organização* (p. ex., Empresa Fornecedor, Agência de Impostos, Administradora de Cartões, Almoxarifado etc.).
3. *Outros sistemas de software* (p. ex., Sistema de Cobrança, Sistema de Estoque de Produtos etc.).
4. *Equipamentos* com os quais o sistema deve se comunicar (p. ex., Leitora de Código de Barras Sensor etc.).

Para todas as categorias de atores listadas anteriormente, os casos de uso representam alguma forma de interação, no sentido de troca de informações entre o sistema e o ator. Normalmente o ator inicia a sequência de interações correspondente a um caso de uso. Uma situação menos frequente, mas também possível, é um evento interno acontecer para que a sequência de interações do caso de uso em questão seja acionada (ver [Seção 4.3.2.1](#)).

Um aspecto importante a ser notado é que um ator corresponde a um *papel* representado em relação ao sistema. Por exemplo, a mesma pessoa pode agir (desempenhar um papel) como um ator em um momento e como outro ator em outro momento. Para ser mais específico, em um sistema de vendas de livros via Internet, um indivíduo pode desempenhar o papel de Cliente que compra mercadorias, assim como pode também assumir o papel de Agendador, o funcionário da loja que agenda a entrega de encomendas. Outro exemplo: uma pessoa pode representar o papel de Funcionário de uma instituição bancária que faz a manutenção de um caixa eletrônico (para reabastecê-lo de numerário, por exemplo). Em outra situação de uso, essa mesma pessoa pode ser o Cliente do banco que realiza o saque de uma quantia no caixa eletrônico.

Levando em conta que um ator é um papel representado por algo (ou alguém) em relação ao sistema, é uma boa prática de modelagem fazer com que o nome dado a esse ator lembre o seu papel, em vez de lembrar quem o representa. Exemplos de bons nomes para atores: Cliente, Estudante, Fornecedor etc. Exemplos de maus nomes para atores: João, Fornecedora, ACME etc.

Um ator pode participar de muitos casos de uso (de fato, essa situação é comum na prática). Do mesmo modo, um caso de uso pode envolver a participação de vários atores, o que resulta na classificação dos atores em *primários* ou *secundários*. Um *ator primário* é aquele que inicia uma sequência de interações de um caso de uso. São eles os agentes externos para os quais o caso de uso traz benefício direto. As funcionalidades principais do sistema são definidas tendo em mente os objetivos dos atores primários. Já um *ator secundário* supervisiona, opera, mantém ou auxilia na utilização do sistema pelo atores primários. Atores secundários existem apenas para que os atores primários possam utilizar o sistema. Por exemplo, considere um programa para navegar na Internet (ou seja, um navegador ou *browser*). Para que o Usuário (ator primário) requisite uma página ao programa, outro ator (secundário) está envolvido: o Servidor Web. Note que o ator primário Usuário é de certa forma auxiliado pelo secundário, Servidor Web, uma vez que é através deste último que o primeiro consegue alcançar seu objetivo (visualizar uma página da Internet). Outro exemplo pode ser encontrado no cenário de caso de uso do [Quadro 4-5](#), no qual Cliente é o ator primário e Sistema de Logística é o ator secundário.

4.1.3 Relacionamentos

Casos de uso e atores não existem sozinhos. Além desses últimos, o modelo de casos de uso possui um terceiro componente, cuja função é relacionar os atores e casos de uso. Esse componente corresponde aos relacionamentos. Sendo assim, um ator deve estar relacionado a um ou mais casos de uso do sistema. Além disso, pode haver relacionamentos entre os casos de uso ou entre os atores de um sistema. A UML define os seguintes relacionamentos para o modelo de casos de uso: *comunicação*, *inclusão*, *extensão* e *generalização*. Os significados (semântica) desses relacionamentos são discutidos nas próximas seções. Antes de passar às próximas seções, entretanto, é bom enfatizar que todos esses relacionamentos possuem uma notação gráfica definida pela UML. Deixamos a apresentação dessas notações para a [Seção 4.2](#).

4.1.3.1 Relacionamento de comunicação

Um relacionamento de comunicação informa a que caso e uso o ator está associado. O fato de um ator estar associado a um caso de uso por meio de um relacionamento de comunicação significa que esse ator interage (troca informações) com o sistema com ajuda daquele caso de uso. Um ator pode se relacionar com mais de um caso de uso do sistema. O relacionamento de comunicação é, de longe, o mais comumente utilizado de todos.

4.1.3.2 Relacionamento de inclusão

O relacionamento de inclusão existe somente entre casos de uso. Para entender o princípio deste relacionamento, é útil uma analogia com um conceito comum em linguagens de programação: a *rotina*. Em uma linguagem de programação, uma sequência de instruções pode ser agrupada em uma unidade lógica chamada rotina. Quando a execução dessa sequência de instruções for necessária, a rotina é chamada de algum ponto do programa. O mecanismo de empacotar uma sequência de instruções em uma rotina evita a repetição dessa sequência em todo lugar do programa em que ela se faz necessária. Na verdade, sempre que a sequência de instruções deve ser executada, a rotina correspondente é chamada.

O princípio subjacente ao relacionamento de inclusão entre casos de uso é o mesmo utilizado no mecanismo de definição de rotinas em linguagens de programação. Quando dois ou mais casos de uso incluem uma sequência comum de interações, essa sequência comum pode ser descrita em outro caso de uso. A partir daí, vários casos de uso do sistema podem *incluir* o comportamento desse caso de uso comum. Isso evita a repetição da descrição de uma mesma sequência de interações e, consequentemente, torna a descrição dos casos de uso como um todo mais simples e de manutenção mais fácil. Como terminologia, o *caso de uso inclusor* é aquele que inclui o comportamento de outro caso de uso; já o *caso de uso incluso* é aquele cujo comportamento é incluído por outros.

Como exemplo de utilização do relacionamento de inclusão, considere um sistema de controle de transações bancárias. Alguns casos de uso desse sistema são Obter Extrato, Realizar Saque e Realizar Transferência. Esses casos de uso têm uma sequência de interações em comum: a que valida a senha do cliente do banco. Essa sequência de interações em comum pode ser descrita em um caso de uso Fornecer Identificação. Dessa forma, todos os casos de uso que utilizam essa sequência de interações podem fazer referência ao caso de uso Fornecer Identificação por meio do relacionamento de inclusão.

Um aspecto relevante acerca do relacionamento de inclusão é relativo à forma de referenciar o caso de uso incluso no caso de uso inclusor. A UML não especifica uma forma padrão de referência. Consequentemente, cada modelador é livre para definir sua forma de fazer referência a um caso de uso incluso. Por outro, é um consenso o fato de que é na descrição do caso de uso inclusor que deve ser feita uma referência para o caso de uso incluso. Recomendamos a utilização da seguinte sintaxe para referência na descrição do caso de uso inclusor: *Include (nome do caso de uso incluso)*. Essa sintaxe indica que no ponto da descrição em que for encontrada, a sequência de interações do caso de uso incluso é inserida (inclusa) no caso de uso inclusor.

4.1.3.3 Relacionamento de extensão

Considere dois casos de uso, A e B. Um relacionamento de extensão de A para B indica que um ou mais dos cenários de B podem incluir o comportamento especificado por A. Nesse caso, diz-se que

A estende B. O caso de uso B é chamado de *estendido*, e o caso de uso A de *extensor*.

O relacionamento de extensão é utilizado para modelar situações em que diferentes sequências de interações podem ser inseridas em um mesmo caso de uso. Cada uma dessas diferentes sequências representa um comportamento *eventual*, ou seja, um comportamento que só ocorre sob certas condições, ou cuja realização depende da escolha do ator. Na realidade, quando o relacionamento de extensão é utilizado de forma adequada, a descrição do caso de uso estendido não deve aparentar falta de completude, ou seja, essa descrição não deve dar a entender que deve (necessariamente) existir uma extensão em seu comportamento. Para concluir o raciocínio: a existência do caso de uso estendido deve ser *independente* da existência de quaisquer casos de uso que estendam o primeiro.

Quando um ator opta por executar a sequência de interações definida no extensor (ou quando certa condição se torna verdadeira), este é acionado. Após a sua execução, o fluxo de interações volta ao caso de uso estendido, recomeçando logo após o ponto em que o extensor foi inserido.

É importante notar que o caso de uso estendido é uma descrição *completa* de uma sequência de interações, com significado em si mesma. Isso quer dizer que não necessariamente o comportamento definido pelo caso de uso extensor é utilizado pelo caso de uso estendido; pode haver cenários em que apenas o comportamento do caso de uso estendido é executado, sem que o de qualquer extensor seja acionado. O que desencadeia a execução do comportamento definido pelo extensor é a ocorrência de alguma condição ou a solicitação explícita do ator.

Uma questão importante diz respeito ao modo como é feita a definição dos pontos de extensão, ou seja, os locais da descrição do caso de uso estendido em que o comportamento do extensor pode ser inserido. O comum é que essa definição seja feita na descrição textual do caso de uso extensor. A vantagem disso é que o caso de uso estendido não precisa ser modificado quando um caso de uso extensor tiver de ser adicionado. No entanto, se o formato de descrição numerada (ver [Seção 4.1.1.1](#)) for utilizado, pode ser que, em uma eventual modificação do estendido, a referência descrita no extensor fique desatualizada. Por exemplo, suponha que a descrição do extensor defina que ele pode ser ativado entre os passos 5 e 6 do estendido. Se um passo anterior do estendido fosse removido ou um novo passo fosse adicionado, a referência feita pelo extensor estaria errada. O modelador deve definir um ponto de extensão como um composto de um nome e de uma descrição. A descrição de um ponto de extensão deve indicar em que ponto (ou pontos) do caso de uso estendido é possível inserir o comportamento do extensor. O seu nome, porém, pode ser apenas um mnemônico. Note que pode haver diversos pontos no estendido em que o extensor pode ser ativado. Em particular, talvez o extensor possa ser ativado entre quaisquer dois passos do estendido. De qualquer maneira, as descrições dos pontos de extensão realizadas no caso de uso extensor devem declarar claramente e sem ambiguidades os locais de extensão.

Como um exemplo de possível utilização do relacionamento de extensão, considere um processador de textos. Considere que um dos casos de uso deste sistema seja Editar Documento, que descreve a sequência de interações que ocorre quando um usuário edita um documento. No cenário típico desse caso de uso, o ator abre o documento, modifica-o, salva as modificações e fecha o documento. Mas, em outro cenário, o ator pode desejar que o sistema faça uma verificação ortográfica no documento. Em outro, ele pode querer realizar a substituição de um fragmento de texto por outro. Tanto a verificação ortográfica quanto a substituição de texto são esporádicas, não usuais, opcionais. Dessa forma, os casos de uso Corrigir Ortografia e Substituir Texto podem ser definidos como extensões de Editar Documento.

Como exemplo, apresentamos a seguir a sequência de interações do caso de uso extensor Substituir Texto, utilizando a primeira alternativa descrita há pouco (o extensor indica em que

ponto(s) pode ser ativado).

1. Em qualquer momento durante Editar Documento, o ator pode optar por substituir um fragmento de texto por outro.
2. O ator fornece o texto a ser substituído e o texto substituto.
3. O ator define os parâmetros de substituição (substituir somente palavras completas ou ocorrências dentro de palavras; substituir no documento todo ou somente na parte selecionada; ignorar ou considerar letras maiúsculas e minúsculas).
4. O sistema substitui todas as ocorrências encontradas no texto.

4.1.3.4 Relacionamento de generalização

Os relacionamentos descritos anteriormente (inclusão e extensão) implicam o reúso do comportamento de um caso de uso na definição de outros casos de uso. Outro relacionamento no qual o reúso de comportamento é evidente é o de generalização. O relacionamento de generalização pode existir entre dois casos de uso ou entre dois atores. Esse relacionamento permite que um caso de uso (ou um ator) herde características de outro, mais genérico, este último normalmente chamado de caso de uso (ator) *base*. O caso de uso (ator) herdeiro pode especializar o comportamento do caso de uso (ator) base. Vamos examinar ambas as alternativas.

Na generalização entre casos de uso, sejam A e B dois casos de uso. Quando B herda de A, as sequências do comportamento de A valem também para B. Quando for necessário, B pode redefinir as sequências de comportamento de A. Além disso, B (o caso de uso herdeiro) participa em qualquer relacionamento no qual A (o caso de uso pai) participa. Ou seja, todo ator que puder realizar o caso de uso pai pode também realizar qualquer caso de uso filho.

Uma consequência da utilização de generalização entre casos de uso é que sequências de comportamento descritas no caso de uso mais genérico são reutilizadas pelos casos de uso herdeiros. Somente o comportamento que não faz sentido ou é diferente para o caso de uso herdeiro precisa ser redefinido. Nota-se que um complicador se estabelece aqui, pois deve haver alguma maneira de especificar na descrição do caso de uso herdeiro que passos do caso de uso pai estão sendo redefinidos pelo primeiro. Isso pode ser feito com o posicionamento de marcadores no caso de uso pai. Esses marcadores podem, então, ser referenciados na descrição do caso de uso filho para especificar que passos estão sendo redefinidos ou onde está sendo definida uma extensão do comportamento do pai.

A UML estabelece que o caso de uso mais genérico em uma generalização pode ser concreto ou abstrato. Um caso de uso abstrato não apresenta comportamento associado. Por outro lado, um caso de uso concreto possui algum comportamento (a discussão no parágrafo anterior pressupõe que o caso de uso pai é concreto). Como boa prática de modelagem, recomendamos que o caso de uso pai de uma generalização seja sempre abstrato. Isso evita complicações com o uso de marcadores. Além disso, ainda podemos usar o relacionamento de generalização para o seu verdadeiro objetivo, que é indicar que dois ou mais casos de uso têm comportamentos semelhantes; o caso de uso abstrato é utilizado apenas para capturar a natureza semelhante entre os casos de uso filhos, estes últimos concretos.

A *generalização entre atores* significa que o ator herdeiro possui o mesmo comportamento (em relação ao sistema) que o ator do qual ele herda. Isso implica que, se dois ou mais atores herdarem de outro ator A, então todos os casos de uso associados a A identificam os atores herdeiros também.

como A. Em outras palavras, é impossível para um caso de uso relacionado a A perceber a diferença entre este e qualquer um de seus atores herdeiros; o ator A e seus herdeiros são percebidos como um só pelo caso de uso.

Outra característica da generalização entre atores é o fato de que o ator herdeiro pode participar em casos de uso nos quais o ator de quem ele herda não participa. Note que a propriedade de assimetria da generalização se aplica aqui: os casos de uso válidos para o ator herdeiro não são válidos para o ator pai.

Assim como na generalização entre casos de uso, um ator também pode ser concreto ou abstrato. Mas, diferentemente daquele primeiro tipo, na generalização entre atores não há complicador envolvido na utilização de atores concretos.

Como um exemplo de generalização entre atores, analise uma biblioteca na qual pode haver dois tipos de usuários: alunos e professores. Os dois tipos de usuário podem realizar empréstimos de títulos de livros e reservas de exemplares. Suponha que, com relação àqueles processos do negócio (emprestimos e reservas), não haja diferença entre um professor e um aluno. Suponha, ainda, que somente o professor pode requisitar a compra de títulos de livros à biblioteca. Nessa situação, poder-se-ia definir um ator chamado Usuário e outro ator chamado Professor, que herdaria de Usuário. Assim, o ator Professor herda todos os casos de uso do ator Usuário, ou seja, um professor pode interagir com todos os casos de uso com os quais um usuário comum interage. Além disso, um professor também interage com o caso de uso de requisição de compra de novos títulos de livros, sendo que este caso de uso é específico para professores.

4.1.3.5 Quando usar relacionamentos no MCU

Uma vantagem comum aos relacionamentos de inclusão, extensão e generalização é que eles tornam possível manter a descrição dos casos de uso a mais simples possível com a fatoração de sequências de interações *comuns*. Sem a utilização desses relacionamentos, as descrições de algumas sequências poderiam se repetir em vários lugares, ou estar aglutinadas em um único caso de uso gigante e descompacto.

Uma dúvida comum é saber que tipo de relacionamento utilizar em dada situação. Na verdade, não há regras para saber quando utilizar um ou outro tipo de relacionamento; há somente heurísticas.² Sendo assim, para a escolha do relacionamento a utilizar, as seguintes heurísticas podem ser seguidas:

- **Inclusão.** Use inclusão quando o mesmo comportamento se repetir em mais de um caso de uso. Por meio do relacionamento de inclusão esse comportamento comum pode ser fatorado em um novo caso de uso, o chamado caso de uso inclusivo. Note que esse comportamento comum está *necessariamente* contido em *todos* os cenários dos casos de uso inclusores, e que estes últimos não são completos se o comportamento do caso de uso não estiver inclusivo.
- **Extensão.** Use extensão quando um comportamento *eventual* de um caso de uso tiver de ser descrito. Note que alguns cenários do caso de uso estendido podem não utilizar esse comportamento eventual, uma vez que o mesmo é opcional. Podemos também pensar em usar o relacionamento de extensão na situação em que precisamos estender o comportamento de um caso de uso preexistente sem modificar sua descrição original. Essa possibilidade é importante, principalmente em um processo de desenvolvimento iterativo (ver Seção 2.3.2). Isso porque, quando iterações subsequentes são realizadas em um projeto de desenvolvimento (ocasião em que novas versões de casos de uso preexistentes são

desenvolvidas), é comum a situação de a equipe ter de adicionar novos comportamentos àqueles casos de uso. Isso pode ser feito com o relacionamento de extensão, sem a necessidade de alterar a descrição do caso de uso original.

- **Generalização entre casos de uso.** Use generalização entre casos de uso quando você identificar dois ou mais casos de uso com comportamentos semelhantes. Crie, então, um caso de uso mais genérico (de preferência abstrato) e o relate por generalização aos casos de uso semelhantes. Note que a generalização entre dois casos de uso implica que o caso de uso herdeiro herda todo o comportamento de seu pai. Portanto, se alguma parte do caso de uso pai não fizer sentido para o caso de uso herdeiro, não existe nexo em utilizar generalização. Se apenas algumas partes do caso de uso pai fizerem sentido para os potenciais herdeiros, considere o uso dos relacionamentos de inclusão da extensão, em vez da generalização.
- **Generalização entre atores.** Use generalização quando precisar definir um ator que desempenhe um papel que já é desempenhado por outro ator em relação ao sistema, mas que também possui comportamento particular adicional.

A [Tabela 4-1](#) resume as possibilidades de existência de relacionamentos entre os elementos do modelo de casos de uso.

Tabela 4-1: Possibilidades de relacionamentos entre os elementos do modelo de casos de uso

	Comunicação	Extensão	Inclusão	Generalização
Caso de uso e caso de uso		X	X	X
Autor e autor				X
Caso de uso e autor	X			

Para finalizar esta seção, é importante dizer que os relacionamentos (entre casos de uso e entre atores) definidos pela UML devem ser utilizados com parcimônia. Se não fizermos isso, corremos o risco de obter um MCU com vários relacionamentos e difícil de ser entendido. Isso porque quanto mais relacionamentos são utilizados, menor é a clareza do modelo. Casos de uso *não* são especificações formais; portanto, descrições repetidas em mais de um caso de uso são aceitáveis, desde que controladas. O importante a notar é que o MCU é uma peça fundamental na *validação* (ver [Seção 2.1.2](#)) de um sistema. Portanto, a clareza e a legibilidade desse modelo devem ser levadas sempre em conta quando estivermos pensando em utilizar qualquer um dos três relacionamentos descritos aqui. Para deixar claro: o modelo ideal é aquele sem redundâncias e legível, mas se tivermos que sacrificar uma dessas qualidades, que seja a primeira.

Nesse ponto, a dica prática que damos é a seguinte: se você tiver absoluta certeza de que duas ou mais funcionalidades do sistema compartilham comportamento comum, ou se é um modelador experiente, então utilize os relacionamentos diretamente. Do contrário, retarde a definição de quaisquer relacionamentos de inclusão, extensão e generalização até o momento em que você já tiver definido uma primeira versão do MCU de seu sistema, contendo os casos de uso concretos e seus atores. Até lá, você terá um melhor entendimento do sistema e poderá decidir com maior discernimento se a utilização de um ou outro relacionamento é compensatória.

Se o modelador usar relacionamentos (de inclusão, extensão e generalização) em excesso na construção de um MCU, há o risco da diminuição da clareza desse modelo. Lembre-se de que o MCU é ferramenta fundamental na validação do sistema, em que a facilidade de comunicação com o usuário (especialistas do domínio) é um fator determinante.

4.1.3.6 Decomposição funcional e relacionamentos entre casos de uso

Um erro bastante comum na identificação de relacionamentos entre casos de uso, principalmente para desenvolvedores acostumados com as técnicas de Análise e Estruturada, é o de considerar o MCU equivalente ao *modelo funcional* utilizado na metodologia estruturada, no qual se utiliza a ferramenta de DFD (Diagrama de Fluxos de Dados) para representar processos do sistema. Para a construção do modelo funcional, utiliza-se o procedimento da *decomposição funcional*. Nesse procedimento, o sistema é interpretado como um grande processo que pode ser dividido em processos menores e mais simples. Cada um desses, por sua vez, pode também ser dividido em outros mais simples ainda, e assim por diante. A aplicação desse procedimento resulta em uma rede hierárquica de processos. Em cada nível dessa hierarquia, existem processos que são mais simples que os do nível acima e mais complexos que os do nível abaixo. Além disso, as funções de cada nível estão indiretamente conectadas por *depósitos de dados*, repositórios de informações que tais funções processam. As funções do sistema se comunicam pelos depósitos de dados. No último nível dessa hierarquia, existem as *primitivas funcionais*, processos simples o suficiente para serem entendidos facilmente e que, por conta disso, não precisam ser particionados. Dessa forma, o modelo funcional acaba por estabelecer a estrutura funcional interna e o fluxo de informações entre as funções do sistema. O modelo funcional, portanto, não provê uma visão externa do sistema (como o faz o modelo de casos de uso), mas sim de seu comportamento interno (embora em um nível de abstração alto).

Outro erro de modelagem resultante da confusão entre modelagem de casos de uso e modelagem funcional é “quebrar” (em dois ou mais casos de uso) funcionalidades que na verdade pertencem a um mesmo caso de uso. Com relação a isso, é importante ter em mente que um caso de uso é uma descrição completa de uma sequência de interações, cuja realização traz um resultado de valor para o ator envolvido; ele não é normalmente um passo ou atividade individual em um processo. Por exemplo, considere um sistema que funciona via Internet e que possibilita a compra de livros por usuário cadastrados. Nessa situação, é bastante possível que a definição de um caso de uso denominado Imprimir Fatura não seja correta. É mais provável que aquela sequência de interações seja uma subsequência contida em uma sequência maior, esta última correspondente a um caso de uso denominado Comprar Produtos.

Outra má interpretação que pode ser feita pelo modelador iniciante é pensar que casos de uso se comunicam, assim como funções (processos) podem se comunicar em um DFD (através de depósitos de dados). Esse erro é um sintoma de que o modelador está tentando definir a estrutura interna do sistema no MCU com o uso de relacionamentos (ver Seção 4.1.3). Em um MCU, quando definimos um relacionamento (de inclusão, extensão ou generalização) entre dois casos de uso, não existe a semântica de troca de dados entre os mesmos. “Esse caso de uso chama o outro”, certa vez ouvi um iniciante em modelagem declarar. Em modelagem de casos de uso, essa interpretação simplesmente não faz sentido.

Em resumo, o enfoque ao utilizar casos de uso e seus relacionamentos é identificar os *objetivos do usuário*, em vez das funções do sistema. Tenha em mente que o MCU define uma visão externa do sistema. Embora essa visão implique uma descrição técnica das ações e das estruturas internas do

sistema, esses aspectos não são considerados nesse modelo. A modelagem de casos de uso não é uma ferramenta para realizar a decomposição funcional do sistema. Em vez disso, casos de uso fornecem uma perspectiva externa do comportamento do sistema, sem prover detalhes sobre a lógica interna de seu funcionamento.

4.2 Diagrama de casos de uso

Na [Seção 4.1](#), descrevemos os principais componentes de um MCU: atores, casos de uso e relacionamentos. Apresentamos também diversos estilos de descrição possíveis para um caso de uso. O conjunto de descrições dos casos de uso e atores corresponde à *perspectiva textual* do MCU. Nesta seção, nosso interesse recai sobre a *perspectiva gráfica* do MCU, representada pelo diagrama de casos de uso (DCU).

O DCU é um dos diagramas da UML e corresponde a uma visão externa de alto nível do sistema. Esse diagrama representa *graficamente* os atores, casos de uso e relacionamentos entre esses elementos. O DCU tem o objetivo de ilustrar em um nível alto de abstração quais elementos externos interagem com que funcionalidades do sistema. Nesse sentido, a finalidade de um DCU é apresentar um tipo de “diagrama de contexto”, que apresenta os elementos externos de um sistema e as maneiras segundo as quais eles as utilizam.

A notação utilizada para ilustrar atores em um DCU é a figura de um boneco, com o nome do ator definido abaixo da figura. Note que essa notação não corresponde ao significado de ator em sua completude, porque um ator nem sempre corresponde a seres humanos, como a notação leva a entender (ver [Seção 4.1.2](#)). Cada caso de uso é representado por uma elipse. O nome do caso de uso é posicionado abaixo ou dentro dessa elipse. Um relacionamento de comunicação é representado por um segmento de reta ligando ator e caso de uso. Um ator pode estar associado por meio do relacionamento de comunicação a vários casos de uso em um DCU. Pela UML, também é possível imprimir um sentido ao segmento de reta correspondente a um relacionamento de comunicação, para denotar o iniciante da sequência de interações. No entanto, essa situação tem pouco uso prático e normalmente o segmento de reta do relacionamento de comunicação é definido sem o sentido. A [Figura 4-2](#) ilustra a notação da UML para representar atores, casos de uso e relacionamentos de comunicação. Esses três elementos são os mais comumente utilizados.

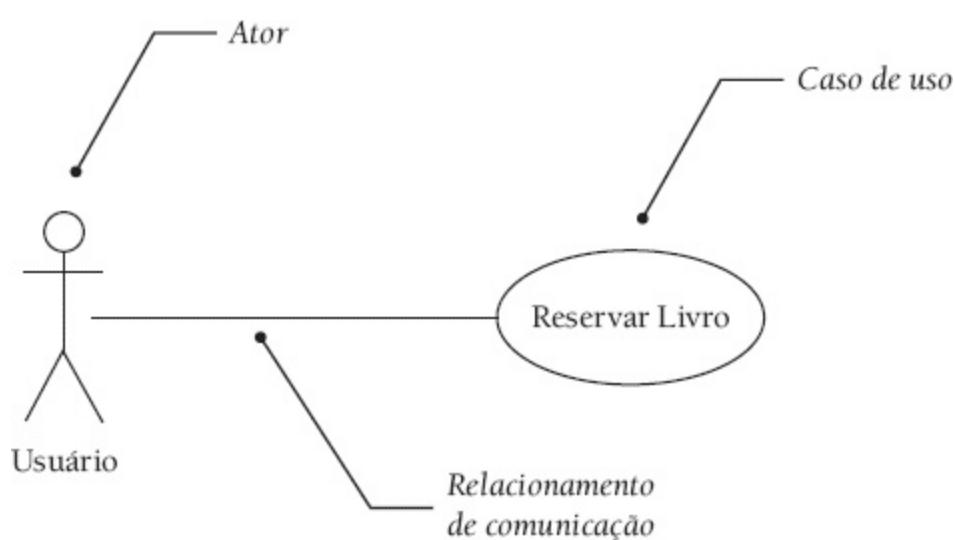


Figura 4-2: Notação para ator, caso de uso e relacionamento de comunicação.

Também é possível representar a fronteira do sistema em um diagrama de casos de uso. Essa fronteira é representada por um retângulo no interior do qual são inseridos os casos de uso. Os atores são posicionados do lado de fora do retângulo, para enfatizar a divisão entre o interior e o exterior do sistema. A [Figura 4-3](#) apresenta um exemplo de diagrama de casos de uso no qual se utiliza uma fronteira.

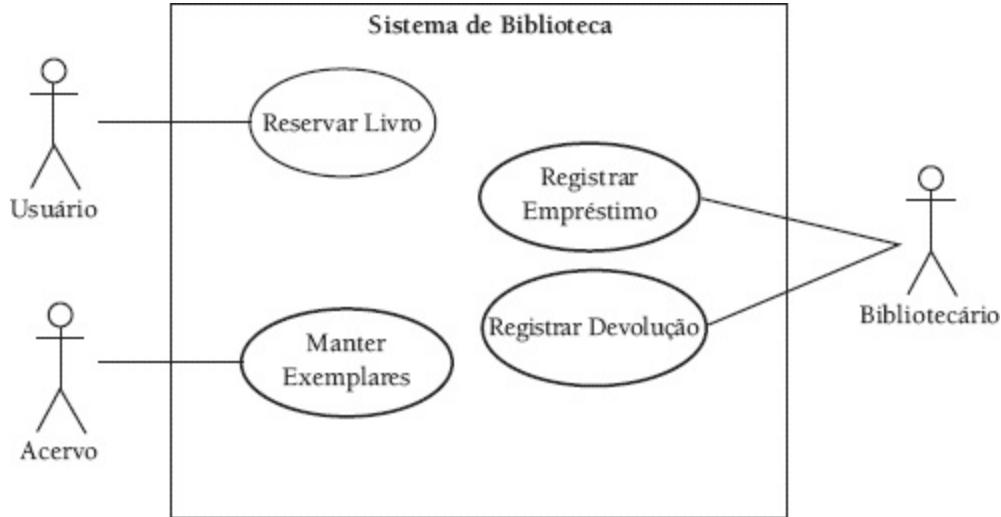


Figura 4-3: Exemplo de diagrama de casos de uso utilizando um retângulo de fronteira.

O relacionamento de inclusão (ver [Seção 4.1.3.2](#)) em que um caso de uso A inclui um caso de uso B é representado por uma seta direcionada de A para B. O eixo dessa seta é tracejado e rotulado com o estereótipo (ver [Seção 3.1](#)) predefinido `include`. A [Figura 4-4](#) ilustra a representação do relacionamento de inclusão em um DCU. Esse diagrama informa que os casos de uso `Obter Extrato`, `Realizar Saque` e `Realizar Transferência` têm uma sequência de interações em comum: a utilizada para autenticar o cliente do banco que está representada pelo caso de uso `Fornecer Identificação`.

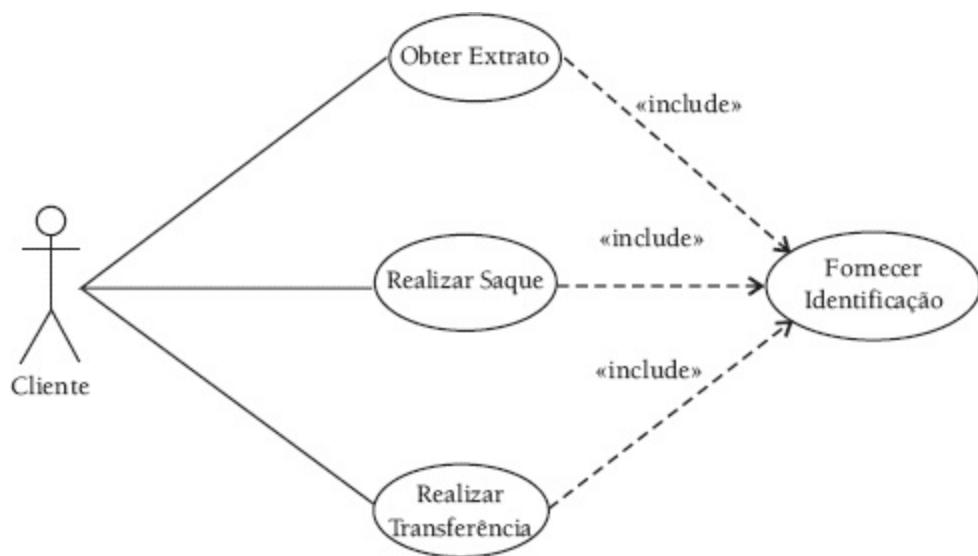


Figura 4-4: Exemplo de relacionamento de inclusão.

O relacionamento de extensão, em que um caso de uso A estende um caso de uso B, é representado por uma seta direcionada de A para B. Essa seta, de eixo também tracejado, é rotulada com outro estereótipo predefinido pela UML, o `extend`. A [Figura 4-5](#) ilustra a representação desse relacionamento. Essa figura mostra que os casos de uso `Corrigir Ortografia` e `Substituir Texto` têm sequências

de interações que são *eventualmente* utilizadas quando o ator Escritor estiver usando o caso de uso Editar Documento.

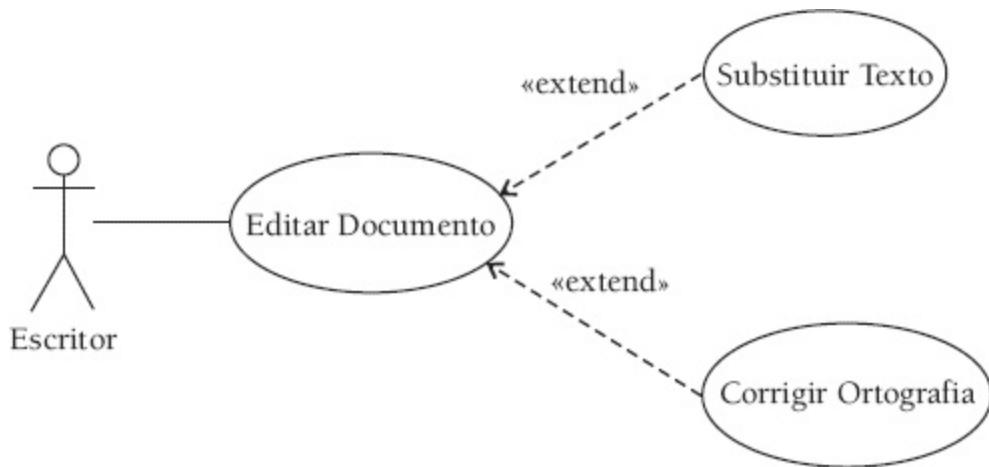


Figura 4-5: Exemplo de relacionamento de extensão.

A Figura 4-6 ilustra exemplos do relacionamento de generalização em suas duas formas: entre casos de uso e entre atores. A generalização entre casos de uso indica que os casos de uso Realizar Pagamento com Cartão de Crédito e Realizar Pagamento com Dinheiro São especiais em relação ao caso de uso Realizar Pagamento. Já a generalização entre os atores Usuário e Professor indica que este último pode interagir com qualquer caso de uso que um usuário comum interage.

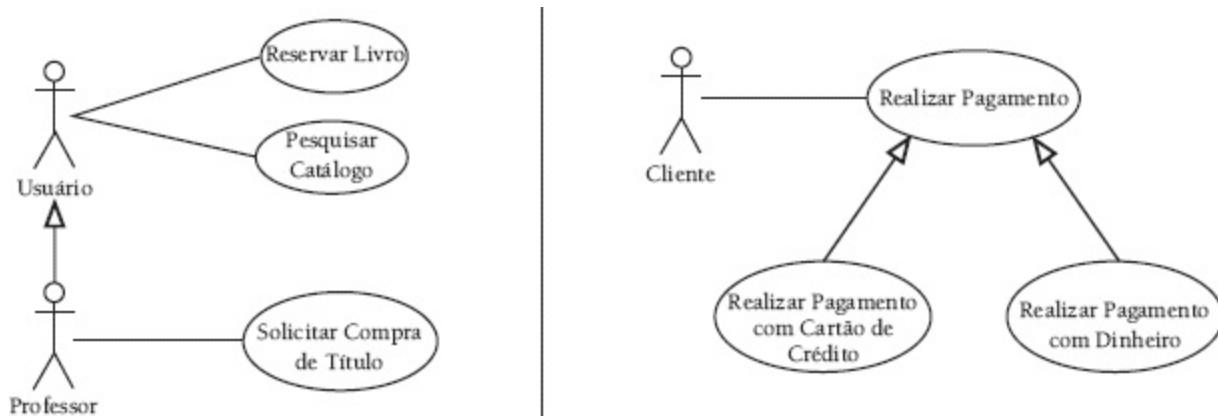


Figura 4-6: Exemplos de utilização do relacionamento de generalização.

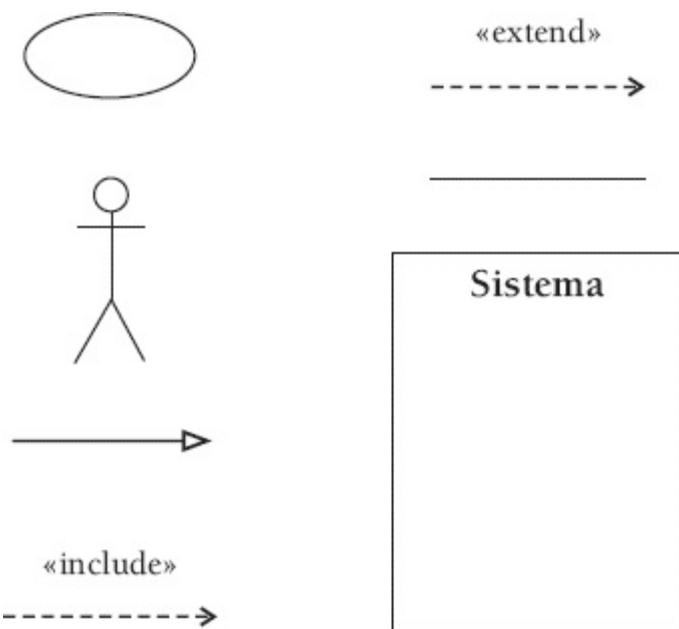


Figura 4-7: Elementos gráficos da UML para o desenho de um DCU.

Além disso, o Professor pode participar em outros casos de uso específicos a ele, como, por exemplo, Solicitar Compra de Título.

4.3 Identificação dos elementos do MCU

Nas [Seções 4.1 e 4.2](#), apresentamos os componentes do MCU e descrevemos detalhes acerca de suas perspectivas textual e gráfica. O domínio das regras definidas sobre essas duas perspectivas é importante para a construção de modelos de casos de uso corretos. Entretanto, tão importante quanto dominarmos a notação do MCU é termos conhecimento de técnicas e boas práticas de modelagem que, se seguidas e utilizadas, nos levam à construção de modelos coerentes com as reais necessidades dos futuros usuários. Nesta seção, estudamos, portanto, algumas técnicas que podem ser aplicadas para a correta identificação dos elementos de um MCU.

Os atores que interagem com um sistema e os casos de uso desse sistema são identificados a partir de informações coletadas na fase de levantamento de requisitos (ver [Seção 2.1.1](#)). Durante essa fase, os analistas de sistemas (ver [Seção 2.2.2](#)) devem identificar as atividades dos processos de negócio que precisam ser automatizadas pelo sistema a ser construído. Os analistas também devem identificar quais os elementos que interagem naqueles processos.

Nesta seção, são apresentadas algumas dicas de como identificar atores e casos de uso. Vale dizer que não há uma regra geral que indique *quantos* casos de uso são necessários para descrever completamente um sistema. A quantidade de casos de uso a ser utilizada depende completamente da *complexidade* do sistema. Sistemas de software de porte médio possuem de quinze a vinte casos de uso, enquanto os realmente complexos chegam a possuir até uma ordem de grandeza a mais que os de porte médio.

4.3.1 Identificação de atores

Para começar a construir o MCU, todos os atores do sistema devem ser identificados. Para isso, o analista de sistemas deve tentar identificar quais as fontes de informações a serem processadas e quais são os destinos das informações geradas pelo sistema. Se o sistema estiver sendo desenvolvido

para uma empresa, o analista deve identificar as áreas dessa empresa que serão afetadas ou utilizarão o sistema. Como, por definição, um ator é todo elemento externo que interage com o sistema, as fontes e os destinos das informações a serem processadas são atores em potencial.

Na identificação de atores há algumas perguntas úteis para as quais os analistas de sistemas devem procurar respostas:

1. Que órgãos, empresas ou pessoas utilizarão o sistema?
2. Que sistemas ou equipamentos irão se comunicar com o sistema a ser construído?
3. Alguém deve ser informado de alguma ocorrência no sistema?
4. Quem está interessado em certo requisito funcional do sistema?

Além de fazer essa identificação inicial, o desenvolvedor deve continuar a pensar sobre atores quando passar para a identificação dos casos de uso, pois nessa atividade é possível que apareçam atores ainda não identificados.

4.3.2 Identificação de casos de uso

A partir da lista de atores, deve-se passar à identificação dos casos de uso. Nessa identificação, é possível distinguir entre dois tipos de casos de uso: primário e secundário. Passamos para a descrição desses casos de uso nas próximas seções.

4.3.2.1 Casos de uso primários

Casos de uso primários são aqueles que representam os *objetivos* dos atores. Esses casos de uso representam os processos da empresa que estão sendo automatizados pelo sistema de software. A seguir são enumeradas algumas perguntas para as quais os analistas de sistemas devem procurar respostas com o intuito de identificar os casos de uso primários de um sistema:

1. Quais são as necessidades e os objetivos de cada ator em relação ao sistema?
2. Que informações o sistema deve produzir?
3. O sistema deve realizar alguma ação que ocorre regularmente no tempo?
4. Para cada requisito funcional, existe um (ou mais) caso(s) de uso para atendê-lo?

Também é possível para o moderador utilizar outras técnicas de identificação. Pode-se considerar as seguintes situações (na descrição das situações a seguir, são dados exemplos considerando um sistema de venda de livros pela Internet):

- *Caso de uso “oposto”*: chama-se caso de uso oposto aquele cuja realização desfaz o resultado da realização de outro caso de uso. Por exemplo, pode ser que um cliente tenha a possibilidade de cancelar um pedido de compra realizado anteriormente. Nesse caso, não é preciso pensar muito para identificar um novo caso de uso, Cancelar Pedido. Deve-se perguntar de forma geral, para cada caso de uso: “As ações realizadas pelo sistema quando da realização deste caso de uso podem ser desfeitas?”.
- *Caso de uso que precede outro caso de uso*: algumas vezes, certas condições devem ser verdadeiras quando da execução de um caso de uso. Por exemplo, para que um cliente realize um pedido de compra, é necessário que ele esteja cadastrado no sistema da livraria virtual. Isso leva a um novo caso de uso para que o cliente se cadastre. Além disso, para

realizar um pedido de compra, o cliente deve ter a possibilidade de obter detalhes de determinados produtos por meio de um mecanismo de busca. De forma geral, a pergunta a ser feita para identificar casos de uso precedentes é, para cada caso de uso, “o que pode ocorrer antes da realização deste caso de uso?”.

- *Caso de uso que sucede a outro caso de uso:* uma outra estratégia de identificação é pensar nas consequências da realização de um caso de uso. Por exemplo, considerando o mesmo exemplo da livraria virtual, quando um cliente realiza uma compra, pode ser que haja a necessidade de agendar sua entrega. Nessa situação, um possível caso de uso decorrente seria Agendar Entrega de Pedido, no qual um funcionário da livraria seleciona alguns pedidos para serem entregues em certa data. A pergunta geral que se deve fazer para cada caso de uso identificado é “o que pode ocorrer após a realização deste caso de uso?”.
- *Caso de uso temporal:* pode haver funcionalidades realizadas pelo sistema que não são iniciadas por um ator. Isso normalmente acontece quando o sistema deve realizar alguma tarefa de tempos em tempos, sem intervenção externa.³ Por exemplo: “O sistema deve gerar um relatório de vendas toda sexta-feira”. A pergunta geral nessa situação é: “Há alguma tarefa que o sistema deva realizar automaticamente?”. Em um caso de uso temporal, normalmente o ator é definido como o agente que recebe a informação resultante (p. ex., um funcionário recebe o relatório de execução do sistema). Alternativamente, pode-se definir um ator fictício, no caso, Tempo, que estará associado ao referido caso de uso (ver [Figura 4-8](#)).

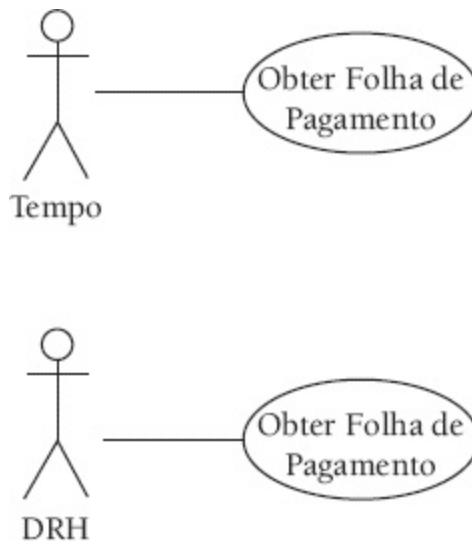


Figura 4-8: Formas alternativas de representar um caso de uso temporal.

- *Caso de uso relacionado a alguma condição interna:* assim como nos casos de uso temporais, esta é uma situação em que não há um ator diretamente envolvido. Nessa situação, o sistema deve realizar alguma funcionalidade de acordo com a ocorrência de algum evento interno. Seguem dois exemplos disso: “O sistema deve notificar o usuário de que há novas mensagens de correio”; “O sistema deve avisar o almoxarife de que um determinado produto chegou no nível de estoque mínimo”.

4.3.2.2 Casos de uso secundários

Um caso de uso secundário é aquele que não traz benefício direto para os atores, mas que é necessário para que o sistema funcione adequadamente. Esses casos de uso se encaixam nas seguintes

categorias:

- *Manutenção de cadastros*: frequentemente há a necessidade de inclusão, exclusão, alteração ou consulta sobre dados cadastrais. Por exemplo, em um sistema de folha de pagamento, deve haver cadastros para funcionários e cargos. Normalmente, o mais adequado é criar um caso de uso que corresponda às quatro operações.⁴ Isso é feito quando todas as operações cadastrais são realizadas pelo mesmo ator. Quando são realizadas por atores diferentes, é melhor criar casos de uso separados para elas. Isso vale de forma geral: sempre é mais adequado agrupar casos de uso de manutenção de cadastros por ator, e não pelo item de informação sendo cadastrado. Por fim, é importante notar que há certa controvérsia na literatura acerca da necessidade de definição de casos de uso de cadastro. Recomendamos a definição explícita desses casos de uso. Isso porque, quando um agente externo precisa cadastrar alguma informação, algum comportamento do sistema é iniciado por esse agente; ele usa o sistema para criar (alterar, excluir ou pesquisar) um item de informação. É fácil ver que essa situação se encaixa nas definições de caso de uso e de ator. Além disso, não considerar certa funcionalidade de cadastro no MCU torna esse modelo incompleto e suscetível a mais de uma interpretação: a funcionalidade será implementada, embora não tenha sido modelada no MCU, ou o fato de ela não aparecer no MCU significa que não será implementada?
- *Manutenção de usuários e de seus perfis*: adição de novos usuários, atribuição de direitos de acesso, configuração de perfis de usuários etc.
- *Manutenção de informações provenientes de outros sistemas*: pode ser o caso em que o sistema deva se comunicar com outro sistema. Por exemplo, em um sistema de venda de produtos pode haver a necessidade de comunicação com um sistema de controle de estoque para saber a quantidade de produtos disponíveis. Em casos assim, as informações em um sistema e no outro devem ser sincronizadas.

Uma observação importante: embora os casos de uso secundários precisem ser considerados, o modelador deve priorizar inicialmente a identificação dos casos de uso primários, que representam os processos do negócio da empresa. Começar a identificação pelos casos de uso secundários é uma indicação de que o modelador está pensando em *como* o sistema deve ser construído. O ponto-chave é considerar que um sistema de software não existe para cadastrar informações, tampouco para gerenciar os seus usuários. O objetivo principal de um sistema é produzir algo de valor para o ambiente no qual está implantado.

4.4 Construção do modelo de casos de uso

Na Seção 4.3, descrevemos algumas heurísticas para identificação dos elementos de um MCU. Entretanto, uma vez que esses elementos estão identificados, como documentá-los para construir o modelo de casos de uso propriamente dito? A definição de um MCU envolve a construção das suas duas perspectivas, a gráfica e a textual. A primeira corresponde ao diagrama de casos de uso, enquanto a segunda corresponde à documentação dos atores e casos de uso. Nas próximas seções descrevemos a construção desses artefatos.

4.4.1 Construção do diagrama de casos de uso

Conforme descrevemos anteriormente, um objetivo importante, talvez o principal, de um MCU é a comunicação. Ele deve prover um veículo que permita a especialistas do domínio e desenvolvedores discutirem as funcionalidades do sistema e o seu comportamento. Por outro lado, o DCU deve servir para dar suporte à parte escrita do modelo, fornecendo uma visão de alto nível do sistema e obviamente sendo coerente com aquela parte escrita. Na [Seção 4.1.3](#), analisamos os relacionamentos possíveis de serem utilizados em um MCU e a (potencial) falta de legibilidade que seu uso em excesso pode trazer. Como o DCU deve ser coerente com a parte escrita, a falta de legibilidade pode se propagar para a parte gráfica do modelo, também. Se o diagrama é um emaranhado indecifrável de elementos gráficos, ele não está cumprindo seu objetivo, que é de comunicar. Nesse sentido, quanto mais fácil for a leitura do DCU, melhor. Portanto, a clareza do DCU (e de qualquer diagrama da UML) deve ser uma preocupação constante do modelador.

Para um sistema de software pequeno ou médio (composto de uma ou duas dúzias de casos de uso), o seu DCU muito provavelmente cabe em uma folha de papel (ou na tela de uma ferramenta CASE) e pode ser visualizado e entendido de uma única vez. Portanto, para esse tipo de sistema, o modelador pode criar um único DCU. Nesse diagrama, uma opção é utilizar um *retângulo de fronteira*. Os casos de uso são desenhados dentro do retângulo, e os atores, do lado de fora. O objetivo dessa disposição é dar uma ideia visual clara da fronteira do sistema. Esse diagrama permite oferecer uma visão global e de alto nível do sistema.

Contudo, para sistemas mais complexos, a quantidade de casos de uso cresce acima desse limite de fácil visualização. Representar todos os casos de uso em um único DCU talvez torne esse diagrama um tanto ilegível. Nessa situação, o modelador pode decidir formar grupos de casos de uso logicamente relacionados, em que cada grupo pode ser visualizado de uma só vez. Nessas situações, o modelador pode adotar a abordagem de criar vários diagramas de casos de uso. A alocação dos casos de uso e dos atores por esses diagramas deve ser feita de acordo com as necessidades de visualização. A seguir descrevemos alguns critérios que podem ser adotados:

- Diagrama que exibe um caso de uso e seus relacionamentos.
- Diagrama que exibe todos os casos de uso para um ator.
- Diagrama que exibe todos os casos de uso a serem implementados em uma iteração de desenvolvimento.
- Diagrama que exibe todos os casos de uso de uma divisão (parte) específica da organização

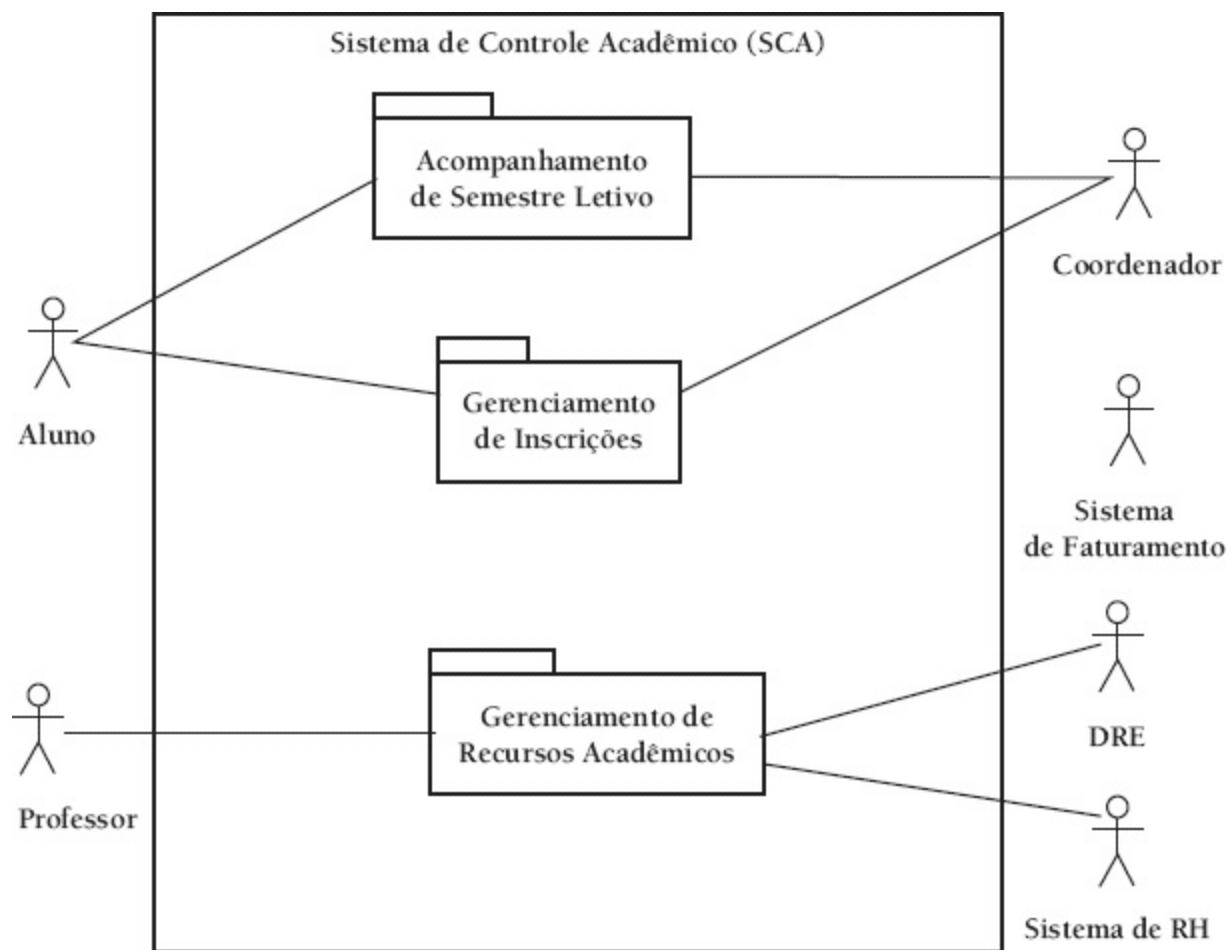


Figura 4-9: Exemplo de diagrama de pacotes para casos de uso.

O uso de pacotes (ver [Seção 3.5](#)) permite formar grupos de casos de uso e atores de tal sorte que o MCU possa ser compreendido e gerenciado por partes. Em sistemas complexos, é possível dividir os elementos do MCU em diversos pacotes. No contexto da modelagem de casos de uso, os pacotes podem ser utilizados com diversos objetivos. A seguir, são listados alguns desses objetivos:

- Para estruturar o modelo de casos de uso de maneira que reflita os tipos de usuários do sistema.
- Para definir a ordem na qual os casos de uso serão desenvolvidos.
- Para definir o grau de correlação entre os casos de uso.

A abordagem de representar os casos de uso em diversos diagramas e agrupar cada diagrama em pacote apresenta a desvantagem de se dificultar a manutenção do MCU. Contudo, uma boa ferramenta CASE (ver [Seção 2.6](#)) deve ajudar nessa tarefa.

O modelador deve sempre tentar maximizar a legibilidade do DCU. Se necessário, deve-se criar mais de um diagrama de casos de uso.

4.4.2 Documentação dos atores

A documentação de atores é relativamente simples. Uma breve descrição (uma frase ou duas) para cada ator deve ser adicionada ao modelo de casos de uso. O nome designado para um ator precisa ser escolhido de tal forma que lembre o papel desempenhado por ele no sistema.

4.4.3 Documentação dos casos de uso

Conforme mencionamos na [Seção 4.1.1](#), a UML não define uma estruturação específica a ser utilizada na descrição de um caso de uso. Por conta disso, há diversas propostas de descrição. Nesta seção é apresentada uma proposta para a descrição de um caso de uso expandido.⁵ No entanto, antes de começar a apresentação, o leitor deve estar atento para o fato de que essa proposta é apenas uma sugestão. Pode ser que uma equipe de desenvolvimento não precise utilizar todos os itens aqui mencionados; pode até ser que mais detalhes sejam necessários. De qualquer modo, a equipe de desenvolvimento deve utilizar os itens de descrição que forem *realmente* úteis e mais inteligíveis para o usuário.

4.4.3.1 Nome

O primeiro item que deve constar da descrição de um caso de uso é o seu nome. Este deve ser o mesmo nome utilizado no DCU. Cada caso de uso precisa ter um nome único.

4.4.3.2 Identificador

O identificador é um código único para cada caso de uso que permite fazer referência cruzada entre diversos documentos relacionados ao MCU (p. ex., a descrição de um cenário do caso de uso pode fazer referência a esse identificador). Uma convenção de nomenclatura que recomendamos é usar o prefixo CSU seguido de um número sequencial. Por exemplo: CSU01, CSU02.

4.4.3.3 Importância

A definição da categoria de importância é atribuída ao caso de uso. A [Seção 4.6](#) detalha as possíveis categorias em que é possível enquadrar um caso de uso.

4.4.3.4 Sumário

Uma pequena declaração do objetivo do ator ao utilizar o caso de uso (no máximo duas frases).

4.4.3.5 Ator primário

O nome do ator que inicia o caso de uso. (Note que talvez o ator não inicie o caso de uso, mas ainda assim pode ser alvo do resultado produzido pelo caso de uso.) Um caso de uso possui apenas um ator primário.

4.4.3.6 Atores secundários

Os nomes dos demais elementos externos participantes do caso de uso, os atores secundários (ver [Seção 4.1.2](#)). Um caso de uso possui zero ou mais atores secundários.

4.4.3.7 Precondições

Pode haver alguns casos de uso cuja realização não faça sentido em qualquer momento, mas ao contrário, somente quando o sistema estiver em um determinado estado com certas propriedades.

Uma precondição de um caso de uso define que hipóteses são assumidas como verdadeiras para

que o caso de uso tenha início. Este item da descrição pode conter zero ou mais precondições.

4.4.3.8 Fluxo principal

O fluxo principal de um caso de uso, por vezes chamado de fluxo básico, corresponde à sua descrição da sequência de passos usual. Isso significa que fluxo principal descreve o que *normalmente* acontece quando o caso de uso é utilizado. Toda descrição de caso de uso deve ter um fluxo principal. O texto descritivo desse fluxo (assim como dos fluxos alternativos e de exceção, descritos a seguir) precisa ser claro e conciso. Além disso, nessa descrição, o modelador deve se ater ao domínio do problema, e não à sua solução. Portanto, o jargão computacional não deve ser utilizado na descrição de casos de uso; ao contrário, casos de uso devem ser escritos do ponto de vista do usuário e usando a sua terminologia.

4.4.3.9 Fluxos alternativos

Por vezes, um caso de uso pode ser utilizado de diversas maneiras possíveis, o que resulta na existência de diversos *cenários* para o mesmo (ver [Seção 4.1.1.4](#)). Esses fluxos podem ser utilizados para descrever o que acontece quando o ator opta por utilizar o caso de uso de uma forma alternativa, diferente da descrita no fluxo principal, para alcançar o seu objetivo. Fluxos alternativos também podem ser utilizados para descrever situações de escolha exclusivas entre si (em que há diversas alternativas e somente uma deve ser realizada). A [Figura 4-10](#) ilustra de forma esquemática essas situações de uso dos fluxos alternativos. As linhas tracejadas representam fluxos alternativos. A linha sólida representa o fluxo principal. Note que é possível que a descrição de um caso de uso tenha somente o fluxo principal, sem fluxos alternativos.

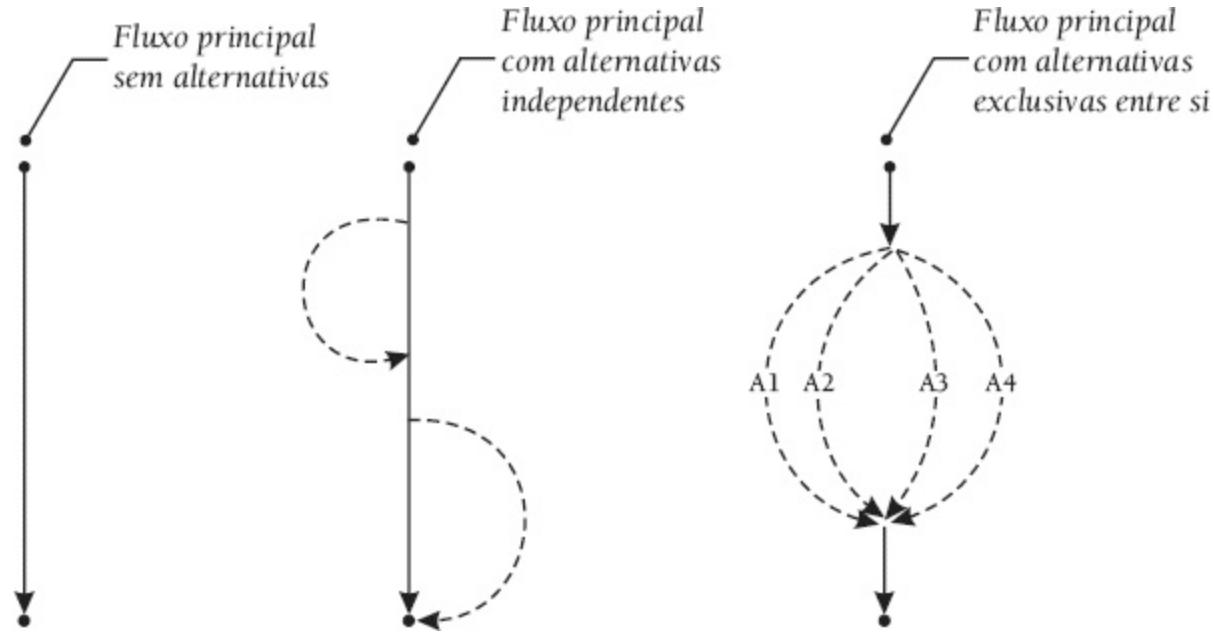


Figura 4-10: Fluxos alternativos em um caso de uso.

Uma dúvida que pode existir durante a descrição de um caso de uso é se um determinado comportamento deve ser descrito como um fluxo alternativo ou como um caso de uso de extensão (ver [Seção 4.1.3.3](#)). Podemos resolver esse dilema recorrendo à definição do relacionamento de extensão. Esse relacionamento implica que, ao comportamento de um caso de uso, pode ser *inserido* o comportamento definido em outro caso de uso. Note a utilização do termo “*inserido*”, significando

que o comportamento do caso de uso extensor não substitui parte alguma do caso de uso estendido, e sim o complementa. Podemos pensar no caso de uso extensor como uma extensão que descreve um comportamento que funciona como uma *interrupção* em relação ao caso de uso de estendido. Por outro lado, um fluxo alternativo descreve um comportamento alternativo para a execução do fluxo principal, que substitui uma parte do comportamento do fluxo principal. De qualquer maneira, a decisão de utilizar um fluxo alternativo ou um caso de uso de extensão não terá tanta importância quanto o fato de ignorar a existência do comportamento adicional.

4.4.3.10 Fluxos de exceção

Um fluxo de exceção é similar a um fluxo alternativo, uma vez que também representa um comportamento executado como um “desvio” a partir do fluxo básico de um caso de uso. No entanto, os primeiros correspondem à descrição de *situações de exceção*. Isso significa que fluxos de exceção descrevem o que acontece quando algo inesperado ocorre na interação entre ator e caso de uso (p. ex., quando um usuário realiza alguma ação inválida).

A importância de fluxos de exceção está no fato de o modelador poder especificar situações não usuais, a partir das quais o sistema pode se recuperar (contornar a situação) ou cancelar a realização do caso de uso em questão.

Um fluxo de exceção possui algumas características importantes, listadas a seguir.

1. Representa um erro de operação durante o fluxo principal do caso de uso.
2. Não tem sentido fora do contexto do caso de uso no qual ocorre.
3. Deve indicar em que passo o caso de uso continua ou, conforme for, indicar explicitamente quando ele termina.

Por exemplo, considere um caso de uso denominado Realizar Pedido, em que um ator usa o sistema para realizar uma encomenda (pedido) de quaisquer produtos. A seguir são listadas algumas situações não usuais que seriam tratadas em fluxos de exceção na descrição desse caso de uso.

- E se o cartão de crédito excede o limite?
- E se a loja não tem a quantidade requisitada para um dos produtos desejados?
- E se o cliente já tem um débito anterior?

4.4.3.11 Pós-condições

Em alguns casos, em vez de gerar um resultado observável, o estado do sistema pode mudar após um caso de uso ser realizado. Essa situação é especificada como uma pós-condição. Uma pós-condição é um estado que o sistema alcança após certo caso de uso ter sido executado.

A pós-condição deve declarar *qual* é esse estado, em vez de declarar *como* ele foi alcançado. Um exemplo típico de pós-condição é a declaração de que uma (ou mais de uma) informação foi modificada, removida ou criada no sistema. Pós-condições são normalmente descritas utilizando o tempo pretérito.

4.4.3.12 Regras do negócio

A descrição de um caso de uso também pode fazer referência cruzada a uma ou mais *regras do*

negócio. Por sua importância, deixamos o detalhamento de regras do negócio para a [Seção 4.5.1](#).

4.4.3.13 Histórico

Este item da descrição do caso de uso pode declarar informações como o autor do caso de uso, a data em que ele foi criado, além de eventuais modificações no seu conteúdo.

4.4.3.14 Notas de implementação

Na descrição dos fluxos (principal, alternativos e de exceção) de um caso de uso, o objetivo é manter a narrativa em um alto nível e utilizar a terminologia do domínio. Entretanto, ao fazer isso, podem vir à mente do modelador algumas considerações relativas à implementação desse caso de uso. A seção *notas de implementação* serve para capturar essas ideias. Note que essa seção não é a especificação da solução para implementar um caso de uso. Ela serve somente para capturar ideias de implementação relevantes que passam pela cabeça do modelador do caso de uso, enquanto o está descrevendo. Note, também, que esta seção (assim como a de histórico) não deve ser utilizada na atividade de validação (consulte a [Seção 2.1.2](#)).

4.5 Documentação suplementar ao MCU

O MCU capture os requisitos funcionais e força o desenvolvedor a pensar em como os agentes externos interagem com o sistema. No entanto, esse modelo corresponde somente aos requisitos funcionais. Outros tipos de requisitos (desempenho, interface, segurança, regras do negócio etc.) que fazem parte do *documento de requisitos* de um sistema não são considerados pelo modelo de casos de uso. Está fora do escopo deste texto introdutório oferecer uma descrição detalhada sobre como documentar todos os possíveis tipos de requisitos de um sistema.

Entretanto, esta seção descreve como alguns dos demais itens da especificação de requisitos podem estar relacionados ao modelo de casos de uso. Os itens aqui considerados são os seguintes:

- Regras do negócio
- Requisitos de interface
- Requisitos de desempenho

4.5.1 Regras do negócio

Regras do negócio são políticas, condições ou restrições que devem ser consideradas na execução dos processos existentes em uma organização (GOTTESDIENER, 2001). As regras do negócio constituem uma parte importante dos processos organizacionais, porque descrevem a maneira como a organização funciona. O termo “regra de negócio” é utilizado mesmo em organizações que não se caracterizam como empresariais.

Cada organização pode ter várias regras do negócio. As regras do negócio de uma organização são normalmente identificadas nas fases de levantamento de requisitos de análise. Essas regras são documentadas no chamado *modelo de regras do negócio*. Nesse modelo, as regras são categorizadas. Cada regra normalmente recebe um identificador (assim como acontece para cada caso de uso). Esse identificador permite que a regra seja facilmente referenciada nos demais artefatos do processo de desenvolvimento.

A descrição do modelo de regras do negócio pode ser feita utilizando-se um texto informal ou alguma forma de estruturação. Alguns exemplos de regras do negócio (não pertencentes a uma mesma organização) são apresentados aqui:

- O valor total de um pedido é igual à soma dos totais dos itens do pedido acrescido de 10% (taxa de entrega).
- Um professor só pode estar lecionando disciplinas para as quais esteja habilitado.
- Um cliente do banco não pode retirar mais de R\$ 1.000,00 por dia de sua conta.
- Os pedidos para um cliente não especial devem ser pagos antecipadamente.
- Para alugar um carro, o proponente deve estar com a carteira de motorista válida.
- O número máximo de alunos por turma é igual a trinta.
- Um aluno deve ter a matrícula cancelada se obtiver dois conceitos D no curso.
- Uma vez que um professor confirma as notas de uma turma, elas não podem ser modificadas.
- Senhas devem ter, no mínimo, seis caracteres, entre números e letras, e devem ser atualizada a cada três meses.

As regras do negócio normalmente têm influência sobre a lógica de execução de um ou mais casos de uso. Por exemplo, considere a última regra ilustrada anteriormente. Essa regra implica que deve haver uma maneira de informar ao usuário quando uma atualização é necessária e um modo pelo qual o usuário possa atualizar a sua senha, o que tem influência no modelo de casos de uso do sistema.

Para conectar uma regra a um caso de uso no qual ela é relevante deve ser utilizado o *identificador da regra do negócio* que influenciar no caso de uso em questão. Exemplos disso podem ser encontrados nos casos de uso do Sistema de Controle Acadêmico, um estudo de caso de modelagem que desenvolvemos neste livro (ver [Seção 4.7.3](#)).

A [Tabela 4-2](#) fornece um formulário que pode ser utilizado para construir o modelo de regras do negócio. Este formulário apresenta o nome da regra de negócio, o seu identificador, a descrição da regra, a fonte de informação que permitiu defini-la (normalmente um especialista do domínio) e um histórico de evolução da regra (p. ex., data de identificação, data de última atualização etc.).

Tabela 4-2: Possível formato para documentação de uma regra de negócio

Nome	Quantidade de inscrições possíveis (RN01)
Descrição	Um aluno não pode se inscrever em mais de seis disciplinas por semestre letivo.
Fonte	Coordenador da escola de informática.
Histórico	Data de identificação: 12/7/2002.

Para finalizar a discussão sobre regras do negócio é importante notar que há outras formas de especificação de uma regra de negócio, além da forma simplesmente descritiva. Por exemplo, o *diagrama de atividades* pode ser utilizado para representar graficamente uma regra do negócio (ver [Capítulo 10](#)). A *OCL* (ver [Seção 3.4](#)) é outra forma de especificar regras.

4.5.2 Requisitos de desempenho

O MCU também não considera *requisitos de desempenho*. Um requisito de desempenho define características relacionadas à operação do sistema. Exemplos: número esperado de transações por

unidade de tempo, tempo máximo esperado para uma operação, volume de dados que deve ser tratado etc.

Alistair Cockburn recomenda em seu livro que se utilize uma tabela para ilustrar os requisitos de desempenho e suas associações com os casos de uso do sistema (COCKBURN, 2004). A [Tabela 4-3](#) exibe uma adaptação de um exemplo encontrado neste livro. Nesse exemplo, são apresentados os identificadores de casos de uso na primeira coluna e requisitos de desempenho nas demais.

Tabela 4-3: Tabela conectando casos de uso a requisitos de desempenho

Identificador do caso de uso	Frequência da utilização	Tempo máximo esperado...	...
CSU01	5/mês	Interativo	...
CSU02	15/dia	1 segundo	...
CSU03	60/dia	Interativo	...
CSU04	180/dia	3 segundos	...
CSU05	600/mês	10 segundos	...
CSU07	500/dia durante 10 dias seguidos	10 segundos	...

4.5.3 Requisitos de interface gráfica

A especificação dos requisitos de um sistema pode conter também uma seção que descreva os requisitos de interface do sistema. Por exemplo, o cliente pode ter definido restrições específicas com respeito à interface do sistema: cor, estilo, interatividade etc.⁶ É possível que os requisitos de interface estejam relacionados a um ou mais casos de uso do sistema. Esse relacionamento pode ser feito de uma forma semelhante à da [Tabela 4-3](#).

4.6 O MCU em um processo de desenvolvimento iterativo

Casos de uso formam uma base natural pela qual é possível planejar e realizar as iterações do desenvolvimento. Para isso, os casos de uso devem ser divididos em grupos. Cada grupo é alocado a uma iteração.⁷ A partir daí, o desenvolvimento do sistema segue a alocação realizada: em cada iteração, o grupo de casos de uso correspondente é detalhado e desenvolvido. O processo continua até que todos os grupos de casos de uso tenham sido desenvolvidos, e o sistema esteja completamente construído.

Conforme mencionado há pouco nesta seção, um fator importante para o sucesso do desenvolvimento do sistema é considerar os casos de uso mais importantes primeiramente. Murray Cantor (1998) propõe uma classificação dos casos de uso identificados para um sistema em função de dois parâmetros: *risco de desenvolvimento* e *prioridades estabelecidas pelo usuário*. Dessa forma, cada caso de uso se encaixa em uma das categorias a seguir:

- Risco alto e prioridade alta:** casos de uso nesta categoria são os mais críticos e, portanto, devem ser considerados o quanto antes.
- Risco alto e prioridade baixa:** embora os casos de uso nesta categoria tenham risco alto, é necessário, antes de começar a considerá-los, negociar com o cliente em relação à sua verdadeira necessidade.

3. **Risco baixo e prioridade alta:** embora os casos de uso tenham prioridade alta, é necessário ter em mente que os de mais alto risco precisam ser considerados primeiro.
4. **Risco baixo e prioridade baixa:** em situações em que o desenvolvimento do sistema está atrasado, estes casos de uso são os primeiros a serem “cortados”.

Na atribuição de importância categorizada por Cantor, um caso de uso não tão importante não será contemplado nas iterações iniciais. Se o requisito correspondente a esse caso de uso for modificado ou não mais precisar ser considerado, os analistas não terão desperdiçado tempo com ele.

Note também que a descrição expandida de um determinado caso de uso é normalmente feita somente na iteração durante a qual ele deve ser implementado. Essa abordagem evita que se perca tempo inicialmente no seu detalhamento. Além disso, essa estratégia é mais adaptável aos *requisitos voláteis*. Isso porque, se todos os casos de uso forem detalhados inicialmente, e se um ou mais requisitos são modificados durante o desenvolvimento, toda a modelagem correspondente a esses casos de uso sofrerá modificações. Por outro lado, se a descrição detalhada é deixada para a iteração à qual o caso de uso foi alocado, uma eventual mudança dos requisitos associados a esse caso de uso não afetará tão profundamente o desenvolvimento.

A construção do MCUS deve se adequar ao processo de desenvolvimento sendo utilizado. Os casos de uso mais arriscados devem ser considerados primeiramente.

4.6.1 O MCUS nas atividades de análise e projeto

O modelo de casos de uso é tipicamente um artefato da fase de análise (ver [Seção 2.1.2](#)) do desenvolvimento de um sistema. Por outro lado, consideramos neste capítulo um tipo especial de modelagem de casos de uso, a *modelagem de casos de uso de sistema* (MCUS). Nessa modelagem o objeto sendo modelado é um sistema de software, que tem o objetivo de automatizar um ou mais processos de negócio de uma organização. No entanto, casos de uso também podem ser utilizados para modelar um objeto em um escopo mais amplo, a saber, a organização como um todo. Essa atividade é conhecida como *modelagem de casos de uso de negócio*.

A MCUN é uma extensão do conceito de casos de uso para descrever os processos do negócio de uma organização. Essa atividade é realizada na fase de *análise do domínio* (também conhecida como *modelagem do negócio* ou *modelagem dos processos do negócio*; ver [Seção 2.1.2](#)). A MCUN interpreta o sistema como sendo a própria organização empresarial; as funcionalidades são os processos empresariais. Um modelo de casos de uso de negócio serve para estabelecer o escopo do sistema desejado, ou seja, que processos do negócio devem ser automatizados, que processos não o são, e que partes devem ser atacadas com mudanças no funcionamento da organização. A modelagem de casos de uso de negócio não pertence ao escopo deste livro, onde tratamos apenas da MCUS. De qualquer modo, um modelo de casos de uso de sistema deve dar suporte a um ou mais modelos de casos de uso de negócio.

Alguns desenvolvedores escrevem as descrições iniciais de casos de uso mencionando detalhes de interface gráfica com o usuário (considerando atores humanos). A justificativa é que a narrativa dos casos de uso segundo essa abordagem fornece uma ideia mais concreta de como se apresentará uma determinada funcionalidade do sistema. Contudo, as desvantagens dessa abordagem se sobrepõem às vantagens. Considere a situação de uma parte da interface gráfica ser modificada, por

alguma razão. Nessa situação, a desvantagem de utilização de casos de uso reais (ver [Seção 4.1.1.3](#)) se torna nítida, pois o fato de a interface ser modificada possivelmente resultará na modificação da narrativa do caso de uso. Além disso, lembre-se do objetivo principal do modelo de casos de uso: dar forma aos requisitos funcionais do sistema. Portanto, casos de uso devem ser independentes do desenho da interface pelo fato de que os requisitos do sistema não devem estar associados a detalhes de interface. Uma melhor abordagem é utilizar inicialmente casos de uso essenciais (ver [Seção 4.1.1.3](#)) para não acoplar os detalhes da interface da aplicação na especificação narrativa das interações de um caso de uso. Nessa especificação, a atenção do modelador deve recair sobre a essência das interações entre atores e o sistema, em vez de sobre como cada interação é realizada fisicamente. Especificações de casos de uso feitas dessa forma tornam-se mais imunes a futuras mudanças na interface com o usuário, além de permitir que o analista de sistemas se concentre no que é realmente importante em uma narrativa de caso de uso: as interações entre ator(es) e sistema. Por exemplo, considere o termo “envia uma requisição” em contraposição com “duplo clique sobre o botão de envio de requisições”. Em resumo, casos de uso que mencionam detalhes de interface gráfica são indesejáveis durante a análise. O mais adequado é utilizar casos de uso essenciais e, posteriormente, na etapa de projeto, transformá-los em reais adicionando mais detalhes.

Na fase de análise, descrições de casos de uso devem capturar os requisitos funcionais do sistema e ignorar aspectos de projeto, como a interface gráfica com o usuário.

Descrevemos agora um procedimento que pode ser utilizado na construção do MCU em um processo de desenvolvimento iterativo. (Para descrição das fases aqui mencionadas, ver [Seção 2.3.2.1](#).)

1. Identifique os atores e casos de uso na fase de concepção. Alguns atores e casos de uso só serão identificados posteriormente, mas a maioria deve ser descoberta nesta fase.
2. Na fase de elaboração:
 - a. Desenhe o(s) diagrama(s) de casos de uso.
 - b. Escreva os casos de uso em um formato de alto nível e essencial.
 - c. Ordene a lista de casos de uso de acordo com prioridade e risco. Cada partição corresponde a um grupo de casos de uso que será implementado em um dos ciclos de desenvolvimento do sistema.
3. Associe cada grupo de casos de uso a uma iteração da fase de construção. Os grupos mais prioritários e arriscados devem ser alocados às iterações iniciais.
4. Na *i*-ésima iteração da fase de construção:
 - a. Detalhe os casos de uso do grupo associado a esta iteração (se necessário, utilize o nível de abstração real).
 - b. Implemente estes casos de uso.

4.6.2 O MCU e outras atividades do desenvolvimento

O modelo de casos de uso direciona a realização de várias outras atividades do desenvolvimento.

4.6.2.1 Planejamento e gerenciamento do projeto

O modelo de casos de uso é uma ferramenta fundamental para o gerente de um projeto no planejamento e controle de um processo de desenvolvimento iterativo. Ao final de cada iteração, o gerente pode avaliar a produtividade na realização das tarefas. Essa avaliação serve como massa de dados para que esse profissional realize a alocação das tarefas e dos recursos para as próximas iterações.

4.6.2.2 Testes de sistema

N a [Seção 2.1.5](#), os testes de sistema foram resumidamente descritos. Em um processo de desenvolvimento iterativo, não há apenas uma única fase de testes de sistema. Ao contrário, esses testes são realizados continuamente durante todo o desenvolvimento, no fim da iteração. Os profissionais responsáveis pelos testes de sistema utilizam o modelo de casos de uso para planejar as atividades de teste. Os casos de uso e seus cenários oferecem *casos de teste*. Quando o sistema está sendo testado, os cenários do sistema podem ser verificados para identificar a existência de erros.

4.6.2.3 Documentação do usuário

Os manuais e guias do usuário também podem ser construídos com base no modelo de casos de uso. Na verdade, se o modelo de casos de uso foi bem construído, provavelmente há uma correspondência clara entre cada caso de uso do sistema e uma seção do manual do usuário. Isso porque esse modelo está baseado na noção de que o sistema é construído para se adequar à perspectiva de seus usuários.

4.7 Estudo de caso

A partir desta seção, um estudo de caso começa a ser desenvolvido. Esse estudo tem o objetivo de consolidar os principais conceitos teóricos descritos e oferecer uma visão prática sobre como os modelos apresentados neste livro são desenvolvidos. Batizamos o sistema de nosso estudo de caso com o nome de *Sistema de Controle Acadêmico* (SCA).

O desenvolvimento de nosso estudo de caso é feito de forma incremental: em cada capítulo deste livro em que houver uma seção denominada “Estudo de caso”, uma parte do desenvolvimento é apresentada. É importante notar que, para manter a descrição em um nível de simplicidade e clareza aceitáveis para um livro didático, muitos detalhes do desenvolvimento são deliberadamente ignorados.

4.7.1 Descrição da situação

O estudo de caso constitui-se de uma instituição de ensino que precisa de uma aplicação para controlar alguns processos acadêmicos, como inscrições em disciplinas, lançamento de notas, alocação de recursos para turmas, etc. Após o levantamento de requisitos inicial desse sistema, os analistas chegaram à seguinte lista de requisitos funcionais:

- R1.** O sistema deve permitir que alunos visualizem as notas obtidas por semestre letivo.
- R2.** O sistema deve permitir o lançamento das notas das disciplinas lecionadas em um semestre letivo e controlar os prazos e atrasos neste lançamento.

- R3.** O sistema deve manter informações cadastrais sobre disciplinas no currículo escolar.
- R4.** O sistema deve permitir a abertura de turmas para uma disciplina, assim como a definição de salas e laboratórios a serem utilizadas e dos horários e dias da semana em que haverá aulas de tal turma.
- R5.** O sistema deve permitir que os alunos realizem a inscrição em disciplinas de um semestre letivo.
- R6.** O sistema deve permitir o controle do andamento das inscrições em disciplinas feitas por alunos.
- R7.** O sistema deve se comunicar com o *Sistema de Recursos Humanos* para obter dados cadastrais dos professores.
- R8.** O sistema deve se comunicar com o *Sistema de Faturamento* para informar as inscrições realizadas pelos alunos.
- R9.** O sistema deve manter informações cadastrais dos alunos e de seus históricos escolares.

4.7.2 Regras do negócio

Algumas regras iniciais do negócio também foram identificadas para o sistema. Essas regras são descritas a seguir, utilizando o formato proposto na [Seção 4.5.1](#) (por simplificação, a fonte e o histórico das regras são omitidos).

Choques de horários (RN00)	
Descrição	Em um semestre letivo, um aluno não pode se inscrever em turmas de disciplinas para as quais haja choque de horários.

Quantidade máxima de inscrições por semestre letivo (RN01)	
Descrição	Em um semestre letivo, um aluno não pode se inscrever em uma quantidade de turmas cuja soma de créditos nas disciplinas correspondentes ultrapasse vinte.

Quantidade de alunos possíveis (RN02)	
Descrição	Uma turma não pode ter mais alunos inscritos do que a capacidade máxima definida para ela.

Pré-requisitos para uma disciplina (RN03)	
Descrição	Um aluno não pode se inscrever em uma turma de uma disciplina para a qual não possua os pré-requisitos necessários. Além disso, um aluno não pode se inscrever em uma turma de alguma disciplina que já tenha cursado com aprovação.

Habilitação para lecionar disciplina (RN04)	
Descrição	Um professor só pode lecionar disciplinas para as quais esteja habilitado.

Cancelamento de matrícula (RN05)	
Descrição	Um aluno deve ter a matrícula cancelada no curso se for reprovado, por média ou por faltas, mais de duas vezes na mesma disciplina.

Política de Avaliação de Alunos (RN06)

Descrição	<p>A nota parcial (NP) de um aluno em uma turma (um valor de 0 a 10) é obtida pela média simples de duas avaliações durante o semestre, A1 e A2, ou pela frequência nas aulas.</p> <ul style="list-style-type: none">• Se o aluno obtém NP maior ou igual a 7,0 (sete), então sua média final (MF) é igual NP, e ele está aprovado por média.• Se o aluno obtém NP menor do que 3,0 (três), então MF é igual a NP e ele está reprovado por média.• Se o aluno obtém NP maior ou igual 5,0 (cinco) e menor que 7,0 (sete), deve fazer o exame final, que resulta em uma nota, NF. Nesse caso, MF é igual à média simples entre NF e NP. Se MF for igual ou maior do que 5,0, está aprovado por média. Em caso contrário, está reprovado por média.• Se o aluno tiver uma frequência menor do que 75% em uma turma, está automaticamente reprovado por faltas, independentemente do valor de MF.
-----------	---

Prioridade em Listas de Espera (RN07)

Descrição	Ao criar uma nova turma para alojar alunos que estão em uma lista de espera, a ordem de entrada nesta lista deve ser respeitada (i.e., alunos esperando a mais tempo têm maior prioridade).
-----------	---

4.7.3 Documentação do MCU

Nesse sistema de controle acadêmico, o analista identificou e documentou os seguintes atores:

- *Aluno*: indivíduo que está matriculado na instituição de ensino com interesse em se inscrever em turmas de disciplinas do curso.
- *Professor*: indivíduo que leciona disciplinas na faculdade.
- *Coordenador*: pessoa responsável por agendar as alocações de turmas e professores e visualizar o andamento de inscrições dos alunos.
- *Departamento de Registro Escolar (DRE)*: departamento da faculdade interessado em manter informações sobre os alunos matriculados e sobre seu histórico escolar.
- *Sistema de Recursos Humanos*: este sistema legado é responsável por fornecer informações cadastrais sobre os professores.
- *Sistema de Faturamento*: este sistema legado tem interesse em obter informações sobre inscrições dos alunos para realizar o controle de pagamento de mensalidades.

O analista também identificou os casos de uso a seguir e os organizou em três pacotes: *Gerenciamento de Inscrições*, *Gerenciamento de Recursos Acadêmicos* e *Acompanhamento de Semestre Letivo*. Os casos de uso que apresentam comentários (entre parênteses e em itálico) são aqueles para os quais *não* fornecemos descrições detalhadas.

- Gerenciamento de Inscrições
 - Realizar Inscrição
 - Cancelar Inscrição
 - Visualizar Grade Curricular (Aluno visualiza a grade curricular atual; ver [Seção 5.7.2](#) para a definição deste termo)

- Visualizar Andamento de Inscrições
- Abrir Turma (Coordenador abre uma turma)
- Fechar Turma (Coordenador fecha uma turma)
- Atender Listas de Espera
- Gerenciamento de Recursos Acadêmicos
 - Manter Grade Curricular (Coordenador define informações sobre uma grade curricular; ver definição deste termo no glossário)
 - Manter Disciplina
 - Manter Aluno (DRE mantém informações sobre aluno)
 - Fornecer Grade de Disponibilidade
 - Fornecer Habilidades (Professor informa as disciplinas da grade curricular que está apta a lecionar)
 - Atualizar Informações sobre Professor
- Acompanhamento de Semestre Letivo
 - Lançar Avaliações
 - Obter Diário de Classe (Professor obtém o diário de classe para determinado mês do semestre letivo corrente. Ver [Seção 5.7.2](#) para a definição do termo “diário de classe”)
 - Visualizar Avaliações
 - Solicitar Histórico Escolar (Aluno solicita a produção de seu histórico escolar)

A seguir são apresentados o diagrama de casos de uso e as descrições de alguns casos no formato essencial e expandido. A descrição dos demais casos de uso fica como exercício para o leitor.

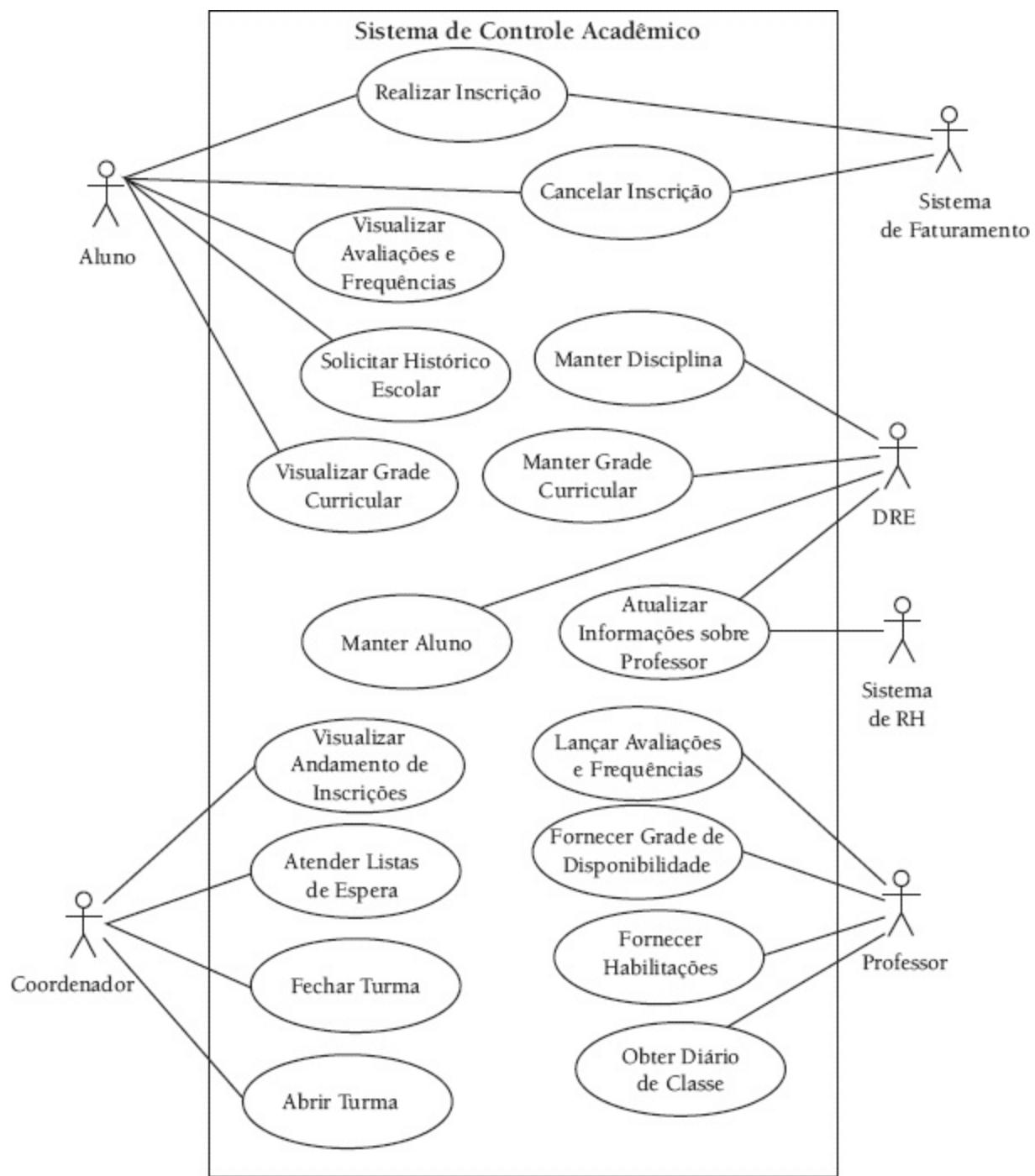


Figura 4-11: Diagrama de casos de uso para o SCA.

Cancelar Inscrições (CSU00)

Sumário: Aluno cancela inscrições em uma ou mais turmas em que havia solicitado inscrição.

Ator Primário: Aluno.

Atores Secundários: Sistema de Faturamento.

Precondições:

Período de cancelamento de inscrições está aberto.

O aluno está identificado pelo sistema.

Fluxo Principal

1. Aluno solicita o cancelamento de inscrições.
2. Sistema apresenta as disciplinas (e respectivos códigos das turmas) em que o aluno está inscrito para o semestre letivo corrente.

3. Aluno define a lista de turmas para as quais deseja cancelar a inscrição e as submete para cancelamento.
4. Sistema realiza o cancelamento das inscrições, envia os dados sobre as mesmas para o Sistema de Faturamento, e o caso de uso termina.

Regras de Negócio: N/A

Realizar Inscrição (CSU01)

Sumário: Aluno usa o sistema para realizar inscrição em disciplinas.

Ator Primário: Aluno.

Atores Secundários: Sistema de Faturamento.

Precondições: O aluno está identificado pelo sistema.

Fluxo Principal

1. Aluno solicita a realização de inscrições.
2. Sistema apresenta as disciplinas (e respectivos códigos das turmas) em que o aluno pode se inscrever (conforme a [RN03](#)).
3. Aluno escolhe a lista de turmas que deseja cursar no próximo semestre letivo e as submete para inscrição.
4. Para cada turma, o sistema informa o professor, os horários e os respectivos locais das aulas.
5. Aluno confirma as inscrições.
6. Sistema registra as inscrições do aluno, envia os dados sobre as mesmas para o Sistema de Faturamento, e o caso de uso termina.

Fluxo Alternativo (4): Não há oferta disponível para alguma disciplina selecionada pelo aluno (conforme [RN02](#)),

- a. Sistema fornece a possibilidade de inserir o aluno em uma lista de espera.
- b. Se o aluno aceitar, o sistema o insere na lista de espera e apresenta a posição na qual o aluno foi inserido na lista. O caso de uso retorna ao passo 4.
- c. Se o aluno não aceitar, o caso de uso prossegue a partir do passo 4.

Fluxo Alternativo (5): Revisão das inscrições

- Aqui, é possível que o caso de uso retorne ao passo 3, conforme o aluno queira revisar (inserir ou remover itens) a lista de disciplinas selecionadas.

Fluxo de Exceção (4): Violação de [RN00](#)

- Aluno selecionou turmas para as quais há choque de horários.
 - a. Sistema informa as turmas em que houve choque de horários, juntamente com os respectivos horários de cada uma, e o caso de uso retorna ao passo 2.

Fluxo de Exceção (4): Violação de [RN01](#)

- Aluno atingiu a quantidade máxima de inscrições possíveis.
 - a. Sistema informa a quantidade de créditos que o aluno pode selecionar, e o caso de uso retorna ao passo 2.

Pós-condições: O aluno foi inscrito em uma turma de cada uma das disciplinas desejadas, ou foi adicionado a uma ou mais listas de espera.

Regras de Negócio: [RN00](#), [RN01](#), [RN02](#), [RN03](#)

Visualizar Avaliações (CSU02)

Sumário: Aluno visualiza avaliação que recebeu (notas e frequência) nas turmas de um semestre letivo.

Ator Primário: Aluno.

Precondições: O Aluno está identificado pelo sistema.

Fluxo Principal

1. O Aluno solicita a visualização das avaliações para as turmas em que participou.
2. O sistema exibe os semestres letivos nos quais o Aluno se inscreveu em pelo menos uma turma cujo lançamento já tenha sido encerrado.
3. O Aluno seleciona os semestres letivos cujas avaliações deseja visualizar.
4. O sistema exibe uma lista de avaliações agrupadas por semestres letivos selecionados e por turma, e o caso de uso termina.

Fluxo de Exceção (2): Aluno sem inscrição

a. Não há semestre letivo no qual o Aluno tenha participado de alguma oferta de disciplina: o sistema reporta o fato, e o caso de uso termina.

Fornecer Grade de Disponibilidades (CSU03)

Sumário: Professor fornece a sua grade de disponibilidade (disciplinas que deseja lecionar, juntamente com dias e horários em que está disponível) para o próximo semestre letivo.

Autor Primário: Professor.

Precondições: N/A.

Fluxo Principal

1. Professor fornece sua matrícula para validação.
2. Sistema apresenta a lista de disciplinas disponíveis (conforme RN04), e a lista de dias da semana e de horários do semestre letivo seguinte.
3. Professor informa (1) cada disciplina que deseja lecionar e (2) cada disponibilidade para o próximo semestre letivo.
4. Professor solicita o registro da grade.
5. Sistema registra a grade e o caso de uso termina.

Fluxo Alternativo (3): Modificação na grade atual

- a. Professor solicita que o sistema apresente a mesma grade do semestre atual.
- b. Sistema apresenta a configuração de grade requisitada.
- c. Professor realiza as modificações que deseja na grade e solicita o seu registro.
- d. Sistema registra a grade alterada e o caso de uso termina.

Fluxo de Exceção (4): O professor não forneceu disciplina alguma.

- a. Sistema reporta o fato e o caso de uso continua a partir do passo 3.

Fluxo de Exceção (4): O professor não forneceu dias e horários.

- a. Sistema reporta o fato e o caso de uso continua a partir do passo 3.

Pós-condições: o sistema registrou a disponibilidade do Professor para o próximo semestre letivo.

Regras de Negócio: [RN04](#)

Lançar Avaliações (CSU04)

Sumário: Professor realiza o lançamento de avaliações (notas e frequências) para alunos de uma turma sob sua responsabilidade no semestre letivo corrente.

Autor Primário: Professor.

Precondições: O professor está identificado pelo sistema.

Fluxo Principal

1. O professor solicita o lançamento de avaliações.
2. O sistema exibe a lista de turmas do semestre letivo corrente nas quais o professor lecionou.
3. O professor seleciona a turma para a qual deseja realizar o lançamento.
4. Para cada aluno inscrito na turma selecionada, o sistema nome e matrícula e requisita a primeira nota (A1), a segunda nota (A2) e a quantidade de faltas.
5. O professor fornece as notas de A1 e de A2 e a quantidade de faltas para cada aluno.
6. O sistema exibe o resultado da avaliação de cada aluno, conforme regra de negócios [RN06](#).
7. O professor confirma o lançamento.
8. O sistema registra as avaliações, e o caso de uso termina.

Fluxo Alternativo (7): Erro no lançamento. (O professor detecta que lançou uma avaliação errada para algum aluno.)

- a. O professor corrige a informação lançada erroneamente.
- b. O sistema aceita a correção, e o caso de uso continua a partir do passo 7.

Fluxos de Exceção (4): Avaliação em branco ou errada.

- a. O professor não fornece alguma nota, ou frequência, ou fornece dados inválidos:

- b. O sistema reporta o fato e o caso de uso retorna ao passo 4.

Pós-condições: as avaliações de uma turma lecionada pelo professor foram lançadas no sistema.

Regras de Negócio: RN05, RN06

Manter Disciplina (CSU05)

Sumário: DRE realiza o cadastro (inclusão, remoção, alteração e consulta) dos dados sobre disciplinas.

Autor Primário: DRE.

Fluxo Principal

1. DRE requisita a manutenção de disciplinas.
2. O sistema apresenta as operações que podem ser realizadas: a inclusão de uma nova disciplina, a alteração dos dados de uma disciplina, a exclusão de uma disciplina e a consulta de disciplinas.
3. O DRE indica a opção a realizar ou opta por finalizar o caso de uso.
4. O DRE seleciona a operação desejada: Inclusão, Exclusão, Alteração ou Consulta.
5. Se o DRE deseja continuar com a manutenção, o caso de uso retorna ao passo 2; caso contrário, o caso de uso termina.

Fluxo Alternativo (4): Inclusão

- a. O DRE requisita a inclusão de uma disciplina.
- b. O sistema apresenta um formulário em branco para que os detalhes da disciplina (código, nome e quantidade de créditos) sejam incluídos.
- c. O DRE fornece os detalhes da nova disciplina.
- d. O sistema apresenta uma lista de disciplinas para que o DRE selecione as que são pré-requisitos para a disciplina a ser criada.
- e. O DRE define zero ou mais disciplinas como pré-requisitos.
- f. O sistema verifica a validade dos dados. Se os dados forem válidos, inclui a nova disciplina; caso contrário, o sistema reporta o fato, solicita novos dados e repete a verificação.

Fluxo Alternativo (4): Remoção

- a. O DRE seleciona uma disciplina e requisita o sistema que a remova.
- b. Se a disciplina pode ser removida, o sistema realiza a remoção; caso contrário, o sistema reporta o fato.

Fluxo Alternativo (4): Alteração

- a. O DRE altera um ou mais dos detalhes sobre uma disciplina e requisita a sua atualização.
- b. O sistema verifica a validade dos dados e, se eles forem válidos, altera os dados na lista de disciplinas da faculdade.

Fluxo Alternativo (4): Consulta

- a. O DRE solicita a realização de uma consulta sobre a lista de disciplinas.
- b. O sistema apresenta uma lista com os códigos de todas as disciplinas, permitindo que o usuário selecione a disciplina desejada.
- c. O DRE seleciona uma disciplina.
- d. O sistema apresenta os detalhes da disciplina e seus pré-requisitos (se existirem) no formulário de disciplinas.

Pós-condições: uma disciplina foi inserida ou removida, ou seus detalhes foram alterados.

Visualizar Andamento de Inscrições (CSU06)

Sumário: O Coordenador usa o sistema para visualizar o andamento de inscrições sendo realizadas pelos alunos em turmas para o próximo semestre letivo.

Autor Primário: Coordenador.

Precondições: O coordenador está identificado pelo sistema.

Fluxo Principal

1. O coordenador solicita a visualização do andamento de inscrições realizadas pelos alunos.
2. O sistema exibe a lista de turmas ofertadas para o próximo semestre (nome professor responsável, horários e locais de aula e situação [aberta ou fechada]).
3. O coordenador seleciona a turma para a qual deseja visualizar o andamento de inscrições.
4. O sistema apresenta a lista de alunos inscritos na turma selecionada (ordenada por data de inscrição). Para cada inscrição, o

Atualizar Informações sobre Professor (CSU07)

Sumário: DRE do sistema usa o sistema para atualizar as informações cadastrais sobre professores a partir do SRH.

Autor Primário: DRE.

Autor Secundário: Sistema de Recursos Humanos (SRH).

Precondições: O DRE está identificado pelo sistema.

Fluxo Principal

1. O DRE solicita ao sistema que obtenha os dados atualizados sobre professores.
2. O sistema se comunica com o SRH e obtém os dados a partir deste.
3. O sistema apresenta os dados obtidos e solicita a confirmação do DRE para realizar a atualização.
4. O DRE confirma a atualização.
5. O sistema atualiza os dados cadastrais dos professores, e o caso de uso termina.

Fluxo de Exceção (2): Houve uma falha na obtenção de dados

- a. O sistema não consegue obter os dados a partir do SRH.
- b. O sistema reporta o fato e o caso de uso termina.

Fluxo Alternativo (4): Desistência de atualização

O DRE declina da atualização, e o caso de uso termina.

Atender Listas de Espera (CSU08)

Sumário: Coordenador cria uma turma para atender às demandas representadas por uma lista de espera por vagas em turma de uma disciplina.

Autor Primário: Coordenador.

Atores Secundários: Sistema de Faturamento.

Precondições: O coordenador está identificado pelo sistema.

Fluxo Principal

1. O sistema apresenta as listas de espera abertas para as disciplinas.
2. O coordenador seleciona uma das listas de espera.
3. O sistema apresenta os detalhes da lista de espera selecionada (data de criação e quantidade de alunos).
4. O coordenador fornece o código, dias e respectivos horários de aula da turma a criar.
5. O sistema apresenta as salas e os professores disponíveis nos dias e horários fornecidos.
6. O coordenador seleciona o professor e uma ou mais salas para a turma, assim como a quantidade de alunos a serem alocados para a criação da nova turma (conforme RN02).
7. O sistema cria a turma e transfere a quantidade de alunos fornecida no passo 7 da lista de espera para a oferta recém-criada, de acordo com a ordem dos alunos nessa lista.
8. O sistema envia os dados de inscrições de alunos para o sistema de faturamento, e o caso de uso termina.

Fluxo de Exceção (7): A quantidade de alunos que o Coordenador fornece é inválida (viola a RN02),

- a. O sistema reporta o fato e solicita um novo valor.
- b. O coordenador corrige o valor e o caso de uso prossegue a partir do passo 8.

Regras de Negócio: RN02

Pós-condições:

- Uma turma foi criada.
- Alunos selecionados foram inscritos na turma recém-criada e removidos da lista de espera.

Fornecer Habilidades (CSU09)

Sumário: Professor informa suas habilitações, i.e., as disciplinas da grade curricular que está apto a lecionar.

Autor Primário: Professor.

Precondições: O professor está identificado pelo sistema.

Fluxo Principal

1. O professor solicita o registro de suas habilitações.
2. O sistema apresenta duas listas de disciplinas: a lista de disciplinas que compõem as habilitações atuais do professor (La); lista de todas as disciplinas da grade curricular atual do curso, à exceção das contidas em La (Lb).
3. O professor solicita a transferência de itens entre (La) e (Lb), conforme queira remover ou adicionar habilitações.
4. O sistema atualiza as listas La e Lb apresentadas, de acordo com as transferências solicitadas.
5. O professor confirma o registro das habilitações, e o caso de uso termina.

Pós-condições:

- A lista de disciplinas correspondente às habilitações do professor foi atualizada.

► EXERCÍCIOS

4-1: Descreva a posição do diagramas de casos de uso no processo de desenvolvimento iterativo. Quando eles são utilizados? Para que são utilizados?

4-2: Construa um modelo de casos de uso para a seguinte situação fictícia: Estamos criando um serviço de entregas. Nossos clientes podem nos requisitar a entrega de volumes. Alguns volumes são considerados de maior valor por nossos clientes, e, portanto, eles querem ter tais volumes segurados durante o transporte. Contratamos uma companhia de seguro para segurar volumes de valor.

4-3: Observe a seguinte narrativa do caso de uso Realizar Saque. Identifique os erros existentes nela. Construa uma nova versão deste caso de uso que não contenha os erros encontrados.

A operação de um caixa eletrônico tem início a partir de uma sessão em que o cliente seleciona a opção de realizar saque. O cliente, então, escolhe uma quantia a ser retirada, a partir de um conjunto de opções de quantia disponíveis.

O sistema verifica se a conta correspondente tem saldo suficiente para satisfazer a requisição. Senão, uma mensagem adequada é reportada, o que acarreta na execução da extensão. Se há dinheiro suficiente, os números da conta e da agência do cliente são enviados ao banco, que aprova ou desaprova a transação. Se a transação é aprovada, a máquina libera a quantia correspondente e emite um recibo. Se a transação é reprovada, a extensão Informar Falha é executada.

O banco é notificado, independentemente de uma transação aprovada ter sido completada ou não pela máquina. Se a transação é concluída, o banco realiza o débito na conta do cliente (BJORK, 1998).

4-4: Qual é a notação da UML para um *caso de uso*? Qual é a notação da UML para um ator? Qual a notação utilizada na UML para o relacionamento de generalização?

4-5: Defina o que significa um ator. O que significa um ator estar associado a um caso de uso por um relacionamento de comunicação?

4-6: Qual o objetivo dos diagramas de *casos de uso*?

4-7: Defina o conceito de requisito. Que tipos de requisitos existem? Explique o que é realizado na fase de levantamento de requisitos de um sistema de informações.

4-8: Que tipo de relacionamento é possível entre um ator e um caso de uso? Que tipo de relacionamento pode haver entre casos de uso? Que tipo de relacionamento pode haver entre atores?

4-9: Descreva a(s) diferença(s) entre os relacionamentos de inclusão, de extensão e de herança.

4-10: Considere um sistema de controle de uma biblioteca. Forneça a descrição narrativa para os seguintes casos de uso: Reservar Livro (situação em que um usuário faz a reserva de um livro), Obter Empréstimo de Livro (situação em que um usuário pega um exemplar de livro emprestado), Cancelar Reserva (situação em que um usuário cancela uma reserva) e Devolver Cópia (situação em que um usuário devolve uma cópia anteriormente adquirida).

4-11: Durante a execução de um caso de uso, podem ocorrer exceções. Considere o caso de uso Realizar Pedido, no qual pode ser que o cliente solicite um produto que está fora de estoque. Como você modelaria tal situação? Desenhe um diagrama de casos de uso.

4-12: Construa o modelo de casos de uso para a seguinte situação. Tente identificar também regras de negócio que se apliquem a ela, de acordo com o texto fornecido.

Uma rede de televisão está requisitando um sistema para gerenciar informações sobre uma de suas produções televisivas (p. ex., uma minissérie ou uma novela).

Uma produção televisiva tem uma verba e é composta de cenas. Cenas são escolhidas em uma determinada sequência. Cada cena, que tem uma duração em minutos, é gravada em uma ou mais fitas. Cada fita possui um número de série e uma capacidade (medida em minutos que podem ser gravados na mesma). Deseja-se saber em que fita(s) se encontra uma determinada cena. Cada cena pode ter sido gravada

muitas vezes (futuramente, na edição da obra, o produtor selecionará uma dessas tomadas de cena para compor a versão final da produção televisiva). Deve-se manter o registro de todas as cenas filmadas, de quais atores e dublês participaram de cada cena. Deseja-se saber, também, que dublê substituiu que ator, em cada cena.

Para uma produção televisiva como um todo, deseja-se manter a informação de quais outros funcionários, os chamados funcionários de apoio, participaram das filmagens. Esses funcionários podem ser de diversos tipos (câmeras, iluminadores, contra-regras etc). Além disso, é possível haver funcionários de apoio que exerçam mais de uma função na mesma produção televisiva.

Atores e dublês ganham por produção televisiva em que participam. Os demais funcionários têm um salário fixo por obra. Também é necessário armazenar essas informações para ter uma ideia do consumo de recursos em relação à verba.

Após o término de uma obra, o sistema deve produzir um relatório com o valor a ser pago para cada funcionário. O sistema também deve produzir um relatório de informações sobre as cenas de uma obra televisiva, e sobre que atores, dublês e demais funcionários participaram dessa obra televisiva.

4-13: O seguinte documento de requisitos foi adaptado do livro (WIRFS-BROCK *et al.*, 1990). Leia o texto com atenção. A seguir, elabore um modelo de casos de uso inicial para o sistema.

O GNU Editor é um editor gráfico interativo. Com ele, usuários podem criar e editar desenhos compostos de linhas, retângulos, elipses e texto.

Há dois modos de operação do editor. Apenas um modo de operação está ativo em um dado momento.

Os dois modos de operação são: modo de seleção e modo de criação. Quando o modo de seleção está ativado, os elementos gráficos podem ser selecionados com o cursor do mouse. Um ou mais elementos gráficos podem ser selecionados e manipulados; se vários elementos gráficos forem selecionados, eles podem ser manipulados como se fossem um único elemento gráfico. Elementos que tenham sido selecionados desse modo são definidos como a “seleção atual”. A seleção atual é indicada visualmente através da exibição dos pontos de controle para o elemento. Um clique seguido de um arrasto de mouse sobre um ponto de controle modifica o elemento ao qual o ponto de controle está associado.

Quando o modo de criação está ativado, a seleção atual está vazia. O usuário pode selecionar um objeto gráfico a partir de um conjunto de objetos gráficos predefinidos.

A criação de um elemento de texto: a posição do primeiro caractere do texto é determinada pela posição na qual o usuário clica o botão do mouse. O modo de criação é desativado quando o usuário clica com o mouse fora do elemento de texto.

Os pontos de controle para um elemento de texto são posicionados nos quatro cantos da região em que o texto é inserido. O arrasto desses pontos de controle muda a região.

Os outros elementos que podem ser criados pelo usuário são linhas, retângulos e elipses. O elemento apropriado começa quando o botão do mouse é pressionado e se completa quando o botão do mouse é liberado. Esses dois eventos criam o “ponto de partida” e o “ponto de parada”.

A “criação de linha” define uma linha do ponto de partida até o ponto de parada. Esses são os pontos de controle. O arrasto de um ponto de controle modifica o ponto extremo correspondente.

A “criação de retângulo” define um retângulo tal que dois dos cantos do retângulo diametralmente opostos do retângulo correspondem ao ponto de partida e ao ponto de parada. Os cantos do retângulo formam os pontos de controle. O arrasto de um ponto de controle modifica o canto correspondente.

A “criação de elipse” define uma elipse que está contida dentro de um retângulo definido pelos dois pontos definidos acima. O raio maior da elipse é metade do comprimento do retângulo, e o seu raio menor é metade da altura do retângulo. Os pontos de controle são os cantos do retângulo que contêm a elipse. O arrasto de um ponto de controle modifica o canto correspondente.

Será assumido que o programa deve fornecer uma tela gráfica do diagrama sendo criado, e que um mouse e um teclado serão utilizados como dispositivos de entrada.

4-14: Considere a seguinte declaração obtida de um gerente de uma empresa que comercializa livros por correio durante o levantamento de requisitos para construção de um sistema de software: *Após a ordem de compra do cliente ter sido registrada, o vendedor envia uma requisição ao depósito com detalhes da ordem de compra.* Que atores em potencial podem ser identificados a partir dessa situação?

4-15: Considere o exemplo de relacionamento de extensão entre casos de uso apresentado na [Seção 4.1.3.3](#), que descreve relacionamentos de extensão entre os casos de uso Editar Documento e os extensores Corrigir Ortografia e Substituir Texto. Desenhe um diagrama de casos de uso para essa situação. Como você faria para estender seu diagrama de casos de uso com um novo requisito, a saber, permitir que o editor de textos possibilite a criação de um índice remissivo sobre um documento sendo editado?

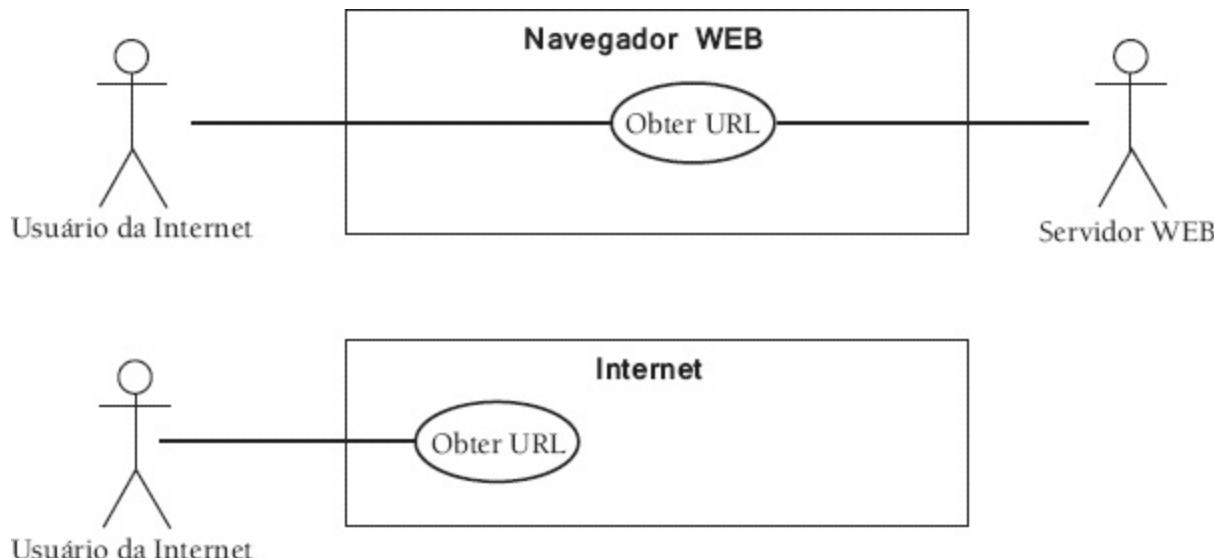
4-16: Em uma empresa, vários projetos são realizados. Os cinquenta empregados da empresa trabalham em pelos menos um projeto. Há um sistema implantado na empresa que permite aos participantes de um determinado projeto marcarem suas

horas de trabalho. Esse sistema também permite que outra pessoa, ao fim do mês, gere os relatórios com os totais de horas trabalhadas de cada participante. Quantos atores você definiria para esse sistema? E quantos papéis?

4-17: O TurboNote+ é um programa *shareware* que permite aos seus usuários criar mensagens de lembrete que permanecem na área de trabalho de seus computadores. (Esse programa funciona como uma versão eletrônica daqueles bloquinhos de papel cujas folhas podem ser afixadas na parede.) Ao criar uma nova folhinha no TurboNote+, o usuário pode preenchê-la com texto. As folhinhas podem ser movidas pela área de trabalho, conforme a vontade do usuário. As folhinhas permanecem na área de trabalho. Toda vez que o usuário inicia o seu computador, as folhinhas estão lá, na área de trabalho. Quando não são mais necessárias, elas podem ser removidas. Se o usuário escrever uma expressão aritmética em uma folhinha, o resultado da expressão é exibido. Desenhe o diagrama de casos de uso para o TurboNote+.

4-18: Suponha que um sistema de vendas deve gerar de forma automática um conjunto de estatísticas para a diretoria da empresa no último dia útil de cada mês. Desenhe o diagrama de casos de uso para essa situação. Há mais de uma maneira de representá-la?

4-19: Na utilização da Internet, um usuário normalmente abre em um programa navegador (browser) que, por sua vez, se comunica com um ou mais servidores Web para fornecer as páginas nas quais o usuário está interessado. O que está errado no diagrama a seguir? Desenhe novos diagramas para representar corretamente a situação, considerando duas alternativas de escopo. Na primeira, o programa navegador é o sistema. Na segunda, a Internet é o sistema.



4-20: Assinale V ou F para as seguintes assertivas:

() pessoas com o mesmo cargo em uma empresa podem representar papéis de diversos atores.

() um ator pode representar pessoas de diferentes cargos.

4-21: Altere os seguintes “nomes de casos de uso” de acordo com as nomenclaturas apresentadas neste capítulo:

- a. Cliente realiza transferência de fundos em um caixa eletrônico.
- b. Clientes compram livros na livraria.
- c. É produzido um relatório de vendas para o gerente.
- d. Hóspede se registra em um hotel.

4-22: Desenhe diagramas de casos de uso para os seguintes sistemas:

- a. A biblioteca de sua universidade.
- b. O seu aparelho celular.
- c. Um sistema de validação de cartões de crédito.

4-23: Suponha que existe um caso de uso Pagar Pedido em um sistema, que é realizado pelo ator Cliente. Neste caso de uso, o cliente realiza o pagamento de um pedido realizado em algum momento do passado. Considerando esta situação, você pode pensar em algum outro caso de uso do sistema?

4-24: Considere o modelo de casos de uso inicial para o Sistema de Controle Acadêmico (ver [Seção 4.7.3](#)). Modifique esse modelo para contemplar as seguintes novidades:

- a. O coordenador informa à equipe de desenvolvimento que há datas inicial e final preestabelecidas dentro de um semestre para que um professor possa lançar notas ou fornecer sua disponibilidade de carga horária para semestre letivo seguinte. É o próprio coordenador que deve estabelecer essas datas.
- b. Da mesma forma, há um período para realização de inscrições e outro para cancelamentos das mesmas. Fora desses períodos, o sistema não deve aceitar tais operações. O coordenador também deve ter a possibilidade de definir esses períodos.
- c. O coordenador declara que precisa ser informado pelo sistema (por e-mail, por exemplo) quando este último cria uma nova lista de espera para uma determinada disciplina.

1. Esta dica se justifica pela grande rapidez com a qual uma tecnologia é substituída por outra: poucas tecnologias de cem anos atrás são usadas hoje, assim como poucas tecnologias atuais serão utilizadas daqui a cem anos.

2. Uma heurística é uma espécie de dica embasada na experiência prática.

3. O leitor familiarizado com a Metodologia de Análise Essencial irá se recordar de um conceito semelhante, o dos denominados *fluxos e eventos temporais*.

4. Esses são conhecidos como *casos de uso* *CRUD* (create-read-update-delete).
5. O formato aqui fornecido é uma adaptação do proposto pelo Grupo Guild (GUILD, 2002).
6. Note que a interface propriamente dita não deve ser específica no documento de requisitos, mas sim restrições e demandas do cliente em relação a essa interface.
7. Pode ser que um único caso de uso, em virtude de sua complexidade, precise ser desenvolvido em mais de uma iteração. Nesse caso, as seções do caso de uso podem servir de unidade de alocação.

Modelagem de classes de análise

O engenheiro de software amador está sempre à procura da mágica, de algum método sensacional ou ferramenta cuja aplicação promete tornar trivial o desenvolvimento de software. É uma característica do engenheiro de software profissional saber que tal panaceia não existe.

– GRADY BOOCHE

O modelo de casos de uso de um sistema é construído para formar a *visão de casos de uso* do sistema (conforme mencionado na [Seção 1.4.1](#)). Esta visão fornece uma perspectiva do sistema a partir de um ponto de vista *externo*. De posse da visão de casos de uso, os desenvolvedores precisam prosseguir no desenvolvimento do sistema.

A funcionalidade externa de um sistema orientado a objetos é fornecida por meio de colaborações entre objetos. Externamente ao sistema, os atores visualizam resultados de cálculos, relatórios produzidos, confirmações de requisições realizadas etc. Internamente, os objetos do sistema colaboram uns com os outros para produzir os resultados visíveis de fora. Essa colaboração pode ser vista sob o *aspecto dinâmico* e sob o *aspecto estrutural estático*, conforme descrito a seguir.

O *aspecto dinâmico* de uma colaboração entre objetos descreve a troca de mensagens (ver [Seção 1.2.2](#)) entre os mesmos e a sua reação a eventos que ocorrem no sistema. O aspecto dinâmico de uma colaboração é representado pelo Modelo de Interações e é estudado nos [Capítulos 7 e 10](#).

O *aspecto estrutural estático* de uma colaboração permite compreender como o sistema está estruturado internamente para que as funcionalidades externamente visíveis sejam produzidas. Esse aspecto é denominado *estático*, porque não apresenta informações sobre como os objetos do sistema interagem no decorrer do tempo (isso é representado no aspecto dinâmico da colaboração). Também é chamado de *estrutural*, porque a estrutura das classes de objetos componentes do sistema e as relações entre elas são representadas.

Os aspectos estáticos e dinâmicos de um sistema orientado a objetos não são independentes. Na verdade, conforme descrito com mais detalhes neste capítulo, a construção de um serve para adicionar detalhes ao outro. Por exemplo, quando, durante a construção do aspecto dinâmico, o modelador detecta a necessidade de uma mensagem entre dois objetos, isso implica a existência de alguma referência estrutural entre os mesmos. Em outras palavras, um objeto precisa ter uma referência para o outro a fim de lhe enviar uma solicitação. Do mesmo modo, a definição da estrutura do relacionamento entre dois objetos feita no aspecto estrutural influencia na forma pela qual esses mesmos objetos podem trocar mensagens.

Este capítulo inicia a descrição do aspecto estrutural estático de um sistema orientado a objetos. Esse aspecto é representado pelo Modelo de Classes, da mesma forma que o aspecto funcional é representado pelo Modelo de Casos de Uso. A ferramenta da UML utilizada para representar o aspecto estrutural estático é o *diagrama de classes*. O *modelo de classes* é composto desse diagrama e da descrição textual associada ao mesmo. Para melhor orientar o leitor, os objetivos principais deste capítulo são enumerados a seguir.

1. Descrever algumas técnicas para identificação de classes de análise.
2. Apresentar alguns dos elementos de notação do diagrama de classes necessários à construção do modelo de classes de análise (outros elementos são descritos em capítulos posteriores). É também uma meta deste capítulo apresentar o *diagrama de objetos*, menos usado na prática, mas também relacionado ao aspecto estrutural estático.
3. Descrever como o modelo de classes pode ser documentado.
4. Descrever a inserção do modelo de classes de análise em um processo de desenvolvimento iterativo e incremental.

Este capítulo está organizado da seguinte forma: na [Seção 5.1](#), descrevemos os estágios pelos quais passa o modelo de classes de um SSOO durante o seu desenvolvimento. Na [Seção 5.2](#) e na [Seção 5.3](#), apresentamos os elementos de notação necessários para a modelagem de classes na etapa de análise. Na [Seção 5.4](#), descrevemos diversas técnicas usadas para identificação de classes. Na [Seção 5.5](#), apresentamos detalhes acerca da construção do modelo de classes, uma vez que foram identificadas as classes iniciais. A [Seção 5.6](#) apresenta a modelagem de classes de análise no contexto de um processo de desenvolvimento, assim com sua relação com as demais atividades de modelagem. Finalmente, a [Seção 5.7](#) apresenta a continuação da modelagem do SCA no contexto da modelagem de classes de análise.

5.1 Estágios do modelo de classes

O modelo de classes é utilizado durante a maior parte do desenvolvimento iterativo e incremental de um SSOO. Mais que isso, esse modelo evolui durante as iterações do desenvolvimento do sistema. À medida que o sistema é desenvolvido, o modelo de classes é incrementado com novos detalhes. Durante essas evolução, há três *estágios sucessivos de abstração* pelos quais o modelo de classes passa: *análise, especificação e implementação*.

1. O *modelo de classes de análise* representa as classes de análise, ou seja, aquelas que se tornam evidentes na medida em que focamos a atenção sobre “o que” o sistema deve fazer. Este modelo é construído na fase de análise (ver [Seção 2.1.2](#)). Por definição, um modelo de classes de análise não leva em consideração restrições inerentes à tecnologia a ser utilizada na solução de um problema. Em seu conjunto, o modelo de casos de uso e o modelo de classes de análise são os dois principais modelos criados na fase de análise (ver [Seção 2.1.2](#)). Um modelo de classes que se encontra nesse estágio representa conceitos do domínio do problema. A participação do especialista do domínio (ver [Seção 2.2.6](#)) é fundamental para a correta construção do modelo de classes de análise.
2. O *modelo de classes de especificação* é um detalhamento do modelo de classes de análise. É também conhecido como *modelo de classes de projeto*. Quando chegamos nesse estágio, normalmente descobrimos a necessidade de criar outras classes, pois começamos a focar nossa atenção sobre “como” o sistema deve funcionar. Além disso, esse detalhamento do modelo de classes também envolve a adição de detalhes às classes identificadas na análise, em função da solução de software escolhida. O modelo de classes de projeto é construído na atividade de projeto (ver [Seção 2.1.3](#)) do desenvolvimento de uma iteração do desenvolvimento. Para entender a diferença entre o modelo de classes de análise e o de

projeto, vale aqui uma analogia com a construção de uma casa. No modelo de análise de uma casa, pensamos em objetos como salas, quartos, banheiros, portas etc. No modelo de especificação desta mesma casa, temos que pensar em outros detalhes, como encanamento, parte elétrica, caixonetes para as portas, ou seja, detalhes que não são evidentes para os ocupantes de uma casa, mas que são necessários para construí-la propriamente. Se voltarmos ao contexto de sistemas de software, um exemplo desse detalhamento é definir de que forma o sistema deve se comunicar com agentes externos a ele, usando possibilidades como servidores de e-mail, servidores de bancos de dados, etc.

3. O *modelo de classes de implementação* é um detalhamento do modelo de especificação. Esse modelo corresponde à *implementação* das classes em alguma linguagem de programação, normalmente orientada a objetos (C++, Java, C#, VB.NET, etc.). O modelo de implementação é construído na atividade de implementação (ver [Seção 2.1.4](#)) de um processo de desenvolvimento iterativo. (Note que aqui estamos considerando o próprio código-fonte do sistema como um modelo.)

Neste capítulo, nosso principal objetivo é estudar o modelo de classes em seu primeiro estágio. O modelo de classes de análise é composto dos objetos identificados na análise do domínio e na análise da aplicação (ver [Seção 2.1.2](#)). Seu objetivo é descrever o problema representado pelo sistema a ser desenvolvido; ele não considera as características da solução a ser utilizada. Já os modelos de especificação e de implementação consideram detalhes da solução de software a ser utilizada. No entanto, ao contrário do modelo de implementação, o de especificação descreve a solução em um nível alto de abstração.

Antes de passarmos para as próximas seções, apresentamos a seguir a nomenclatura de identificadores dos elementos do modelo de classes utilizada neste livro. Entretanto, é importante notar que essa nomenclatura é apenas uma possibilidade. A equipe de desenvolvimento pode escolher qualquer estilo de nomenclatura que desejar. O importante é que, uma vez escolhida uma nomenclatura, ela seja utilizada consistentemente.

1. Para identificadores, quaisquer espaços em branco e preposições do nome são removidos.
2. Para nomes de classes e nomes de relacionamentos, as palavras componentes do nome são escritas começando por letra maiúscula. Exemplos: Cliente, ItemPedido, Pedido, OrdemServiço, Reside, Realiza etc.
3. Para nomes de atributos e nomes de operações: deve-se escrever a primeira palavra do nome de atributo em minúsculas. Escreva as palavras subsequentes em maiúsculas. No entanto, siglas são mantidas inalteradas. Exemplos: quantidade, precoUnitário, CPF, nome, dataNascimento, obterTotal etc.

5.2 Diagrama de classes

O *diagrama de classes* é utilizado na construção do modelo de classes desde o nível de análise até o nível de especificação. De todos os diagramas da UML, esse é o mais rico em termos de notação. Nesta seção, são apresentados os elementos do diagrama de classes utilizados para a construção do modelo de classes de nível de análise.

5.2.1 Classes

Uma classe é representada por uma “caixa” com, no máximo, três compartimentos exibidos. No primeiro compartimento (de cima para baixo) é exibido o seu nome. Por convenção, esse nome é apresentado no singular e com as palavras componentes começando por maiúsculas. No segundo compartimento, são declarados os atributos que correspondem às informações que um objeto armazena. Finalmente, no terceiro compartimento, são declaradas as operações, que correspondem às ações que um objeto sabe realizar.

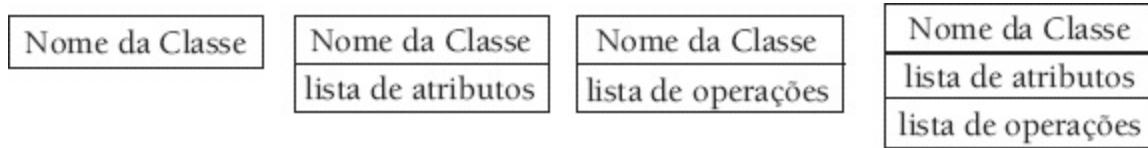


Figura 5-1: Possíveis notações para uma classe na UML.

As possíveis notações da UML para representar classes são apresentadas na [Figura 5-1](#). O grau de abstração desejado em certo momento do desenvolvimento do modelo de classes direciona a utilização de uma ou outra notação.

Estruturalmente, uma classe é composta de *atributos* e de *operações*. Os atributos correspondem à descrição dos dados armazenados pelos objetos de uma classe. A cada atributo de uma classe está associado um conjunto de valores que esse atributo pode assumir. As operações correspondem à descrição das ações que os objetos de uma classe sabem realizar. Ao contrário dos atributos (para os quais cada objeto tem o seu próprio valor), objetos de uma classe compartilham as mesmas operações. O nome de uma operação normalmente contém um verbo e um complemento, terminando com um par de parênteses. (Na descrição do modelo de classes de especificação, é apresentada a verdadeira utilidade desse par de parênteses.)

A [Figura 5-2](#) ilustra exemplos de representação de uma mesma classe, ContaBancária, em diferentes graus de abstração.

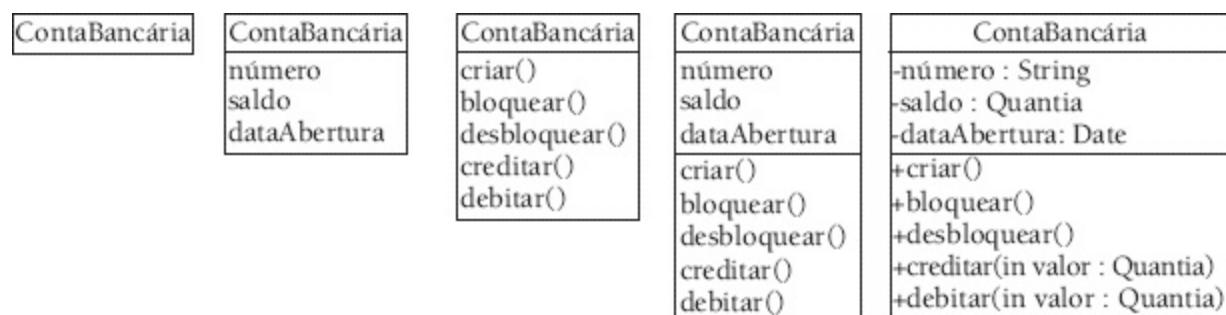


Figura 5-2: Diferentes graus de abstração na notação de classe.

5.2.2 Associações

Da [Seção 1.2.1](#), sabe-se que cada ocorrência de uma classe é chamada de *objeto* ou *instância*. Um ponto importante a respeito de objetos de um sistema é o fato de que eles podem se relacionar uns com os outros. A existência de um relacionamento entre dois objetos possibilita a troca de mensagens (ver [Seção 7.5.1](#)) entre os mesmos. Portanto, em última análise, relacionamentos entre objetos permitem que eles colaborem entre si para produzir as funcionalidades do sistema.

No diagrama de classes, podemos representar a existência de relacionamentos entre objetos. Para representar o fato de que objetos podem se relacionar uns com os outros, existe mais um elemento na

notação do diagrama de classes, a *associação*. Esse elemento representa relacionamentos que são formados entre objetos durante a execução do sistema. Uma associação é representada no diagrama de classes por uma linha (normalmente um segmento de reta) ligando as classes às quais pertencem os objetos relacionados. Na [Figura 5-3](#), apresentamos alguns exemplos de associações entre objetos: (1) no domínio de vendas, um *cliente* compra *produtos*; (2) no domínio bancário, uma *conta-corrente* possui um *histórico de transações*; (3) em um hotel, há vários *hóspedes*, assim como há vários *quartos*. Os *hóspedes* do hotel ocupam *quartos*.

É importante notar que, embora as associações sejam representadas entre classes do diagrama, elas representam na verdade ligações possíveis entre *objetos* das classes envolvidas. Por exemplo, quando ligamos no desenho acima as classes *Hóspede* e *Quarto*, isso significa que, durante a execução do sistema, haverá a possibilidade de troca de mensagens entre objetos dessas classes. Ou seja, objetos dessas classes poderão colaborar para realizar alguma (parte de uma) tarefa do sistema.

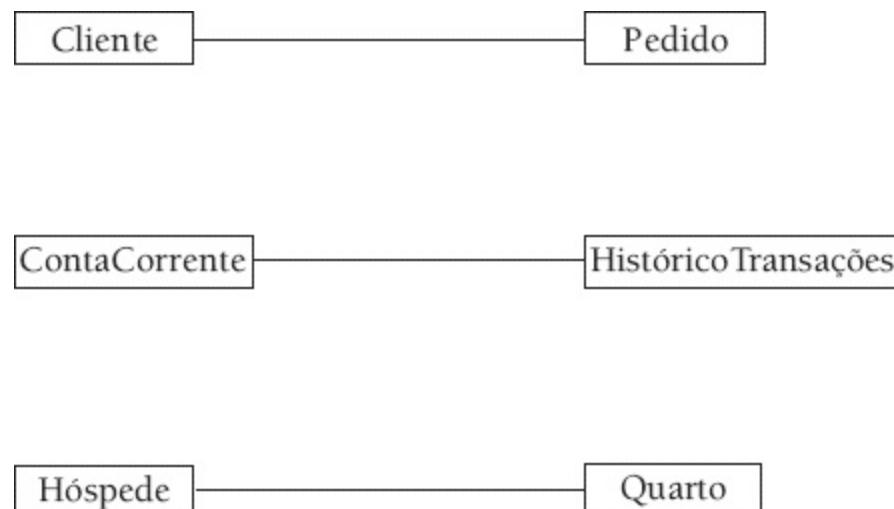


Figura 5-3: Exemplos de associações entre objetos.

Associações possuem diversas características importantes: multiplicidades, nome, direção de leitura, papéis, tipo de participação e conectividade. Nas próximas seções, descrevemos essas características.

5.2.2.1 Multiplicidades

As associações permitem a representação da informação dos limites inferior e superior da *quantidade de objetos* aos quais outro objeto pode estar associado. Esses limites são chamados de *multiplicidades* na terminologia da UML.¹ Cada associação em um diagrama de classes possui duas multiplicidades, uma em cada extremo da linha que a representa. Os símbolos possíveis para representar uma multiplicidade estão descritos na [Tabela 5-1](#). Note que, em dois dos casos apresentados nessa Tabela, a UML define mais de um símbolo para representar a mesma multiplicidade. Primeiramente, o símbolo “1” é equivalente à utilização do símbolo “1..1”. Outra equivalência ocorre entre os símbolos “*” e “0..*”.

Tabela 5-1: Simbologia para representar multiplicidades

Nome	Simbologia
Apenas Um	1

Zero ou Muitos	0..*
Um ou Muitos	1..*
Zero ou Um	0..1
Intervalo Específico	$l_i..l_s$

Como exemplo, observe a [Figura 5-4](#), que exibe duas classes, Pedido e Cliente, e uma associação entre as mesmas. A leitura dessa associação nos informa que pode haver um objeto da classe Cliente que esteja associado a vários objetos da classe Pedido (isso é representado pelo * no símbolo 0..*). Além disso, essa mesma leitura nos informa que pode haver um objeto da classe Cliente que não esteja associado a pedido algum (isso é representado pelo 0 no símbolo 0..0). O diagrama da [Figura 5-4](#) também representa a informação de que um objeto da classe Pedido está associado a um, e somente um, objeto da classe Cliente.



Figura 5-4: Exemplo de utilização dos símbolos de multiplicidade.

O * denota, em todos os símbolos nos quais aparece, que não há um limite superior predefinido para a quantidade máxima de objetos com os quais outro objeto pode se associar. Por exemplo, na [Figura 5-4](#), não há um limite superior, pelo menos na teoria, para a quantidade de pedidos que um cliente pode realizar. Logicamente, a tempo de execução do sistema que está sendo modelado, essa quantidade máxima será sempre limitada pela quantidade de memória do dispositivo computacional no qual a aplicação está executando. No entanto, quando utilizamos o símbolo * em um diagrama de classes, queremos dizer que a quantidade máxima de associações é um valor finito maior ou igual a zero, e que não nos importa qual é esse valor.

A [Tabela 5-1](#) também exibe a forma geral do símbolo de multiplicidade para intervalos específicos. As expressões l_i e l_s devem ser substituídas por valores correspondentes aos limites inferior e superior, respectivamente, do intervalo que se quer representar. Por exemplo, vamos representar informações sobre velocistas e corridas nas quais eles participem. Suponha ainda que em uma corrida deve haver, no máximo, seis velocistas participantes. O fragmento de diagrama de classes que representa esta situação é exibido na [Figura 5-5](#). O valor para l_i é dois (uma corrida está associada a, no mínimo, dois velocistas), e o valor para l_s é seis (uma corrida está associada a, no máximo, seis velocistas).

Uma lista de intervalos também pode ser especificada na multiplicidade de uma associação. Por exemplo, a multiplicidade “1, 3, 5..9, 11” é perfeitamente válida e significa que um objeto pode se associar a uma quantidade de objetos que esteja no conjunto {1, 3, 5, 6, 7, 8, 9, 11}. Note também que, por convenção, os valores especificados em uma multiplicidade estão sempre em ordem crescente quando lidos da esquerda para a direita.

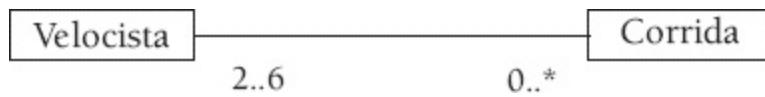


Figura 5-5: Exemplo de utilização de intervalo de valores para multiplicidade.

Existem infinitas possibilidades de associação entre objetos (todas as combinações possíveis entre os números inteiros). No entanto, essas associações podem ser agrupadas em apenas três tipos: “*muitos para muitos*”, “*um para muitos*” e “*um para um*”. Denomina-se *conectividade* o tipo de associação entre duas classes. A conectividade da associação entre duas classes depende dos símbolos de multiplicidade que são utilizados na associação. A Tabela 5-2 exibe as correspondências entre os tipos de conectividade e os símbolos de multiplicidade.

Tabela 5-2: Conectividades versus multiplicidades

Conectividade	Multiplicidade de um extremo	Multiplicidade do outro extremo
Um para um	0.. 1 ou 1	0..1 ou 1
Um para muitos	0..1 ou 1	* ou 1..* ou 0..*
Muitos para muitos	* ou 1..* ou 0..*	* ou 1..* ou 0..*

A Figura 5-6 apresenta três exemplos, um para cada conectividade possível. No primeiro (um para um), indica-se que um empregado pode gerenciar um e somente um departamento. Um departamento, por sua vez, possui um único gerente. No segundo exemplo (um para muitos), um empregado está lotado em um único departamento, mas um departamento pode ter diversos empregados. Finalmente, no terceiro exemplo (muitos para muitos), um projeto pode ter diversos empregados. Além disso, um empregado pode trabalhar em diversos departamentos.

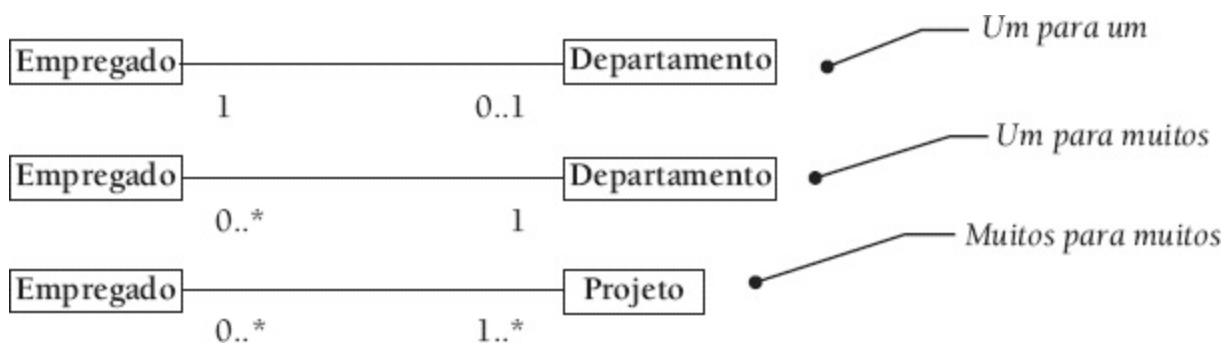


Figura 5-6: Exemplos de cada tipo de conectividade.

5.2.2.2 Participações

Uma característica importante de uma associação está relacionada à necessidade ou não da existência dessa associação entre objetos. Essa característica é denominada *participação*. A participação pode ser *obrigatória* ou *opcional*. Se o valor mínimo da multiplicidade de uma associação é igual a 1 (um), significa que a participação é obrigatória. Caso contrário, a participação é opcional.

Por exemplo, considere a associação entre Empregado e Departamento na parte superior da Figura 5-6. A multiplicidade de valor 1 próxima a Empregado indica que um objeto da classe Departamento só pode existir se estiver associado a um objeto Empregado. Ou seja, para objetos da classe Departamento, a participação é *obrigatória*. No entanto, para objetos de Empregado, essa mesma associação é *parcial* (pode haver empregados que não estejam associados a um departamento). Isso é indicado pelo símbolo 0..1 no extremo próximo à classe Departamento.

É preciso ter cuidado com a utilização de participações obrigatórias, pois estas “engessam” o modelo resultante, o que dificulta futuras extensões do mesmo. Se há dúvidas sobre as

multiplicidades de uma associação, pode-se adiar essa decisão para quando mais informações do sistema forem conhecidas. Em particular, as informações obtidas com a *modelagem dinâmica* do sistema servem para definir ou validar as multiplicidades das associações.

5.2.2.3 Nome de associação, direção de leitura e papéis

Para melhor esclarecer o significado de uma associação no diagrama de classes, a UML define três recursos de notação: *nome de associação*, *direção de leitura* e *papel*. O nome da associação é posicionado na linha da associação, a meio caminho das classes envolvidas.

A direção de leitura indica como a associação deve ser lida. Essa direção é representada por um pequeno triângulo posicionado próximo a um dos lados do nome da associação. O nome de uma associação deve fornecer algum significado semântico à mesma.

Quando um objeto participa de uma associação, ele tem um papel específico nela. Uma característica complementar à utilização de nomes e de direções de leitura é a indicação de *papéis* (*roles*) para cada uma das classes participantes em uma associação.

A [Figura 5-7](#) apresenta uma associação na qual são representados os papéis (contratante e contratado) e o seu nome (Contrata). É também indicada a direção de leitura, que indica que uma organização contrata indivíduos, e não o contrário.

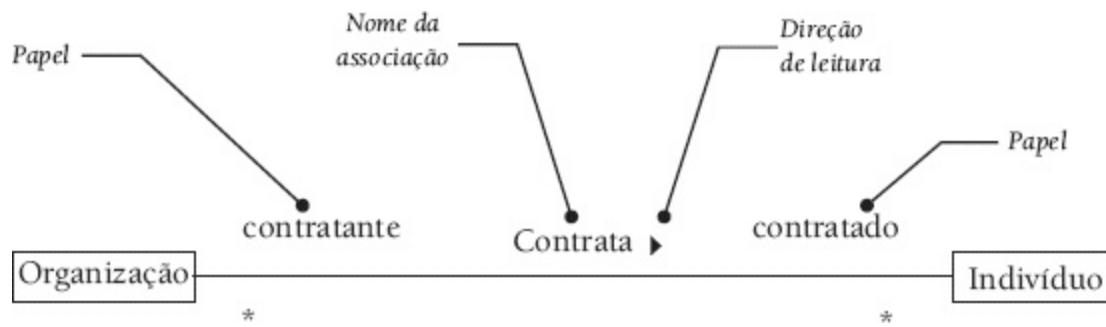


Figura 5-7: Exemplo de utilização de nome de associação, direção de leitura e de papéis.

Sempre que o significado de uma associação não for fácil de inferir, é recomendável representar papéis ou pelo menos o nome da associação. Isso evita que haja interpretações erradas no significado da associação. Além disso, essa prática aumenta a legibilidade do diagrama. O nome da associação deve ser simples e exprimir o significado da mesma. É preferível não nomear associações com usar nomes vagos ou óbvios demais. O mesmo vale para os papéis: em situações em que o significado da associação for intuitivo, a utilização de papéis só serve para “carregar” o diagrama. O ponto é tentar equilibrar clareza e concisão.

Embora não seja usual, pode haver diversos motivos pelos quais um objeto precisa saber a respeito do outro. Consequentemente, pode haver diversas associações definidas entre duas classes no diagrama de classes. Por exemplo, considere duas classes: Empregado e Departamento. Considere, ainda, que um departamento precisa saber quais são os seus empregados e quem é o seu gerente. Nesse caso, há duas associações diferentes, embora as classes envolvidas sejam as mesmas (ver [Figura 5-8](#)).

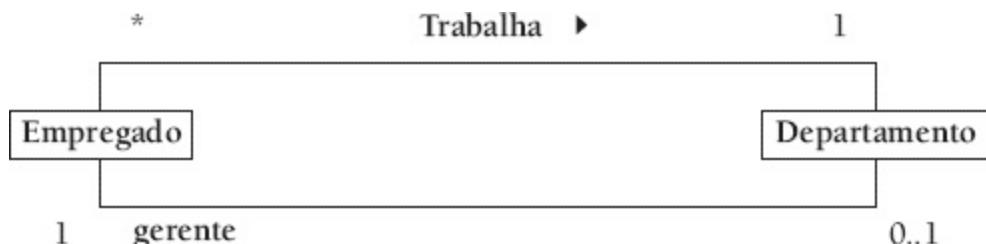


Figura 5-8: Associações diferentes entre duas classes.

5.2.2.4 Classes associativas

Classes associativas são classes que estão ligadas a associações, em vez de estarem ligadas a outras classes. São também chamadas de *classes de associação*. Esse tipo de classe normalmente aparece quando duas ou mais classes estão associadas e é necessário manter informações dessa associação. Embora seja mais comum encontrar classes associativas ligadas a associações de conectividade *muitos para muitos*, uma classe associativa pode estar ligada a associações de qualquer conectividade.

Na UML, uma classe associativa é representada pela mesma notação utilizada para uma classe comum. A diferença é que esta classe é ligada por uma linha tracejada a uma associação. Para ilustrar a utilização de uma classe associativa em um diagrama de classes, apresentamos a [Figura 5-9](#). Esse diagrama informa que uma pessoa é empregada em várias empresas. Uma empresa, por sua vez, tem vários empregados. A classe associativa Emprego permite saber, para cada par de objetos [empregado, empregador], qual o salário e a data de contratação do empregado em relação àquele empregador. De forma geral, se precisarmos representar a existência de uma informação que somente faz sentido quando considerarmos todos os objetos participantes da associação, podemos utilizar o conceito de classe associativa.

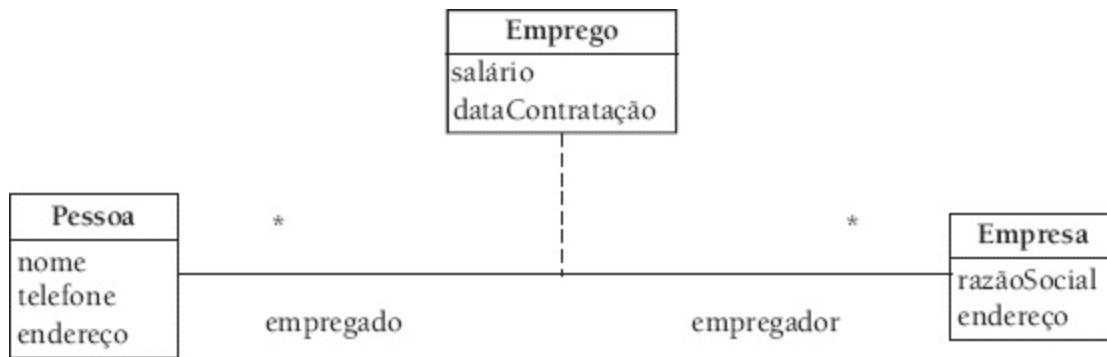


Figura 5-9: Exemplo de classe associativa.

Como dica de modelagem, não se deve nomear a linha da associação de uma classe associativa. Nesse caso, somente o nome escolhido para a classe associativa deve ser suficiente para expressar o significado desejado. Alternativamente, papéis (ver [Seção 5.2.2.3](#)) também podem ser utilizados para dar significado à associação.

Note que uma classe associativa pode participar de outros relacionamentos. Por exemplo, observe o diagrama da [Figura 5-10](#), no qual podemos verificar que há funcionários com várias especialidades. Os funcionários realizam consertos em automóveis, mas é necessário saber que especialidade foi utilizada pelo funcionário em certo trabalho. Para isso, uma classe associativa é criada entre as classes Funcionário e Automóvel. Além disso, essa mesma classe associativa está associada à classe especialidade para permitir conhecer qual a especialidade utilizada em um

conserto.

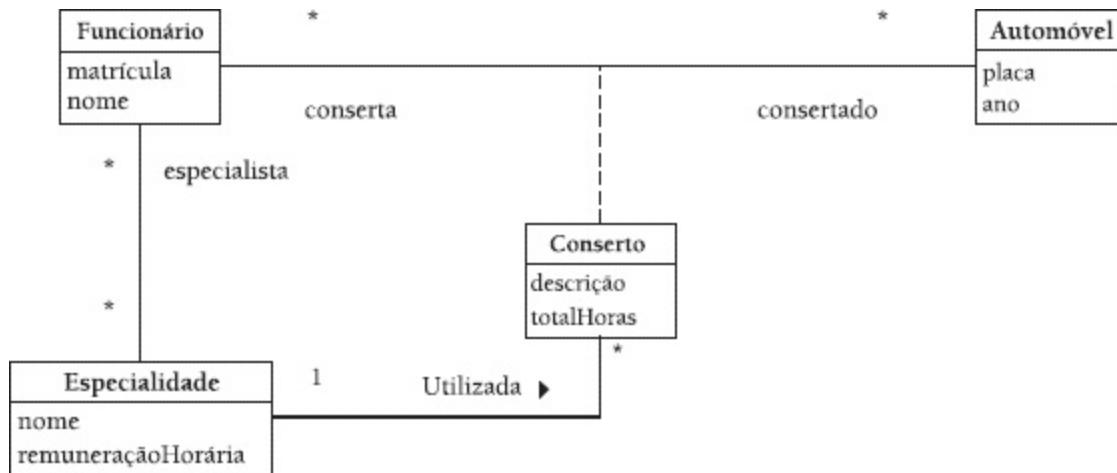


Figura 5-10: Exemplo de classe associativa.

Pelos exemplos anteriores, pode-se notar que uma classe associativa é um elemento híbrido: tem características de uma classe, mas também de uma associação.

Por fim, de forma geral, um diagrama de classes que contém uma classe associativa pode ser modificado para retirá-la, sem perda de informação no modelo em questão. Isso pode ser feito em dois passos: (1) eliminação da associação correspondente à classe associativa; e (2) criação de associações diretas desta última com as classes que antes eram conectadas pela associação eliminada no passo 1. Como exemplo de aplicação desses passos, a [Figura 5-11](#) apresenta um diagrama equivalente ao da [Figura 5-9](#), em que a classe associativa foi substituída por uma ordinária. Note que a classe Emprego tem participação obrigatória em ambas as associações.

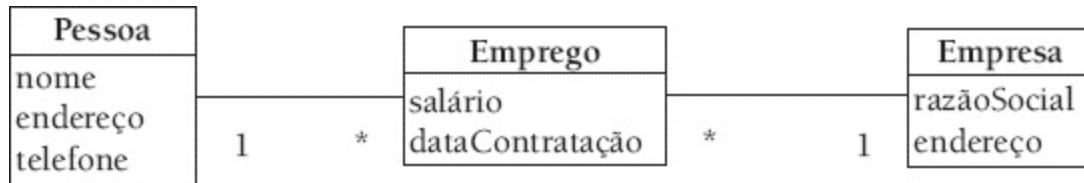


Figura 5-11: Uma classe associativa pode ser substituída por uma classe ordinária.

5.2.2.5 Associações ternárias

Define-se o *grau* de uma associação como a quantidade de classes envolvidas na mesma. Na maioria dos casos práticos de modelagem, as associações normalmente são binárias, ou seja, representam a ligação entre objetos de duas classes (tem grau igual a dois). Entretanto, de forma geral, uma associação pode ter grau maior que dois. Quando o grau de uma associação é igual a três, dizemos que ela é ternária. Associações ternárias são necessárias quando é preciso associar três objetos distintos. A [Figura 5-12](#) ilustra a notação da UML para associações ternárias.

Dada uma associação ternária (ou de mais alto grau) entre as classes A, B e C, a multiplicidade do extremo C indica quantos objetos da classe C podem estar associados a um par particular de objetos (b, c), em que b é um objeto da classe B , e c é uma instância da classe C .

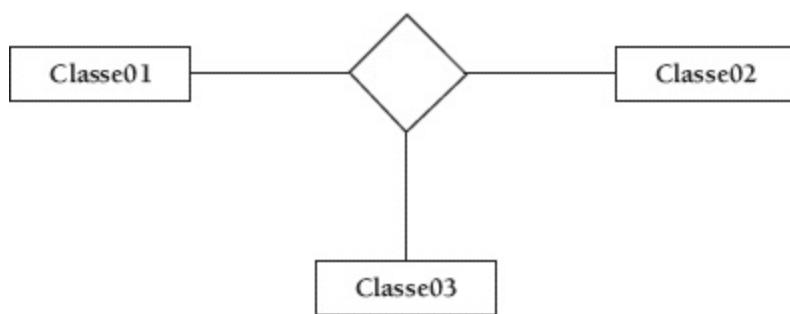


Figura 5-12: Notação para representar associações ternárias no diagrama de classes.

Um exemplo de associação ternária é apresentado na [Figura 5-13](#). A figura ilustra o fato de que um técnico utiliza exatamente um computador para cada projeto em que trabalha. Cada computador pertence a um técnico para cada projeto. Note que um técnico pode trabalhar em muitos projetos e utilizar diferentes computadores para diferentes projetos.

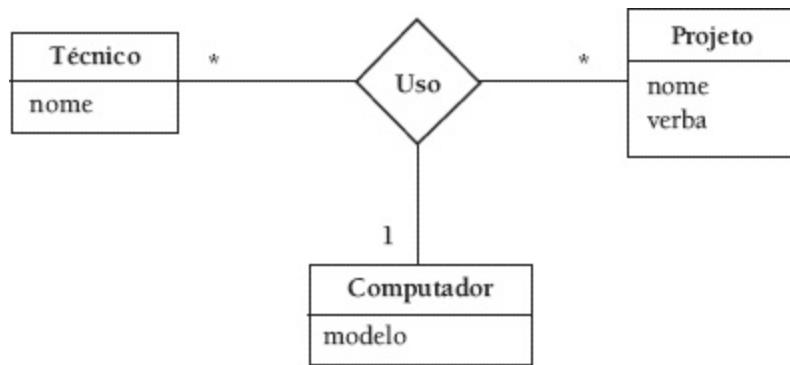


Figura 5-13: Exemplo de associação ternária.

Outro exemplo de associação ternária é apresentado na [Figura 5-14](#). Desta vez, vê-se que cada empregado associado a um projeto trabalha exclusivamente em uma localização, mas pode estar em diferentes localizações em projetos diferentes. Em uma localização em particular, pode haver muitos empregados associados a um dado projeto.

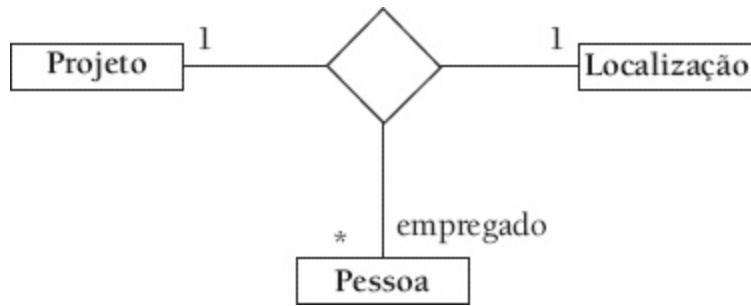


Figura 5-14: Exemplo de associação ternária.

Finalmente, os conceitos de classe associativa e de associação ternária podem ser misturados no conceito de *classe associativa ternária*, no qual existe uma classe associativa ligada a uma associação ternária. A [Figura 5-15](#) ilustra um exemplo dessa situação. Há três classes ligadas por uma associação ternária (Estudante, Instrutor e Curso). Além disso, existe a classe associativa SeçãoCurso, que permite armazenar informações para cada instância da associação.

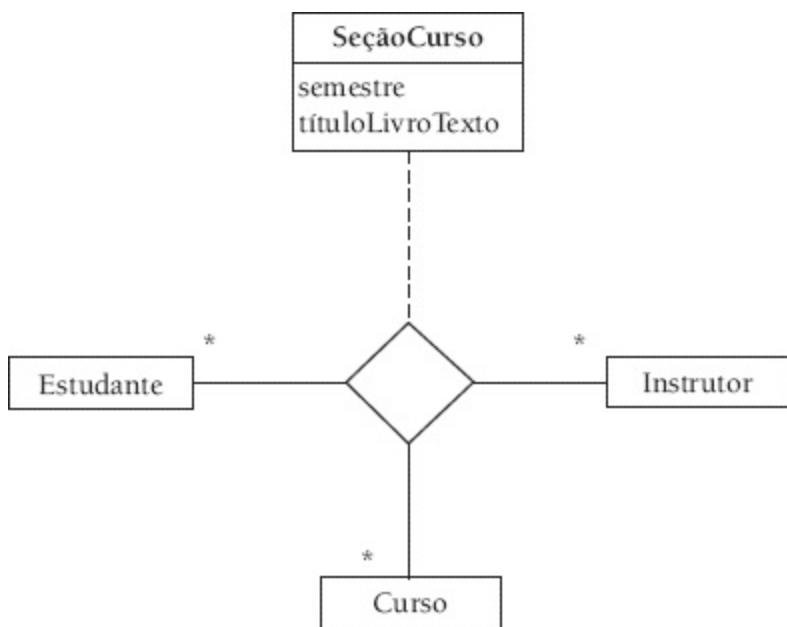


Figura 5-15: Exemplo de classe associativa ternária.

5.2.2.6 Associações reflexivas

Uma associação reflexiva (também denominada *autoassociação*) liga objetos da mesma classe. Cada objeto tem um papel distinto nessa associação. Por exemplo, considere a [Figura 5-16](#), que exibe uma associação reflexiva entre objetos de Empregado. Nessa figura, apresentamos um exemplo de uso de autoassociação, em que há objetos que assumem o papel de supervisor e outros que tomam para si o papel de supervisionado. Nesse exemplo, o nome da associação poderia ter sido omitido, uma vez que os papéis foram definidos (ver [Seção 5.2.2.3](#)). Além disso, conforme mostra a [Figura 5-16](#), em associações reflexivas a utilização de papéis é fundamental para evitar qualquer confusão na leitura da associação.

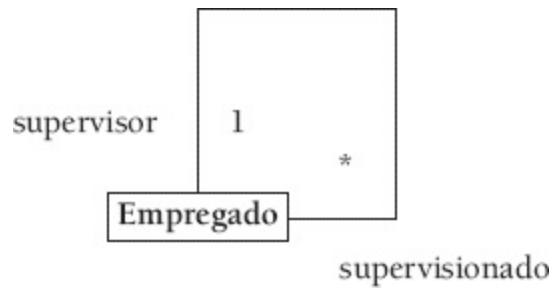


Figura 5-16: Exemplo de associação reflexiva.

Não se deve confundir o significado de uma associação reflexiva. Ela *não* indica que um objeto se associa a ele próprio (um empregado não é supervisor de si mesmo; uma disciplina não é seu próprio pré-requisito etc.). Em vez disso, uma autoassociação indica que um objeto de uma classe se associa com outros objetos da mesma classe.

5.2.2.7 Agregações e composições

À toda associação, podemos atrelar uma *semântica*. A semântica de uma associação corresponde ao seu significado, ou seja, à natureza conceitual da relação que existe entre os objetos que participam daquela associação. Quando damos um nome a uma associação (ou alternativamente definimos os papéis dos objetos associados pela mesma), fazemos isso com o objetivo de esclarecer a sua

semântica.

De todos os significados diferentes que uma associação pode ter, há uma categoria especial de significados que representa relações *todo-parte*. Esse tipo de relação entre dois objetos indica que um deles *está contido* no outro. Podemos também dizer que um objeto *contém* o outro.

A UML define dois tipos de *relacionamentos todo-parte*, a *agregação* e a *composição*. A agregação e a composição são casos especiais da associação. Consequentemente, todas as características válidas para esta última (multiplicidades, participações, papéis etc.) valem para as primeiras. Além disso, sempre que for possível utilizar uma agregação ou composição, uma associação também poderá ser utilizada. De qualquer modo, a seguir são relacionadas algumas características particulares das agregações e composições que as diferem das associações simples.

- Agregações/composições são *assimétricas*, no sentido de que, se um objeto A é parte de um objeto B, o objeto B não pode ser parte do objeto A.
- Agregações/composições propagam comportamento, de forma que um comportamento que se aplica a um todo automaticamente se aplica também às suas partes.
- Nas agregações/composições, as partes são normalmente criadas e destruídas pelo todo. Na classe do objeto todo, são definidas operações para adicionar e remover as partes.

Em uma situação prática, podemos aplicar a seguinte regra para verificar se faz sentido utilizarmos um relacionamento todo-parte (agregação ou composição): sejam duas classes associadas, X e Y. Se uma das perguntas a seguir for respondida com um sim, provavelmente há um relacionamento todo-parte envolvendo X e Y, no qual X é o todo e Y é a parte. (Do contrário, é melhor utilizarmos a associação simples.)

- 1) X tem um ou mais Y?
- 2) Y é parte de X?

Graficamente, a UML fornece notações diferentes para a agregação e a composição. Uma agregação é representada como uma linha que conecta as classes relacionadas, com um diamante (losango) branco perto da classe que representa o todo. Já uma composição é representada na UML por meio de um diamante negro, para contrapor com o diamante branco da agregação.

Como exemplo de agregação, analise a [Figura 5-17](#), que apresenta um fragmento de diagrama de classes. Esse diagrama indica que uma associação esportiva é formada por diversas equipes. Cada uma delas é formada por diversos jogadores. Por outro lado, um jogador pode fazer parte de diversas equipes.

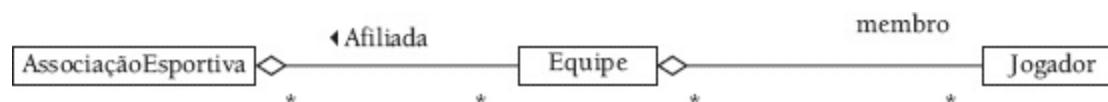


Figura 5-17: Exemplo de agregação.

Como exemplo de composição, considere os itens de um pedido de compra. É comum esse tipo de pedido incluir vários itens. Cada item diz respeito a um produto faturado. Os itens têm identidade própria (é possível distinguir um item de outro no mesmo pedido). Essa situação é representada no diagrama da [Figura 5-18](#). Outro exemplo de composição é apresentado na [Figura 5-19](#). Esse

diagrama informa que um automóvel possui várias partes: um motor e quatro rodas.

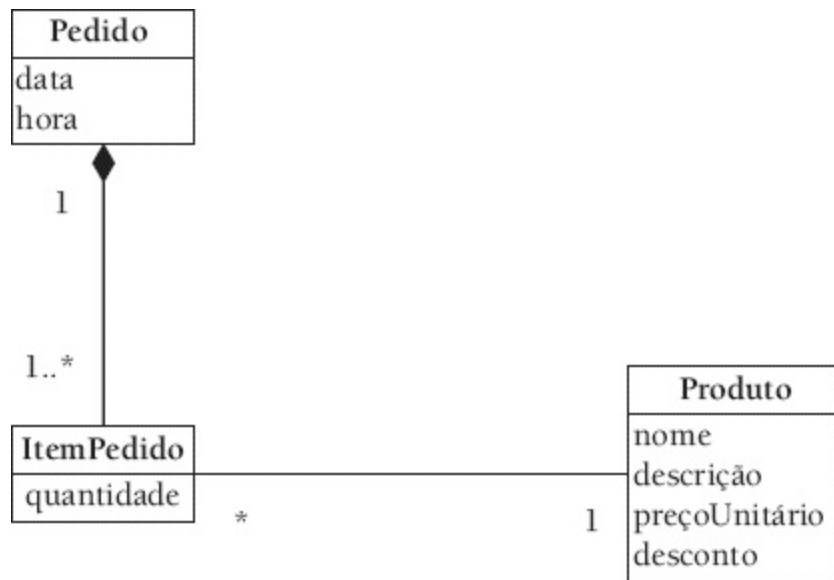


Figura 5-18: Composição entre Pedido e ItemPedido.

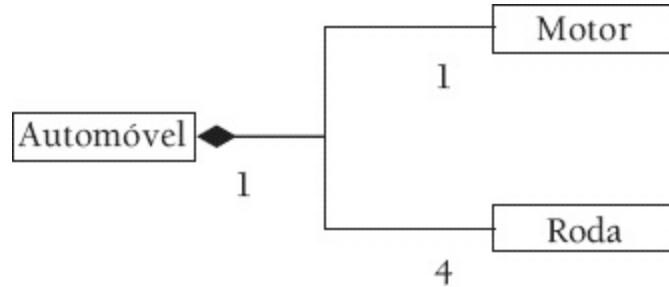


Figura 5-19: Exemplo de utilização de composição. Todo: Automóvel. Partes: Motor e Roda.

Uma agregação pode se estender por diversos níveis. Isso significa que é possível haver um objeto A composto de objetos B, ou que este último seja composto por objetos C, e assim por diante. Isso gera uma *hierarquia de agregações*. O exemplo da [Figura 5-17](#), em que há dois níveis de agregação, permite notar essa característica. Assim como as agregações, composições também podem se estender por diversos níveis. Dessa forma, pode-se falar em uma *hierarquia de composições*. Um exemplo de hierarquia de composição é a [Figura 5-20](#), que mostra que capítulos são compostos de seções, e estas, por sua vez, são compostas de parágrafos.

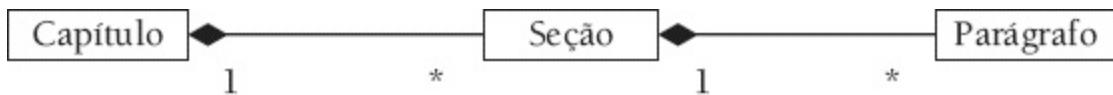


Figura 5-20: Exemplo de hierarquia de composição.

Se um modelador constata que uma associação tem uma semântica todo-parte, ele deve decidir que relacionamento utilizar, se uma agregação ou uma composição. Infelizmente, tanto na teoria quanto na prática, as diferenças entre a agregação e a composição não são bem definidas. Por conta disso, em contextos práticos, se o modelador estiver em dúvida acerca de que tipo de relacionamento todo-parte usar, isso é um indício de que não deve usar nenhum dos dois. Em vez disso, o modelador pode usar uma associação, já que ela é o caso geral da composição e da agregação. Mesmo assim, para guiar o modelador em sua escolha, descrevemos a seguir as duas diferenças mais marcantes entre os

tipos de relacionamentos todo-parte.

1. Na agregação, a destruição de um objeto todo não implica necessariamente a destruição do objeto parte. Por exemplo, considere a [Figura 5-17](#). Se uma das equipes das quais um jogador é membro for extinta por algum motivo, ele ainda poderá continuar membro de outras equipes. Por outro lado, considere o exemplo da [Figura 5-18](#). Nessa situação, os itens não têm existência independente do pedido ao qual estão conectados. Quando o pedido deixa de existir, o mesmo acontece com os seus itens.
2. Na composição, os objetos *parte* pertencem a um único *todo*. Por essa razão, a composição é também denominada *agregação não compartilhada*. Por outro lado, é possível que, em uma agregação, um mesmo objeto participe como componente de vários outros objetos. Por essa razão, a agregação é também denominada *agregação compartilhada*.

5.2.2.8 Restrições sobre associações

Restrições podem ser adicionadas sobre uma associação para adicionar mais semântica a ela. Duas das restrições sobre associações predefinidas pela UML são subset (subconjunto) e xor (ou exclusivo). Ambas são representadas por uma linha pontilhada que liga duas ou mais associações. Embora a descrição a seguir faça referência a associações, essas restrições também podem ser utilizadas em agregações e composições.

A restrição subset indica que os objetos conectados por uma associação constituem um subconjunto dos objetos conectados através de uma outra associação. A [Figura 5-21](#) ilustra o uso da restrição subset. Neste exemplo há duas classes ligadas pela restrição subset, indicando que os objetos associados por Administra formam um subconjunto dos objetos associados por Reside. (Portanto, a flecha parte da associação correspondente ao subconjunto.)

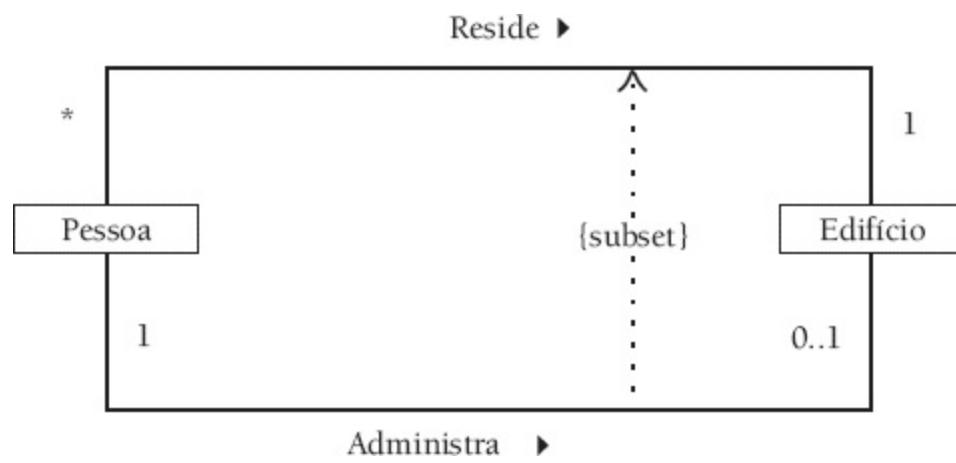


Figura 5-21: Exemplo de utilização da restrição subset.

Na restrição xor, há duas ou mais classes ligadas pela linha pontilhada. Essas classes devem ter associações com uma classe em comum. Essa restrição significa que somente uma das associações envolvidas pode ocorrer entre os objetos. Como exemplo da restrição xor, considere a [Figura 5-22](#) (por simplificação, atributos e operações são omitidos). Esse exemplo indica que um objeto ContaBancária pode estar associado a objetos Pessoa ou a objetos Instituição, mas nunca a objetos dos dois tipos simultaneamente.

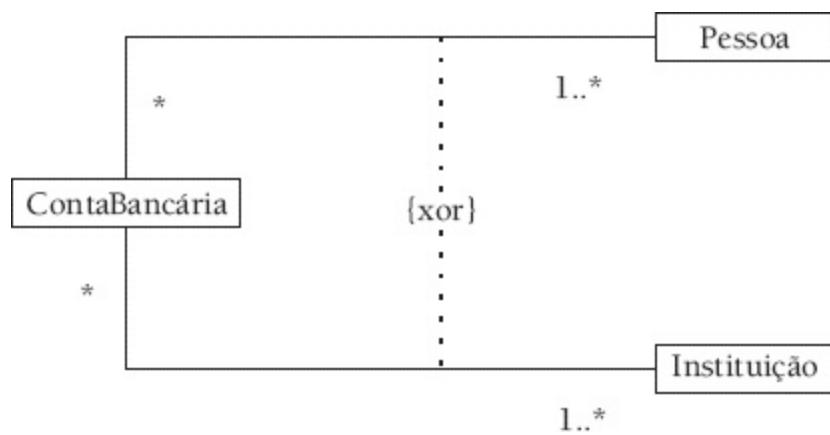


Figura 5-22: Exemplo de utilização da restrição *xor*.

As restrições também podem ser definidas formalmente em OCL (ver [Seção 3.6](#)). Expressões nessa linguagem podem ser definidas utilizando-se uma expressão da forma Item.seletor, que permite o acesso às propriedades de uma classe (atributos, operações e associações). Essas expressões também podem ser combinadas recursivamente para formar outras, mais complexas. Além disso, os operadores aritméticos (+, -, *, /) e lógicos (>, .=, <,,=, <>, =) também estão disponíveis para definir expressões na OCL. A seguir, descreveremos vários exemplos de diagramas de classes nos quais são definidas expressões em OCL.

Na [Figura 5-23](#), a expressão em OCL define que, para um objeto Voo, a quantidade de horas de treinamento do piloto deve ser maior ou igual à quantidade mínima de horas exigida para o avião a ser pilotado.

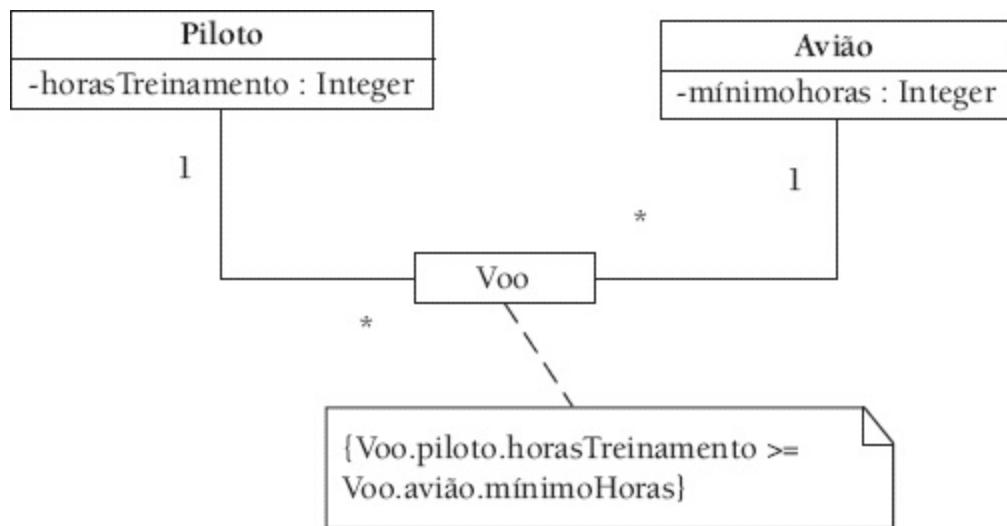


Figura 5-23: Exemplo de utilização da OCL para definir restrições.

No exemplo da [Figura 5-24](#), tem-se uma restrição em OCL definida sobre o elemento de associação entre as classes do diagrama. Essa restrição indica que um indivíduo deve ter mais de 21 anos para participar de certa sociedade.

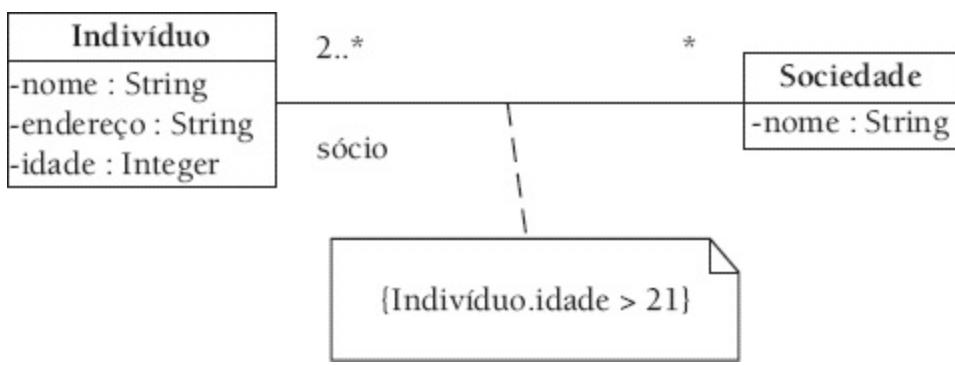


Figura 5-24: Exemplo de definição de restrição em OCL.

Outro exemplo (adaptado do documento de especificação da UML) é ilustrado na [Figura 5-25](#), em que há duas classes, Empresa e Pessoa. A restrição em OCL deste exemplo implica que uma pessoa que trabalhe para uma empresa empregadora e o gerente dessa pessoa precisam trabalhar para a mesma empresa.

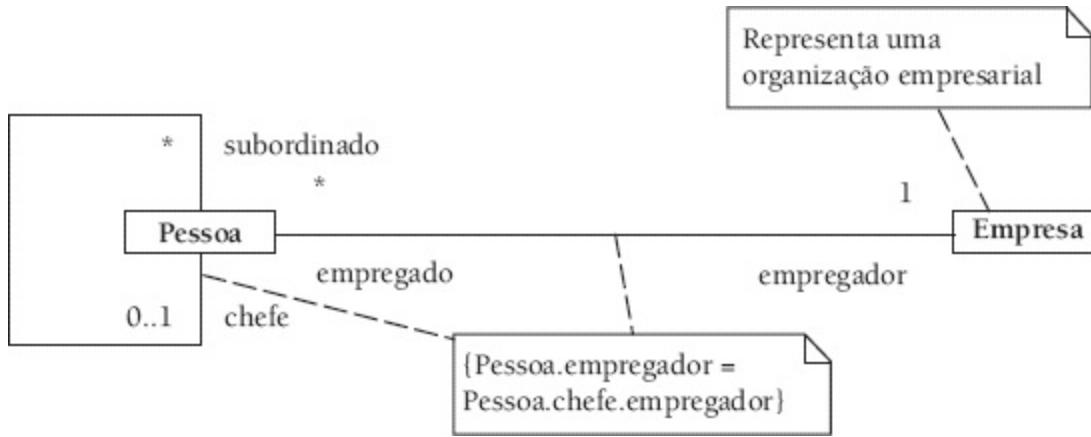


Figura 5-25: Exemplo de definição de restrição em OCL.

5.2.3 Generalizações e especializações

Além de relacionamentos entre objetos, o modelo de classes também pode representar relacionamentos entre classes. Esses últimos denotam relações de generalidade ou especificidade entre as classes envolvidas. Por exemplo, o conceito *mamífero* é mais genérico que *ser humano*. Outro exemplo: o conceito *carro* é mais específico que *veículo*.

O relacionamento de herança é também chamado de relacionamento de *generalização/especialização*, ou simplesmente *gen/espec*. Isso porque a generalização e a especialização são dois pontos de vista do mesmo relacionamento: dadas duas classes A e B, se A é uma generalização de B, então B é uma especialização de A.

Os termos para denotar o relacionamento de herança são bastante variados. O termo *subclasse* é utilizado para denotar a classe que herda as propriedades de outra classe por meio de uma generalização. Diz-se ainda que a classe que possui propriedades herdadas por outras classes é a *superclasse*. Utilizam-se ainda os nomes *supertipo* e *subtipo* como sinônimos para superclasse e subclasse, respectivamente. Outros termos (mais utilizados em linguagens de programação orientada a objetos) são *classe base* (sinônimo para superclasse) e *classe herdeira* (sinônimo para subclasse). Diz-se também que uma subclasse é uma *especialização* de sua superclasse (a subclasse especializa a superclasse), e que uma superclasse é uma *generalização* de suas subclasses (a superclasse generaliza as subclasses). Os termos *ancestral* e *descendente* fazem referência ao uso do

relacionamento de generalização em vários níveis (ver [Seção 5.2.3.1](#)).

No diagrama de classes, a herança é representada na UML por uma flecha partindo da subclasse em direção à superclasse. Alternativamente, podemos juntar as flechas de todas as subclasses de uma classe, como mostra a [Figura 5-26](#). A utilização de um ou de outro estilo é questão de gosto.

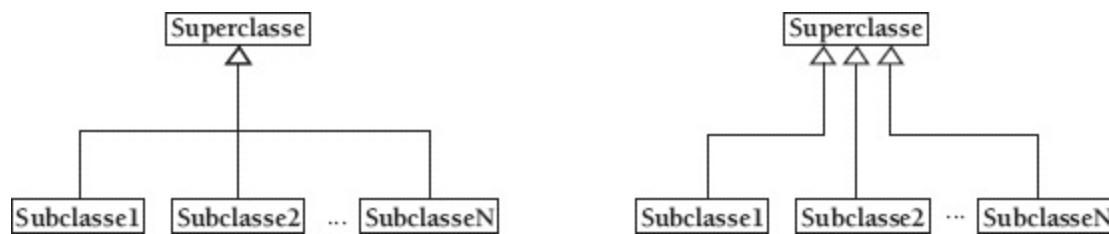


Figura 5-26: Representações alternativas para o relacionamento de generalização na UML.

Para entender a semântica de um relacionamento de herança, é preciso lembrar que uma classe representa um conjunto de objetos que partilham um conjunto comum de propriedades (atributos, operações, associações). No entanto, alguns objetos, embora bastante semelhantes a outros, podem possuir um conjunto de propriedades que esses outros não possuem. Por exemplo, todos os funcionários de uma empresa partilham os atributos nome, endereço, númeroMatrícula e salárioBase. No entanto, um tipo especial de funcionário – o dos vendedores – possui atributos específicos, por exemplo, zonaGeográfica (zona em que trabalham) e taxaComissão (porcentagem de comissão que recebem pelas vendas realizadas). Os vendedores pertencem à classe dos funcionários, mas formam por si próprios uma outra classe, a dos vendedores. Portanto, vendedores possuem todas as características inerentes a funcionários, além de possuir características próprias. Diz-se, assim, que a classe de vendedores *herda* propriedades da classe de funcionários.

É importante notar que não somente atributos e operações são herdados pelas subclasses, mas também as associações que estão definidas na superclasse. A [Figura 5-27](#) é um exemplo dessa propriedade: existe uma associação entre Cliente e Pedido. Não há necessidade de se criar uma associação entre Pedido e ClientePessoaFísica e entre Pedido e ClientePessoaJurídica, pois as subclasses herdam a associação de sua superclasse. Por outro lado, se existe uma associação que não é comum a todas as subclasses, mas sim particular a algumas delas, então essa associação deve ser representada em separado, envolvendo somente as subclasses em questão.

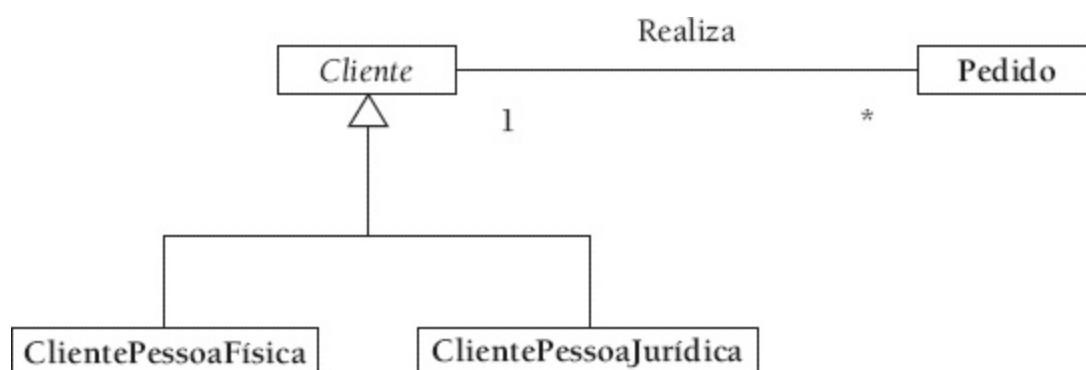


Figura 5-27: Não há necessidade de se criar uma associação entre Pedido e as subclasses de Cliente.

Um conceito que pode ser utilizado em situações de modelagem é a *classe abstrata*. Uma classe abstrata normalmente é utilizada para organizar a hierarquia de classes ou para fins de modelagem, como um contêiner de definições de propriedades. Classes abstratas não geram objetos diretamente (embora objetos de uma subclasse de uma classe abstrata sejam também considerados objetos dessa

última). Na notação da UML, uma classe abstrata é representada com seu nome em *itálico*. Por exemplo, na [Figura 5-27](#), a classe Cliente é abstrata. As classes Conta e ContaBancária na [Figura 5-28](#) também são abstratas. O conceito de classe abstrata tem importância fundamental no contexto da modelagem de classes de projeto. Na [Seção 8.5.2](#), descrevemos o conceito de classe abstrata naquele contexto.

É importante notar que o relacionamento de herança difere do de associação, porque o primeiro se trata de um relacionamento *entre classes*. Quando se diz, por exemplo, que gerentes são tipos especiais de funcionários significa que objetos da classe gerente possuem características comuns a todos os funcionários, além de poderem ter características próprias de um gerente. Por outro lado, os relacionamentos de associação, agregação e composição são definidos *entre objetos*. Isso equivale a dizer que objetos específicos de uma classe se associam entre si ou com objetos específicos de outras classes.

O relacionamento de herança acontece entre *classes*. Já o relacionamento de associação acontece entre *instâncias de classes* (i.e., entre objetos).

5.2.3.1 Propriedades do relacionamento de herança

O relacionamento de herança possui duas propriedades importantes, *transitividade* e *assimetria*, que descrevemos a seguir.

A propriedade da *transitividade* indica que uma classe em uma hierarquia herda tanto propriedades e relacionamentos de sua superclasse imediata quanto de suas não imediatas (classes em um nível mais alto da hierarquia). A propriedade de transitividade entre classes também pode ser vista sob a perspectiva de que toda instância de uma classe também é uma instância de suas superclasses. Ou seja, uma instância de uma classe C também é instância de todos os ancestrais de C. Uma classe é uma *generalização* de outra classe se toda instância desta última for também da primeira. Pela propriedade de transitividade, a herança pode ser aplicada em vários níveis, ou seja, uma classe que herda propriedades de uma outra classe pode ela própria servir como superclasse. A [Figura 5-28](#) ilustra uma *hierarquia de herança* de três níveis. Nessa hierarquia, são apresentadas cinco classes. Note que ContaPoupança possui um atributo próprio e mais três herdados de Conta. Note também a disposição das classes no diagrama, no qual as classes mais genéricas estão posicionadas acima das mais específicas. Essa disposição não é obrigatória, pois o sentido da seta indica qual é a classe mais genérica. No entanto, é recomendável dispor as classes dessa forma para enfatizar a hierarquia existente entre superclasses e subclasses.

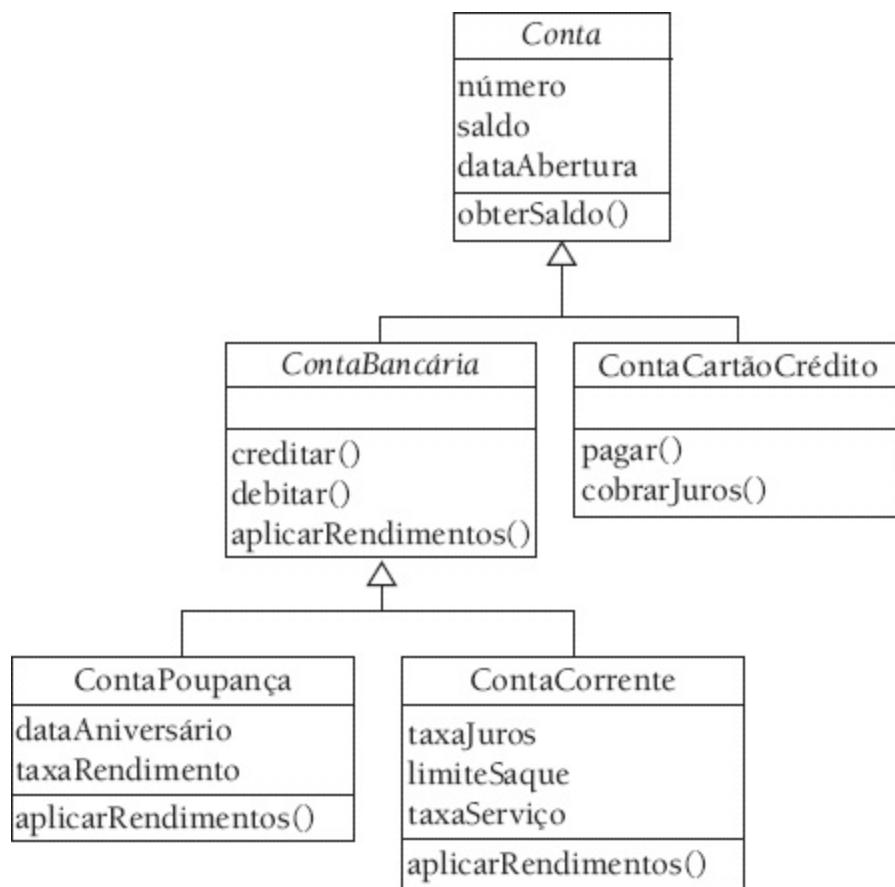


Figura 5-28: Exemplo de hierarquia de herança.

Outra propriedade importante da herança é a *assimetria*. Essa propriedade significa que dadas duas classes A e B, se A for uma generalização de B, então B não pode ser uma generalização de A. Ou seja, *não* pode haver ciclos em uma hierarquia de herança.

5.2.3.2 Refinando o modelo de classes com gen/espec

Graças a relacionamentos de herança, podem ser feitos refinamentos no modelo de classes. A ideia básica é identificar abstrações mais genéricas ou mais específicas que outras no diagrama de classes do sistema. Essa identificação de abstrações corresponde a refinamentos no modelo de classes que podem ser obtidos a partir de duas estratégias alternativas e complementares: a *generalização* e a *especialização*.

Primeiramente, pode ser que duas ou mais classes semelhantes tenham sido identificadas. Neste caso, talvez seja adequado criar uma *generalização*, ou seja, uma classe mais genérica, e definir as classes anteriores como subclasses desta última. Pode ser que a superclasse não gere instâncias próprias e esteja sendo utilizada somente para organizar classes semelhantes em uma hierarquia. Se for assim, a superclasse pode ser definida como *abstrata* (ver [Seção 8.5.2](#)), o que significa que ela tem o objetivo de organizar o modelo.

Em segundo lugar, também é possível aplicar a *especialização*, que corresponde ao processo de criar classes mais específicas a partir de uma classe preexistente. Por exemplo, talvez tenha sido identificada uma nova propriedade de uma classe preexistente que justifique a criação de uma subclass na qual esta propriedade seja definida.

Seja qual for o processo de derivação utilizado, generalização ou especialização, uma dica prática para confirmarmos se há legitimamente um relacionamento de gen/espec entre duas classes é aplicar a chamada *regra da substituição* que apresentamos a seguir. (Por conta dessa regra, o

relacionamento gen/espec também é conhecido como *relacionamento é-um*.)

Regra da substituição: sejam duas classes A e B, onde A é uma generalização de B. Não pode haver diferenças entre utilizar instâncias de B ou de A, do ponto de vista dos clientes de A.

Uma consequência da regra da substituição é a seguinte: é inadequado o uso do relacionamento gen/espec, onde nem todas as propriedades e comportamentos da superclasse fazem sentido para a subclasse.

Outro teste que pode ser realizado para identificar se duas classes se relacionam por gen/espec, e que equivale ao da regra da substituição, é perguntar o seguinte: *X é um tipo de Y?* Se a resposta for “sim”, provavelmente a classe X deve ser definida como uma subclasse de Y.

Deve-se também considerar a conformidade da subclasse em relação às propriedades definidas em sua superclasse. Ou seja, é inadequado o uso de gen/espec na situação em que nem todas as propriedades da superclasse fazem sentido para a subclasse. Essa restrição é uma interpretação da regra da substituição.

Como um exemplo de uso adequado de gen/espec (e que respeita a regra da substituição), observe o diagrama de classes da [Figura 5-29](#). Note que tanto ContaCorrente quanto ContaPoupança são tipos de ContaBancária. Observe também que os relacionamentos e propriedades da superclasse valem para ambas as subclasses.

Por meio da generalização e da especialização, classes mais gerais podem ser definidas a partir de classes mais específicas, e classes mais específicas também podem ser definidas a partir de classes mais gerais. Entretanto, na aplicação de generalização ou especialização, deve-se evitar a construção de hierarquias de herança muito profundas (com mais de três níveis), pois elas dificultam a leitura do diagrama.

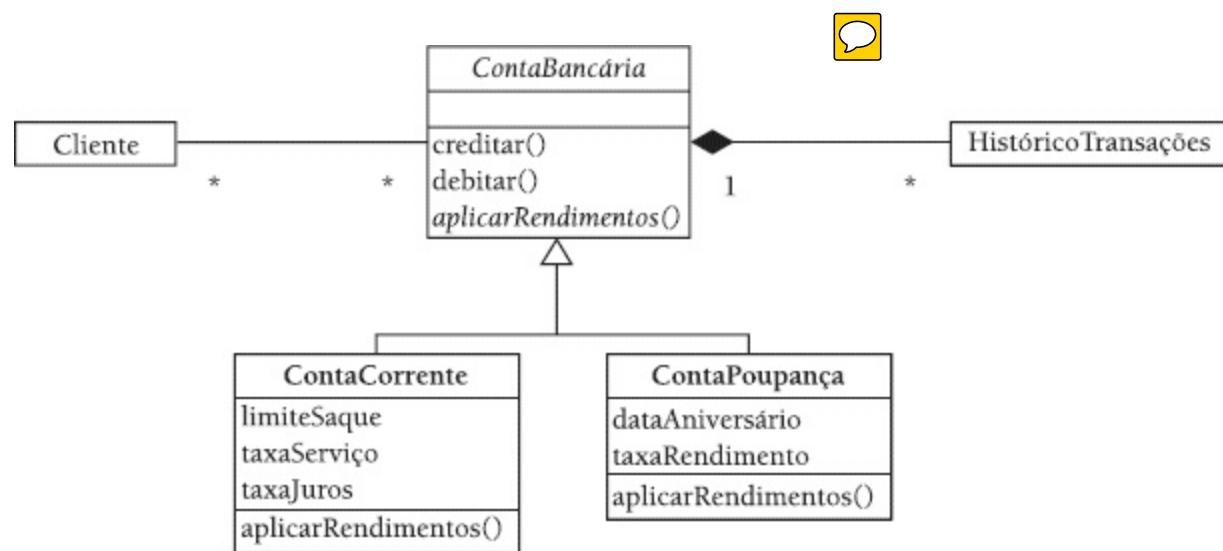


Figura 5-29: Conformidade das subclasses à superclasse.

Outra dica: papéis (ver [Seção 5.2.2.3](#)) e subclasses não devem ser confundidos. Um papel corresponde ao uso de certa classe em uma associação. Uma classe pode assumir vários papéis. O modelador precisa evitar a criação de subclasses em situações que podem ser resolvidas pela utilização de papéis. Veja o exemplo da [Figura 5-30](#).

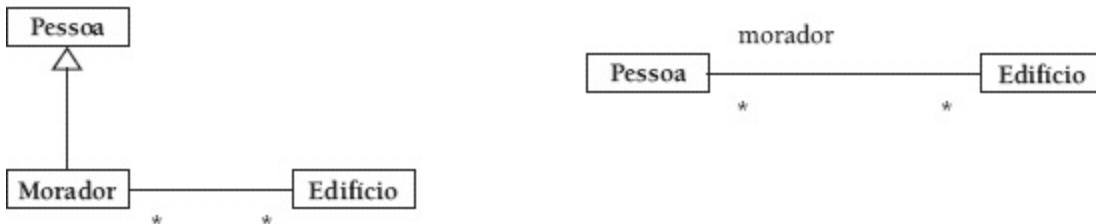


Figura 5-30: Situação em que é mais adequado utilizar um papel do que uma subclasse.

5.2.3.3 Definição de restrições sobre gen/espec

Conforme descrito na [Seção 3.4](#), a UML permite que determinadas restrições sejam associadas a elementos de um modelo. Nesta seção, descreveremos algumas restrições predefinidas pela UML que se aplicam ao relacionamento gen/espec.

As restrições sobre gen/espec são representadas no diagrama de classes, próximas à linha do relacionamento. Essas restrições são apresentadas entre chaves. As restrições predefinidas pela UML para hierarquias de herança são descritas na [Tabela 5-3](#).

Tabela 5-3: Restrições predefinidas para generalizações

Restrição	Significado
sobreposta	Posteriormente podem ser criadas subclasses que herdem de mais de uma subclasse (herança múltipla).
disjunta	Quaisquer subclasses criadas posteriormente poderão herdar de somente uma subclasse.
completa	Todas as subclasses possíveis foram enumeradas na hierarquia.
incompleta	Nem todas as subclasses foram enumeradas na hierarquia.

As duas primeiras restrições (sobreposta e disjunta) e as duas últimas (completa e incompleta) são exclusivas entre si. Isso quer dizer que não há sentido em definir subclasses que sejam sobrepostas e disjuntas ao mesmo tempo, e que não há razão para definir subclasses simultaneamente com as restrições completa e incompleta. A seguir, veremos diversos exemplos de hierarquias de herança em que se utilizam restrições.

O exemplo da [Figura 5-31](#) informa que há outras subclasses de Veiculo além das que foram enumeradas na hierarquia.

Na [Figura 5-32](#), as restrições completa e disjunta indicam que (1) não há objeto dentro desta hierarquia que não seja nem homem nem mulher (completa) e (2) não há objeto nesta hierarquia que seja homem e mulher ao mesmo tempo (disjunção).

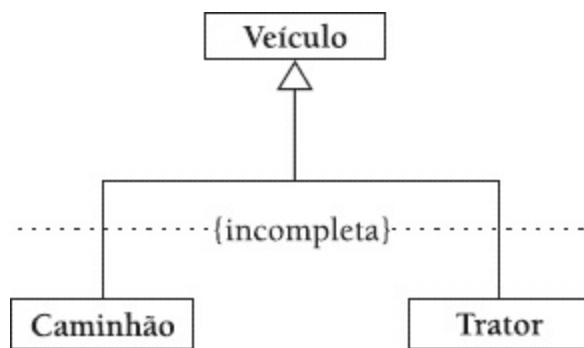


Figura 5-31: Herança incompleta.

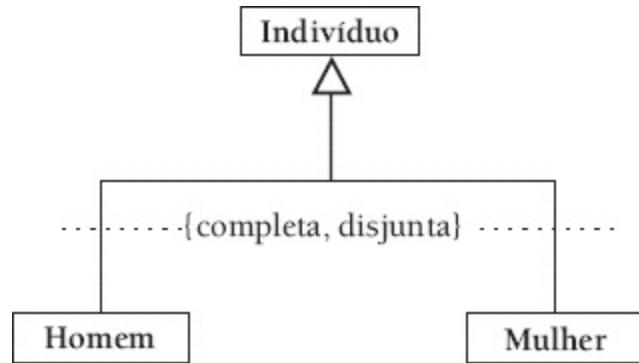


Figura 5-32: Herança completa e disjunta.

Na [Figura 5-33](#), o uso da restrição sobreposta significa que um Atleta pode ser tanto Nadador quanto Corredor. Além disso, há outras subclasses de Atleta não enumeradas no diagrama (incompleta).

Na [Figura 5-34](#), Elipse, Quadrado e Círculo são tipos de FiguraGeométrica. Além disso, existem outras subclasses não especificadas na hierarquia.

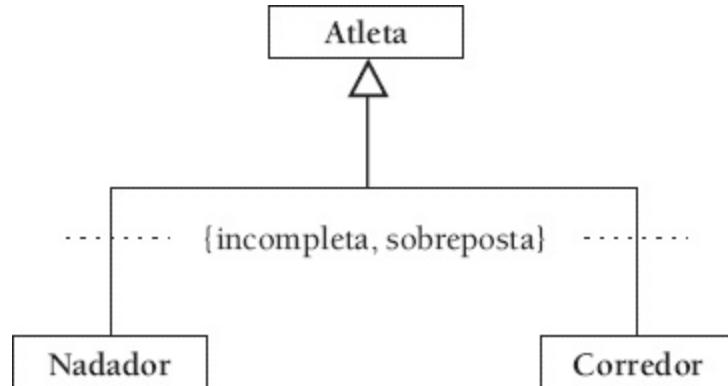


Figura 5-33: Herança incompleta e sobreposta.

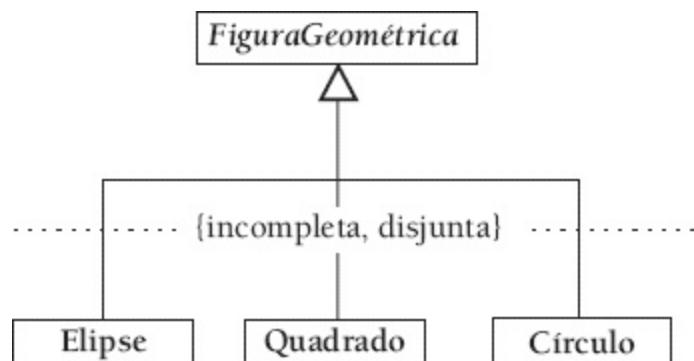


Figura 5-34: Herança incompleta e disjunta.

5.3 Diagrama de objetos

Além do diagrama de classes, a UML define outro tipo de diagrama estrutural: o *diagrama de objetos*. Esses diagramas podem ser vistos como instâncias de diagramas de classes, da mesma forma que objetos são instâncias de classes.² Assim como diagramas de classes, os de objetos são estruturas estáticas. Um diagrama de objetos exibe uma “fotografia” do sistema em certo momento, exibindo as ligações formadas entre objetos conforme estes interagem e de acordo com os valores dos seus atributos.

Em um diagrama de objetos, cada objeto é representado por um retângulo com dois compartimentos. No compartimento superior, a identificação do objeto é exibida. No inferior (cuja utilização é opcional), aparecem valores para os atributos definidos na classe do objeto.

A identificação do objeto deve ser sempre sublinhada. Por convenção, o nome da classe começa com letra maiúscula e pode ser omitido quando for conveniente. Os dois formatos possíveis para a identificação de um objeto, juntamente com exemplos de cada um, são apresentados na [Tabela 5-4](#).

Tabela 5-4: Possíveis formatos para identificação de instâncias em um diagrama de objetos

Formato	Exemplo
:NomeClasse	:Pedido
nomeObjeto: NomeClasse	umPedido: Pedido

O segundo compartimento do retângulo que representa um objeto, quando utilizado, exibe uma lista de pares da forma *nome do atributo: valor do atributo*.

Como exemplo, a [Figura 5-35](#) exibe um diagrama de objetos instanciado a partir do diagrama de classes da [Figura 5-18](#). Este diagrama de objetos mostra uma instância de pedido ligada a três instâncias de item de pedido. Cada item, por sua vez, está ligado a um produto. Note que são apresentados valores para cada atributo de um objeto.

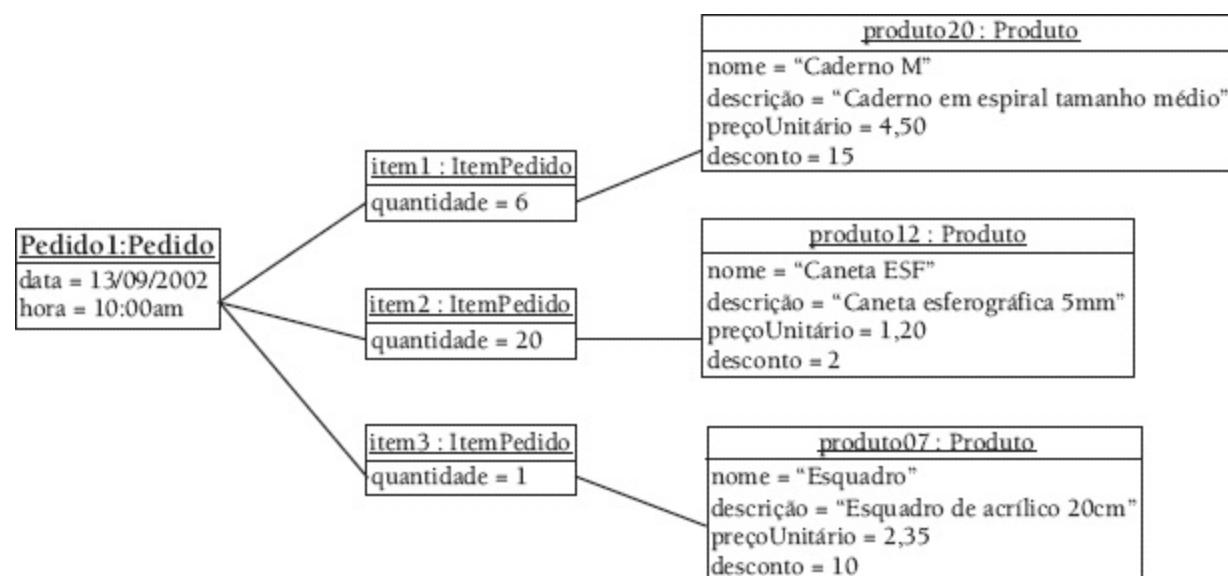


Figura 5-35: Exemplo de diagrama de objetos.

A [Figura 5-36](#) exibe outro exemplo de diagrama de objetos, em que os compartimentos dos valores de atributos são omitidos. Esse diagrama é uma instância do diagrama de classes da [Figura](#)

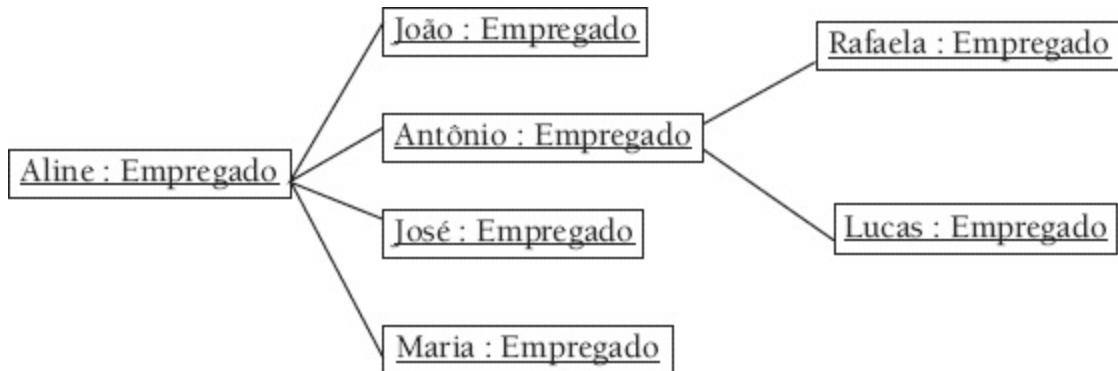


Figura 5-36: Exemplo de diagrama de objetos em que os valores dos atributos não são exibidos.

O diagrama de objetos é uma contribuição do método de Booch (ver [Seção 1.3](#)) à UML. Considerados isoladamente, esses diagramas são raramente utilizados na prática. A única utilidade prática e direta dos diagramas de objetos é a de ilustrar a formação de relacionamentos complexos de um diagrama de classes, como associações reflexivas. Com o objetivo de facilitar a atividade de *validação* (ver [Seção 2.1.2](#)), podemos construir diagramas de objetos para ilustrar e esclarecer certos aspectos de um diagrama de classes. Entretanto, fora desse contexto de validação, a descrição do diagrama de objetos ainda é relevante, pois o diagrama de interação (mais particularmente, o de comunicação), descrito no [Capítulo 7](#), utiliza a mesma notação que o primeiro.

5.4 Técnicas para identificação de classes

Na [Seção 5.2](#), descrevemos alguns dos elementos do diagrama de classes. No entanto, apenas o conhecimento da notação existente para produção daquele diagrama não é o bastante para a construção do modelo de classes de um *sistema de software orientado a objetos* (SSOO). De fato, uma das tarefas mais importantes e difíceis no desenvolvimento de um SSOO é a identificação das classes necessárias e suficientes para compor esse sistema.

Nesta seção, nosso objetivo é estudar algumas técnicas de identificação de classes. Antes de continuarmos, porém, é importante oferecermos uma explicação do termo “identificação de classes”. Por definição, um SSOO é composto de uma coleção de objetos que colaboram para realizar as tarefas desse sistema. Por outro lado, sabemos que todo objeto de um SSOO pertence a uma classe. Portanto, quando se fala em “identificação das classes”, o objetivo na verdade é saber quais objetos irão compor o sistema em questão.

Independentemente da técnica utilizada, a tarefa de identificação de classes se divide em duas atividades. Primeiramente, *classes candidatas* (ou seja, entidades que podem se tornar classes) são identificadas. Depois disso, são aplicados alguns princípios para eliminar classes candidatas desnecessárias. A segunda atividade é, sem dúvida, a mais complicada: identificar *possíveis* classes para um sistema não é a parte mais difícil; o problema é eliminar desse conjunto o que não é necessário. Além disso, é importante notar que as atividades para identificação de classes *não* são sequenciais (uma não começa quando a outra acaba). Ao contrário, os desenvolvedores intercalam a realização dessas atividades, identificando novas candidatas e removendo candidatas previamente identificadas. Parafraseando Bertrand Meyer: “Como um jardineiro, o [desenvolvedor] deve, a todo o momento, alimentar as plantas boas e eliminar as ervas daninhas” (MEYER, 1997).

5.4.1 Análise textual de Abbott

Em 1983, Russell Abbott propôs essa técnica de identificação que hoje leva seu nome (ABBOTT, 1983). Na chamada *Análise Textual de Abbott* (ATA), utilizam-se diversas fontes de informação sobre o sistema: documento de requisitos, modelos do negócio (se existirem; ver [Seção 4.6.1](#)), glossários, conhecimento a respeito do domínio etc. Para cada um desses documentos, os nomes (substantivos e adjetivos) que aparecem no mesmo são destacados. (São também consideradas locuções equivalentes a substantivos.) Após isso, os sinônimos são removidos (permanecem os nomes mais significativos para o domínio do negócio em questão). Cada termo remanescente se enquadra em uma das situações a seguir:

1. O termo se torna uma classe (ou seja, são classes candidatas).
2. O termo se torna um atributo.
3. O termo não tem relevância alguma com relação aos requisitos do SSOO.

Abbott segue na descrição de sua proposta para aplicá-la também na identificação de *operações* de uma classe e de *associações* entre classes. Para isso, ele sugere que destaquemos os verbos no texto. Verbos com sentido de ação (como, por exemplo, calcular, confirmar, cancelar, comprar, fechar, estimar, depositar, sacar etc.) são operações em potencial. Verbos com sentido de “ter” são agregações ou composições (ver [Seção 5.2.2.7](#)) em potencial. Verbos com sentido de “ser” são generalizações (ver [Seção 5.2.3](#)) em potencial. Os demais verbos são associações em potencial.

Conforme descrito no parágrafo anterior, uma operação é identificada mediante análise de um verbo que denota ação. A classe na qual essa operação deve ser definida pode ser encontrada através do sujeito da frase em que o verbo é utilizado. Além disso, informações necessárias à realização dessa operação normalmente podem ser encontradas no complemento da frase.

Parte do Texto	Componente	Exemplo
Nome próprio	Objeto	Eduardo Bezerra
Nome simples	Classe	aluno
Verbos de Ação	Operação	registrar
Verbo Ser	Herança	é um
Verbo Ter	Todo-parte	tem um

Uma vantagem da análise de Abbott é que essa abordagem é bastante simples: para cada documento, destacam-se termos que detectam componentes do modelo de classes. No entanto, apesar da facilidade de aplicação dessa técnica, uma desvantagem é que seu resultado (as classes candidatas identificadas) depende de o documento utilizado como fonte ser completo. Além disso, levando em consideração o estilo que foi utilizado para escrever esse documento, essa técnica pode identificar diversas classes candidatas que não gerarão classes. Pior que isso, a análise do texto de um documento talvez não deixe explícita uma classe importante para o sistema. Isso porque, em linguagem natural, as variações linguísticas e as formas de expressar uma mesma ideia são bastante numerosas.

Para contornar os problemas na identificação de classes por meio da análise textual, uma solução é aplicar outra técnica para validar o que foi descoberto inicialmente e para identificar novas

classes.

5.4.2 Análise dos casos de uso

Essa técnica é um caso particular da Análise Textual de Abbott, descrita na [Seção 5.4.1](#). A análise de caso de uso é preconizada pelo processo de desenvolvimento denominado RUP (*Rational Unified Process*). Essa técnica é também chamada de *identificação dirigida por casos de uso*. Na análise dos casos de uso, o MCU (ver [Capítulo 4](#)) é utilizado como ponto de partida. Conforme vimos no [Capítulo 4](#), um caso de uso corresponde a um comportamento específico do sistema. Em um SSOO, esse comportamento somente pode ser produzido por objetos que compõem o sistema. Em outras palavras, a realização de um caso de uso é responsabilidade de um conjunto de objetos que devem colaborar para que se obtenha o resultado daquele caso de uso. Com base nisso, o modelador que aplica a técnica de análise dos casos de uso tenta identificar as classes necessárias para produzir o comportamento que está documentado na descrição do caso de uso.

A técnica de análise dos casos de uso é aplicada da seguinte maneira: o modelador estuda a descrição textual de cada caso de uso para identificar classes candidatas. Para cada caso de uso, seu texto (fluxos principal, alternativos e de exceção, pós-condições e precondições etc.) é analisado. Ao fazer isso, o modelador tenta identificar classes que possam fornecer o comportamento de certo caso de uso. Ao longo desse processo, as classes do SSOO são identificadas. Quando todos os casos de uso tiverem sido analisados, todas as classes (ou pelo menos a maioria delas) terão sido identificadas.

Essa técnica tem como base o fato de que a existência de uma classe só pode se justificar se ela participar de alguma forma do comportamento externamente visível do sistema. Dessa forma, o modelo de classes é obtido a partir da descrição dos casos de uso.

De forma resumida, os passos na análise de casos de uso podem ser descritos da seguinte forma:

1. Suplemente as descrições dos casos de uso. Esse passo envolve complementar a descrição dos casos de uso com o objetivo de torná-los completos e facilitar a identificação de todas as classes envolvidas no mesmo.
2. Para cada caso de uso:
 - a. Identifique classes a partir do comportamento do caso de uso.
 - b. Distribua o comportamento do caso de uso pelas classes identificadas.
3. Para cada classe de análise resultante a. Descreva suas responsabilidades. b. Descreva atributos e associações.
4. Unifique as classes de análise identificadas em um ou mais diagramas de classes.

No passo em que se identificam as classes a partir do comportamento do caso de uso, há uma categorização de objetos que pode servir como ponto de partida para tal tarefa. Descrevemos essa categorização na [Seção 5.4.2.1](#). Além disso, a *modelagem de interações* na fase de análise se encaixa no contexto da realização dos passos descritos anteriormente. Descrevemos os diagramas de interação no [Capítulo 7](#).

5.4.2.1 Categorização BCE

Uma técnica de identificação que pode ser combinada com a análise de casos de uso é a *Análise de*

Robustez. Essa técnica foi proposta por Ivar Jacobson (JACOBSON *et al.*, 1992) e posteriormente adotada pelo processo de desenvolvimento denominado ICONIX (ROSENBERG e SCOTT, 2001). De acordo com essa técnica, os objetos que compõem um SSOO podem ser divididos em três categorias: *fronteiras*, *controladores* e *entidades*. Chamamos essa categorização de BCE (para lembrar os nomes originais das categorias: *boundary*, *control* e *entity*). Essa categorização fornece uma espécie de “arcabouço” que podemos tomar como ponto de partida para a identificação de classes em cada caso de uso de um sistema.

No contexto do diagrama de classes, a UML fornece estereótipos (ver [Seção 3.1](#)) textuais e gráficos predefinidos para cada categoria BCE. A [Figura 5-37](#) apresenta de forma esquemática os modos de representar objetos em cada uma dessas três categorias.

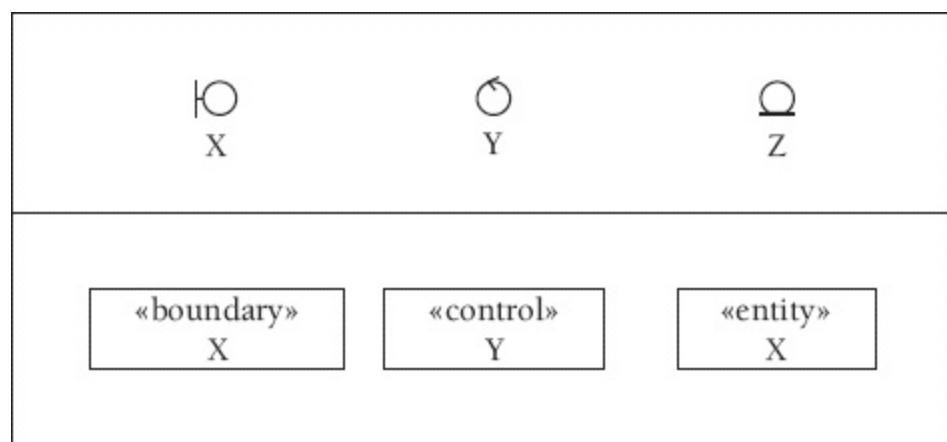


Figura 5-37: Notação da UML para objetos, segundo a categorização BCE.

5.4.2.2 Fronteiras

Fronteiras realizam a comunicação do sistema com atores. Em outras palavras, são objetos de fronteira que permitem ao sistema interagir com seu ambiente. Há três tipos principais de classes de fronteira: as que realizam a interface com o usuário (atores humanos), as que realizam a interface com sistemas externos e as que realizam comunicação com dispositivos atrelados ao sistema.

Para dar um exemplo, considere nosso estudo de caso do SCA (Sistema de Controle Acadêmico). Mais especificamente, analise o caso de uso Realizar Inscrição (ver [Seção 4.7](#)). Este caso de uso apresenta dois atores, Aluno e Sistema de Faturamento. Para que o sistema se comunique com esses atores, é necessário que existam objetos para realizar essa comunicação. Por essa razão, criamos as classes FormulárioInscrição e SistemaFaturamento (consulte a [Figura 5.47](#)).

Fronteiras têm tipicamente as seguintes responsabilidades: (1) notificar aos demais objetos os eventos gerados pelo ambiente do sistema; e (2) notificar aos atores o resultado de interações entre os demais objetos. Em resumo, fronteiras são especializadas em realizar *comunicação com o ambiente* do SSOO. Nomes de classes de fronteira não devem conter verbos nem sugerir ações. Em vez disso, esse nome precisa servir para lembrar qual é o canal de comunicação com o mundo externo que a classe representa. Exemplos: FormulárioInscrição, LeitoraCartões, SistemaFaturamento etc.

Fronteiras se comunicam com atores e com controladores (ver [Seção 5.4.2.3](#)). Normalmente existem somente durante a realização do caso de uso (isso é particularmente verdadeiro para objetos de fronteira que correspondem a interfaces gráficas com o usuário).

Cabe uma observação importante sobre fronteiras: durante a análise da aplicação (ver [Seção 2.1.2](#)), o modelador deve abstrair os detalhes dessa categoria de objetos. Isso significa que, durante a

análise, fronteiras devem ser consideradas somente como o ponto de contato com o ambiente do sistema. Em particular, os detalhes de *como* será feita essa comunicação com o ambiente (por interface gráfica, por algum protocolo de comunicação etc.) precisam ser ignorados na fase de análise. O objetivo nesta fase é identificar as classes e suas responsabilidades em um nível alto de abstração, sem comprometer eventuais soluções técnicas.

5.4.2.3 Controladores

Controladores servem como uma ponte de comunicação entre fronteiras e entidades. São responsáveis por coordenar a execução de alguma funcionalidade específica do sistema. Normalmente (embora não necessariamente), essa parte do sistema corresponde a um caso de uso. Esses objetos decidem o que o sistema deve fazer quando ocorre um evento externo relevante. Eles realizam o controle do processamento, ou seja, servem como “gerentes” dos outros objetos para a realização de um ou mais cenários de um caso de uso. Assim como fronteiras, essa categoria de objetos tem vida curta: em geral, existem somente durante a realização de um caso de uso.

Nomes normalmente utilizados para uma classe de controle devem lembrar qual caso de uso ela é responsável por coordenar. Dois exemplos de nomes de classes de controle são `RealizarInscriçãoControlador` e `ReservasCarrosControlador`.

Algumas responsabilidades típicas de controladores são: (1) realizar monitorações a fim de responder a eventos externos ao sistema (gerados por fronteiras); (2) coordenar a realização de um caso de uso por meio do envio de mensagens a fronteiras e a entidades; (3) criar associações entre entidades (embora isso possa ser feito pelas próprias entidades; ver mais adiante); (4) manter valores acumulados ou derivados durante a realização de um caso de uso; (5) manter o estado da realização do caso de uso.

5.4.2.4 Entidades

Uma entidade representa um conceito encontrado no domínio do problema (ver [Seção 2.1.1](#)). Normalmente servem como um repositório para alguma informação manipulada pelo sistema. Mas isso não é tudo. Nessas classes é que são alocadas as responsabilidades mais importantes do sistema: as que dizem respeito à *lógica do negócio*. Esses objetos são encontrados durante a *análise de domínio* (ver [Seção 2.1.2](#)). É comum existirem várias instâncias de uma mesma classe de entidade coexistindo no sistema.³ Por exemplo, em um sistema de venda de produtos, é possível haver milhares de objetos (instâncias) da classe Produto.

Entidades frequentemente participam de vários casos de uso e têm um ciclo de vida longo (ou seja, armazenam informações persistentes do sistema). Por exemplo, um objeto Aluno pode participar na realização dos casos de uso Realizar Inscrição e Lançar Notas (ver o estudo de caso do SCA). Além disso, uma vez criado, esse objeto pode existir por diversos anos ou mesmo tanto quanto o próprio sistema de que é componente. (No entanto, algumas entidades podem ter ciclo de vida curto e não ser persistentes.)

Algumas responsabilidades típicas de entidades são as seguintes: (1) informar valores de seus atributos a objetos requisitantes; (2) realizar cálculos ou impor restrições relativas às *regras do negócio* (ver [Seção 4.5.1](#)), normalmente com a colaboração de objetos de entidade associados; (3) criar e destruir objetos-parte (considerando que o objeto de entidade em questão seja um objeto-todo de uma agregação ou composição).

5.4.2.5 Categorização BCE na identificação de classes

A categorização proposta por Jacobson implica que cada objeto é especialista em realizar um dos três tipos de tarefa: comunicar-se com atores (fronteira), manter as informações (entidade) ou coordenar a realização de um caso de uso (controle).

Pelo exposto nas [Seções 5.4.2.2, 5.4.2.3 e 5.4.2.4](#), podemos concluir que a categorização BCE funciona como uma espécie de “receita de bolo” para identificar objetos participantes da realização de um caso de uso. Em particular, essa técnica preconiza que, para cada caso de uso, as seguintes regras podem ser aplicadas na análise:

1. Adicionar um objeto de fronteira para cada ator participante do caso de uso. Dessa forma as particularidades de comunicação com cada ator do caso de uso ficam encapsuladas no objeto de fronteira correspondente.
2. Adicionar um objeto de controle para o caso de uso, por ele representar um determinado processo do negócio. O controlador de um caso de uso tem então a atribuição de coordenar a realização desse processo do negócio.

A [Figura 5-38](#) ilustra de forma esquemática o que acontece quando os objetos de um caso de uso de um SSOO são identificados conforme a categorização BCE. O ator se comunica com um objeto de fronteira, que repassa (pelo envio de mensagens) as ações desse ator para o objeto de controle (controlador). O controlador, por sua vez, coordena a colaboração de uma ou mais entidades para a realização da tarefa desejada pelo ator. Note que, de acordo com o esquema dessa Figura, as entidades também se comunicam entre si com o envio de mensagem. Perceba também que a realização de uma tarefa pode envolver o envio de mensagens para outras fronteiras, para que o sistema se comunique com outros atores (p. ex., outros sistemas) envolvidos no caso de uso em questão. Após a conclusão da tarefa, o objeto de controle repassa o resultado para um objeto de fronteira (que pode ser o mesmo no qual a realização começou, ou outro objeto). Este objeto de fronteira, por sua vez, apresenta o resultado para o ator.

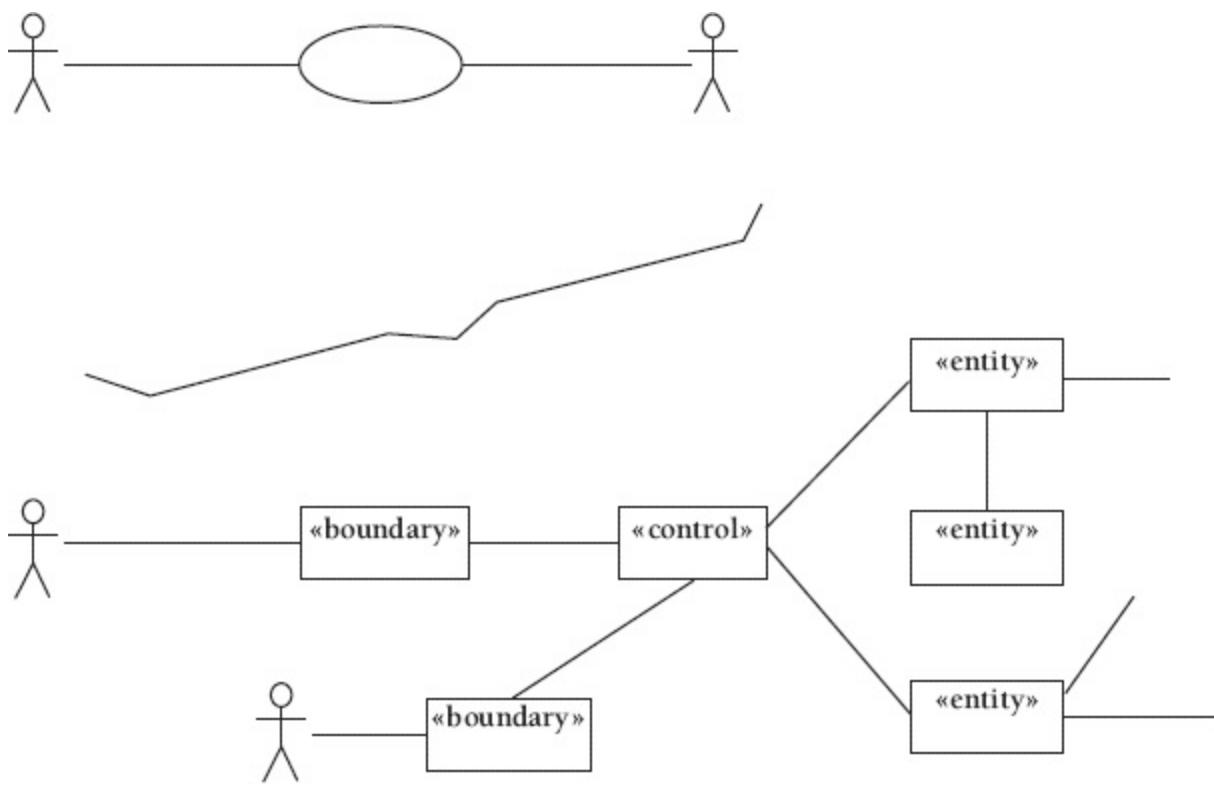


Figura 5-38: A realização de um caso de uso envolve objetos de fronteira, de controle e de entidade.

Classes de entidade, assim como seus atributos, são relativamente fáceis de identificar, porque tipicamente correspondem a conceitos pertencentes ao domínio do problema. Essas classes são identificadas na *análise do domínio*. Já as classes de entidade e de controle são normalmente encontradas na *análise da aplicação*. (Para detalhes acerca da análise de domínio e da análise da aplicação, ver Seção 2.1.2.)

5.4.3 Técnicas baseadas em responsabilidades

As técnicas de identificação de classes apresentadas nesta seção se baseiam fortemente no paradigma da orientação a objetos, em que objetos colaboram uns com os outros para que uma tarefa seja realizada (veja a Seção 1.2). A ideia básica dessas técnicas é a de que as responsabilidades atribuídas a um SSOO devem, em última análise, ser atribuídas aos objetos componentes do mesmo. Sendo assim, podemos partir das responsabilidades que atribuímos ao sistema e, para cada uma delas, identificar classes para assumi-las (ver Figura 5-39).

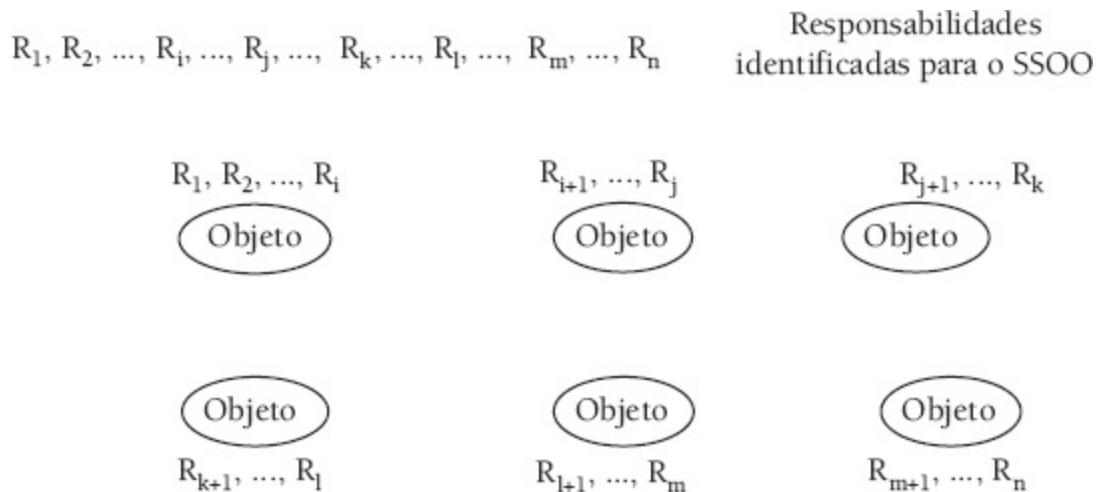


Figura 5-39: As responsabilidades de um sistema de software orientado a objeto devem ser alocadas aos objetos componentes do mesmo.

Nas técnicas baseadas em responsabilidades, as diversas responsabilidades de um SSOO são atribuídas a classes. Essas técnicas permitem *antropomorfizar* o conceito de classe. Ou seja, objetos são verdadeiramente interpretados como entidades ativas que encapsulam comportamento.

5.4.3.1 Projeto dirigido por responsabilidades

Na técnica de projeto dirigido por responsabilidades (*Responsibility Driven Design, RDD*), a ênfase está na identificação de classes a partir de seus *comportamentos* relevantes para o sistema. O esforço do modelador recai sobre a identificação das *responsabilidades* que cada classe deve ter dentro do sistema. Esse método foi proposto por Rebecca Wirfs-Brock e Brian Wilkerson (Wirfs-Brock e Wilkerson, 1989). Nas palavras desses autores: “O método dirigido a responsabilidades enfatiza o encapsulamento da estrutura e do comportamento dos objetos.” Ou seja, essa metodologia utiliza o princípio do encapsulamento, descrito na [Seção 1.2.3.1](#): a ênfase está na identificação das responsabilidades de uma classe que são úteis externamente a ela. Os detalhes internos à classe (*como* ela faz para cumprir suas responsabilidades) devem ser abstraídos.

Em um SSOO, os objetos encapsulam tanto dados quanto comportamento. O comportamento de um objeto é definido de tal forma que possa cumprir com suas *responsabilidades*. Uma responsabilidade de um objeto é uma obrigação que ele tem para com o sistema no qual está inserido. Graças às suas responsabilidades, um objeto colabora com outros objetos para que os objetivos do sistema sejam alcançados. Na prática, uma responsabilidade é algo que um objeto *conhece* ou *faz* (*sozinho ou sendo ajudado por outro(s) objeto(s)*). Como exemplo do conceito de responsabilidade, considere um objeto Cliente. Esse objeto provavelmente *conhece* seu nome, seu endereço, seu telefone etc. Considere um objeto Pedido; esse objeto conhece sua data de realização. Ele também deve fazer o cálculo do seu valor total.

Um objeto cumpre com suas responsabilidades a partir das informações que ele possui ou das informações que pode derivar de colaborações com outros objetos.

Em alguns casos, um objeto tem uma responsabilidade com a qual ele não pode cumprir sozinho. Nessas situações, o objeto deve requisitar *colaborações* de outros objetos do sistema para cumprir com a sua responsabilidade. Por exemplo, quando a impressão da fatura de um pedido é requisitada em um sistema de vendas, vários objetos precisam colaborar. O diagrama de classes da [Figura 5-40](#) ilustra as classes de alguns objetos que estão envolvidos nessa colaboração. Um objeto Pedido pode ter a responsabilidade de fornecer o seu valor total; um objeto Cliente fornece seu nome; cada ItemPedido informa a quantidade do produto correspondente e o valor de seu subtotal; os objetos Produto também colaboram fornecendo seu nome e preço unitário.

De uma forma geral, cada objeto tem um conjunto de responsabilidades dentro de um SSOO. Esse objeto pode ser capaz de cumprir sozinho algumas dessas responsabilidades. Já para outras, ele necessita da colaboração de outros objetos.⁴ Estes últimos são seus *colaboradores*. A [Figura 5-41](#) ilustra a correspondência entre classes (de objetos), responsabilidades e colaborações.

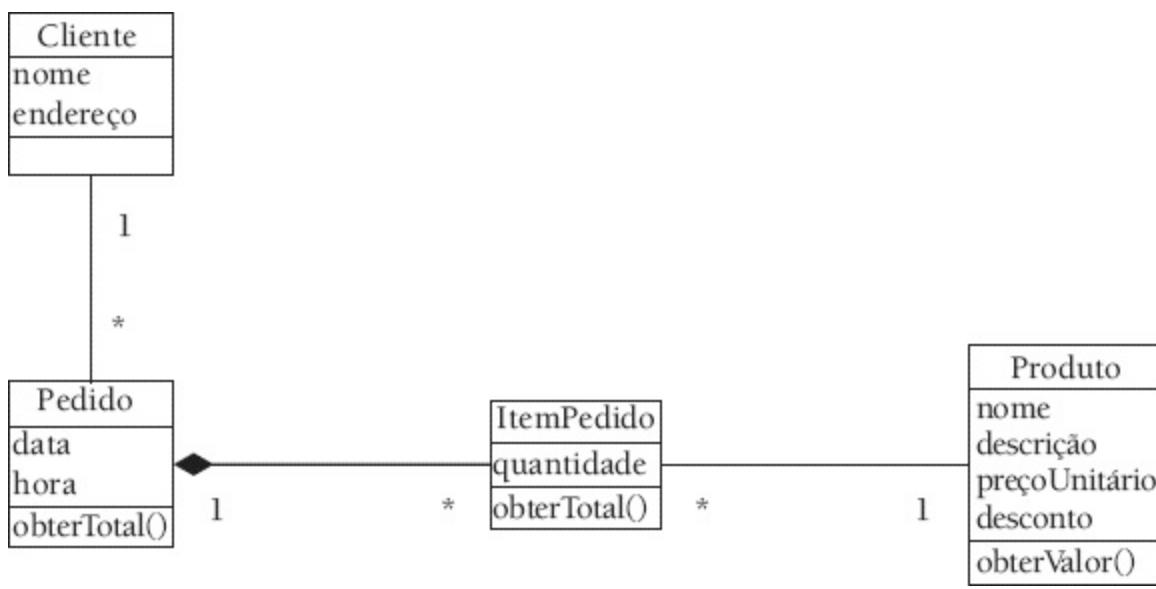


Figura 5-40: Objetos colaboram entre si para cumprir com suas responsabilidades.

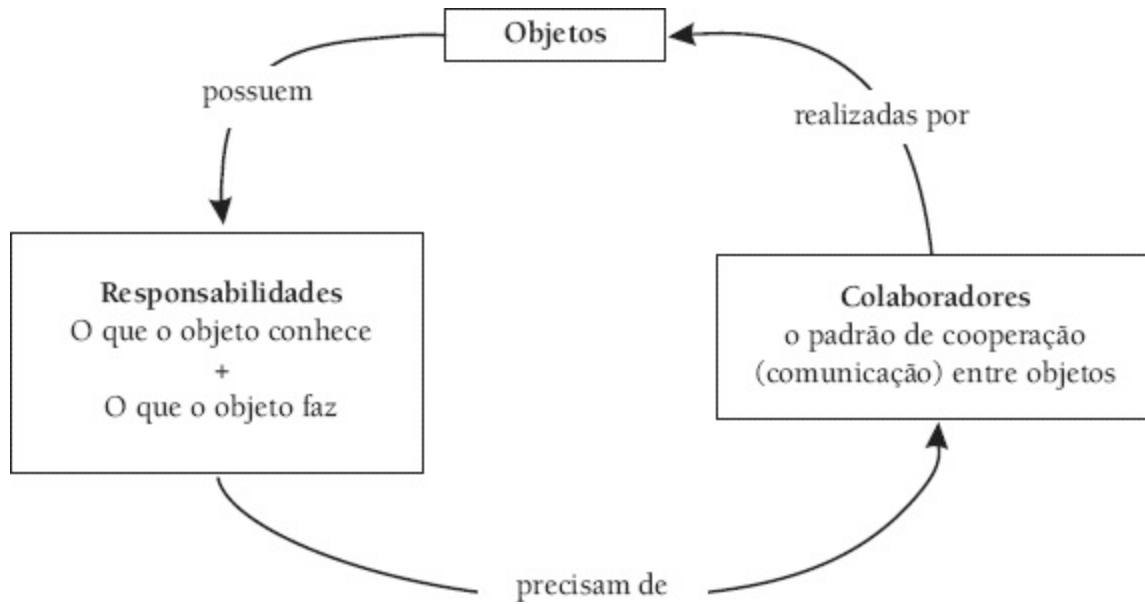


Figura 5-41: Relação entre objetos, responsabilidades e colaboradores.

A identificação de classes dirigida a responsabilidades normalmente utiliza uma técnica que permite a participação de especialistas do domínio e analistas. Essa técnica, a *modelagem CRC*, é descrita na [Seção 5.4.3.3](#).

5.4.3.2 Projeto dirigido ao domínio

O Projeto dirigido ao domínio (*Domain Driven Design*, DDD) é uma abordagem para desenvolvimento de sistemas de software complexos por meio do uso de padrões e de boas práticas de modelagem e desenvolvimento. Essa abordagem foi apresentada por Eric Evans em seu livro publicado em 2003 (EVANS, 2003). Em princípio, as ideias apresentadas por Evans não necessariamente estão confinadas ao contexto da orientação a objetos, mas a descrição delas se baseia fortemente nesse paradigma.

O ponto de partida do DDD é a construção de um *modelo do domínio* (ou *modelo conceitual*), que corresponde a uma representação abstrata do domínio de aplicação no qual se insere o sistema de software a ser desenvolvido. Na prática, o modelo de domínio é um conjunto de representações

de diferentes perspectivas do software a ser construído, que podem ser apresentadas por meio de diagramas de UML (embora o DDD não enfatize o uso da UML nessa construção).

De acordo com o DDD, o modelo de domínio deve ser construído em conjunto por especialistas do domínio e desenvolvedores de software. Além disso, essa construção deve usar expressões bem definidas e não ambíguas, provenientes do que se chama no DDD de **linguagem universal** ou **linguagem ubíqua** (tradução para *ubiquitous language*). O propósito dessa linguagem é facilitar a comunicação entre especialistas do domínio e desenvolvedores. Essa linguagem deve conter expressões provenientes do domínio da aplicação. Por exemplo, no SCA, quando desenvolvedores e especialistas usam termos provenientes desse domínio, como SemestreLetivo ou Disciplina, para se comunicar, essas duas partes devem ter o mesmo entendimento acerca do significado desses termos.

O DDD apresenta o conceito de **projeto estratégico** (*strategic design*), que descreve práticas adequadas para aplicar quando há vários modelos de domínio. As técnicas descritas no projeto estratégico do DDD visam facilitar a evolução e a coexistência desses diversos modelos, além de descrever as ações das equipes envolvidas no desenvolvimento dos diversos modelos de domínio eventualmente existentes. Em seu livro, Evans descreve essas práticas e as denomina de **padrões estratégicos**.

Além da linguagem universal e do projeto estratégico, o livro de Eric Evans traz o conceito **padrão tático**. Os padrões estratégicos e táticos são chamados conjuntamente de **blocos de construção do DDD**. Padrões táticos são relevantes para a tarefa de identificação de classes baseada em responsabilidades. Sendo assim, o restante desta [Seção 5.4.3.2](#) se restringe à apresentação de detalhes sobre essa família de padrões.

Cada padrão tático corresponde a um conjunto de famílias de classes nas quais, para cada classe pertencente a uma família, espera-se que ela cumpra com um conjunto pré-estabelecido de responsabilidades. Os principais padrões táticos do DDD são os seguintes: entidades, objetos valor, agregados, fábricas, repositórios e serviços. No restante desta seção, apresentamos detalhes acerca desses padrões. No [Capítulo 7](#) e no [Capítulo 8](#), alguns desses padrões são reconsiderados no contexto da etapa de projeto. O [Capítulo 11](#) também revisita esses padrões em um contexto arquitetural.

Entidade (*Entity*)

Na terminologia do DDD, uma entidade é um objeto que representa um conceito do domínio da aplicação e que possui uma identidade única, independente dos valores de seus atributos. Uma entidade é caracterizada por possuir um ciclo de atividade associadas. Por exemplo, no SCA, pode haver dois ou mais objetos que representam alunos de mesmo nome, José da Silva. Entretanto, sabemos que esses objetos não representam o mesmo indivíduo. Além disso, alunos passam por avaliações, realizam suas matrículas em disciplinas a cada período letivo, etc. Nesse exemplo, Aluno é uma classe que representa entidades. Outros exemplos de classes que representam entidades no SCA são Turma, Professor e Disciplina (veja a [Seção 5.7](#)).

Objeto Valor (*Value Object*)

Objetos valor são objetos (normalmente representantes de conceitos do domínio, mas não necessariamente) cujo estado não pode ser alterado após sua construção. Objetos valor representam conceitos de diferentes naturezas, como medidas, descrições, números, datas, montantes e cadeias de

caracteres. Geralmente são objetos cuja estrutura é simples. Sua identidade é baseada em seu *estado* (i.e., em uma configuração específica de valores de seus atributos; veja a [Seção 1.2.2](#)): se dois objetos valor possuem o mesmo estado, então eles são considerados iguais. Dessa forma, é possível que existam em um sistema vários objetos valor da mesma classe que representam o mesmo valor conceitual.

Por exemplo, no SCA, objetos das classes `SemestreLetivo` e `Intervalo` são objetos valor. Em particular, considere a classe `Intervalo`, responsável por representar um intervalo de tempo. É possível haver em uma sessão de uso do SCA vários objetos da classe `Intervalo` que representam o intervalo de tempo “das 18h20 às 20h”. Esses objetos podem ser considerados iguais para todos os motivos práticos. Como outro exemplo, considere um objeto que represente a quantia monetária correspondente a R\$ 10,00. Esse objeto deveria ser considerado igual a outro que represente o mesmo valor. Outros exemplos de conceitos que poderiam ser modelados como objetos valor são números de CPF, números de CNPJ, contas de correio eletrônico e números de ISBN (de livros).

A modelagem de conceitos simples por meio de classes (em vez de usar tipos primitivos de alguma linguagem de programação) permite associar comportamento a esses conceitos. Por exemplo, em uma classe `Email`, que representa contas de correio eletrônico, é possível atribuir a responsabilidade para validar a sintaxe em um endereço de e-mail. Essa forma de pensamento, que considera até mesmo conceitos simples como classes, está em consonância com o padrão objeto valor em particular, e com o paradigma da orientação a objetos em geral.

Agregado (*Aggregate*)

Como o próprio nome deixa transparecer, um agregado corresponde a classes organizadas em aglomerados, que podem envolver entidades ou objetos valor. Em geral, um agregado pode ser bastante complexo do ponto de vista de sua composição. Em um agregado, há sempre uma classe que corresponde à *raiz*, por meio da qual outros objetos podem manipular os demais objetos desse agregado.

Um exemplo de agregado é o composto pelas classes `ItemPedido` e `Pedido`, apresentada na [Figura 5-39](#). Repare que a classe `Pedido`, raiz desse aglomerado, está associada a vários objetos da classe `ItemPedido`. Quaisquer manipulações sobre objetos desta última classe devem ser realizadas por meio de mensagens enviadas ao objeto `Pedido` correspondente. Dessa forma, esse objeto `Pedido` assume a responsabilidade de manter a integridade da coleção de itens de pedido associados. Validações ou regras de negócio associadas à coleção de itens de pedido podem ser definidas como responsabilidades da classe `Pedido`.

No SCA, encontramos outro exemplo de agregado: a classe `Turma`, raiz desse agregado, encapsula objetos das classes `SemestreLetivo`, `Professor`, `Aula`, `Inscricao`, dentre outras. Em particular, para inscrever um aluno em uma turma, uma mensagem deve ser enviada para a raiz do agregado. Esse objeto raiz então se encarrega de criar um objeto da classe `Inscricao`, que representa um evento de inscrição de um aluno na turma. Veja a [Figura 5-49](#) e a [Figura 7-25](#). Repare que, ao assumir a responsabilidade de inscrever um aluno, a raiz desse agregado pode manter a consistência interna; em particular, pode ser validada a restrição acerca de um aluno não se inscrever em duas turmas de uma mesma disciplina em um mesmo período letivo.

Fábrica (*Factory*)

Fábricas são objetos do domínio cuja responsabilidade é criar objetos de entidade ou objetos

valor. Fábricas são úteis quando o processo de criação de algum objeto do domínio é muito complexo, por conta de envolver também a criação e associação de objetos componentes. Isso acontece comumente no caso de agregados. Quando for constatado que a atribuição da responsabilidade de criação de um objeto a sua própria classe tornaria essa classe muito complicada, uma alternativa é criar uma fábrica para assumir essa responsabilidade.

Repositório (*Repository*)

Um **repositório** é uma classe que representa uma coleção de entidades, agregados ou mesmo de objetos valor. Sua finalidade é fornecer uma abstração para o fato de haver um mecanismo de armazenamento persistente ao qual a aplicação faz acesso. Um repositório assume a responsabilidade de *adicionar* e *remover* objetos de certo tipo. Além disso, outra responsabilidade de um repositório é responder a consultas que solicitam subconjuntos específicos da coleção de objetos do domínio nele armazenada. Normalmente repositórios são criados para manipular agregados, e não para toda e qualquer classe de domínio.

Por exemplo, no SCA são manipuladas várias coleções de objetos do domínio: turmas oferecidas em um semestre letivo, alunos, professores, disciplinas. Re却itórios são criados para essas classes para manter seus objetos. Em particular, em um repositório para alunos representado por uma classe denominada `AlunoRepositorio`, podemos alocar responsabilidades para adicionar e remover alunos da coleção. Podemos também atribuir a essa classe responsabilidades como (1) resgatar um aluno por sua matrícula e (2) resgatar uma coleção de alunos que ingressaram no mesmo período letivo.

Os repositórios correspondem a um padrão tático importante, pois mantêm o modelo de domínio separado de aspectos relativos à tecnologia utilizada para persistência de objetos. De fato, durante a modelagem, quando um contêiner de objetos de domínio de certo tipo é necessário, podemos representá-lo como um repositório, sem considerações adicionais sobre a forma pela qual esses objetos serão armazenados de forma persistente. Um termo normalmente usado nesse contexto é *ignorância de persistência (persistence ignorance)*: re却itórios mantêm o modelo de classes de domínio separado de detalhes técnicos de persistência dos objetos. Repare que essa separação não diminui a importância da realização da especificação técnica acerca de como objetos serão efetivamente armazenados em (e resgatados de) um mecanismo persistente (e.g., um SGBD); apenas que a etapa de análise não é o momento adequado para abordar esse problema. (Veja também a discussão no [Capítulo 12](#) sobre camadas de persistência e frameworks para mapeamento objeto-relacional.)

Serviço do domínio (*Domain Service*)

Durante a atribuição de responsabilidades às classes de um sistema, surgem situações em que uma determinada responsabilidade não é adequada às classes já existentes, por conta de envolver a aplicação de lógica do negócio a objetos de diferentes tipos. Nessas situações, é possível criar uma classe de serviço para assumir essa responsabilidade.

Como exemplo, mais uma vez no contexto do SCA, considere o caso de uso `Realizar Inscrição` (veja a [Seção 4.7.3](#)). Nesse caso de uso, surge a necessidade de alguma classe assumir a responsabilidade de produzir a lista de turmas que devem ser apresentadas a um aluno para que ele selecione aquelas em que deseja se inscrever. Repare que nem todas as turmas oferecidas são possíveis a um aluno. Em particular, um aluno não pode se inscrever em uma turma de uma disciplina para a qual já tenha sido aprovado. Um aluno também não pode se inscrever em uma turma de uma disciplina para a qual não

tenha obtidos os créditos dos pré-requisitos. Sendo assim, há uma lógica de negócio que não é trivial e que deve ser atribuída a algum objeto do domínio. Uma possibilidade de modelagem nesse contexto é criar uma classe denominada `TurmasPossiveisServico` que assume essa responsabilidade de determinar quais são as disciplinas em que é possível um aluno se inscrever. Repare que essa classe deverá manipular coleções de objetos de diferentes tipos, como `Aluno`, `Turma` e `Disciplina`.

Por fim, classes que seguem o padrão tático *Serviço do Domínio* não devem ser confundidas com as classes da camada da aplicação, também denominados *Serviços da Aplicação*, que são discutidos no [Capítulo 11](#). Como o próprio nome diz, um Serviço do Domínio é um objeto do domínio, enquanto um serviço da aplicação é um objeto da camada da aplicação. Neste livro, adotamos a convenção de usar os sufixos `Controlador` e `Servico` nos nomes dos serviços da aplicação e do domínio, respectivamente (p. ex., `RealizarInscricaoControlador` e `TurmasPossiveisServico`).

5.4.3.3 Modelagem CRC

A *modelagem CRC* (sigla para *Classes, Responsabilidades e Colaboradores*) foi proposta em 1989 por Kent Beck⁵ e Ward Cunningham (Beck e Cunningham, 1989). A princípio, essa técnica de modelagem foi apresentada como uma forma de ensinar a iniciantes o paradigma da orientação a objetos. Contudo, a sua simplicidade de notação a tornou particularmente interessante para ser utilizada na identificação de classes.

A modelagem CRC é eficaz quando profissionais que não têm tanta experiência com o paradigma da orientação a objetos (o que normalmente é o caso para especialistas do domínio) estão envolvidos na identificação de classes.

Na modelagem CRC, especialistas do domínio e desenvolvedores trabalham em conjunto para identificar objetos, suas responsabilidades e colaboradores. Para aplicar essa técnica, esses profissionais se reúnem em uma sala, onde tem início uma *sessão CRC*.

Uma sessão CRC é uma reunião que envolve cerca de seis pessoas. Entre os participantes estão especialistas de domínio, projetistas, analistas e o moderador da sessão.

A cada pessoa é entregue um cartão de papel como o da [Tabela 5-4](#).⁶ Esse cartão é denominado *cartão CRC*. Cada um mede aproximadamente 10 cm x 15 cm e corresponde a uma das classes do SSOO.

Tabela 5-5: Formato típico de um cartão CRC

Nome da classe	
Responsabilidades	Colaboradores
1a responsabilidade	1o colaborador
2a responsabilidade	2o colaborador
...	..
n -ésima responsabilidade	m -ésimo colaborador

A [Tabela 5-6](#) ilustra um exemplo de cartão CRC para a classe `ContaBancária`. A coluna de colaboradores desse cartão foi definida em função das responsabilidades alocadas à classe.

Tabela 5-6: Cartão CRC para a classe Conta Bancária

ContaBancária

Responsabilidades	Colaboradores
1. Conhecer o seu cliente	Cliente
2. Conhecer o seu número	Transação
3. Conhecer o seu saldo	
4. Manter um histórico de transações	
5. Aceitar saques e depósitos	

Uma vez distribuídos os cartões pelos participantes, um conjunto de cenários (ver [Seção 4.1.1.4](#)) de determinado caso de uso é selecionado. Então, para cada cenário, uma sessão CRC é iniciada. (Se o caso de uso não for tão complexo, ele pode ser analisado em uma única sessão.)

Note que um cartão CRC é dividido em várias partes, conforme podemos ver na [Tabela 5-5](#). Na parte superior do cartão, aparece o nome de uma classe. A parte inferior do cartão é dividida em duas colunas. Na coluna da esquerda, o indivíduo ao qual foi entregue o cartão deve listar as responsabilidades da classe. Finalmente, na coluna da direita, o indivíduo aponta os nomes de outras classes que colaboram com a classe em questão para que ela cumpra suas responsabilidades.

É normal existirem algumas classes candidatas já identificadas para determinado cenário (obtidas com a aplicação de outras técnicas de identificação).⁷

A sessão CRC começa com algum dos participantes simulando o ator primário que dispara a realização do caso de uso. À medida que esse participante simula a interação do ator com o sistema, os demais presentes na sessão encenam a colaboração entre objetos que ocorre internamente ao sistema, para que o objetivo do ator seja alcançado. Graças a essa encenação desempenhada pelos participantes da sessão CRC, as classes, responsabilidades e colaborações são identificadas.

Durante uma sessão CRC, para cada responsabilidade atribuída a uma classe, o seu proprietário (o indivíduo ao qual foi alocado o cartão correspondente) deve questionar se tal classe é capaz de cumprir a responsabilidade sozinha. Se ela precisar de ajuda, um colaborador a oferece. Os participantes da sessão decidem então que outra classe pode fornecer tal ajuda. Se essa classe existir, ela recebe uma nova responsabilidade, necessária para que forneça ajuda. Do contrário, um novo cartão (ou seja, uma nova classe) é criado para cumprir com tal responsabilidade de ajuda.

Uma classe pode participar em mais de um caso de uso. À medida que os cenários de casos de uso são encenados, as responsabilidades (e os colaboradores) dessa classe vão se acumulando. Ou seja, as responsabilidades e os colaboradores necessários para a realização dos cenários aparecem de acordo com o desenrolar da sessão CRC. Além disso, durante uma sessão, uma ou mais das seguintes ações podem ser necessárias:

- Adicionar uma responsabilidade identificada a uma classe já existente.
- Criar uma classe para realizar uma responsabilidade identificada.
- Mover responsabilidades quando uma classe se tornar muito complexa.

5.4.3.4 Dicas gerais para atribuição de responsabilidades

A aplicação das técnicas de identificação baseada em responsabilidades é uam tarefa bastante dinâmica. Em cada simulação de um comportamento específicos do sistema (p. ex., um caso de uso), responsabilidades podem passar de um objeto para outro, novos objetos podem ser criados para assumir outras responsabilidades, objetos preexistentes podem não ser mais necessários etc. A

seguir, descrevemos algumas dicas gerais que podem ser usadas durante a análise para atribuir responsabilidades a classes.

Associe responsabilidades com base na especialidade da classe. Dependendo da especialidade (fronteira, controle ou entidade) da classe, há um conjunto típico de responsabilidades com as quais ela deve cumprir (ver [Seção 5.4.2.1](#)). O modelador pode tomar esse conjunto como base para atribuir responsabilidades às classes do sistema.

Distribua a inteligência do sistema. A inteligência do sistema deve ser uniformemente distribuída. Isso quer dizer que o conjunto de responsabilidades do sistema deve ser disposto o mais uniformemente possível pelas classes. Com efeito, uma classe não deve ser sobrecarregada com responsabilidades demais. Se isso começar a acontecer, considere a criação de outras classes para assumirem algumas responsabilidades que pertencem à classe sobrecarregada. Uma quantidade pequena de classes complexas demais significa que as mesmas encapsulam mais conhecimento (inteligência) do que o desejado e são menos reutilizáveis. Por outro lado, uma quantidade maior de classes mais simples significa que cada classe encapsula uma fração menor da inteligência do sistema e é, portanto, mais reutilizável.

Como exemplo, considere que a um objeto Pedido foi atribuída a responsabilidade de conhecer o seu valor total. Para isso, é necessário que todos os subtotais dos itens de pedido sejam calculados. Cada item de Pedido pode ficar responsável por calcular o seu total e informar o resultado ao objeto para que ele faça a totalização. Por sua vez, o item de pedido pode pedir ajuda ao objeto Produto para que este calcule o seu valor unitário (necessário ao cálculo do subtotal). A situação pode ser resumida da seguinte forma:

1. Pedido sabe o seu valor total.
2. ItemPedido sabe o seu valor total.
3. Produto sabe o seu valor unitário.

Nessa situação, cada classe fica com uma parte da responsabilidade relacionada à tarefa de computar o valor total de um pedido. A inteligência necessária para o cálculo desse valor é distribuída pelo sistema.

Agrupe as responsabilidades conceitualmente relacionadas. Responsabilidades conceitualmente relacionadas devem ser mantidas em uma única classe. Por exemplo, as informações (responsabilidades *de conhecer*) pertinentes a um cliente devem estar definidas em uma classe Cliente, e não espalhadas por diversas classes. Da mesma forma, é mais adequado criar as classes chamadas Pedido e Fatura em vez de uma única classe para manter informações dos dois conceitos. Essa dica está diretamente relacionada ao conceito de *coesão* de uma classe, que detalhamos na [Seção 7.5.2](#).

Evite responsabilidades redundantes. Responsabilidades redundantes não devem ser criadas. Se duas ou mais classes precisam de alguma informação, analise as seguintes decisões alternativas: criar uma nova classe ou escolher uma das classes preexistentes para manter a informação. Em ambos os casos, a informação fica em um único lugar, e os outros objetos devem enviar mensagens

para o objeto que possui tal informação para obtê-la.

A decisão de criar uma nova classe ou utilizar uma preexistente para manter a informação deve ser tomada com cuidado. É recomendável que o modelador tente sempre reutilizar uma classe preexistente para atribuir uma responsabilidade. No entanto, é preciso observar outra dica anterior: não se deve adicionar uma responsabilidade que não esteja relacionada ao propósito da classe; nesse caso, é melhor criar uma nova classe.

Como comentário final desta seção, repare que todas as responsabilidades identificadas durante a análise devem ser alvo de validação e refinamento na atividade de modelagem de interações do sistema. No [Capítulo 7](#), estudamos essa atividade tão importante da modelagem de um SSOO e descrevemos sua correlação com as responsabilidades identificadas na modelagem de classes de análise (por RDD, DDD ou CRC). De fato, a tarefa de identificação de responsabilidades começa na modelagem de classes de análise, mas continua durante a de interações. Portanto, é adequado seguir as mesmas dicas aqui apresentadas durante a modelagem de interações.

5.4.4 Padrões de análise

Após produzir diversos modelos para um mesmo domínio de problema, é natural que um modelador comece a identificar características comuns entre esses modelos. Em particular, um mesmo conjunto de classes ou colaboração entre objetos costuma recorrer, com algumas pequenas diferenças, em todos os sistemas desenvolvidos para o domínio do problema em questão. Por exemplo, quantos modelos de classes já construídos possuem os conceitos Cliente, Produto, Fornecedor, Departamento etc.? Quantos outros possuem os conceitos Ordem de Compra, Ordem de Venda, Orçamento, Contrato etc.?

A identificação de características comuns é um processo natural e acontece em consequência do ganho de experiência do modelador em determinado domínio de problema. Ao reconhecer processos e estruturas comuns em um domínio de problema, o modelador pode descrever (catalogar) a parte essencial dos mesmos, dando origem a um *padrão de software*. Quando o problema correspondente a um padrão de software acontecer, um modelador pode utilizar a essência da solução descrita pelo padrão (descrição essa possivelmente catalogada por outro modelador). A aplicação desse processo permite que o desenvolvimento de determinado aspecto de um SSOO seja feito de forma mais rápida e menos suscetível a erros. Isso porque a reutilização de uma mesma solução já comprovada anteriormente livra o modelador da tarefa de repensar (ou reinventar) tal solução.

Um padrão de software pode, então, ser definido como uma descrição essencial de um problema recorrente no desenvolvimento de softwares. Padrões vêm sendo estudados há anos e atualmente são utilizados em diversas atividades do desenvolvimento de um sistema, inclusive na análise. Padrões de software utilizados na análise são chamados *padrões de análise*. Um padrão de análise normalmente é composto, entre outras partes, de um fragmento de diagrama de classes que pode ser customizado para uma situação de modelagem em particular.

Pois bem, a técnica de identificação de classes pela utilização de padrões de análise consiste em identificar padrões que podem ser aplicados à modelagem do SSOO em questão. Sendo assim, o que ocorre não é exatamente uma identificação de classes, mas sim a identificação de problemas cujas soluções podem ser encontradas em padrões de análise.

Nos últimos anos, padrões de análise vêm sendo extensivamente estudados e catalogados. Está fora do escopo deste livro um estudo detalhado sobre padrões de análise. No entanto, com o objetivo de dar ao leitor um entendimento em alto nível sobre o assunto, nas próximas seções descrevemos

alguns padrões existentes.

5.4.4.1 O padrão Party

Esse padrão é normalmente utilizado para representar componentes de uma organização e os relacionamentos entre eles. Esse padrão trabalha com o conceito de *parte* (tradução para *party*). Uma parte é uma generalização de uma pessoa ou de uma organização de interesse para a aplicação sendo modelada. A [Figura 5-42](#) apresenta a estrutura de classes correspondente ao padrão Party.

N a [Figura 5-42](#), a classe Parte possui uma autoassociação que representa relações de subordinação: entre pessoas e organizações, entre pessoas (p. ex., quem supervisiona quem), ou entre organizações (p. ex., que organizações são subsidiárias de outras).

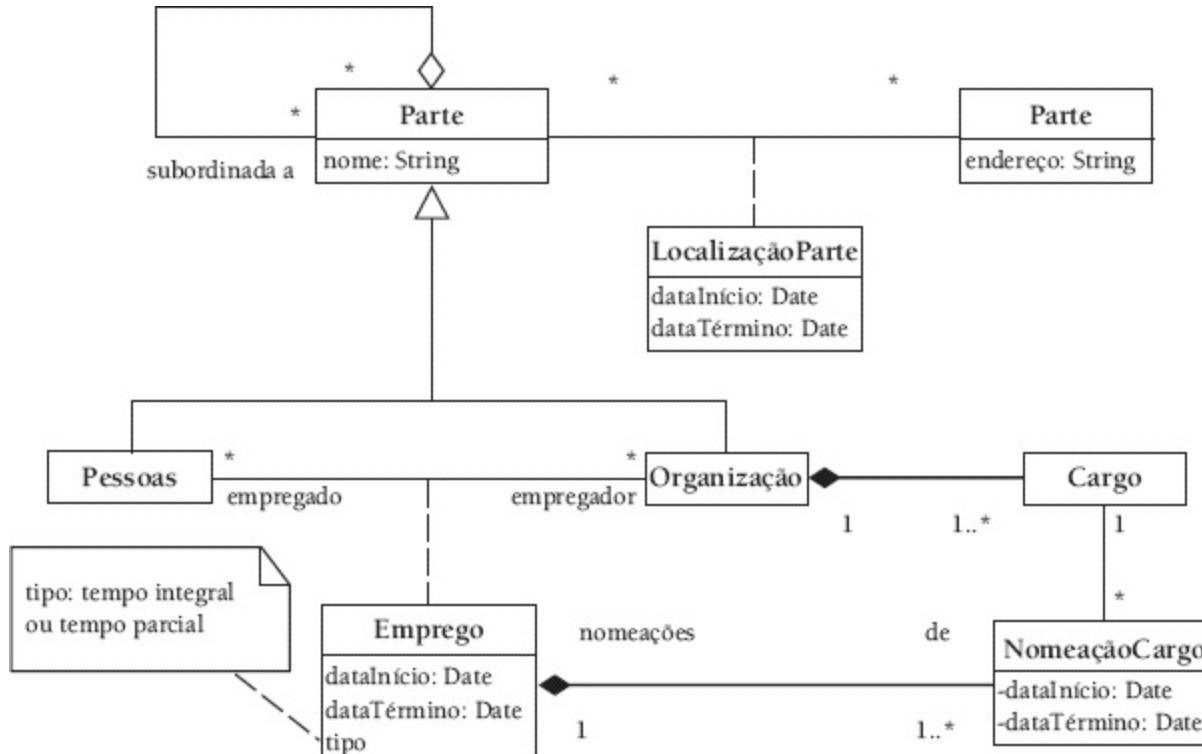


Figura 5-42: O padrão Party.

Embora a [Figura 5-42](#) não mostre isso, tanto Pessoa quanto Organização podem ter quaisquer atributos que sejam necessários a uma aplicação particular do padrão Party. Note também a existência da classe Emprego, que representa associações entre empregados (pessoas) e empregadores (organizações). Essa mesma classe está associada a NomeaçãoCargo, o que permite identificar que empregados assumiram determinados cargos em quais organizações e em que períodos.

5.4.4.2 O padrão Metamodel

Considere um conjunto de itens quaisquer. Considere, ainda, que cada item possui várias propriedades. Um caso particular dessa situação é aquele em que uma classe possui um grupo (fixo) de atributos, mas, no geral, as propriedades de cada item podem ser diferentes das propriedades dos outros itens do conjunto.

Por exemplo, analise os seguintes itens (neste caso, equipamentos): Monitor, Modem, Teclado e Processador. Obviamente, as propriedades desses produtos não são as mesmas. É desejável criar no modelo um único conceito para representar Produto, sem deixar de modelar as propriedades

particulares de cada tipo de produto. Uma possível solução para esse problema é criar uma classe denominada Produto e diversas subclasses, uma para cada tipo de produto existente. Dessa forma, cada subclasse representaria as propriedades particulares do tipo de produto correspondente. Uma desvantagem dessa solução é que novos produtos estão sempre aparecendo e outros desaparecendo, o que levaria a uma frequente modificação na estrutura de modelagem.

Para complicar ainda mais, podem surgir situações em que as propriedades de um mesmo item podem variar com o tempo (ou seja, novas propriedades podem ser adicionadas, ou as já existentes podem ser removidas). Se a solução descrita no parágrafo anterior fosse utilizada, haveria várias mudanças no esquema do banco de dados, o que muito provavelmente acarretaria alterações nas aplicações que utilizassem esse esquema.

O padrão Metamodel permite a “modificação” da estrutura de um modelo de objetos sem que o esquema desses objetos seja realmente modificado. O que acontece é que o esquema de classes representa um *metamodelo*. Esse padrão é apresentado na [Figura 5-43](#).

Vamos agora descrever o funcionamento do padrão Metamodel. Para isso, considere os equipamentos de computador citados no exemplo anterior. Considere ainda que *dimensões*, *velocidade de transferência*, *quantidade de teclas* e *velocidade* são atributos de Monitor, Modem, Teclado e Processador, respectivamente. Através do padrão Metamodel, esses atributos são representados como instâncias da classe Propriedade, cada item é representado como uma instância da classe Item. Finalmente, a classe valor é utilizada para representar o valor de uma propriedade de determinado item.

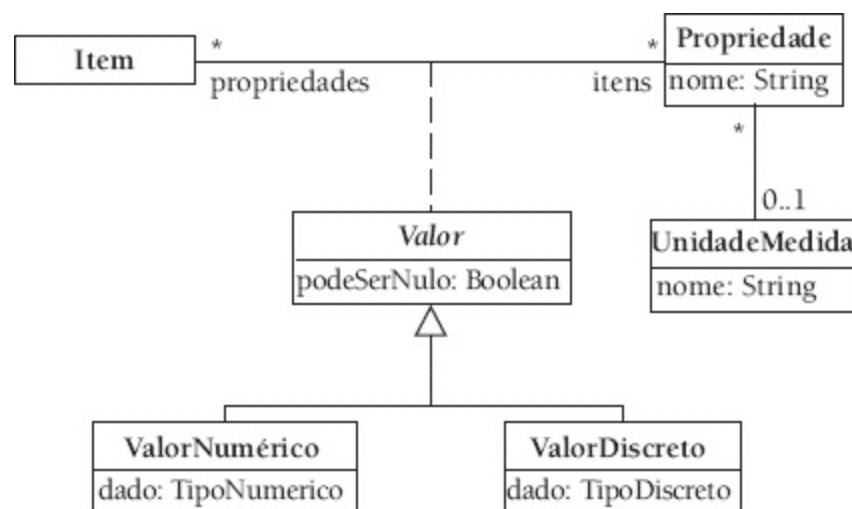


Figura 5-43: O padrão Metamodel.

Note que propriedades podem ser adicionadas (ou removidas) do metamodelo cuja base está no padrão Metamodel. Por exemplo, para adicionar o atributo formato ao item Monitor (para indicar se o monitor possui tela plana ou não), basta adicionar uma nova linha à tabela correspondente ao mapeamento da classe Propriedade. Observe também que é responsabilidade da aplicação apresentar os dados como se eles fossem atributos da instância de Item. Pode haver inclusive uma aplicação que permita que seus usuários adicionem (ou removam) atributos.

5.4.5 Outras técnicas de identificação

Além das técnicas de identificação descritas nas seções anteriores, diversas outras abordagens podem ser aplicadas com o objetivo de identificar as classes de um SSOO. Um exemplo é a

utilização de uma taxonomia de conceitos:

- *Conceitos concretos*, como edifícios, carros, salas de aula etc.
- *Papéis* desempenhados por seres humanos, como professores, alunos, empregados, clientes etc.
- *Eventos*, ou seja, ocorrências em uma data e em uma hora particulares, como reuniões, pedidos, aterrissagens, aulas etc.
- *Lugares*, ou seja, áreas reservadas para pessoas ou coisas, como escritórios, filiais, locais de pouso, salas de aula etc.
- *Organizações*, ou seja, coleções de pessoas ou de recursos, como departamentos, projetos, campanhas, turmas etc.
- *Conceitos abstratos*, isto é, princípios ou ideias não tangíveis, como reservas, vendas, inscrições etc.

Além de estudar taxonomias de conceitos, é uma boa abordagem analisar também as funcionalidades e a documentação de sistemas similares no mesmo domínio do problema. O estudo do vocabulário dos especialistas do domínio é igualmente uma boa fonte para identificação de classes.

5.4.6 Discussão

Nas seções anteriores, descrevemos diversas técnicas para identificação de classes. No entanto, uma questão que aflige diversos modeladores (tanto iniciantes quanto experientes) é a seguinte: é possível aplicar as técnicas de modelagem de dados para identificação de classes? Talvez a principal fonte para esta dúvida seja a semelhança de notação que existe entre o diagrama de entidade e relacionamentos (conhecida ferramenta de modelagem de dados) e o diagrama de classes. Alistair Cockburn (1996) descreve uma experiência que envolveu dois grupos de profissionais, um de modelagem de dados e o outro de modelagem orientada a objetos. Nessa experiência, foi requisitado aos dois grupos que elaborassem um modelo de domínio para uma dada situação. Nessa experiência, Cockburn constatou que os dois grupos produziram modelos similares *sob o aspecto estrutural*. No entanto, quando o *aspecto funcional* foi comparado (os profissionais de modelagem de dados utilizaram o DFD na experiência), os resultados foram bastante diferentes.

De fato, a questão levantada no parágrafo anterior é das mais complexas, e para ela há respostas conflitantes na literatura. Posto isso, o que exponho nas próximas linhas é minha opinião sobre o assunto. A diferença mais significante entre um modelo de dados e um modelo de objetos é o fato de este último representar o *comportamento* dos objetos, além de seus dados.

Em minha experiência prática em modelagem orientada a objetos, constato que as entidades que seriam identificadas com a modelagem de dados frequentemente são identificadas (sob a forma de classes) com a modelagem orientada a objetos. Isso porque ambas as abordagens têm princípios semelhantes em relação ao que consideram um bom modelo do ponto de vista estrutural. Mas outra constatação minha é que objetos do domínio que têm um viés mais comportamental dificilmente são identificados com a técnica de modelagem conceitual de dados. Isso se justifica pelo simples fato de essa técnica priorizar o aspecto dos dados relacionados a uma dada entidade do mundo real. Além disso, um modelo conceitual de dados representa a estrutura estática, enquanto modelos de classes representam uma fotografia (estática) das estruturas (relacionamentos) que são criadas durante o

comportamento dinâmico do sistema. Além disso, diferentemente do modelo conceitual de dados, algumas entidades representadas em um modelo de classes são transientes (ou seja, não “vivem” além de uma sessão do sistema, armazenadas em um SGBD, por exemplo).

Não se trata de afirmar que a modelagem conceitual de dados seja uma técnica ruim; o fato é que ela deixa para outras técnicas (p. ex., a decomposição funcional com a ferramenta de Diagrama de Fluxos de Dados) a responsabilidade de considerar o aspecto funcional do sistema que está sendo modelado. Em minha opinião, não há nada de errado em aplicar as técnicas de modelagem conceitual de dados para construir o aspecto estrutural do modelo de classes. No entanto, não se pode pensar que chegamos ao final da tarefa quando tivermos feito isso. É importante considerar também o aspecto comportamental desse modelo. E isso deve ser feito com a aplicação de técnicas de identificação que levem em conta esse aspecto estrutural, algumas das quais foram descritas nesta seção (análise textual de Abbott, análise de casos de uso e identificação dirigida a responsabilidades).

Modelos de classes são perigosamente similares a modelos de dados. A maioria dos princípios que resultam em um bom modelo de dados também resultam em um bom modelo de classes. O perigo é justamente usar apenas técnicas de modelagem de dados para desenvolver um modelo de classes orientado a dados, e não a responsabilidades.

Outro ponto importante: é recomendável que o modelador aplique duas ou mais técnicas de identificação durante a identificação de classes. O objetivo disso é fazer com que a aplicação de uma técnica valide a aplicação de outra técnica. A utilização de mais de uma técnica também permite descobrir classes que seriam ignoradas se apenas uma técnica fosse aplicada. Além disso, é fundamental a participação de um ou mais especialistas do domínio durante essa atividade. É muito pouco provável que um analista tenha conhecimento equiparável a uma especialista no domínio de conhecimento deste último.

Uma primeira questão que podemos analisar é como conciliar as técnicas de identificação dirigida a responsabilidades com a análise textual de Abbott. Ou seja, será que essas duas técnicas podem ser aplicadas em conjunto para identificar as classes de análise? No meu entender, a resposta é positiva. Isso pode ser feito em duas etapas. Na primeira, identificamos as classes mais óbvias através da ATA (podemos utilizar o documento de especificação de requisitos, descrições de casos de uso, manuais, enfim, quaisquer fontes de informação textual). Note que na técnica ATA, os verbos que indicam ação podem ser interpretados como responsabilidades em potencial. As classes assim identificadas servem de base para a segunda etapa, quando ocorre a aplicação da identificação dirigida a responsabilidades. No início de uma sessão CRC, cada indivíduo participante deve ser responsável por ao menos uma classe identificada pela aplicação da ATA. São duas as vantagens dessa abordagem. Em primeiro lugar, as classes identificadas pela ATA são validadas pela aplicação da técnica de modelagem baseada em responsabilidades. Em segundo lugar, a sessão CRC começa com um conjunto de classes (e possivelmente responsabilidades) identificadas.

O mesmo raciocínio que empregamos no parágrafo anterior para argumentar sobre a possibilidade de aplicação da ATA juntamente com a modelagem CRC pode ser utilizado em relação a esta última e à análise de casos de uso. Em particular, as responsabilidades de um sistema estão codificadas nos casos de uso identificados para o mesmo. Portanto, outra estratégia válida para identificação de classes é aplicar a análise de casos de uso e, subsequentemente, validar os resultados com a

aplicação da modelagem baseada em responsabilidades.

Ainda quanto à conciliação de técnicas de identificação, até aqui apresentamos situações em que as técnicas baseadas em responsabilidades são aplicadas como forma de validação de outra técnica. Isso não quer dizer que essas técnicas sejam menos importantes que as outras. Na verdade, é justo comentar que há domínios em que a modelagem baseada em responsabilidades é mais adequada e aplicável que as outras. Por exemplo, na identificação de objetos para um jogo computacional, para um sistema de automação, ou para um sistema de tempo real, as técnicas baseadas em responsabilidades apresentam uma forma bastante mais natural de capturar todas as responsabilidades do sistema na forma de objetos.

É importante notar também que as técnicas de identificação de classes descritas na [Seção 5.3](#) normalmente são aplicadas de um modo *iterativo*. Iterativo porque é virtualmente impossível para um modelador construir um modelo perfeito (no sentido de contemplar todos os aspectos relevantes) da primeira vez. Em vez disso, o modelo de classes adequado para um SSOO é construído por refinamentos sucessivos. Cada refinamento aperfeiçoa a versão anterior do modelo.

Por último, é relevante fazer um comentário a respeito da diferença conceitual que há entre as classes que definimos durante a modelagem de classes do domínio e durante a modelagem de classes da aplicação. Em um SSOO, é razoável categorizar seus objetos em dois tipos: *objetos do domínio* e *objetos da aplicação*. Com relação à identificação de classes, quando estamos realizando a modelagem de classe de domínio, normalmente *descobrimos* classes, no sentido de que as classes que definimos são abstrações de entidades reais existentes no domínio. Por outro lado, quando estamos na modelagem de classes da aplicação, a tendência é *inventarmos* classes. Isso porque as classes que definimos nessa atividade *não* têm correspondente no domínio do problema (esse é normalmente o caso das classes de controle e de fronteira).

5.5 Construção do modelo de classes

Após a identificação de classes e atribuição de responsabilidades, o modelador deve verificar a consistência entre as classes para eliminar incoerências e redundâncias. Como dica, o modelador deve estar apto a declarar as razões de existência de cada classe identificada e, para cada classe, argumentar sobre a existência de cada uma de suas responsabilidades e de cada um de seus colaboradores. Do contrário, o modelador deve reconsiderar se a classe é verdadeiramente necessária.

Em seguida, os analistas devem começar a definir o mapeamento das responsabilidades e dos colaboradores de cada classe para os elementos do diagrama de classes. Esse mapeamento gera um diagrama de classes que apresenta uma estrutura estática relativa a todas as classes que foram identificadas anteriormente como participantes da realização de um ou mais casos de uso. A seguir, descrevemos o mapeamento de responsabilidades e colaboradores, levando em consideração os diversos elementos do diagrama de classes apresentados na [Seção 5.2](#).

5.5.1 Definição de propriedades

Uma propriedade é um nome genérico que se aplica aos membros de uma classe, tanto atributos quanto operações. Uma questão importante é como podemos mapear (transformar) responsabilidades atribuídas a uma classe para propriedades da mesma.

Com relação às operações, elas correspondem a um modo mais detalhado de explicitar as

responsabilidades *de fazer* de uma classe. Uma operação também pode ser vista como uma pequena contribuição da classe para uma tarefa mais complexa representada por um caso de uso. Algumas operações de uma classe podem ser identificadas facilmente. No entanto, uma definição mais detalhada e completa das operações de uma classe só pode ser feita após a construção do *modelo de interação*. (Esse modelo representa detalhes sobre a ordem das *colaborações* entre os objetos que participam da realização de um caso de uso. A descrição do modelo de interação e sua construção são apresentadas no [Capítulo 7](#).) Portanto, inicialmente o adequado é que o modelador se atenha a realizar a identificação e a descrição precisas das responsabilidades de cada classe, sem dispensar muito tempo com detalhes sobre a transformação delas em uma ou mais operações ou atributos; esses detalhes são definidos na modelagem de interações. Dito isso, no restante desta seção há o mapeamento de responsabilidades para atributos; enquanto o aspecto do detalhamento de operações está nos capítulos seguintes.

Normalmente uma responsabilidade *de conhecer* é mapeada para atributos ou para associações. Essa tarefa de mapeamento é relativamente fácil, uma vez que estejam corretamente identificadas as responsabilidades de fazer e os colaboradores da classe (ver [Seção 5.4.3](#)). Após mapear responsabilidades para atributos, o modelador deve verificar se cada atributo aplica-se a *todos* os objetos da classe; cada atributo de uma classe deve fazer sentido para todo e qualquer objeto dessa classe. Por exemplo, é incoerente ter o atributo salário em uma classe Pessoa (pois nem toda pessoa tem um salário).

É comum ao iniciante em modelagem de classes mapear uma responsabilidade *de conhecer* como atributos, quando uma melhor modelagem seria mapeá-la como uma associação. Tome por exemplo a responsabilidade “conhecer o seu cliente” atribuída à classe Pedido. Considere também o mapeamento dessa responsabilidade para o atributo nomeCliente na classe Pedido. Nesse caso, é mais adequado criar uma associação entre Pedido e Cliente, sendo que esta última possui o atributo nome. De forma geral, se duas classes estão associadas, não há necessidade de criar um atributo em uma classe para fazer referência a informações da outra.⁸ A [Figura 5-44](#) ilustra um exemplo dessa confusão entre atributos e associações.

Outro erro comum para iniciantes em modelagem orientada a objetos é a utilização de *atributos de identificação*; por exemplo, em uma classe Cliente, definir idCliente como um atributo. Embora haja a necessidade de identificar unicamente cada objeto de uma classe, o modelo de classes de domínio não é o lugar para definir esse tipo de atributo. Nesse modelo, somente informações que existem no domínio do negócio devem ser definidas, e as características de implementação abstraídas. Isso não quer dizer que atributos que correspondem a identificadores naturais e que existem no domínio do negócio não devam ser definidos. Exemplos desses atributos são CPF, código (de um produto), número-Matrícula (de um aluno) etc.

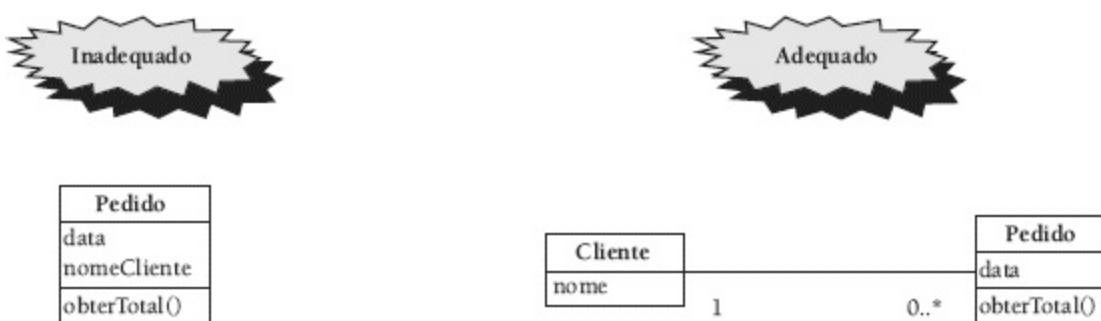


Figura 5-44: Confusão entre atributos e associações.

Ainda sobre atributos, é preciso notar também que a sua definição para uma classe é dependente do contexto. Uma mesma classe, por exemplo, Pessoa, pode ter atributos diferentes em sistemas diferentes. Para um sistema que mantém informações sobre pessoas suspeitas de terem cometido crimes, os atributos dessa classe poderiam ser: fotografia, altura, peso, alibi, registro policial etc. Já em um sistema de recursos humanos que mantém informações sobre os empregados de uma companhia, os atributos poderiam ser: matrícula, salário, data de contratação etc.

5.5.2 Definição de associações

N a [Seção 5.4.3](#), vimos que uma responsabilidade de conhecer pode ser mapeada para uma associação. Por outro lado, o fato de uma classe possuir colaboradores indica que devem existir relacionamentos entre estes últimos e a primeira. Isso porque um objeto precisa conhecer o outro para poder lhe fazer requisições. Portanto, para criar associações, verifique os colaboradores de uma classe, conforme identificados pela técnica de identificação dirigida a responsabilidades. O raciocínio para definir associações reflexivas, ternárias e agregações é o mesmo, com a ressalva de que, em uma agregação, a classe em questão é o todo e o seu colaborador corresponde à parte.

Classes de associação (ver [Seção 5.2.2.4](#)) surgem a partir de responsabilidades de saber (e, mais raramente, *de fazer*) que o modelador não conseguiu atribuir a alguma classe exclusivamente. Nessas situações, a solução é criar uma classe associativa para que ela cumpra tal responsabilidade. Por exemplo, considere a situação em que pessoas trabalham em projetos, formando uma associação de conectividade “muitos para muitos”. Se houver a necessidade de conhecer quantas horas semanais cada pessoa trabalha em cada um dos projetos, essa responsabilidade não pode ser atribuída à classe Pessoa exclusivamente, pois isso violaria as propriedades 2 e 3 de um atributo, definidas na [Seção 5.5.1](#) (é possível haver pessoas que não estejam alocadas em projetos; além disso, uma pessoa pode ter diversas cargas horárias diferentes, uma para cada projeto no qual trabalhe). Um raciocínio análogo se aplica à atribuição da responsabilidade em questão à classe Projeto, que também não é adequada para cumprir com a mesma. Portanto, há uma responsabilidade *de conhecer* que não se aplica exclusivamente a nenhuma das classes envolvidas. A solução é criar uma classe associativa para realizar essa responsabilidade. A [Figura 5-45](#) ilustra essa situação.



Figura 5-45: Definindo classes associativas para cumprir responsabilidades órfãs.

5.5.3 Organização da documentação

Uma vez que classes e suas responsabilidades e colaboradores tenham sido identificados, esses elementos devem ser organizados em um ou mais diagramas de classes e documentados. O conjunto de todos os diagramas de classes identificadas na análise de um sistema, juntamente com sua documentação (e eventuais diagramas de objetos; ver [Seção 5.3](#)), corresponde ao *modelo de classes de análise*.

Na diagramação, sempre que possível, as classes devem ser posicionadas de tal forma que as

associações sejam lidas da esquerda para a direita ou de baixo para cima. Isso facilita a leitura, principalmente se o diagrama tiver de ser entendido por pessoas de cultura ocidental (onde se costuma ler da esquerda para a direita e de cima para baixo).

Conforme podemos concluir pelo conteúdo deste Capítulo, algumas responsabilidades mais simples de uma classe podem ser modeladas por meio de elementos gráficos do diagrama de classes, como atributos, associações, restrições etc. Por outro lado, durante a análise, pode ser que a documentação de determinadas responsabilidades seja mais adequada ou mesmo possível se feita de forma textual. Algumas ferramentas CASE ([Seção 2.6](#)) fornecem esse artifício, como é o caso da *Enterprise Architect™*. Nessa ferramenta, o modelador pode adicionar um elemento gráfico que representa uma classe com um compartimento extra (além dos já existentes para definir nome da classe, atributos e operações) para definir dentro dele as responsabilidades da classe de forma textual. A Figura 5.51 apresenta um exemplo desse tipo de notação, bastante útil, mas que, por não ser um padrão, é encontrado em poucas ferramentas CASE. Na falta de uma ferramenta CASE que dê suporte à definição de responsabilidades no próprio diagrama de classe, uma alternativa é registrar essas responsabilidades em um documento textual mesmo, complementar ao(s) diagrama(s) de classes.

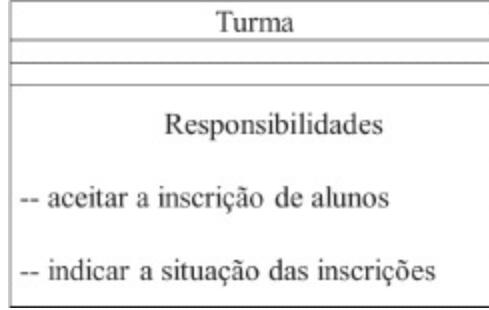


Figura 5.46: Classe com compartimento de responsabilidades.

A construção de um único diagrama para representar todas as classes do sistema pode ter um resultado bastante complexo e de difícil interpretação. Uma alternativa é criar uma *visão de classes participantes* (VCP) para cada caso de uso. Uma VCP é um diagrama de classes cujas instâncias (objetos) participam da realização de um caso de uso. Se essa estratégia for adotada, a elaboração de cada caso de uso resulta em uma VCP. Em uma VCP, além das classes conceituais (p. ex., das classes de entidade), podem ser apresentadas também as classes de fronteira e de controle do caso de uso correspondente (veja a [Seção 5.4.2.1](#)).

As classes conceituais identificadas em cada caso de uso podem ser reunidas para formar um único diagrama de classes, denominado **diagrama de classes conceituais**. Se esse diagrama ficar muito grande, o modelador pode optar por criar um diagrama de classes por subdomínio. Se isso fizer sentido, o recurso de modularização representado pelo mecanismo de *pacotes* (ver [Seção 3.5](#)) é também útil.

Além do diagrama, as classes devem ser documentadas textualmente. Uma definição precisa deve ser criada para cada classe, assim como para cada um de seus atributos cujo significado não seja fácil de inferir pelo nome. À medida que o desenvolvimento evolui, esse *glossário* resultante se torna útil para eliminar eventuais confusões acerca do significado de cada termo encontrado no sistema (ver [Seção 5.7](#) para um exemplo de glossário no contexto do SCA).

Os nomes escolhidos para as classes devem estar no singular (pois uma classe já representa vários objetos), iniciando com letra maiúscula. As classes devem preferencialmente receber nomes

provenientes do domínio, ou inspirados nele. Também é importante que as responsabilidades atribuídas a cada classe (que resultam em atributos e operações) recebam nomes provenientes ou inspirados no domínio.

Por fim, é também útil adicionar uma seção de detalhes de implementação ao modelo de classes, assim como para o modelo de casos de uso (ver [Seção 4.4.3.14](#)). Embora os detalhes de implementação não façam parte nem devam ser considerados no modelo de classes de análise, esses detalhes inevitavelmente surgem na mente dos analistas enquanto estão construindo esse modelo. Esta seção de implementação serve como repositório de ideias e soluções relativas ao modelo de classes que podem ser utilizadas (ou mesmo descartadas) nas atividades futuras do desenvolvimento.

5.6 Modelo de classes no processo de desenvolvimento

Em um desenvolvimento *dirigido a casos de uso* (ver [Seção 4.6](#)), após a descrição dos casos essenciais, é possível realizar a identificação de classes. Nesse ponto, uma ou mais das técnicas de identificação de classes (descritas na [Seção 5.4](#)) podem ser aplicadas. Durante a aplicação dessas técnicas, as classes identificadas são refinadas para retirar inconsistências e redundâncias. Finalmente, as classes são documentadas, e o diagrama de classes inicial é construído, resultando no modelo de classes de análise.

Depois que a primeira versão do modelo de classes de análise está completa, o modelador deve retornar ao modelo de casos de uso e verificar a consistência entre os dois modelos. Uma inconsistência indica a existência de algum erro no modelo de casos de uso ou no modelo de classes. Um caso de uso em que não foram identificados objetos ou uma classe que não participa da realização de algum caso de uso são exemplos de inconsistências que devem ser eliminadas.

Pelo que foi exposto anteriormente, pode-se notar que as construções do modelo de casos de uso e do modelo de classes são *retroativas* entre si. De fato, na realização de uma sessão de identificação das classes de um sistema, novos casos de uso podem ser identificados, ou pode-se identificar a necessidade de modificação de casos de uso preexistentes. A [Figura 5-47](#) ilustra a interdependência entre a construção do modelo de casos de uso e o modelo de classes. De fato, esta é mais uma consequência do modo de pensar orientado a objetos: detalhes são adicionados aos poucos, à medida que o problema é entendido.

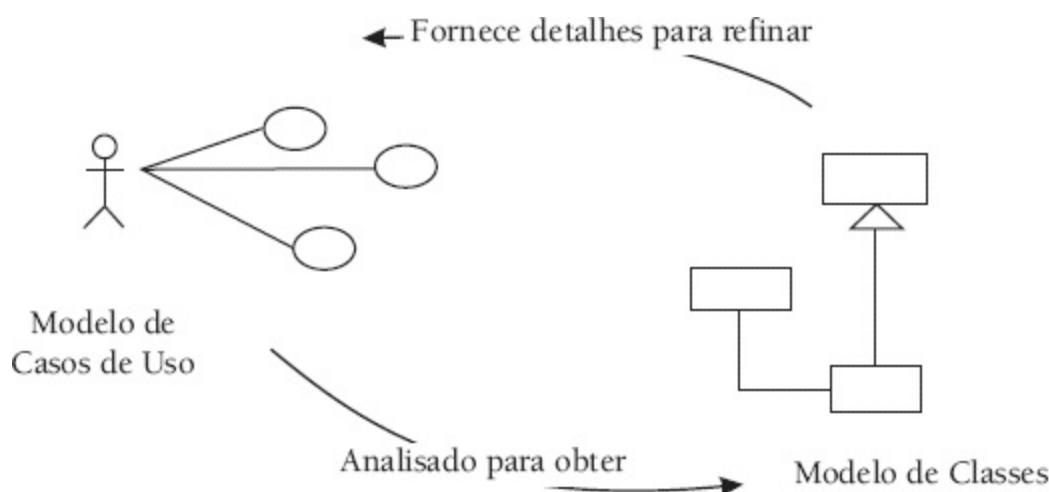


Figura 5-47: Interdependência entre o modelo de casos de uso e o modelo de classes.

Na [Seção 2.3.2](#), descrevemos o ciclo de vida de desenvolvimento iterativo. Nesse ciclo de vida,

os artefatos de software são construídos aos poucos, iteração após iteração. Além dessa característica de construção incremental, os diversos modelos do SSOO, quando construídos, fornecem informações úteis para refinar os modelos preexistentes. Em relação ao modelo de classes de análise, nada acontece de forma diferente. Na construção deste, a ênfase inicial recai em identificar os objetos do domínio. Após isso, alguns relacionamentos, atributos e mesmo operações desses objetos podem ser identificados nesta fase. Em seguida, alguns objetos da aplicação (particularmente fronteiras e controladores; veja a [Seção 5.4.2](#)) são também identificados na análise. Entretanto qualquer elemento identificado na modelagem de classes de análise está sujeito a modificações e refinamentos, em função das informações obtidas com a *modelagem dinâmica*. Primeiramente, a modelagem dinâmica fornece detalhes adicionais que podem ser utilizados para refinar os elementos já identificados do modelo de classes. Em segundo lugar, pode ser que alguma classe ou responsabilidade importante para o sistema só venha a ser descoberta quando o aspecto dinâmico do sistema for considerado.

O aspecto dinâmico de um SSOO é representado pelo *modelo de interações* e pelo *modelo de estados*. A construção desses modelos é possível por meio de informações obtidas com a construção do modelo de classes. Por outro lado, com a construção desses modelos, o modelador obtém informações suficientes para retornar ao modelo de classes e refiná-lo.

Assim como a modelagem estática (modelagem de classes), a dinâmica também começa na fase de análise, sendo representada pelo *modelo de interações* e pelo *modelo de estados*. O modelo de interações é objeto de estudo do [Capítulo 7](#). A descrição do modelo de estados está no [Capítulo 10](#).

Para finalizar esta seção, deixo um comentário de viés um tanto filosófico. Podemos perceber que não só um SSOO é composto de entidades que colaboram entre si. A própria modelagem orientada a objetos é também constituída de entidades (modelos), cada uma “colaborando” com informações para a construção da outra. Sendo assim, o aspecto dinâmico depende do aspecto estrutural estático, e vice-versa. Nenhum é mais importante que o outro; um serve para completar o outro.

5.7 Estudo de caso

A identificação das classes conceituais envolvidas na realização de certo caso de uso se dá mediante a aplicação de uma ou mais das técnicas descritas na [Seção 5.4](#). Por exemplo, quando utilizamos a técnica de análise de casos de uso (ver [Seção 5.4.2](#)), estudamos a descrição de cada passo do fluxo principal do caso de uso em que o sistema realiza alguma ação. Cada um desses passos implica classes e responsabilidades dentro do sistema. Durante a identificação de classes a partir do texto de um caso de uso, os fluxos alternativos e de exceção devem também ser analisados.

Esta seção continua o desenvolvimento da modelagem do estudo de caso iniciado na [Seção 4.7](#). Aqui, apresentamos diversas versões sucessivas do modelo de classes de análise inicial desse estudo de caso. Conforme descrito neste capítulo, o processo de construção do modelo de classes pode ser realizado por meio da técnica denominada análise dos casos de uso. Para cada um deles, os modeladores identificam quais classes são necessárias para que os resultados externamente visíveis sejam obtidos. Nessa seção, descrevemos a aplicação desta e de outras técnicas para identificar classes de entidade nos três seguintes casos de uso do SCA:

- Fornecer Grade de Disponibilidades,
- Realizar Inscrição,
- Lançar Avaliações.

Por questões de espaço, os demais casos de uso do SCA não são considerados aqui. Como exercício, o leitor é convidado a identificar as classes participantes dos demais casos de uso descritos na [Seção 4.7.3](#). Para acompanhar de forma adequada a descrição apresentada a seguir, recomendamos que ao leitor a leitura das descrições dos casos de uso acima, que são apresentadas na [Seção 4.7.3](#).

Repare também que os diagramas de classes aqui apresentados não mostram operações das classes. Embora o mapeamento de determinadas operações das classes já possa ser feito neste momento, essa tarefa é adiada para os capítulos seguintes, quando descrevemos o modelo de interações ([Capítulo 7](#)) e o modelo de classes de projeto ([Capítulo 8](#)). Isso porque é graças à construção do modelo de interações de um sistema que identificamos as operações que uma classe deve possuir.

5.7.1 Análise do caso de uso Fornecer Grade de Disponibilidades

Comecemos a análise pelo caso de uso Fornecer Grade de Disponibilidades. O diagrama de classes resultante da análise desse caso de uso é apresentado na [Figura 5.48](#). O restante desta seção descreve o processo de raciocínio que resultou neste diagrama.

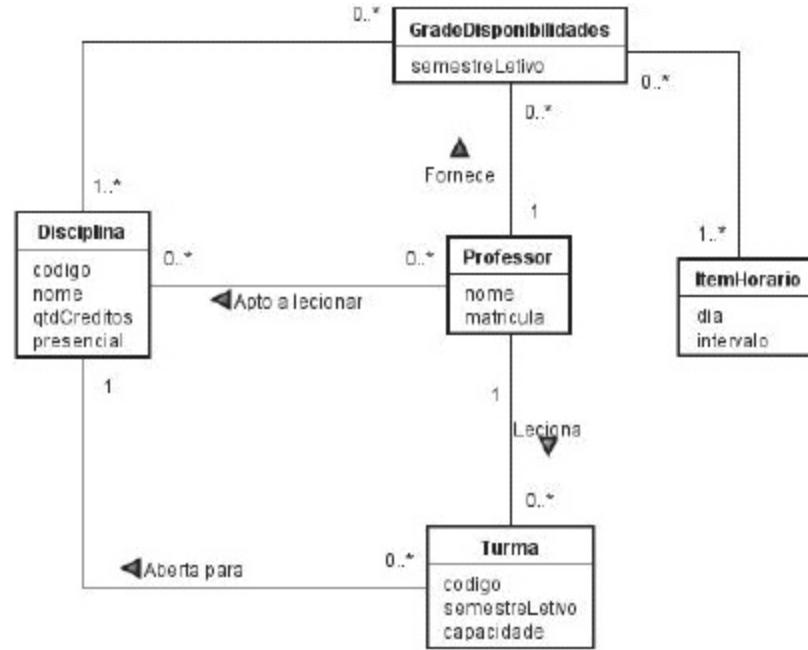


Figura 5-48: Diagrama de classes do SCA (1^a versão).

No primeiro passo do fluxo principal desse caso de uso, lemos: “Professor fornece sua matrícula para validação”. Essa frase suscita a decisão de criar uma primeira classe, Professor, e definir nela um atributo para representar a matrícula de um professor.

O próximo passo do caso de uso é: “Sistema apresenta a lista de disciplinas disponíveis (conforme RN04), e a lista de dias da semana e de horários do semestre letivo”. Esse passo leva o modelador a concluir que é necessária uma classe para representar disciplinas. Além disso, é possível concluir também que o sistema deve ter conhecimento acerca dos horários em que pode haver alocação de uma turma de alguma disciplina. Como consequência, é criada uma classe Disciplina e outra denominada ItemHorario. Outra conclusão é que esse mesmo sistema também deve ter informações acerca de que disciplinas um professor está apto a lecionar, o que resulta na criação de uma associação entre Professor e Disciplina. A semântica dessa associação corresponde às habilitações

de um professor, i.e., quais disciplinas da grade curricular ele está apto a lecionar. Repare também que essa associação é uma forma de refletir no modelo a regra de negócio cujo identificador é RN04 (veja a [Seção 4.7.2](#)).

Quando passamos para analisar o terceiro passo do caso de uso, encontramos: “Professor informa (1) cada disciplina que deseja lecionar e (2) cada disponibilidade para o próximo semestre letivo”. Ao ler esse passo, uma interpretação razoável é que o sistema deve registrar duas listas de informações associadas ao professor. A primeira lista corresponde às disciplinas que o professor selecionou. A segunda lista corresponde aos dias da semana (e respectivos horários de início e término) em que este mesmo professor está disponível para ser alocado em turmas de disciplinas componentes da primeira lista. O modelador então decide unificar essas duas listas como a definição de um novo conceito denominado grade de disponibilidades. Como resultado, a classe GradeDisponibilidades é criada, assim como são criadas duas novas associações (para representar as duas listas de informações previamente mencionadas), uma com Disciplina e outra com ItemHorario. Finalmente, uma terceira associação é criada, entre Professor e GradeDisponibilidades.

5.7.2 Análise do caso de uso Realizar Inscrição

Agora o caso de uso que usamos como exemplo é o Realizar Inscrição. A descrição a seguir pode ser acompanhada juntamente com a visualização do diagrama apresentado na [Figura 5-49](#), que corresponde a uma extensão do diagrama apresentado na [Figura 5-48](#), considerando também as classes relevantes para o caso de uso Realizar Inscrição.

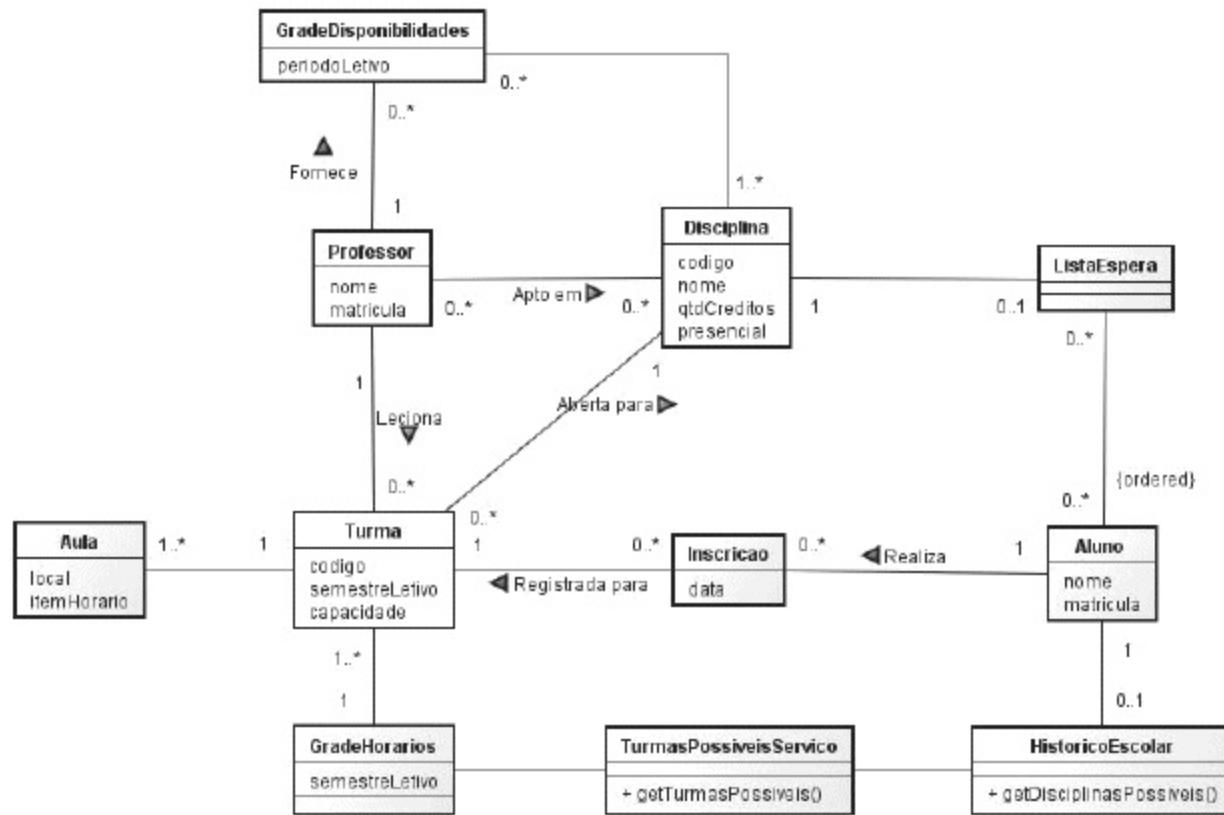


Figura 5-49: Diagrama de classes do SCA (2^a versão).

Os passos 1, 2 e 3 desse caso de uso sugerem que deve haver alguma forma de o sistema registrar cada solicitação de matrícula realizada por um aluno. Como resultado, a classe Turma foi criada. Essa classe é a contrapartida temporal de uma disciplina: toda vez que uma disciplina deve ser ofertada

em um semestre letivo, uma nova turma é criada. Foram também criadas a classe Inscrição e as associações desta com as classes Aluno e Turma. Repare ainda que o passo 2 faz menção à regra de negócio RN03 (veja [Seção 4.7.2](#)). Essa restrição é contemplada no modelo por meio da adição de uma autoassociação na classe Disciplina.

Ainda sobre o passo 3, a análise de sua descrição nos permite concluir que é responsabilidade do sistema saber cada turma (e respectiva disciplina) que pode ser apresentada a um determinado aluno para inscrição. De fato, um aluno não pode ser inscrever em alguma turma de uma disciplina para a qual não tenha cursados todos os pré-requisitos. Além disso, não deve ser apresentada a um aluno alguma turma cuja disciplina ele já cursou com aprovação. Portanto, a obtenção da lista de turmas possíveis necessária no passo 3 é uma tarefa não trivial. Isso nos motiva a definir uma classe denominada HistoricoEscolar para assumir a responsabilidade de gerar a *lista de disciplinas* em cujas turmas um aluno pode se inscrever. Essa classe está associada a Aluno. Uma vez de posse dessa lista, o próximo passo é determinar que turmas (no período letivo corrente) estão abertas para cada uma dessas disciplinas. Visto que essa responsabilidade envolve a manipulação de várias turmas, decidimos atribuí-la a uma nova classe GradeHorarios e criamos uma associação dessa última com Turma. Portanto, repare que a obtenção da lista de turmas possíveis para um aluno é um procedimento de dois passos. Primeiro deve-se definir a lista de disciplinas e, depois disso, definir uma lista de turmas. Para orquestrar a realização desses dois passos, decidimos criar um *Serviço de Domínio* (veja a [Seção 5.4.3.2](#)) denominado TurmasPossiveisServico. Neste ponto, está claro que um objeto dessa classe irá interagir com objeto da classe HistoricoEscolar e com outro da classe GradeHorarios, mas não de que forma ocorrerá essa interação. Portanto, definição completa dessa parte do modelo será postergada para quando iniciar a atividade de modelagem de interações (veja o [Capítulo 7](#)).

O passo 4 desse caso de uso sugere que o sistema deve manter informação acerca do professor, assim dos horários e respectivos locais de aula de cada turma. Por conta disso, as classes Aula e Local são adicionadas ao modelo e associações são criadas para contemplar essa necessidade de informação. A classe ItemHorario, que surgiu durante a análise do caso de uso Fornecer Grade de Disponibilidades, é reusada aqui com o propósito de registrar, para cada aula, os horários em que ela ocorre.

O procedimento para identificação de classes apresentado até agora apenas analisou o fluxo principal do caso de uso em questão. Entretanto, todos os fluxos do caso de uso também devem ser analisados na busca por classes necessárias. Por exemplo, por meio da análise do fluxo alternativo Inclusão em lista de espera (ver [Seção 4.7.3](#)), identificamos a necessidade da classe ListaEspera, responsável por manter uma fila de alunos que estão esperando pela abertura de mais uma turma para uma determinada disciplina. Note que essa classe nem mesmo é mencionada no fluxo principal do caso de uso, o que enfatiza a necessidade de avaliarmos todos os fluxos de um caso de uso durante a identificação. A [Figura 5-49](#) também apresenta a classe ListaEspera, associada a duas classes que já haviam sido identificadas, Aluno e Disciplina. Dessa forma, uma lista de espera é criada para registrar a espera de alunos pela abertura de uma turma para a disciplina associada. Ao mesmo tempo, uma lista de espera está associada aos alunos que aguardam a abertura dessa turma. Um detalhe relevante nessa modelagem é o uso da restrição {ordered} na associação entre ListaEspera e Aluno, no extremo próximo a esta última classe. Essa restrição, predefinida na UML, indica que a ordem dos objetos na coleção de alunos associada a uma lista de espera deve ser mantida pelo sistema.

Um aspecto importante a notar ainda sobre a análise do fluxo alternativo Inclusão em lista de espera é que as classes e associações aqui identificadas já são as necessárias e suficientes para atender ao caso de uso Atender Lista de Espera. De fato, a análise desse último caso de uso não resultaria na

identificação de nenhuma classe nova em relação às já identificadas até o momento.

Conforme mencionado na [Seção 5.4.2.5](#), cada caso de uso tem, a princípio, um objeto de fronteira para cada ator e um objeto controlador. Com base nessa heurística, as primeiras classes identificadas para o cenário principal desse caso de uso foram RealizarInscriçãoControlador, RealizarInscriçãoFormulário e SistemaFaturamento. Essas duas últimas são classes de fronteira do sistema que realizam a interface com os atores Aluno e Sistema de Faturamento, respectivamente. Note que essas classes de fronteira têm naturezas completamente diferentes. Uma corresponde possivelmente a uma interface gráfica com o usuário. A outra corresponde à implementação de algum protocolo de comunicação. Entretanto, não devemos nos ater a esses detalhes durante a análise, pois eles precisam ser abordados apenas na etapa de projeto do sistema. O importante neste momento da modelagem é apenas identificar que tais classes são necessárias para prover as funcionalidades (requisitos funcionais) do sistema.

5.7.3 Análise do caso de uso Lançar Avaliações

Agora vamos analisar o caso de uso Lançar Avaliações. Ao considerarmos a descrição do fluxo principal desse caso de uso, podemos perceber que os passos de 1 até 4 não demandam a criação de novas classes. Entretanto, o passo 5 (“O Professor fornece as notas de A1 e de A2 e a quantidade de faltas solicitadas.”) sugere que o sistema precisa armazenar as avaliações (notas parciais, denominadas A1 e A2 no caso de uso) e a frequência para cada aluno inscrito em um turma. Sendo assim, o resolvemos criar a classe denominada Avaliacao para registrar essas informações. Além disso, a associação com Inscricao permite atrelar essas informações a cada aluno inscrito em uma turma.

Repare também que os limites máximos das multiplicidades dessa associação estão coerentes. Isso porque, quando o registro de uma avaliação é realizado, o objeto inscrição ao qual essa avaliação deve ser associada já existe. Por outro lado, quando um objeto Inscricao é criado, a avaliação correspondente ainda não existe. O diagrama de classes resultante é apresentado na [Figura 5-50](#), na qual a classe Avaliacao está destacada.

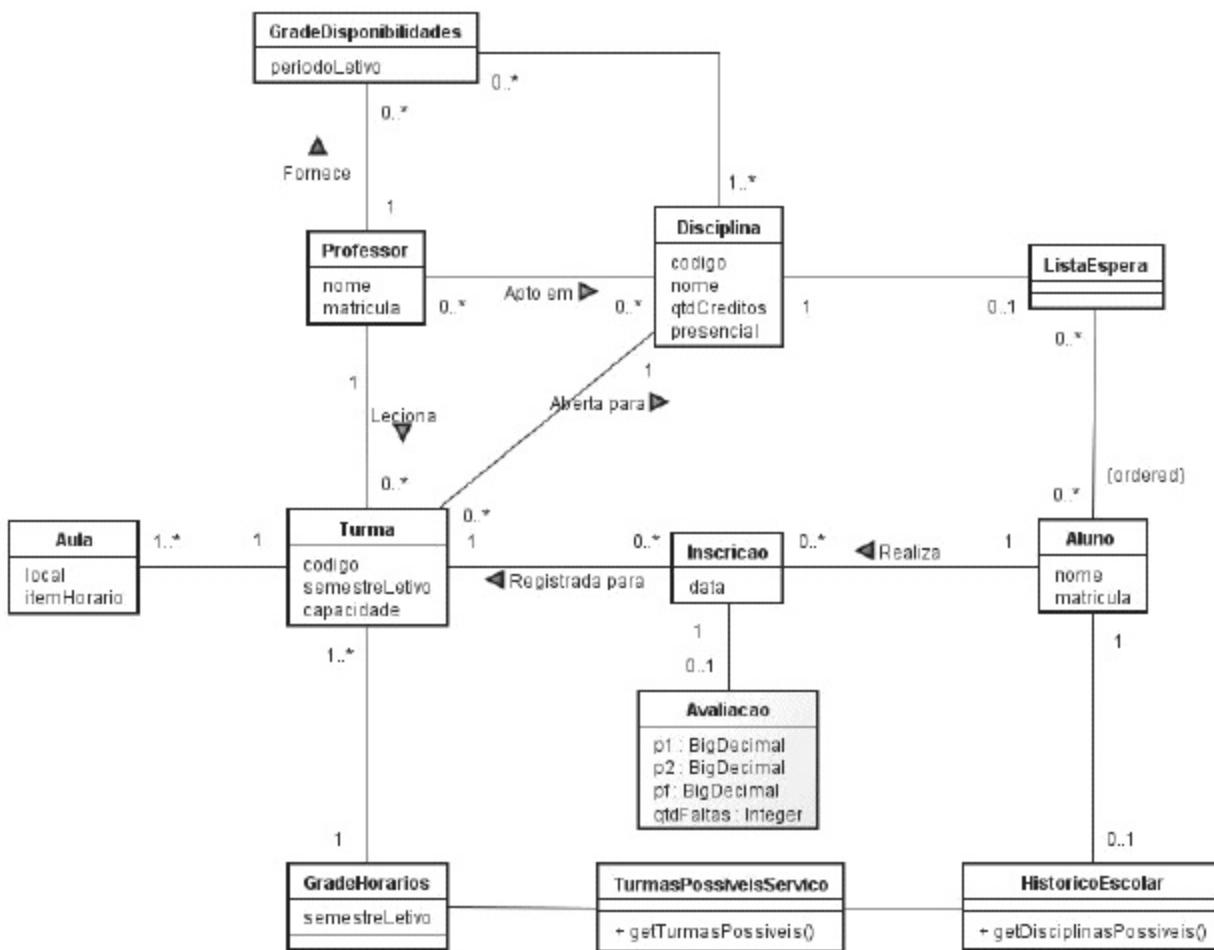


Figura 5-50: Diagrama de classes do SCA (3^a versão).

5.7.4 Análise das regras do negócio

Durante a identificação de classes, é preciso analisar também o modelo de regras de negócio (MRN, veja a [Seção 4.5.1](#)) à procura de novos elementos necessários ao modelo de classes. Em um MCU em que os casos de usos fazem referência às regras de negócio (conforme ilustrado na [Seção 4.7.3](#)), então a maior parte das informações e restrições documentadas no MRN já foram contempladas e incorporadas ao modelo de classes durante a análise dos casos de uso. Mesmo assim, é recomendável que os modeladores dêem uma repassada no MRN à procura de informações que irão gerar alterações no modelo de classes. A seguir apresentamos para cada regra de negócio do SCA (veja a [Seção 4.7.2](#)) seu efeito no modelo de classes de análise inicial.

- **RN00:** essa regra sugere que o sistema deve ser responsável por detectar se há choques de horários em uma coleção de inscrições realizadas por um aluno em um determinado semestre letivo.
- **RN01:** a estrutura do modelo de classes construído contempla essa regra. Isso porque cada objeto da classe Aluno está associado às inscrições que ele realizou, e cada inscrição está associada a uma turma que, por sua vez, está associada a uma disciplina, que conhece sua quantidade de créditos. Uma vez fixada uma coleção de inscrições para um dado semestre letivo, é possível determinar o somatório da quantidade créditos para as disciplinas correspondentes.
- **RN02:** essa regra implica em que uma turma tenha a responsabilidade de conhecer sua capacidade máxima. Um dos elementos de modelagem necessários para satisfazer essa

responsabilidade é o atributo capacidade da classe Turma, que permite à turma conhecer a capacidade máxima de inscrições que pode ter. Além disso, uma turma possui referências para suas inscrições correspondentes, por meio da associação Registrada para.

- **RN03:** essa regra se traduz em uma responsabilidade, atribuída à classe Disciplina, correspondente à necessidade dos objetos dessa classes de saber quais são as disciplinas que servem como seus pré-requisitos (caso existam). O elemento de modelagem criado aqui foi a autoassociação entre objetos da classe Disciplina.
- **RN04:** essa regra faz com que seja necessário que o sistema tenha conhecimento a respeito das disciplinas que um professor está apto a lecionar. Para isso, foi criada a associação de nome Apto a lecionar entre as classes Disciplina e Professor.
- **RN05:** nessa regra, está implícita uma responsabilidade do sistema para que este conheça a quantidade de reprovações de um aluno. Repare que, do ponto de vista estrutural, é possível saber se um aluno A foi reprovado em uma determinada inscrição para uma turma T : basta determinar o objeto inscrição correspondente ao par (A, T) e verificar a situação do objeto Avaliacao associado. Repare também que no modelo de casos de uso inicial, não foi considerado um caso de uso associado a essa funcionalidade de cancelamento de matrícula de um aluno. Temos aqui, portanto, um exemplo de situação em que a modelagem de classes gera informações para o modelador que podem ser usadas para voltar ao modelo de casos de uso e refinar este artefato. Essa situação ocorre comumente em análise de sistema, conforme já adiantado na [Seção 5.6](#) (veja também a [Figura 5-47](#)).
- **RN06:** essa regra implica em outra responsabilidade do sistema, que deve saber determinar situação de um aluno inscrito em uma turma. A classe mais adequada para satisfazer essa responsabilidade é Avaliacao, posto que está associada tanto a Turma quanto a Aluno.
- **RN07:** essa regra resultou na utilização da restrição {ordered} atrelada à associação entre Aluno e ListaEspera (veja também a [Seção 5.7.2](#)).

Pelas descrições fornecidas nos itens acima, repare que, para algumas responsabilidades, já é claro em que classe(s) elas devem ser alocadas. Entretanto, para outras ainda não é seguro afirmar em que classe(s) elas devem ser atribuídas. É algo comum na fase de análise e o modelador não deve dispensar tempo excessivo com isso, já que a tarefa de identificação de novas classes e de atribuições de responsabilidades continua na fase de projeto, momento em que essa atribuição de responsabilidade deve ser considerada por completo. Nesse contexto, é relevante ler novamente o último parágrafo da [Seção 5.6](#).

5.7.5 Documentação das responsabilidades

Conforme mencionado na [Seção 5.5.3](#), algumas responsabilidades simples podem ser representadas na análise por meio de atributos, restrições etc., enquanto outras devem ser identificadas e modeladas em um nível alto de abstração, para posteriormente serem detalhadas na fase de projeto. Esse detalhamento normalmente ocorre por meio da criação de uma ou mais operações, em classes já identificadas durante a análise, ou mesmo em novas classes definidas durante a etapa de projeto. A seguir, apresentamos uma possível forma de documentação de responsabilidades em tempo de análise para algumas classes do SCA.

1. Conhecer seus pré-requisitos;
2. Conhecer seu código, nome e quantidade de créditos.

ListaEspera

1. Conhecer a disciplina para a qual foi criada;
2. Manter os alunos em espera pela abertura de vagas.

Aluno

1. Conhecer seu número de registro e nome;
2. Conhecer as disciplinas que já cursou.

HistoricoEscolar

1. Permitir o lançamento de trancamentos, aprovações e reprovações;
2. Conhecer quais são as disciplinas da grade curricular que ainda não foram cursadas pelo aluno;
3. Conhecer quais são as disciplinas da grade curricular que já foram cursadas e as que não foram pelo aluno.

Turma

1. Conhecer o seu código e situação;
2. Conhecer a disciplina para a qual é ofertada;
3. Inscrever um aluno;
4. Conhecer o semestre letivo em que acontece;
5. Conhecer dias, horários e salas de aula em que acontece;
6. Conhecer sua capacidade, por exemplo, sua quantidade máxima de alunos;
7. Conhecer seu professor.

5.7.6 Glossário de conceitos

Durante a modelagem de classes, os termos relevantes do domínio do problema (i.e. os correspondentes a classes identificadas) devem ser definidos em um glossário. No caso do SCA, a definição inicial desse glossário é apresentada a seguir:

1. **Aluno:** representa um aluno, que se inscreve em turmas.
2. **Avaliacao:** representa notas e frequência obtidas por um aluno em uma turma. A avaliação é lançada por um professor e reflete o aproveitamento do aluno na turma correspondente.
3. **Aula:** representa o acontecimento de um dos encontros semanais de uma de alguma turma. Todas aula acontece em um local e está associada a um item de horário (ItemHorario).
4. **DiarioClasse:** corresponde a um formulário que contém os nomes dos alunos inscritos em determinada turma. Nesse formulário, o professor lança as avaliações (notas e quantidade de faltas) de cada aluno.
5. **Disciplina:** uma disciplina componente da grade curricular. Uma disciplina pode possuir *pré-requisitos*, que também são disciplinas.
6. **GradeDisponibilidades:** representa as informações fornecidas por um professor para indicar em que dias e horários está disponível para lecionar, assim como quais disciplinas está disponível para lecionar. Uma grade de disponibilidade é definida para um determinado SemestreLetivo e composta pelos dias da semana (e os respectivos intervalos de tempo) nos quais um professor está disponível para lecionar (p. ex., ser alocado a alguma turma).

Representa também as disciplinas que o professor está disponível para lecionar nesse mesmo semestre letivo. As informações de grades de disponibilidade de um determinado período letivo são usadas para montar a grade de horários daquele período.

7. **GradeHorarios:** Todas as turmas ofertadas para um determinado semestre letivo (com seus horários, locais de aula e professores responsáveis) compõem a grade de horários daquele semestre. A cada período letivo, é definida uma nova grade de horários. A grade de horários de um período letivo é necessária para permitir a inscrição de alunos em turmas daquele período.
8. **GradeCurricular:** uma grade curricular corresponde a um conjunto de disciplinas, juntamente com os pré-requisitos de cada uma delas. De tempos em tempos, a instituição de ensino atualiza a grade curricular do curso. Nessa atualização novas disciplinas podem ser criadas, assim como disciplinas existentes podem ser removidas da grade curricular. É também possível que pré-requisitos sejam redefinidos em uma alteração da grade curricular. Uma grade curricular entra em vigência (passa a valer) em um determinado semestre letivo e sai de vigência em outro semestre letivo.
9. **Intervalo:** corresponde a um intervalo de tempo. Formado por um horário de início e um horário de término. Serve para delimitar a duração da ocorrência de determinado evento com precisão de minutos. Por exemplo, o intervalo de tempo (18h20, 21h50) possui duração igual a 210 minutos.
10. **ItemHorario:** um item de horário é a unidade mínima de alocação de aulas para uma turma em uma semana. Um item de horário é formado por um dia da semana e por um intervalo de tempo de duração fixa (p. ex., 50 minutos). As aulas de uma turma ocorrem em um ou mais itens de horário semanais, que podem ser contíguos ou não. Por exemplo, a tabela apresentada a seguir contém 17 itens de horário, cada um deles com duração de 50 minutos e distribuídos em turnos (manhã, tarde ou noite).

ID	Manhã	Tarde	Noite
1°	07:00 - 07:50	12:40 - 13:30	18:20 - 19:10
2°	07:55 - 08:45	13:35 - 14:25	19:10 - 20:00
3°	08:50 - 09:40	14:30 - 15:20	20:00 - 20:50
4°	09:55 - 10:45	15:35 - 16:25	21:00 - 21:50
5°	10:50 - 11:40	16:30 - 17:20	21:50 - 22:40
6°	11:45 - 12:35	17:25 - 18:15	

11. **HistoricoEscolar:** um histórico escolar está associado a um aluno e representa o registro do desempenho acadêmico desse aluno nas diferentes turmas em que ele participou. É responsabilidade dessa classe registrar as aprovações e reprovações obtidas pelo aluno. É também sua responsabilidade manter o conhecimento acerca de quais disciplinas da grade curricular o aluno já *venceu*. Diz-se que um aluno *venceu uma disciplina* se ele cursou com aprovação alguma turma ofertada para essa disciplina.
12. **ListaEspera:** representa uma fila de alunos que estão esperando a abertura de uma Turma para determinada Disciplina.
13. **Local:** representa um dos locais que podem ser utilizados para as aulas de uma ou mais turmas.

14. **Inscricao:** representa o registro da inscrição de um aluno em alguma turma. Uma inscrição é posteriormente associada a uma avaliação (Avaliacao).
15. **Professor:** representa o indivíduo que leciona as disciplinas ofertadas por meio de turmas.
16. **SemestreLetivo:** representa um semestre letivo. Por exemplo, 2013.1 ou 2014.2.
17. **Turma:** representa uma turma existente em algum semestre letivo para realizar as atividades de determinada disciplina. Uma turma é a contrapartida temporal de uma (e apenas uma) disciplina da grade curricular.

► EXERCÍCIOS

5-1: Descreva a posição do diagrama de classes no processo de desenvolvimento incremental e iterativo. Quando eles são utilizados e para quê?

5-2: O nome “entidade” aparece neste Capítulo em dois contextos diferentes, a saber, a categorização BCE e na descrição dos padrões táticos do DDD. Discuta as diferenças e semelhanças existentes entre esses usos.

5-3: Construa o modelo de classes de domínio de um sistema de informações para controlar o campeonato da Fórmula 1.

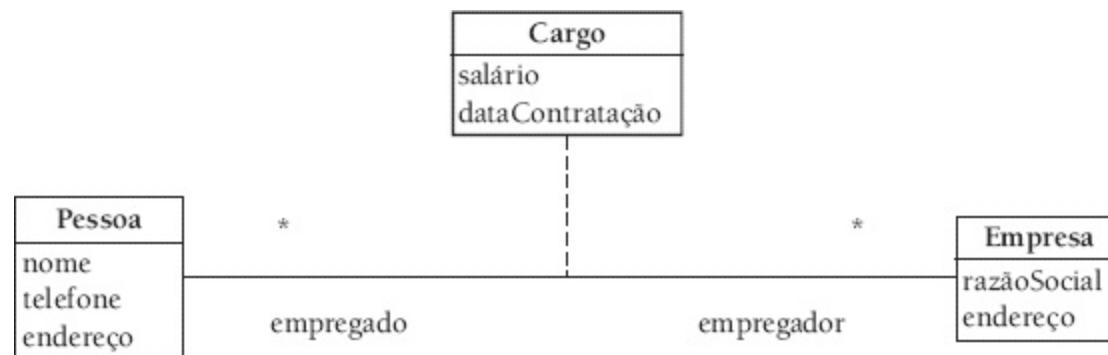
5-4: Desenhe um diagrama de classes com relacionamentos, nomes de papéis e multiplicidades para as seguintes situações:

Uma Pessoa pode ser casada com outra Pessoa.

Uma Disciplina é pré-requisito para outra Disciplina.

Uma Peça pode ser composta de diversas outras Peças.

5-5: Considere o diagrama de classes a seguir, que exibe uma classe associativa entre as classes Pessoa e Empresa. Crie um diagrama de classes equivalente ao fornecido a seguir, mas sem utilizar uma classe associativa.



5-6: Construa um diagrama de classes inicial para a seguinte situação: Pacotes são

enviados de uma localidade a outra. Pacotes têm um peso específico. Localidades são caracterizadas pelas facilidades de transporte (p. ex., rodoviárias, aeroportos e autoestradas). Algumas localidades são vizinhas, isto é, existe uma rota direta de transporte entre tais localidades. A rota de transporte entre as localidades tem certo comprimento (a distância entre as localidades). Treins, aviões e caminhões são usados para o transporte de pacotes. Cada um destes meios de transporte pode suportar uma carga máxima de peso. A cada momento, durante o seu transporte, é necessário saber a posição (localidade) de cada pacote. Também é necessário manter o controle de que meio de transporte está sendo utilizado em cada parte da rota para um certo pacote.

5-7: Considere o seguinte discurso relativo a um sistema de partidas de tênis: “Num torneio de tênis, cada partida é disputada entre dois jogadores. Pretende-se manter informação sobre o nome e a idade dos jogadores; data da partida e atribuição dos jogadores às partidas. O máximo de partidas que um jogador poderá realizar são seis e o mínimo uma”. Desenhe o diagrama de classes correspondente.

5-8: Desenhe um diagrama equivalente ao da [Figura 5-10](#) de duas formas:

- a) Utilizando uma classe ordinária para substituir a classe associativa.
- b) Utilizando uma associação ternária.

5-9: Identifique classes e/ou relacionamentos a partir das seguintes regras do negócio:

- a) Pedidos são compostos de vários itens de pedido.
- b) Um item de pedido diz respeito a um e exatamente um produto.
- c) Um pedido pode conter até 20 itens.

5-10: Considere um sistema de software para controlar um hotel. Normalmente, um *hóspede* ocupa um *quarto* por *estada*. Suponha, porém, que uma nova regra foi criada no negócio: agora, um hóspede pode utilizar até três quartos. Desenhe o diagrama de classe para essas duas situações.

5-11: Reflita sobre a seguinte afirmação: “O tamanho do cartão CRC ajuda a limitar e a restringir a complexidade das classes identificadas nas sessões CRC.”

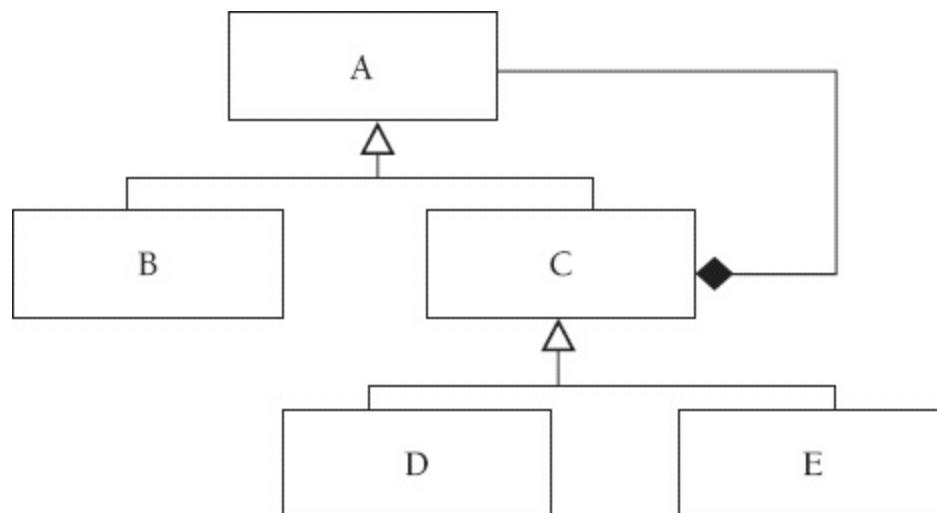
5-12: Reflita e discuta com algum colega sobre a seguinte afirmação: “Atributos são similares a associações. Um atributo de uma classe é apenas uma notação para associá-la a um conceito que tem um valor atômico.”

5-13: A seguir, são enumeradas diversas responsabilidades típicas de serem encontradas em objetos de um sistema de software. No contexto da categorização BCE, discuta qual das categorias de objetos (fronteira, controle ou entidade) é mais

adequada para cumprir cada uma dessas responsabilidades:

- a) Criação ou destruição de um objeto.
- b) Formação ou destruição de associações entre objetos de entidade.
- c) Obtenção ou modificação de valores de atributos de um objeto de entidade.
- d) Exibição de mensagens para o ator.
- e) Realização de cálculos complexos.

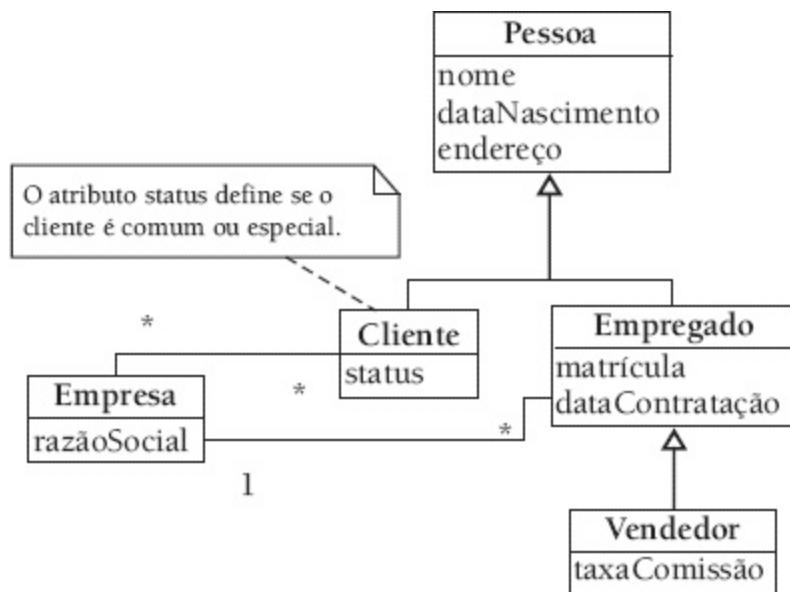
5-14: Considere as instâncias das classes do diagrama exibido a seguir.



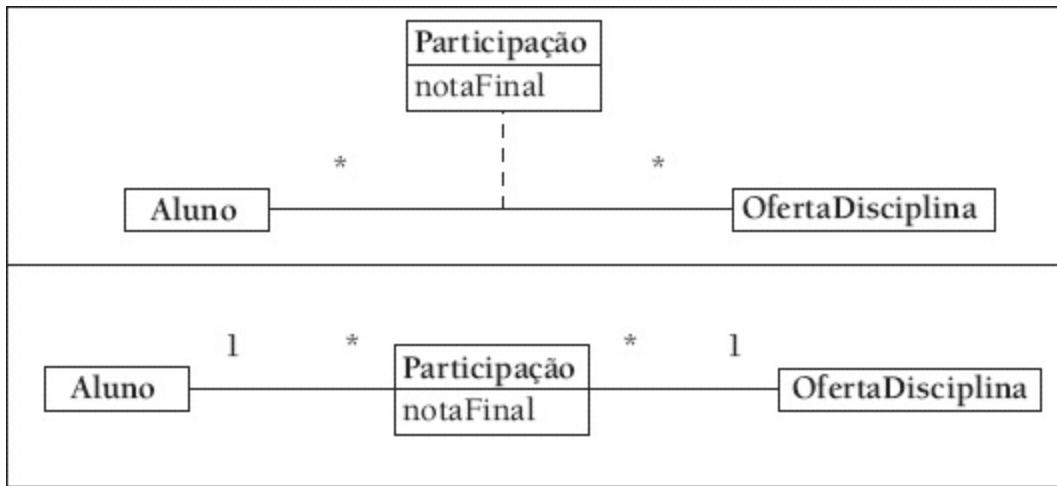
Para o diagrama de classes apresentado, qual das seguintes situações são possíveis?

- 1. e1 contém um d1, o qual contém um e2, o qual contém um b2.
- 2. a1 contém um c1, o qual contém um d1.
- 3. b1 contém um d1, o qual contém um e2 .
- 4. c1 contém um a1, o qual contém b1.

5-15: Considere o diagrama de classes a seguir. Há como construir um modelo equivalente, sem o uso de gen/spec? Qual seria a vantagem dessa transformação?



5-16: Analise os dois fragmentos de diagrama de classes a seguir. Eles são equivalentes? Explique sua resposta.



- As multiplicidades são bastante semelhantes ao conceito de cardinalidade encontrado em outras notações para modelagem de dados (p. ex., a notação de Peter Chen para o *diagrama de entidades e relacionamentos*, bastante conhecido na comunidade de Bancos de Dados).
- Diagramas de objetos são também chamados *diagramas de instâncias*.
- Esses objetos normalmente são armazenados em algum meio persistente. O [Capítulo 12](#) descreve como esses objetos podem ser mapeados e armazenados em um banco de dados relacional.
- É por meio da *troca de mensagens* de um objeto a outro que essas colaborações são acionadas para que as funcionalidades do sistema estejam disponíveis externamente. A modelagem de mensagens entre objetos é descrita no [Capítulo 7](#).
- Esse autor também é o principal expoente de um processo de desenvolvimento que está se tornando bastante famoso, o XP (sigla para *Extreme Programming*).
- De preferência, cada participante deve ficar com um único cartão. Isso faz com que o participante se identifique com a classe correspondente.
- Classes candidatas iniciais também podem ser encontradas em entrevistas com o usuário, no documento de requisitos, no modelo de regras do negócio etc.
- Talvez esse erro comum esteja associado à correlação que as pessoas fazem com o modelo relacional, no qual existe o conceito de chave estrangeira (ver [Seção 12.2](#)). No entanto, esse é um conceito que não deve ser utilizado no modelo de classes.

Passagem da análise para o projeto

Há duas maneiras de fazer o projeto de um sistema de software. Uma delas é fazê-lo tão simples que obviamente não há deficiências. E a outra é fazê-lo tão complexo que não há deficiências óbvias. O primeiro método é de longe o mais difícil.

– C.A.R. Hoare

No desenvolvimento de sistemas orientados a objetos, a mesma representação é utilizada durante a análise e o projeto para representar os conceitos relevantes do sistema: o modelo de classes é utilizado para representar a estrutura das classes do sistema em ambas as fases.¹ A vantagem disso é que há uma uniformidade na modelagem do sistema durante o desenvolvimento, ao contrário do que acontece no desenvolvimento de sistemas que optam pela metodologia de *Análise e Projeto Estruturados*, que utiliza na modelagem várias notações diferentes.

No entanto, essa uniformidade de representação tem a desvantagem de tornar bem menos nítida a separação entre o que é feito na análise e o que é feito no projeto. Além disso, a utilização de um processo de desenvolvimento iterativo torna ainda mais difícil essa separação. Há uma quantidade significativa de tarefas no desenvolvimento de um sistema orientado a objetos que se situa em uma área difusa entre a análise e o projeto. Pode-se dizer que a *análise vai se transformando em projeto* à medida que o desenvolvimento evolui. Em um processo de desenvolvimento iterativo aplicado a um SSOO, a análise precede o projeto inicialmente. Entretanto, em uma determinada iteração, análise e projeto podem estar acontecendo simultaneamente.

De qualquer forma, o modelo de classes de análise ([Capítulo 5](#)) e o modelo de casos de uso ([Capítulo 4](#)) são dois dos artefatos construídos na fase de análise de uma iteração em um processo de desenvolvimento incremental. Também são produtos da fase de análise outros artefatos, como o documento de regras do negócio ([Seção 4.5.1](#)). Na modelagem de classes de análise, estamos interessados em *identificar* as classes que correspondem a conceitos do mundo real e que devem ser manipulados pelo SSOO. Já no projeto, o interesse é *refinar* essas classes conceituais, além de criar outras que se tornam necessárias para aproximar o modelo de sua implementação. Sendo assim, mesmo que durante a análise e o projeto a equipe de desenvolvimento do SSOO trabalhe sobre o mesmo modelo, essas atividades se distinguem pela quantidade de detalhes que são incluídos nesse modelo.

Os modelos construídos na análise esclarecem o problema a ser resolvido. No entanto, as perspectivas do sistema fornecidas por esses modelos não são suficientes para se ter uma visão completa do sistema para que a implementação comece. Antes disso, diversos aspectos referentes à solução a ser utilizada devem ser definidos. É na fase de projeto (ver a [Seção 2.1.3](#)) de uma iteração que essas definições realmente são feitas. O projeto é uma etapa para definir a solução do problema relativo ao desenvolvimento do SSOO. O objetivo é encontrar alternativas para que o sistema atenda aos *requisitos funcionais*, respeitando ao mesmo tempo as restrições definidas pelos *requisitos não funcionais*. Além disso, essa etapa deve aderir a certos princípios para alcançar uma qualidade desejável no produto de software final. As principais atividades realizadas na fase de projeto são

enumeradas a seguir. Após a realização dessas atividades, os modelos estarão em um nível de detalhamento grande o suficiente para que o sistema possa ser implementado.

1. Detalhamento dos aspectos dinâmicos.
2. Refinamento dos aspectos estáticos e estruturais.
3. Detalhamento de aspectos arquiteturais.
4. Definição das estratégias para armazenamento, gerenciamento e persistência dos dados manipulados pelo sistema.
5. Realização do projeto da interface gráfica com o usuário.
6. Definição dos algoritmos a serem utilizados na implementação.
7. Determinação dos padrões de projeto que devem ser aplicados.

Por serem muito extensos, os tópicos que abordam as atividades enumeradas anteriormente estão divididos em vários capítulos seguintes neste livro. Para alguns desses tópicos, sua descrição está contida em um capítulo ou seção deste livro. Para outros, sua descrição está fragmentada em diversas seções. Este capítulo fornece ao leitor uma visão geral desses tópicos antes de abordá-los em mais detalhes. Antes de apresentar essa visão geral, é importante deixar claro que os próximos capítulos não tratam exclusivamente de atividades da fase de projeto. Algumas atividades relativas à análise ainda são descritas. Por exemplo, nos próximos capítulos descrevemos a modelagem de interações (que pode ser realizada com o diagrama de interações, o diagrama de estados e o diagrama de atividades), que também é uma atividade que começa na fase de análise de um SSOO e continua durante a etapa de projeto.

6.1 Detalhamento dos aspectos dinâmicos

O aspecto dinâmico de um SSOO diz respeito ao comportamento de seus objetos a tempo de execução, por exemplo, de que modo esses objetos trocam mensagens para cumprir com as responsabilidades identificadas para o sistema. Como ilustração disso, note que uma parte do aspecto dinâmico do SCA tem a ver com a forma pela qual o sistema reage quando recebe a solicitação de inscrever um aluno em uma turma. Na modelagem de classes de análise, considera-se os aspectos dinâmicos do sistema quando os modeladores utilizam técnicas de identificação de classes como a identificação dirigida por domínio (ver a [Seção 5.4.3](#)). Isso porque essas técnicas simulam a colaboração entre objetos na realização de um caso de uso. Embora o estudo dos aspectos dinâmicos do sistema já comece na etapa de análise, é na fase de projeto que esse estudo se concretiza e onde se realiza o detalhamento das colaborações nas quais cada classe participa.

Para fazer o detalhamento desse aspecto dinâmico, o *modelo de interações*, o *modelo de estados* e o *modelo de atividades* do sistema devem ser construídos. Esses modelos são descritos nos [Capítulos 7, 9 e 10](#), respectivamente. As atividades de construção desses modelos são conhecidas em conjunto como *modelagem dinâmica* do sistema.

6.2 Refinamento dos aspectos estáticos e estruturais

O modelo de classes de análise ([Capítulo 5](#)) descreve as classes do sistema em um nível alto de

abstração, por meio da definição de suas responsabilidades. A passagem do modelo de análise para o modelo de projeto não é tão direta. Por exemplo, pode haver uma classe de análise que resulte em várias classes de projeto. Embora seja uma situação mais rara, pode também haver classes de análise que resultem em uma única classe de projeto. De qualquer forma, tanto para classes que sobrevivem à passagem para o modelo de projeto quanto para as classes criadas durante a etapa de projeto, vários detalhes devem ser definidos sobre elas. Desse detalhamento resulta o **modelo de classes de projeto**. Na modelagem de classes de projeto, são especificados detalhes dos aspectos estrutural e estáticos do modelo de classes: tipos para os atributos, especificações para as operações e as navegabilidades das associações entre classes. Em particular, as responsabilidades de uma classe que foram especificadas em alto nível na etapa de análise devem ser representadas por atributos e operações.

Por outro lado, o próprio detalhamento dos aspectos dinâmicos do sistema gera material (informações, por exemplo) para refinar os aspectos estático e estrutural definidos no modelo de classes de análise. O *detalhamento de atributos e operações* de classes é descrito no [Capítulo 8](#), no qual também se descreve o *detalhamento das associações* existentes entre objetos.

O modelo de classes de análise também pode ser refinado pela identificação de *similaridades* (como atributos e/ou comportamento em comum) entre as classes existentes. Esse refinamento é feito pelo uso de relacionamentos de *generalização e especialização*. No [Capítulo 5](#), descrevemos sucintamente o relacionamento de generalização/especialização. Vimos que esse relacionamento é útil quando queremos fatorar características e comportamentos comuns a um grupo de classes. No entanto, quando passamos a considerar detalhes da solução na fase de projeto, diversos aspectos relativos ao relacionamento de generalização/especialização devem ser definidos. Além disso, na fase de projeto podem ser utilizados outros elementos de modelagem para definir similaridades entre classes, como o conceito de *interface de tipos*. Descrevemos esses aspectos também na [Seção 8.5](#).

De uma forma geral, o projeto de um sistema é a atividade em que diversos detalhes devem ser adicionados aos modelos. Alguns conceitos que também abordamos no [Capítulo 8](#) e que são fundamentais durante o detalhamento dos modelos provenientes da análise são: acoplamento, coesão, polimorfismo, classificação dinâmica, classes abstratas, interfaces de tipos, implementação de associações, interfaces e padrões de projeto.

6.3 Projeto da arquitetura

Outro aspecto a considerar na modelagem de um SSOO é a forma como esse sistema pode ser recursivamente decomposto em seus diversos subsistemas componentes. Há dois tipos de decomposição no contexto arquitetural possíveis, a lógica e a física.

A decomposição lógica de um sistema diz respeito à disposição das camadas de software que compõem o mesmo, independentemente das diversas localizações físicas em que o sistema será executado. Nesse contexto, surgem os termos camada de apresentação, camada de serviço, camada de domínio etc.

Por outro lado, nos dias atuais, é comum o desenvolvimento de aplicações distribuídas, ou seja, sistemas cujos componentes estão fisicamente espalhados por diversos nós de processamento (ou processadores). Se existem vários nós de processamento disponíveis para a execução de um sistema, a forma como o sistema deve ser fisicamente distribuído por esses diversos nós também é relevante e deve ser considerada em sua modelagem. Essa parte da modelagem da arquitetura é denominada

definição da decomposição física.

Em geral, damos o nome de *arquitetura* à forma como um sistema está disposto e subdividido, física e logicamente. O projeto arquitetural de um sistema é a atividade do processo de desenvolvimento na qual a arquitetura desse sistema é definida. Aspectos importantes a serem definidos no projeto da arquitetura são: subsistemas, interfaces, camadas de software e dependências entre subsistemas. Outras expressões que surgem no projeto arquitetural incluem distribuição, controle de falhas, redundância etc. A definição da arquitetura de um SSOO também envolve a definição de meios pelos quais os seus objetos possam se comunicar se estiverem localizados em máquinas (como nós de processamento) diferentes (essa questão é relevante no contexto de sistemas de arquitetura distribuída). Também é encargo dela a escolha de componentes reutilizáveis (como frameworks e bibliotecas) a serem utilizados durante a implementação. Questões relativas às arquiteturas física e lógica de um sistema são abordadas no [Capítulo 11](#).

6.4 Persistência de objetos

Outro aspecto importante no projeto de um SSOO é relativo à forma pela qual objetos desse sistema são armazenados de forma persistente. Nesse contexto, há dois tipos de objeto em um SSOO: os persistentes e os transientes. Esses últimos são aqueles cujo tempo de vida é o de uma sessão de uso do sistema. Ou seja, objetos transientes simplesmente desaparecem quando o processo que os criou termina.

De forma diferente do que acontece com os objetos transientes, os objetos persistentes devem existir por várias sessões do sistema. Para objetos que devem ser persistentes, a questão que se põe é como manter suas informações do sistema entre sessões de uso do mesmo. Na prática, um SSOO precisa controlar vários aspectos relativos a objetos persistentes. Alguns desses aspectos são enumerados a seguir.

- De que forma realizar o mapeamento de representações entre a estrutura de classes e a estrutura esperada pelo mecanismo de armazenamento persistente.
- Como as transações, como operações sobre vários objetos que devem ser realizadas de forma atômica, são controladas.
- Quando e como objetos persistentes devem ser enviados para o mecanismo de armazenamento persistente.
- Quando e como os objetos persistentes devem ser lidos do mecanismo de armazenamento persistente.
- Quando e como os objetos persistentes são removidos.

Examinamos mais a fundo as questões sobre a *persistência de objetos* no [Capítulo 12](#), no qual apresentamos uma discussão a respeito das principais estratégias de mapeamento de objeto relacional existentes. No entanto, damos especial importância à estratégia de persistência que usa um sistema de gerenciamento de bancos de dados relacional. São apresentadas questões relativas ao *mapeamento de classes* para um sistema de gerência de banco de dados relacional. Em particular, descrevemos o uso de um framework de mapeamento objeto-relacional durante o desenvolvimento de um SSOO.

6.5 Projeto de interface gráfica com o usuário

Uma característica importante sobre modelos de casos de uso é que associações entre casos de uso e atores implicam a necessidade de interfaces. Quando o ator é um ser humano, são necessárias, então, telas (formulários), relatórios etc. para dar suporte à associação. O projeto da interface gráfica do usuário é uma atividade cujo objetivo é definir a aparência do sistema relativamente aos seus usuários. Nessa atividade, a meta principal é definir um sistema de alta usabilidade e facilidade de operação. Outros objetivos importantes nessa atividade são: padronização de cores, de mensagens de erro, das dimensões dos controles gráficos, formatação para entrada de dados etc.

Não é escopo desse livro o detalhamento de aspectos relativos ao projeto da interface gráfica de um sistema. De fato, há toda uma teoria desenvolvida na área de pesquisa denominada Interface Homem-Máquina (*Human-Machine Interface*). No entanto, é importante notar que o projeto da interface gráfica de um sistema é uma atividade fundamental, visto que a correta realização dessa atividade resulta na qualidade da parte visível do sistema. Este ponto é crítico para o projeto e pode determinar o sucesso ou total fracasso do produto de software.

Em março de 2013, foi adotado um novo padrão pela OMG, a mesma organização internacional que adotou a UML como padrão de modelagem de sistemas (veja a [Seção 1.4](#)). Esse novo padrão é denominado *Interaction Flow Modeling Language*, ou, abreviadamente, IFML. A IFML dá suporte à especificação de interfaces gráficas com o usuário de modo independente de plataforma. Maiores detalhes sobre esse padrão para modelagem da interface gráfica podem ser encontrados em WAZLAWICK (2014).

6.6 Projeto de algoritmos

O projeto de algoritmos é outra atividade da atividade de projeto. Aspectos que influenciam na escolha dos algoritmos para implementar um sistema são: (1) complexidade computacional, (2) facilidade de entendimento e (3) flexibilidade. A especificação desses algoritmos pode ser feita tanto formal quanto informalmente. O diagrama de atividades da UML pode ser utilizado durante essa especificação (ver [Seção 10.2.3](#)). Neste livro, não tratamos de detalhes do projeto de algoritmos. Para mais detalhes sobre o assunto, recomendamos a referência SZWARCFITER e MARKENZON (2010).

6.7 Padrões de software

O uso de padrões de software tem ao longo dos anos se consolidado na comunidade de Engenharia de Software como uma forma adequada de alcançar o reúso e de desenvolver sistemas de qualidade. Padrões de software são aplicáveis em diversas situações durante o desenvolvimento de sistemas orientados a objetos. Neste livro, descrevemos alguns padrões de software e seu contexto de uso. A lista a seguir resume os padrões de software descritos neste livro. É importante notar que os padrões de software descritos neste livro nem de perto esgotam o assunto tão vasto. Desse modo, também apresentamos na lista a seguir referências para estudo adicional sobre padrões de software em diferentes catálogos.

- *Model-View-Controller* (MVC), padrão arquitetural para organizar aplicações que devem fornecer interfaces com o usuário, na maior parte das vezes gráficas (BUSCHMANN, 1996). Apresentamos esse padrão na [Seção 11.1.3](#).

- *Front Controller* e *View Helper*, padrões de projeto utilizáveis na implementação do MVC em aplicações para Web. Veja a [Seção 11.1.3](#). Esse é um padrão do catálogo JEE (ALUR et al., 2003).
- Padrões *Composite*, *Observer*, *Strategy*, *Factory Method*, *Mediator*, *Facade*. descritos na [Seção 8.6](#). Esses padrões fazem parte do catálogo GoF, que documenta os 23 padrões de projeto clássicos da orientação a objetos (GAMMA et al., 2000). Outro padrão desse catálogo é o *Proxy*, que mencionamos na [Seção 11.2.3](#), no contexto da distribuição de objetos.
- *DAO (Data Access Object)*, padrão de projeto cuja finalidade é desacoplar a aplicação do mecanismo de armazenamento persistente específico utilizado. Esse é outro padrão do catálogo JEE (ALUR et al., 2003). Sua descrição é apresentada na [Seção 12.2.3](#).
- *DTO (Data Transfer Object)*, padrão de projeto para transferência de informações entre nós de processamento. Esse é outro padrão do catálogo JEE (ALUR et al., 2003). Veja a [Seção 11.2.3](#).
- Padrões táticos do DDD (*Entity*, *Value Object*, *Aggregate*, *Domain Service*, *Factory*, *Repository*). Descritos na [Seção 5.4.3.2](#). O livro de Eric Evans (EVANS, 2003) é a fonte principal de consulta acerca desses padrões e do DDD em geral.
- *Metamodel* (MULLER, 1999) e *Party* (FOWLER, 1997; HAY, 1996), dois padrões de análise, ambos descritos na [Seção 5.4.4](#).

1. Note que aqui o termo *fase* é utilizado para denotar análise e projeto. Mas é importante não confundir com o significado do termo *fase* utilizado na descrição do processo de desenvolvimento iterativo e incremental.

Modelagem de interações

Nossas capacidades intelectuais são bastante voltadas para dominar relações estáticas, e nossas capacidades de visualizar processos em evolução no tempo são relativamente pouco desenvolvidas. Por essa razão é que devemos fazer (como sábios programadores, conscientes de nossas limitações) o máximo para diminuir a distância conceitual entre o programa estático e o processo dinâmico, para fazer a correspondência entre o programa (distribuído no espaço) e o processo (distribuído no tempo) tão trivial quanto possível.

– EDSGER W. DIJKSTRA.

Em capítulos anteriores, dois modelos são descritos: o de casos de uso ([Capítulo 4](#)) e o de classes de análise ([Capítulo 5](#)). Vamos resumir o que esses dois modelos fornecem de informação acerca de um sistema. O modelo de casos de uso detalha os requisitos funcionais do sistema e descreve as entidades do ambiente (atores) que interagem com ele. Este modelo nos informa também quais são as ações do sistema conforme percebidas pelos atores, e quais as ações do ator, conforme percebidas pelo sistema. Com esse modelo, podem ser respondidas questões a respeito do que o sistema deve fazer e para quem. No entanto, o modelo de casos de uso nada informa sobre qual deve ser o comportamento interno do sistema para que uma determinada funcionalidade se realize. Ou seja, para que um caso de uso seja realizado, produzindo um resultado de valor para o ator, as questões a seguir são relevantes. (Note que, para essas perguntas, não encontramos respostas no modelo de casos de uso de um sistema, não importa quão detalhado esse modelo seja.)

1. Quais são as operações que devem ser executadas internamente ao sistema?
2. A que classes essas operações pertencem?
3. Quais objetos participam da realização desse caso de uso?

Por outro lado, o modelo de classes de análise fornece uma visão estrutural e estática inicial do sistema. A construção deste modelo resulta em um esboço das classes e de suas responsabilidades. No entanto, algumas questões também não são respondidas por esse modelo:

1. De que forma os objetos colaboram para que um determinado caso de uso seja realizado?
2. Em que ordem as mensagens são enviadas durante essa realização?
3. Que informações precisam ser enviadas em uma mensagem de um objeto a outro?
4. Será que há responsabilidades ou mesmo classes que ainda não foram identificadas?

A leitura dos parágrafos anteriores dá a entender que o modelo de casos de uso e o modelo de classes são representações incompletas do sistema. E de fato essa é a realidade. Para responder às questões levantadas nos parágrafos anteriores, o *modelo de interações* do sistema precisa ser criado. Esse modelo representa as *mensagens* (ver a [Seção 1.2.2](#)) trocadas entre os objetos para a execução de cenários dos casos de uso do sistema. A modelagem de interações é uma parte da *modelagem dinâmica* de um SSOO (a outra parte corresponde à modelagem de estados, que descrevemos no

[Capítulo 9](#)). A modelagem dinâmica é realizada com o propósito de entender de que forma o sistema irá se comportar em tempo de execução.

A construção do modelo de interações pode ser vista como uma consolidação do entendimento dos aspectos dinâmicos do sistema. Em particular, alguns aspectos iniciais relativos à dinâmica de interação entre objetos já é conhecida na modelagem de classes de análise. Entretanto, os aspectos dinâmicos identificados com aquela técnica ainda são incipientes e incompletos. Por meio da construção do modelo de interações, as classes, as responsabilidades e os colaboradores identificados previamente podem ser validados. Esse modelo permite ainda refinar o modelo de classes, pois as operações (e até alguns atributos) de cada classe são identificadas na construção do modelo de interações. Em resumo, estudar de que forma objetos interagem no decorrer do tempo fornece informações para completar a perspectiva estática e estrutural da modelagem.

Neste capítulo, apresentamos detalhes da modelagem de interações. Na [Seção 7.1](#), começamos por descrever os elementos de modelagem utilizados nessa atividade. Nas [Seções 7.2](#) e [7.3](#), apresentamos, respectivamente, os elementos de notação do diagrama de sequência e do diagrama de comunicação, dois dos principais diagramas de interação existentes na UML. Na [Seção 7.4](#), finalizamos a descrição de detalhes de notação com apresentação de elementos gráficos úteis para modularização da modelagem de interações. Na [Seção 7.5](#), passamos a descrever aspectos relativos à construção propriamente dita do modelo de interações de um SSOO. Na [Seção 7.6](#), apresentamos uma contextualização da modelagem de interações em um processo de desenvolvimento. Na [Seção 7.7](#), finalizamos este capítulo com a continuação da apresentação de aspectos da modelagem de interações de nosso estudo de caso, o SCA.

7.1 Elementos da modelagem de interações

A interação entre objetos para dar suporte à funcionalidade de um caso de uso denomina-se *realização de um caso*. A realização de um caso de uso descreve o comportamento de um ponto de vista *interno* ao sistema. A realização de um caso de uso é representada por diagramas de interação.

Os diagramas da UML que dão suporte à modelagem de interações são conhecidos genericamente como *diagramas de interação*. Até a versão 1.X da UML havia dois diagramas para dar suporte à construção do modelo de interações: o *diagrama de sequência (sequence diagram)* e o *diagrama de comunicação (communication diagram)*; na UML 1.X, o diagrama de comunicação era chamado de *diagrama de colaboração*). De forma geral, esses dois tipos de diagramas de interação representam mensagens enviadas entre objetos. A diferença entre esses dois tipos está na ênfase dada às interações entre os objetos. No diagrama de sequência, a ênfase está na ordem temporal das mensagens trocadas entre os objetos. Já o diagrama de comunicação enfatiza os relacionamentos que existem entre os objetos que participam da realização de um cenário. O diagrama de sequência e o diagrama de comunicação são equivalentes entre si. De fato, um diagrama de comunicação pode ser transformado em um diagrama de sequência equivalente. Da mesma forma, um diagrama de sequência pode ser transformado em um diagrama de comunicação equivalente. No entanto, a popularidade e o uso do diagrama de sequência são bem mais amplos do que no caso do diagrama de comunicação. A [Figura 7-1](#) e a [Figura 7-2](#) apresentam esboços das estruturas típicas do diagrama de sequência e do diagrama de comunicação, respectivamente.

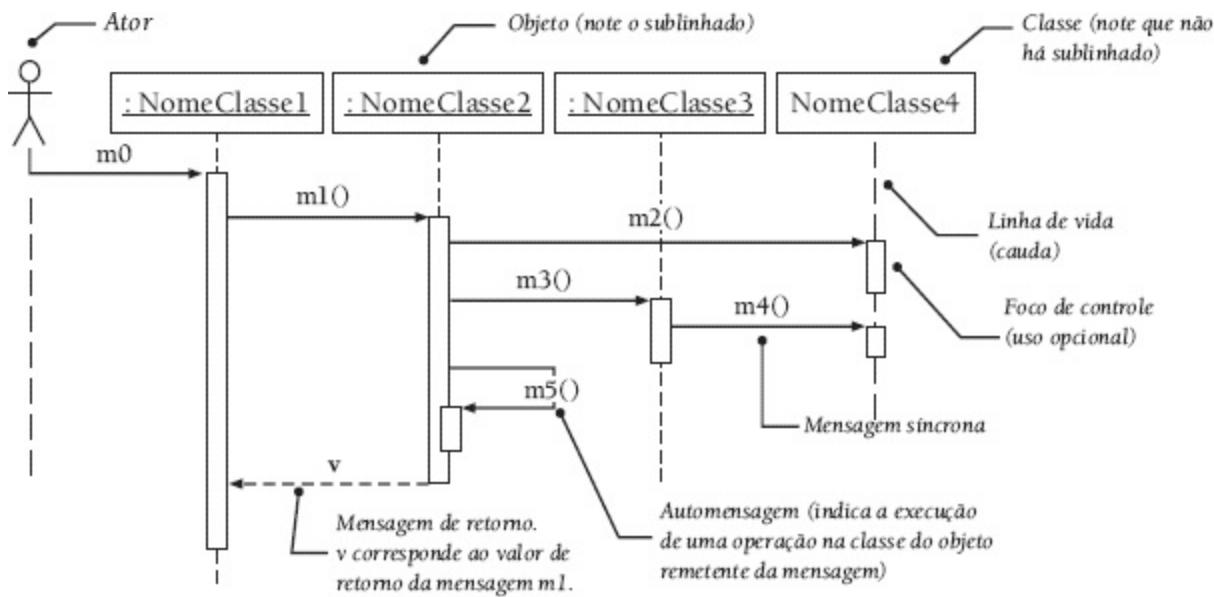


Figura 7-1: Estrutura típica de um diagrama de sequência.

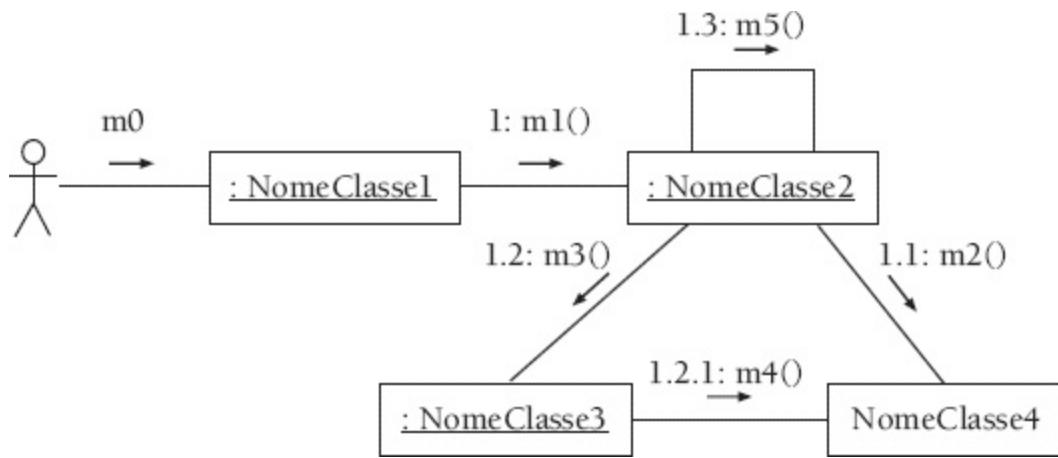


Figura 7-2: Estrutura típica de um diagrama de comunicação.

A versão 2.0 da UML apresenta outro tipo de diagrama de interação, o *diagrama de visão geral de interação* (tradução para *interaction overview diagram*). Esse último pode ser utilizado para apresentar uma visão geral de diversas interações entre objetos, cada uma delas representada por um diagrama de interação. Esse diagrama é útil para **modularizar** a construção dos diagramas de sequência (ou de comunicação). A [Seção 7.4.2](#) apresenta mais detalhes do diagrama de visão geral da interação.

Os diagramas de interação ajudam a documentar e a entender os aspectos dinâmicos do sistema de software. Mais especificamente, descrevem a sequência de mensagens enviadas e recebidas pelos objetos que participam em um caso de uso. Certas características de um caso de uso (como o controle de execução e a concorrência no envio de mensagens) também são esclarecidas pela construção de diagramas de interação.

Diagramas de interação mostram como os objetos do sistema agem internamente para que um ator atinja seu objetivo na realização de um caso de uso. A modelagem de um sistema de software orientado a objetos com a UML normalmente contém diversos diagramas de interação. O conjunto de todos os diagramas de interação de um sistema constitui o seu *modelo de interações*.

Um diagrama de interação é utilizado para modelar a lógica de um *cenário* de caso de uso ([Seção 4.1.1](#)). Um diagrama de interação também pode ser utilizado para modelar a troca de mensagens entre objetos *em uma parte de um cenário*, se ele for particularmente complexo. Sendo assim, dado um caso de uso, pode haver vários diagramas de interação a ele relacionados, um para cada cenário (ou parte de um cenário) considerado relevante. No entanto, se o caso de uso for simples, o diagrama de interação pode modelar o caso de uso como um todo.

O enfoque do diagrama de sequência está em como as mensagens são enviadas no decorrer do tempo. O enfoque do diagrama de comunicação está em como as mensagens são enviadas entre objetos que estão relacionados.

Os diagramas de sequência e de comunicação possuem muitas notações em comum. O restante desta seção descreve os diversos componentes de notação que são comuns a ambos os diagramas de interação. Na [Seção 7.2](#) e na [Seção 7.3](#), descrevemos as notações particulares de cada um desses diagramas de interação.

7.1.1 Mensagens

Conforme vimos no [Capítulo 4](#), o conceito fundamental no MCU é o caso de uso. De forma análoga, o elemento mais importante e o princípio básico da interação entre objetos é o conceito de *mensagem*. Conforme descrito na [Seção 1.2.2](#), uma mensagem é uma solicitação de execução de uma operação em outro objeto.¹ É por meio do envio de mensagens que os objetos do sistema colaboram entre si para prover as funcionalidades esperadas. Em outras palavras, um sistema de software orientado a objetos pode ser visto como uma rede de objetos. As funcionalidades desse sistema são realizadas por seus objetos componentes, que só podem interagir por meio de mensagens.

Uma mensagem representa a requisição enviada de um *objeto remetente* a um *objeto receptor* para que este último execute alguma operação definida em sua classe (ou em algum tipo ao qual pertença). Uma mensagem deve conter informação suficiente para que a operação do objeto receptor possa ser executada.

Uma mensagem enviada a um objeto invoca a execução de uma de suas *operações*. Podemos fazer uma correlação do conceito de *envio de mensagem* com o de *chamada de rotinas* das linguagens de programação. Uma mensagem pode ser vista como uma chamada a uma rotina definida para uma classe de objetos. Na prática, uma operação é uma rotina implementada em alguma linguagem de implementação e definida dentro do escopo de uma classe. Aliás, a diferença entre operação de uma classe e uma rotina está justamente no fato de que a operação é definida dentro de uma classe, em vez de ser um dos componentes de um módulo de um programa.

Quando uma mensagem é representada em um diagrama de interação, há normalmente um objeto remetente, com outro objeto servindo de receptor da mensagem. Além disso, o conteúdo da mensagem enviada pelo remetente especifica informações a serem passadas para a operação que deve ser executada no objeto receptor. Na UML, a sintaxe para representar mensagens é comum a ambos os tipos de diagrama de interação (sequência e comunicação). Por essa razão, os tipos de mensagens e a sintaxe de especificação são descritos separadamente nesta seção, antes do

detalhamento desses diagramas de interação propriamente ditos.

7.1.1.1 Natureza de uma mensagem

A versão 2.0 da UML define o conceito de *natureza da mensagem* (*message sort*). A natureza da mensagem indica o tipo de comunicação que foi utilizado para gerar a mensagem. As naturezas possíveis para uma mensagem são descritas a seguir.

1. Uma *mensagem síncrona* indica que o objeto remetente espera que o objeto receptor processe mensagem antes de recomeçar o seu processamento. Ou seja, o remetente fica bloqueado até que o receptor termine de atender à requisição. Uma mensagem síncrona está tipicamente relacionada à chamada de uma operação definida na classe do objeto receptor da mensagem.
2. Uma *mensagem assíncrona* é aquela na qual o objeto remetente não espera a resposta para prosseguir com o seu processamento.
3. Uma *mensagem de sinal* é aquela usada para enviar um sinal. Um sinal pode representar o envio de uma requisição entre dois módulos (p. ex., cliente e servidor) em um sistema distribuído.
4. Uma *mensagem de retorno* é aquela utilizada para especificar o retorno (término) de uma mensagem enviada anteriormente.

Um objeto pode enviar uma mensagem para ativar uma operação definida em sua própria classe (ou eventualmente em alguma de suas superclasses). Quando isso ocorre, diz-se que o objeto está enviando uma *mensagem reflexiva*. Isso significa que, em vez de requisitar a execução de uma operação definida em outra classe, o objeto em questão está requisitando a execução de uma operação definida na sua própria classe.

7.1.1.2 Sintaxe da UML para mensagens

Em ambos os tipos de diagramas de interação, cada mensagem é representada graficamente por uma seta cujo sentido vai do objeto remetente para o objeto receptor. Além disso, em ambos os diagramas as setas possuem *rótulos* que especificam a mensagem sendo enviada. Esse rótulo pode ser visto como a especificação das informações que são passadas pelo objeto remetente ao objeto receptor no envio de uma mensagem. O rótulo de uma mensagem pode ser definido simplesmente como o nome de uma operação a ser executada no objeto receptor. Essa forma de rótulo é a normalmente utilizada nos diagramas de interação de fase de análise. Por outro lado, o rótulo de uma mensagem pode corresponder à assinatura completa da operação a ser executada. A sintaxe para essa especificação mais completa, normalmente utilizada nos diagramas de interação da fase de projeto, é a seguinte:

[[expressão-sequência] controle:] [v:=] nome [(argumentos)]

Na especificação apresentada acima, os elementos delimitados por colchetes são opcionais. Com efeito, somente o nome da mensagem é obrigatório. Por outro lado, dependendo do grau de detalhe desejado na modelagem, a especificação de uma mensagem pode conter desde o nome da mensagem, apenas, até a definição de uma expressão que contenha todos os elementos possíveis da sintaxe. A descrição dos elementos da sintaxe anterior é dada a seguir.

7.1.1.2.1 Expressão de sequência

A uma mensagem pode estar associada uma expressão de sequência que serve para eliminar ambiguidades acerca de quando a mensagem foi enviada em relação às demais. Por exemplo, se as mensagens m_1 , m_2 e m_3 possuem expressões de sequência 1, 2 e 3, respectivamente, pode-se afirmar que a mensagem m_1 foi enviada antes de m_2 que, por sua vez, foi enviada antes de m_3 .

As expressões de sequência podem também ser definidas por um esquema de numeração em níveis. Por exemplo: 1.2, 1.2.1, 1.3, 2.4 etc. Esse esquema é útil quando desejamos modelar que mensagens são enviadas por conta de uma mensagem anterior ter sido enviada, o que nos permite facilmente definir a ordem correta de envio das mensagens envolvidas. Por exemplo, suponha que uma mensagem cuja expressão de sequência 1 é emitida. As mensagens 1.1, 1.2 etc. são enviadas pela operação que foi invocada pela mensagem 1. As mensagens 1.1.1, 1.1.2 etc. são enviadas pela operação que foi invocada pela mensagem 1.1 e assim por diante.

O formato de representação em níveis pode ser sufixado com letras para indicar o paralelismo do envio de duas ou mais mensagens. Por exemplo, as expressões de sequência 1.2a e 1.2b indicam mensagens que estão sendo enviadas em paralelo dentro da ativação 1.2.

7.1.1.2.2 Controle

Algumas vezes é necessário indicar que o envio de uma mensagem está condicionado ao valor de uma expressão lógica (ou seja, uma expressão cujo valor de avaliação é *verdadeiro* ou *falso*). Outras vezes é preciso indicar quantas vezes uma mensagem deve ser enviada consecutivamente. O elemento *controle* da sintaxe de uma mensagem pode ser utilizado com esses dois objetivos. Esse elemento de controle é especificado com a *OCL* (ver [Seção 3.6](#)). Há dois tipos de controle: a *cláusula-condição* e a *cláusula-iteração*. Ambos são descritos a seguir.

- Se o elemento *cláusula-condição* aparecer na especificação de uma mensagem, esta é enviada se, e somente se, o valor da expressão lógica for *verdadeiro*. Caso seja *falso*, a mensagem não é enviada. Por exemplo, $[a > b]$ é uma cláusula de condição; se, e somente se, a for maior que b, a mensagem correspondente é enviada. A cláusula-condição tem outros nomes: *condição de guarda* ou simplesmente *guarda*. A sintaxe utilizada para esse elemento de controle é a seguinte: $[\text{cláusula-condição}]$.
- A cláusula-iteração representa uma repetição do envio de uma mensagem é representada por um asterisco. Isso indica que a mensagem é emitida múltiplas vezes, geralmente para objetos receptores diferentes. Opcionalmente, a iteração pode apresentar o elemento *cláusula-iteração*, que define o limite inferior e o limite superior de uma sequência de repetições com uma variável de iteração. Por exemplo, a expressão $*[i=1..10]$ indica que a mensagem correspondente será enviada 10 vezes. A sintaxe utilizada para esse elemento de controle é $*[\text{cláusula-iteração}]$.

Alternativamente ao uso da *OCL*, tanto a cláusula de condição quanto a de repetição podem ser especificadas em pseudocódigo. Por exemplo:

- ***[para cada f em F] desenhar()**
- ***[enquanto x>0] transformar(x)**
- **[senha é válida] abrirJanelaPrincipal()**

Embora ainda sejam válidos na UML 2.0, os elementos de controle (cláusula de condição e cláusula de iteração) foram parcialmente substituídos por outros elementos definidos nessa última versão da linguagem de modelagem unificada, em particular os *fragmentos combinados* e os *operadores de interações*. Para mais detalhes desses novos elementos, ver [Seção 7.4](#).

7.1.1.2.3 Variável

O elemento *v* na sintaxe de uma mensagem corresponde a um identificador de variável que recebe o valor retornado pela operação a ser executada no objeto receptor. A existência desse elemento pode ser justificada por uma situação bastante comum em programação: a utilização de variáveis temporárias. Uma variável temporária é utilizada para armazenar certa informação que deve ser utilizada em um momento futuro. Em particular, podemos usar uma variável temporária para armazenar o valor de retorno de uma rotina para uso posterior. Pois bem, durante a modelagem do diagrama de interação, esse artifício de programação pode ser representado pelo elemento *v* que aparece na sintaxe de uma mensagem.

7.1.1.2.4 Nome e argumentos

O elemento *nome* na sintaxe corresponde à expressão de chamada de uma operação definida na classe do objeto receptor. Esse elemento pode conter uma lista (possivelmente vazia) de argumentos após o seu nome. Essa lista é delimitada por parênteses. Se houver mais de um parâmetro na lista, eles devem ser delimitados por vírgulas. Note que os argumentos definidos em um rótulo de mensagem correspondem aos da operação que deve ser executada. (Os detalhes da sintaxe completa para operações estão na [Seção 8.3](#).)

7.1.1.2.5 Exemplos

Alguns exemplos correspondentes à sintaxe de mensagens são apresentados a seguir. A [Seção 7.2](#) apresenta mais exemplos de mensagens utilizadas em diagramas de interação.

- Mensagem simples, sem cláusula alguma.
 1: **adicionarItem(item)**
- Mensagem com cláusula de condição. A mensagem somente é enviada se a condição associa for verdadeira.
 3 [a > b]: **trocar(a, b)**
- Mensagem com cláusula de iteração e com limites indefinidos.
 2 *: **desenhar()**
- Mensagem com cláusula de iteração e com limites definidos. A mensagem desenhar é enviada 10 vezes consecutivas.
 2 *[i:= 1..10]: **figuras[i].desenhar()**
- Mensagem aninhada com valor de retorno armazenado na variável *x*.
 Nesse exemplo, a operação recebe o argumento *e*.

7.1.2 Atores

Os atores que participam da realização de um caso de uso podem ser apresentados no diagrama de interação. Normalmente o ator primário (ver [Seção 4.1.2](#)) é o responsável por enviar a mensagem inicial que inicia a interação entre os objetos. Os atores são representados nos diagramas de

interação pela mesma notação gráfica utilizada no diagrama de casos de uso (ver [Seção 4.2](#)).

7.1.3 Objetos

Objetos em um diagrama de interação são representados da mesma forma que nos diagramas de objetos. De acordo com a UML 2.0, a expressão definida dentro do retângulo do objeto deve obedecer à sintaxe a seguir:

nome_objeto[seletor]: nome_classe ref ocorrência_interação

Na expressão anterior, temos os seguintes componentes:

1. O elemento `nome_objeto` especifica o nome de uma instância (objeto) envolvida na interação. Ess elemento é opcional, o que implica que objetos podem ser *anônimos* ou *nomeados*. Como o próprio nome diz, um objeto nomeado é aquele ao qual foi dado um nome. Normalmente, esse tipo de objeto é utilizado quando o mesmo precisa ser referenciado em mais de um lugar no diagrama. Nesse caso, o nome do objeto é separado do nome de sua classe por um sinal de dois pontos. Quando não há essa necessidade, os objetos anônimos são utilizados. A representação para objetos anônimos e nomeados é apresentada na [Figura 7-3](#).
2. O elemento `nome_classe` corresponde ao nome da classe dessa instância.
3. O elemento `seletor` é opcional; se for especificado, serve para fazer referência a uma instância de uma classe que está armazenada em uma coleção de objetos, como uma lista (para mais detalhes sobre coleções de objetos, ver [Seção 7.2.4](#)). Veja um exemplo na [Figura 7-3](#) e note a utilização de um índice (a letra `i`) para indicar o *i*-ésimo elemento da coleção.
4. Finalmente, o elemento `ocorrência_interação` representa uma *ocorrência de interação*. Este elemento que também é opcional, serve para fazer referência a outro diagrama de sequência, que mostra detalhes a respeito de como essa instância manipula as mensagens que recebe. Quando este elemento for utilizado, a palavra chave “`ref`” precisa ser posicionada a sua esquerda, conforme a sintaxe anterior. Para mais detalhes sobre o uso de ocorrências de interação, ver [Seção 7.4](#).

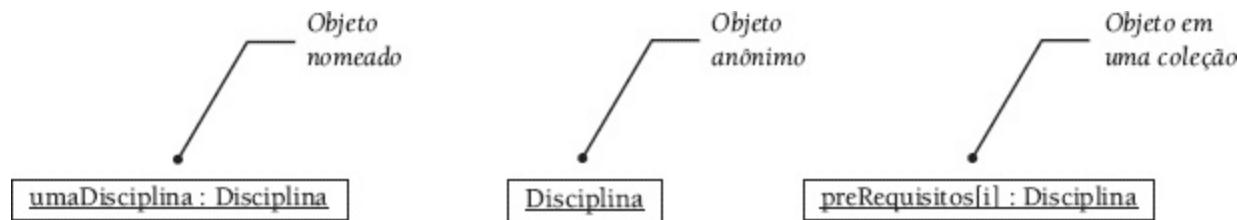


Figura 7-3: Representação para objetos e diagramas de sequência e de comunicação.

7.1.4 Classes

Na maioria das situações encontradas na prática, somente objetos são representados em um diagrama de interação. No entanto, quando a mensagem for endereçada para uma classe (e não para um objeto), a própria classe deve ser representada no diagrama. A participação de uma classe em um diagrama de interação se justifica pelo fato de a própria classe (em vez de um objeto) poder atender uma mensagem. Uma mensagem para uma classe dispara a execução de uma *operação estática* (operações estáticas são descritas na [Seção 8.3.1](#)). A representação de uma classe em um diagrama

de sequência é a mesma utilizada para objetos; porém, o nome da classe não é sublinhado (ver extrema-direita da Figura 7-1).

7.1.5 Coleções de objetos

Além de permitir a representação de objetos e classes, um diagrama de interação também pode apresentar *multiobjetos*. Um multiobjeto representa uma *coleção* de objetos de uma mesma classe. Um multiobjeto é ele próprio um objeto. Portanto, uma mensagem pode ser enviada para a coleção como um todo, em vez de para um único objeto da coleção. Um multiobjeto pode ser utilizado para representar:

1. o lado “muitos” de uma associação de conectividade “um para muitos”;
2. uma lista (temporária ou não) de objetos sendo formada em uma interação.

Um multiobjeto é representado graficamente por dois retângulos superpostos. O nome do multiobjeto é apresentado no retângulo que fica por cima e segue a mesma sintaxe utilizada para objetos. A convenção é que se utilize o mesmo nome da classe dos objetos componentes para nomear o multiobjeto, pois a superposição dos retângulos evita a confusão entre objeto e coleção. Como exemplo, considere o fragmento de diagrama de comunicação da Figura 7-4, no qual se utiliza um multiobjeto ItemPedido.

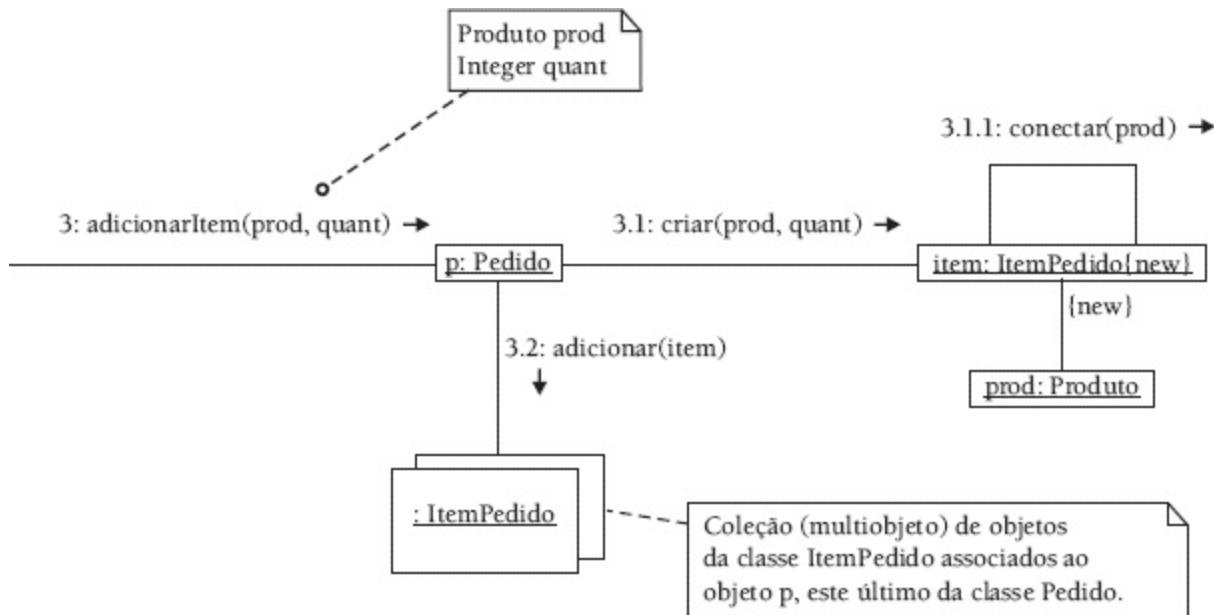


Figura 7-4: Exemplo de uso de um multiobjeto.

A UML não especifica precisamente que operações podem ser invocadas em um multiobjeto. Entretanto, multiobjetos são normalmente implementados por meio de alguma estrutura de dados que manipule uma coleção de objetos. Portanto, algumas mensagens típicas que podemos esperar que um multiobjeto aceite são as seguintes:

- Posicionar o cursor da coleção no primeiro elemento.
- Retornar o i -ésimo objeto da coleção.
- Retornar o próximo objeto da coleção.
- Encontrar um objeto de acordo com um identificador único.

- Adicionar um objeto na coleção.
- Remover um objeto na coleção.
- Obter a quantidade de objetos na coleção.
- Retornar um valor lógico que indica se há mais objetos a serem considerados.

Para dar uma visão mais clara das possibilidades de operações em um multiobjeto observe o [Quadro 7-1](#), no qual apresentamos uma interface da linguagem Java denominada List (detalhamos o conceito de *interface* na [Seção 8.5.4](#)). Note que diversas operações para manipulação de itens de uma lista (coleção) de objetos são declaradas.

Quadro 7-1: A interface List da linguagem Java apresenta operações típicas de um multiobjeto

```
public interface List<E> extends Collection<E> {
    E get(int index);
    E set(int index, E element);
    boolean add(E element);
    void add(int index, E element); E remove(int index);
    abstract boolean addAll(int index, Collection<? extends E>c);
    int indexOf(Object o);
    int lastIndexOf(Object o);
    ListIterator<E>listIterator();
    ListIterator<E>listIterator(int index);
    List<E> subList(int from, int to);
}
```

Um multiobjeto representa uma coleção de instâncias de certa classe. Em diagramas de interação, uma mensagem enviada a um multiobjeto é emitida à coleção, e não a cada instância individual. Entretanto, um símbolo de asterisco (*) pode ser adicionado perto do extremo de uma ligação onde há um multiobjeto para indicar que a mensagem está sendo enviada iterativa a cada instância da coleção. A [Figura 7-5](#) ilustra os dois casos aqui mencionados.

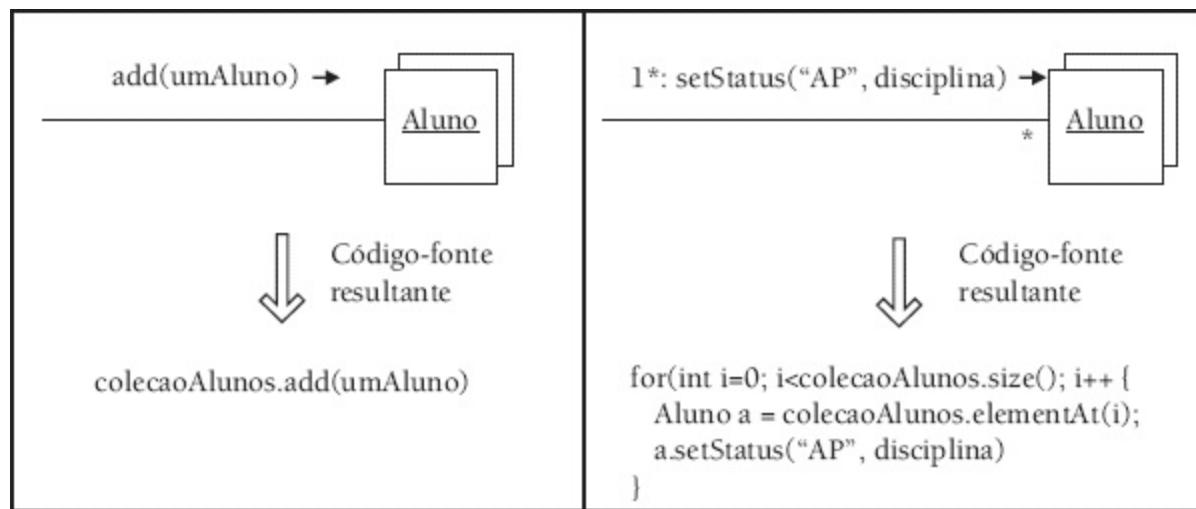


Figura 7-5: Mensagem enviada para um multiobjeto e para instâncias de um multiobjeto.

A versão 2.0 na UML introduziu uma maneira mais conveniente de fazer referência a um dos elementos de uma coleção de objetos (p. ex., um multiobjeto). Essa maneira é ilustrada na [Figura 7-6](#). Note que o nome do objeto é indexado para enfatizar que ele é um componente de uma coleção que contém diversos objetos.

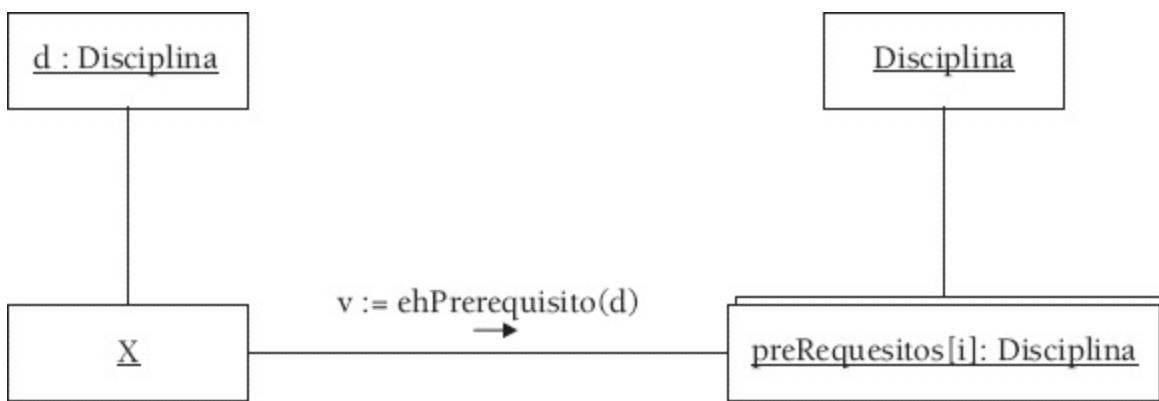


Figura 7-6: Forma para representar um objeto componente de uma coleção (multiobjeto).

Para finalizar esta seção, é importante notar que nem todas as ferramentas CASE ([Seção 2.6](#)) que dão suporte à UML fornecem o recurso de multiobjeto. Nesses casos, pode ser criado um tipo que representa explicitamente uma coleção de objetos de certa classe para ser usada na modelagem de alguma interação. Isso é feito nos diagramas apresentados na [Seção 7.7](#).

7.2 Diagrama de sequência

O objetivo do diagrama de sequência é apresentar as interações entre objetos na ordem temporal em que elas acontecem. Assim como os outros diagramas da UML, o diagrama de sequência possui um conjunto de elementos gráficos. Na construção de um diagrama de sequência, podemos utilizar as notações descritas na [Seção 7.1](#). Podemos também usar algumas notações particulares ao diagrama de sequência (ou seja, que não valem para o de comunicação). Algumas situações para as quais existem notações particulares no diagrama de sequência são as seguintes: linhas de vida, envio de mensagens, ocorrências de execução, criação e destruição de objetos. Descrevemos a notação e o significado desses elementos de notação particulares nesta seção (dicas para a construção propriamente dita de um diagrama de sequência são apresentadas na [Seção 7.5.3](#)).

7.2.1 Linhas de vida

Uma vez identificados, os objetos que participam da realização de um determinado caso de uso devem ser posicionados no diagrama de sequência correspondente. A notação gráfica utilizada para representar esses objetos é denominada linha de vida. Uma linha de vida é composta de duas partes, a *cabeça* e a *cauda*. A cabeça de uma linha de vida é representada pela mesma notação que utilizamos para objetos no *diagrama de objetos* (ver [Seção 5.3](#)).

A representação gráfica de uma linha de vida também possui uma *cauda*, que corresponde a uma linha vertical tracejada, conforme esquematizado na [Figura 7-7](#). Para o caso de atores representados em um diagrama de sequência, é comum pendurar uma linha tracejada nesses elementos, da qual partem as mensagens para o interior do sistema. Em um diagrama de sequência, normalmente o ator é posicionado na extrema esquerda do diagrama de sequência (ver também a [Figura 7-1](#)).

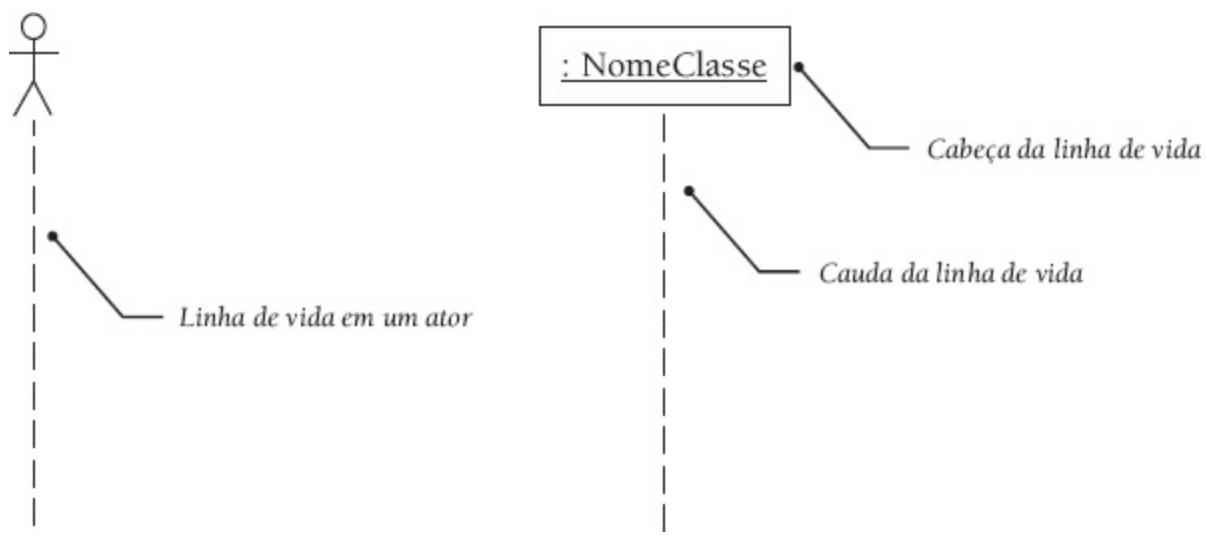


Figura 7-7: Representação de linhas de vida em um diagrama de sequência.

A ordem horizontal utilizada para posicionar os objetos no diagrama de sequência não tem nenhum significado predefinido. Entretanto, a ordem normalmente utilizada é a seguinte (da esquerda para a direita): ator primário, objeto(s) de fronteira com o(s) qual(is) o ator interage, objeto(s) de controle, objetos de entidade e atores secundários. Essa ordem de disposição facilita a leitura do diagrama.

7.2.2 Mensagens

A notação para uma mensagem em um diagrama de sequência é uma flecha (geralmente desenhada na horizontal) ligando uma linha de vida a outra. O objeto do qual parte a seta é aquele que está enviando a mensagem (objeto remetente). O objeto para o qual a seta aponta é aquele que está recebendo a mensagem (objeto receptor). O formato da “ponta” da seta indica o tipo de mensagem que está sendo enviada (ver [Seção 7.1.1](#)). Os formatos possíveis em um diagrama de sequência estão ilustrados na [Figura 7-8](#). O *rótulo da mensagem* (descrito na [Seção 7.1.2](#)) é posicionado acima dessa linha.

→	Mensagem síncrona
→	Mensagem assíncrona
←-----	Mensagem de retorno
«create» -----→	Mensagem de criação de objeto

Figura 7-8: Notações para os diversos tipos de mensagem em um diagrama de sequência.

A passagem do tempo é percebida no diagrama de sequência observando-se a direção vertical no sentido de cima para baixo. Quanto mais abaixo uma mensagem aparece no diagrama, mais tarde no tempo ela foi enviada.

A [Seção 7.1.1](#) apresenta o significado de uma *mensagem reflexiva*. A [Figura 7-9](#) ilustra o uso desse tipo de mensagem em um diagrama de sequência. Pode-se notar que uma mensagem reflexiva é

representada por uma seta saindo e retornando para o próprio objeto.

Opcionalmente, o texto do cenário de uso pode ser adicionado do lado esquerdo do diagrama de sequência, com o objetivo de esclarecer as trocas de mensagens. Esse texto pode ser definido por *notas explicativas* (que fazem parte dos mecanismos de uso geral da UML e são descritas na [Seção 3.2](#)). Se isso for feito, o modelador deve tentar alinhar verticalmente as partes da descrição com as mensagens correspondentes.

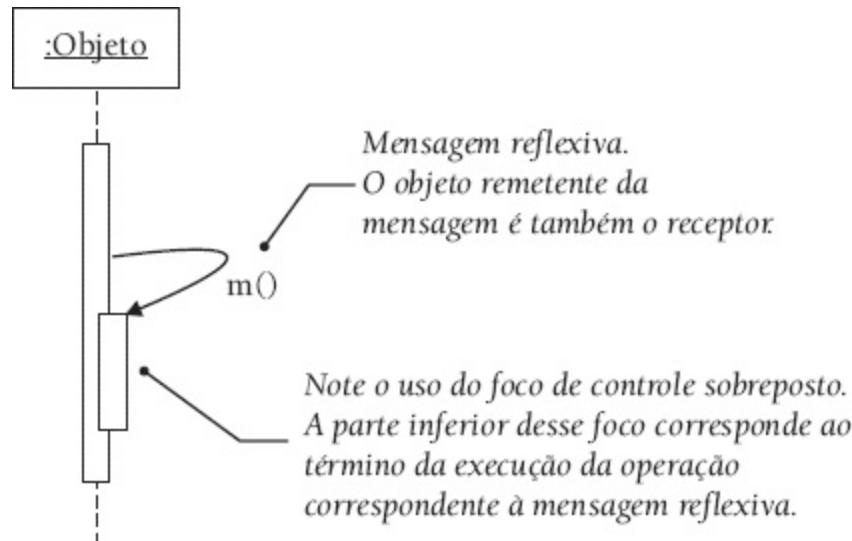


Figura 7-9: Mensagem reflexiva em um diagrama de sequência.

7.2.3 Ocorrências de execução

Uma ocorrência de execução representa o tempo em que o objeto está ativo, ou seja, o tempo em que ele realiza alguma operação. Graficamente, ocorrências de execução correspondem a blocos retangulares posicionados sobre a linha de vida de um objeto. O topo de uma ocorrência de execução coincide, no receptor, com o recebimento de uma mensagem. A parte de baixo dessa ocorrência de execução corresponde ao término de uma operação realizada pelo objeto.

O modelador pode optar por não utilizar ocorrências de execução nos objetos presentes em certo diagrama de sequência. Por outro lado, a utilização de ocorrências de execução muitas vezes torna desnecessário o uso de mensagens de retorno. Isso porque o final de uma ocorrência de execução corresponde ao retorno do fluxo de controle para o emissor da mensagem (no caso de mensagens síncronas).

7.2.4 Criação e destruição de objetos

Duas operações frequentes em um SSOO são a criação e destruição de objetos. A criação de um objeto é o momento em que esse objeto passa a existir no sistema e passa a realizar suas responsabilidades. Por exemplo, em nosso estudo de caso, a realização do caso de uso Abrir Turma tem o potencial de criar objetos da classe Turma, conforme vimos na [Seção 5.7](#). Já a destruição de um objeto é o momento no qual este objeto deixa de existir no sistema.

Algumas linguagens de programação orientadas a objetos normalmente definem rotinas ou operadores especializados para a criação e/ou destruição de objetos. Outras possuem mecanismos para destruição automática de objeto, liberando o programador dessa tarefa. De qualquer forma, as operações de criação e destruição de objetos são importantes o suficiente para terem notações específicas no diagrama de sequência. Descrevemos essas notações a seguir.

Se o objeto existe desde o início da interação (ou seja, se não é criado durante a interação sendo modelada), ele deve ser posicionado no topo do diagrama. No entanto, pode ser que a própria interação crie um ou mais objetos de certa classe. Nesse caso, a posição *vertical* de um objeto em um diagrama de sequência indica o momento no qual ele é criado (instanciado) e começa a participar da interação. Ou seja, o retângulo que representa o objeto deve ser posicionado mais abaixo no diagrama. A instanciação de um objeto é sempre requisitada por outro objeto participante da interação por uma mensagem. A mensagem de criação pode ser rotulada de duas maneiras alternativas: (1) com o estereótipo <<create>> ou com o nome do método construtor que será ativado (descrevemos métodos construtores na [Seção 8.3.4](#)). Além disso, a flecha da mensagem é pontilhada e aponta para o objeto em vez de para a linha de vida. Na [Figura 7-10](#) apresentamos um exemplo da notação de criação de objetos. Repare que, em vez do estereótipo <<create>>, é possível especificar o rótulo da mensagem de tal forma que ele siga a sintaxe de instanciação específica de alguma linguagem de programação. Por exemplo, se o objeto a ser criado é da classe Inscricao do SCA, e se a linguagem empregada é Java, o rótulo da mensagem de criação pode ser `Inscricao(aluno)`, pois, nessa linguagem de programação, construtores possuem o mesmo nome da classe em que são definidos.

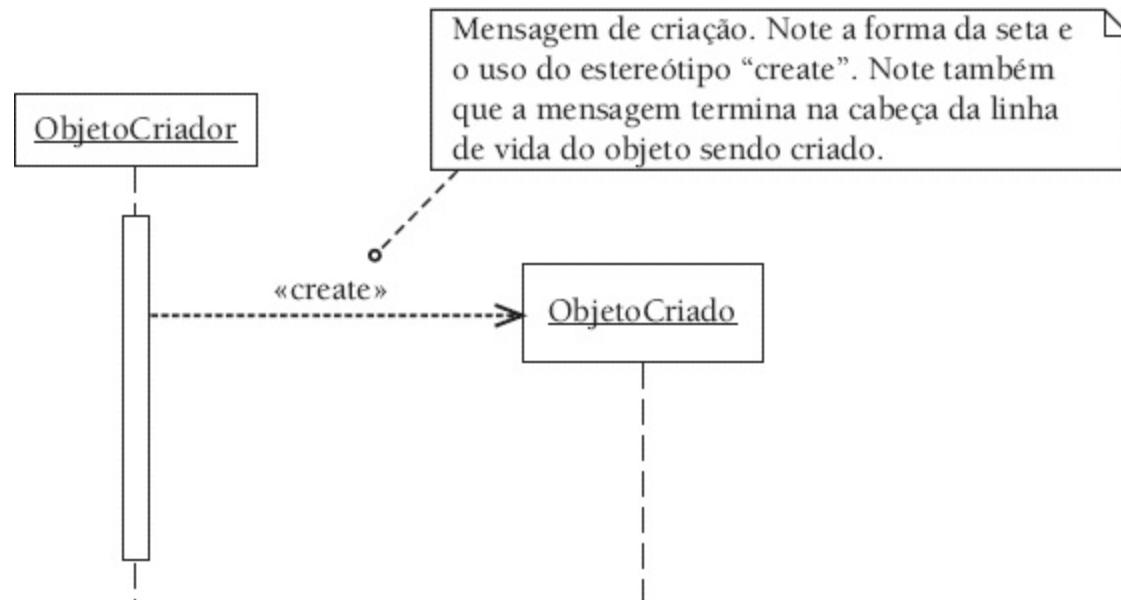


Figura 7-10: Criação de objeto em um diagrama de sequência.

Um diagrama de sequência pode mostrar também a destruição de objetos no decorrer de uma interação. Um objeto normalmente é destruído quando não é mais necessário na interação. O símbolo X na parte de baixo da linha de vida de um objeto significa que ele está sendo destruído. Além disso, a mensagem de destruição é rotulada com o estereótipo <<destroy>>. A [Figura 7-11](#) fornece um exemplo da notação para destruição de objetos.

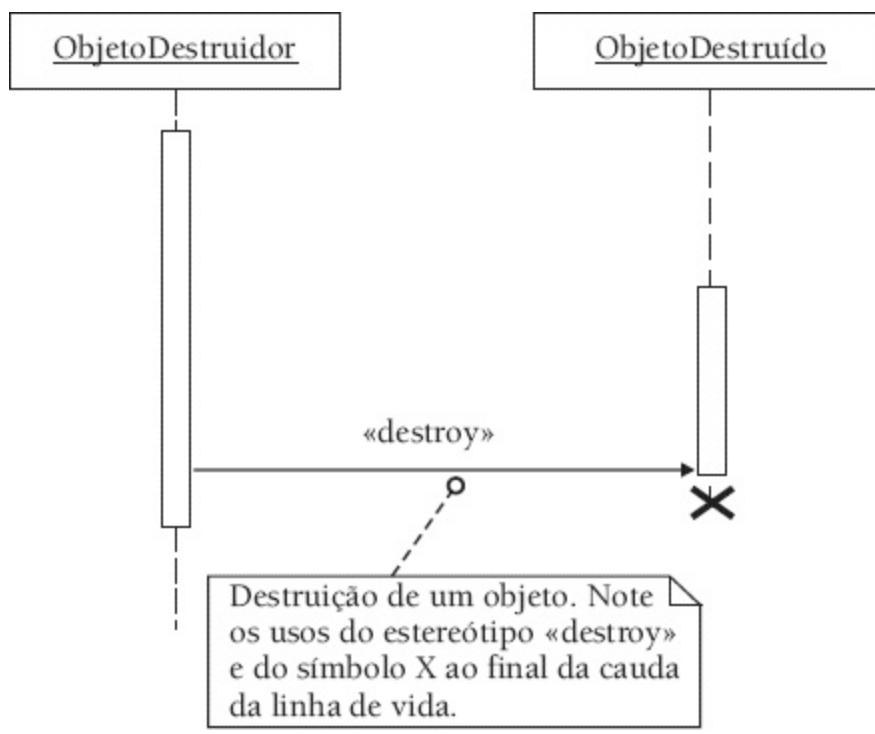


Figura 7-11: Destrução de objeto em um diagrama de sequência.

7.3 Diagrama de comunicação

O diagrama de comunicação mostra os objetos relevantes para a realização de um caso de uso (ou seu cenário), assim como as ligações entre os mesmos. Sendo um tipo de diagrama de interação, o diagrama de comunicação mostra as mensagens trocadas entre objetos. A [Figura 7-2](#) exibe um esquema genérico de um diagrama de comunicação, que permite perceber como, estruturalmente, um diagrama de comunicação é bastante semelhante a um diagrama de objetos ([Seção 5.3](#)). A diferença é que são adicionadas setas e rótulos de mensagens nas ligações entre esses objetos.

Objetos e classes podem aparecer em um diagrama de comunicação, assim como em um diagrama de sequência, e a notação para esses elementos é a mesma utilizada neste último. Da mesma forma, pode haver objetos nomeados e anônimos, multiobjetos e referências para elementos de uma coleção.

Um elemento particular do diagrama de comunicação é a *ligação* entre objetos. As ligações são representadas graficamente por linhas entre objetos no diagrama de comunicação e correspondem a relacionamentos entre os objetos: associações, agregações, composições ou dependências (uma descrição desses relacionamentos é apresentada na [Seção 5.2.2](#)).

Outra diferença dos diagramas de comunicação e de sequência diz respeito à forma de “ler” a ordem de envio das mensagens. Um diagrama de comunicação mostra as interações entre objetos de maneira semelhante ao diagrama de sequência. No diagrama de comunicação, não há como saber a ordem de envio das mensagens a não ser pelas expressões de sequência. Ao contrário, no diagrama de sequência, a posição vertical das mensagens permite deduzir a ordem na qual elas são enviadas (embora a utilização de expressões de sequência para facilitar a leitura e eliminar eventuais ambiguidades em um diagrama de sequência também seja possível). No entanto, o diagrama de comunicação não mostra o tempo como uma dimensão separada. Para compensar isso, todas as mensagens nesse diagrama devem conter obrigatoriamente *expressões de sequência* (ver [Seção 7.1.1.2.1](#)). Sendo assim, a ordem de envio de mensagens em um diagrama de comunicação é deduzida a partir das expressões de sequência. A [Figura 7-12](#) apresenta outro exemplo de diagrama de

comunicação com expressões de sequência.

O rótulo de uma mensagem também pode mostrar um elemento de controle (de condição ou de iteração), conforme descrito na Seção 7.1.2.2. O sentido da mensagem é indicado por uma seta posicionada próxima ao rótulo da mensagem. Essa seta aponta para o objeto receptor da mensagem. Como exemplo, considere a [Figura 7-13](#).

É comum na modelagem de interações surgirem situações em que precisamos representar a criação e destruição de objetos ou ligações entre os mesmos. Com esse objetivo, existem restrições predefinidas na UML que podem ser utilizadas para representar a criação e destruição de objetos, ou de ligações entre objetos em um diagrama de comunicação. Essas restrições são `{new}`, `{destroyed}` e `{transient}`. Veja a descrição a seguir.

Expressões de sequência permitem definir a ordem de envio das mensagens em um diagrama de comunicação.

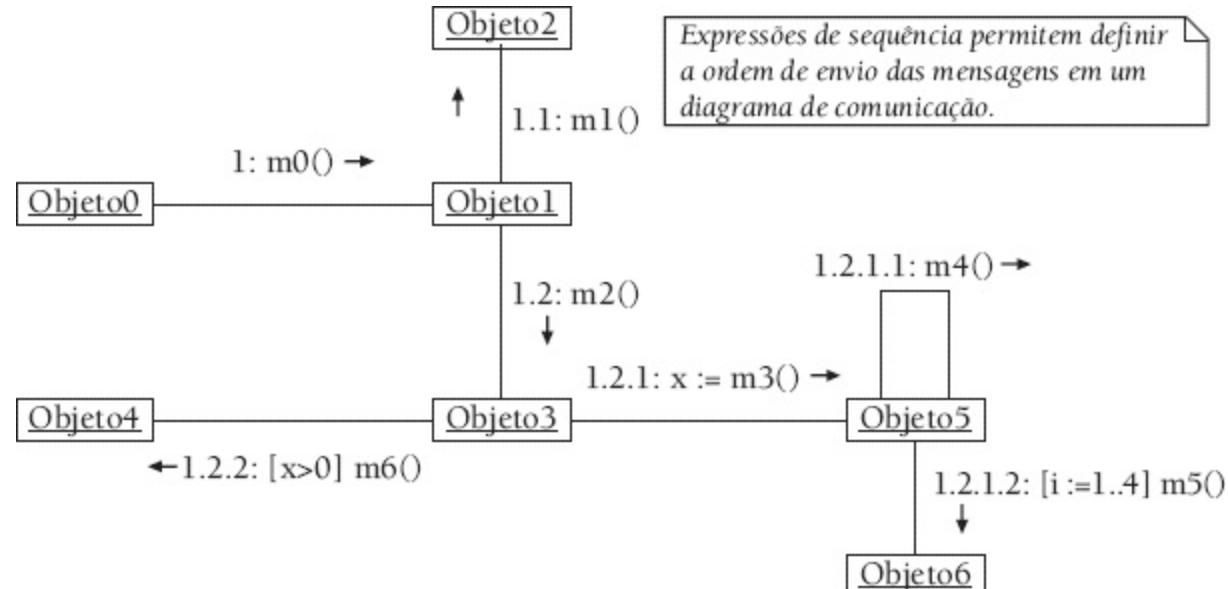


Figura 7-12: Expressões de sequência em um diagrama de comunicação.

Mensagem com expressão de sequência e guarda. Neste exemplo, a mensagem `m6` é enviada somente se `x` for maior que zero.

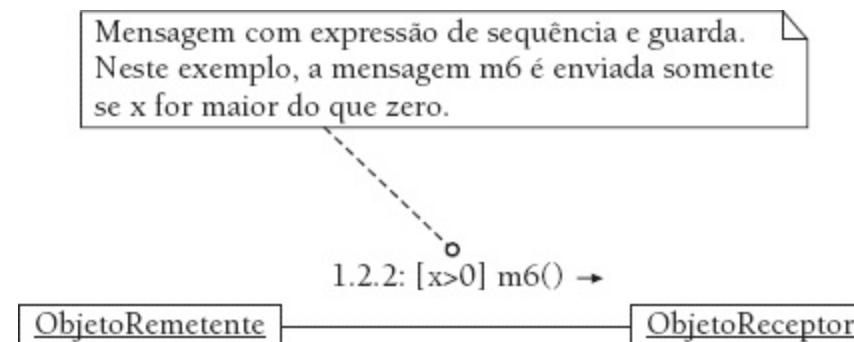


Figura 7-13: Mensagens em um diagrama de comunicação.

1. Objetos ou ligações criados durante a interação podem ser rotulados com a restrição `{new}`.
2. Objetos ou ligações destruídos durante a interação podem ser rotulados com a restrição `{destroyed}`.
3. Objetos ou ligações destruídos e criados durante uma mesma interação podem ser rotulados co-

a restrição {transient}.

Graficamente, a restrição é posicionada à direita no interior do retângulo do objeto. Para uma ligação, a restrição é posicionada próxima à linha correspondente à ligação. A Figura 7-14 ilustra um exemplo de diagrama de comunicação que usa a restrição {new}. Nesse diagrama, o uso dessa restrição na ligação entre os objetos item e GradeDisponibilidade enfatiza que essa ligação está sendo criada durante a interação representada no diagrama.

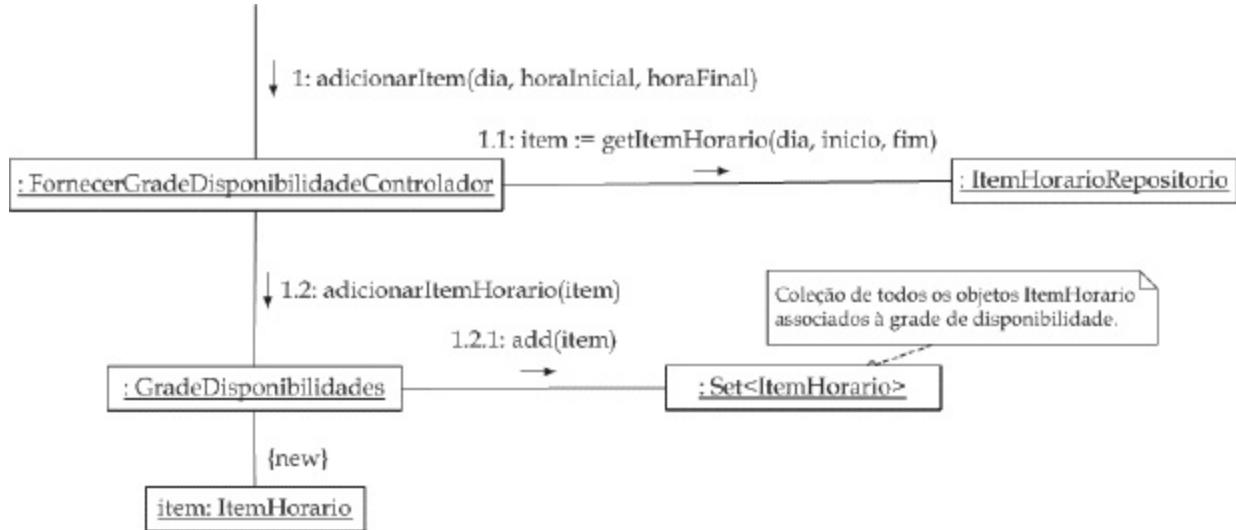


Figura 7-14: Uso da restrição {new} em um diagrama de comunicação.

7.4 Modularização de interações

A UML 2.0 introduziu diversos novos elementos gráficos para construção modular de diagramas de interação: quadros de interação, fragmentos combinados, referências, operadores etc. Nesta seção descrevemos esses elementos. Note que, embora os exemplos apresentados nesta seção sejam relativos a diagramas de sequência, quadros também podem ser utilizados na construção de diagramas de comunicação.

7.4.1 Quadros

Um *quadro de interação* (tradução para *interaction frame*) serve para encapsular um diagrama de sequência. Um quadro fornece um contexto ou fronteira no interior do qual podemos posicionar os elementos de um diagrama de sequência, como linhas de vidas e mensagens. Graficamente, um *quadro* é um retângulo. A notação gráfica para um quadro definida pela UML 2.0 é apresentada na Figura 7-15.

Um diagrama (ou um nome de um diagrama) é posicionado no interior do quadro.

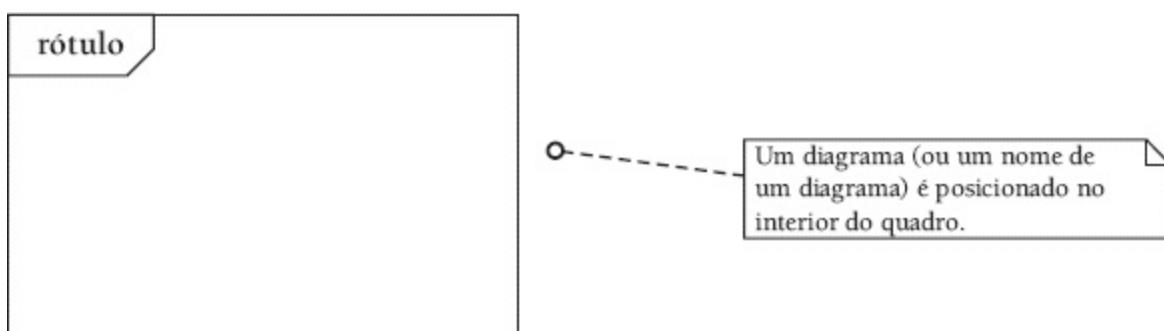


Figura 7-15: Notação para um quadro na UML.

Conforme podemos perceber na [Figura 7-15](#), na parte superior esquerda de um quadro há uma espécie de “orelha” na forma de um pentágono. No interior desse pentágono devemos definir um rótulo que pode ser utilizado com três diferentes objetivos pelo modelador. Enumeramos esses objetivos abaixo. Nos próximos parágrafos desta seção, damos detalhes acerca de cada um desses objetivos.

- Para dar um nome ao diagrama que aparece dentro do quadro.
- Para fazer referência a um diagrama definido separadamente.
- Para definir o fluxo de controle da interação.

Vamos começar detalhando o primeiro objetivo (*dar um nome ao diagrama dentro do quadro*). A partir da UML 2.0, todo diagrama de interação pode ser posicionado dentro de um quadro. Se isso acontecer, a orelha do quadro deve conter uma expressão da forma *TipoDiagrama NomeDiagrama*. Nessa expressão, o elemento *NomeDiagrama* é o nome dado ao diagrama cuja interação está definida dentro do quadro. Qualquer diagrama de interação pode ser posicionado dentro de um quadro. Para identificar qual é o tipo do diagrama contido no quadro, utilizamos o elemento *TipoDiagrama* da expressão. Os valores possíveis para este elemento são os seguintes: *sd* (para diagramas de sequência), *comm* (para diagramas de comunicação) e *activity* (para diagramas de atividade). O objetivo de dar um nome a um diagrama de interação é permitir que este seja referenciado durante a definição de outros diagramas.

O segundo objetivo da orelha de um quadro é *fazer referência a um diagrama definido separadamente*. De fato, além de permitir que seja atribuído um nome a um diagrama, a orelha de um quadro também pode indicar que este corresponde a uma *ocorrência de interação*. Uma *ocorrência de interação* é uma interação que é referenciada por outra. Podemos utilizar ocorrências de interação para fatorar interações comuns a vários diagramas de sequência em um único quadro e reutilizar esse quadro diversas vezes. Graficamente, uma ocorrência de interação é um quadro rotulado com a palavra-chave *ref*. O nome da interação é posicionado no interior desse quadro rotulado com a palavra-chave *ref* à sua esquerda. Essa palavra-chave indica que uma interação está sendo referenciada dentro de outra interação. Na [Figura 7-16](#), observa-se que o quadro rotulado como *InteraçãoA* faz referência a duas outras interações. Essas, por sua vez, são representadas como ocorrências de interação, estão indicadas no diagrama como *InteraçãoB* e *InteraçãoC*.

A ocorrência de interação implica que um quadro pode conter outros quadros em seu interior, ou seja, *subquadros*. Na verdade, a UML 2.0 define dois tipos de subquadros. Um deles é a ocorrência de interação, à qual nos referimos há pouco. O outro tipo corresponde ao chamado *fragmento combinado*. Este elemento é utilizado para alcançar o terceiro objetivo enumerado anteriormente, *definir o fluxo de controle da interação*. Este elemento fornece ao modelador a capacidade de

definir lógica procedural (fluxo de controle) nos seus diagramas de interação. Graficamente, um fragmento combinado corresponde a uma ou mais interações (sequências de mensagens) encapsuladas em um quadro. O operador da interação é representado dentro do pentágono do quadro correspondente ao fragmento combinado. Os operandos do fragmento combinado (que são as interações) são posicionados dentro do quadro. Se houver mais de um operando, eles são separados por uma linha tracejada. Veja a notação definida pela UML 2.0 para um fragmento combinado na [Figura 7-17](#).

InteraçãoB e InteraçãoC são nomes de diagramas que apresentam mensagens trocadas entre os objetos Objeto1 e Objeto2. Note que os quadros correspondentes são rotulados com “ref” e posicionados sobre as linhas de vida dos objetos.

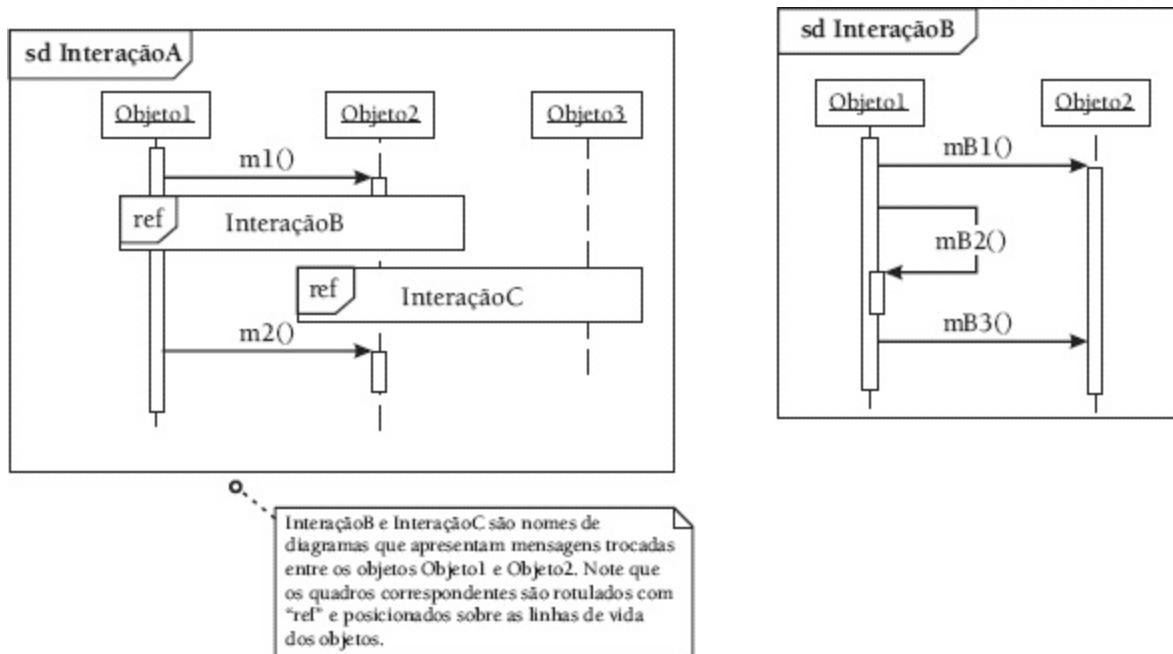


Figura 7-16: Ocorrências de interação.

Aqui entra o operador do fragmento combinado (loop, alt, opt etc.)

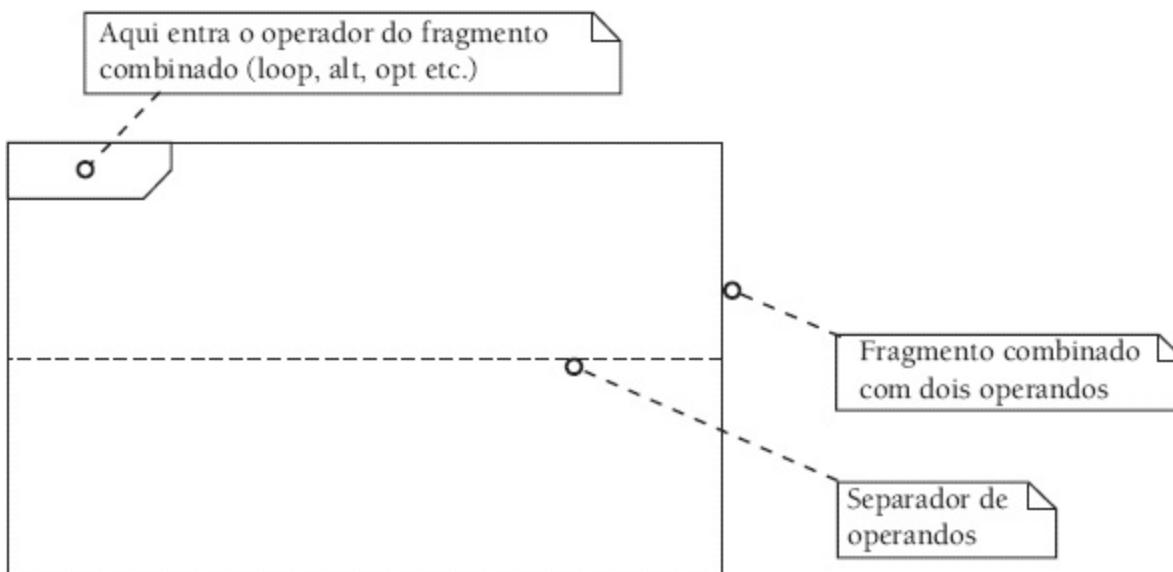


Figura 7-17: Notação da UML para um fragmento combinado.

Internamente, um fragmento combinado é composto de um ou mais *operандos da interação*, de

zero ou mais *condições de guarda*, de um *operador de interação*. Um *operando* em um fragmento combinado corresponde a uma sequência de mensagens. Essa sequência é executada (ou seja, as mensagens correspondentes são enviadas) somente sob circunstâncias específicas. A forma de execução dessa sequência é definida pelo *operador de interação* e pelas *condições de guarda* utilizadas. Uma *condição de guarda*, também chamada de *restrição da interação*, é uma expressão condicional (de valor verdadeiro ou falso) que é associada a um operando da interação em um fragmento combinado. Uma condição de guarda impõe uma restrição da execução do operando ao qual está associada. Essa restrição deve ser suficiente para definir se o operando (interação) correspondente deve ou não ser executado. Se a condição tiver valor verdadeiro, o operando é executado; do contrário, não é. A definição de uma condição de guarda é opcional, de tal forma que um operando sem condição de guarda é executado sempre. Um *operador de interação* define a semântica de um fragmento combinado e determina como usar os operandos da interação contidos nesse fragmento combinado. Em particular, a quantidade de operandos que podem ser definidos em um fragmento combinado depende do operador associado a esse fragmento. Uma descrição completa de todos os operadores de interação definidos pela UML 2.0 está fora do escopo desse livro e pode ser encontrada em OMG (2005). Alguns desses operadores de interação definidos na UML 2.0 são os seguintes: *alt*, *opt*, e *loop*. Descrevemos seus significados a seguir.

O operador *alt* modela construções procedimentais do tipo *se-então-senão*. Dos operandos definidos com esse operador, apenas um é executado. Cada operando é avaliado com base em uma condição de guarda associada ao mesmo. Uma condição de guarda é uma expressão lógica e, se predefinida, pode ser utilizada. Esse guarda avalia para VERDADEIRO quando todas as demais guardas avaliam para FALSO. Os operandos são separados um do outro dentro do quadro com uma linha tracejada. Veja um exemplo na [Figura 7-18](#).

O operador *opt* modela construção procedural do tipo *se então*. Esse operador é similar ao operador *alt*. A diferença é que o operador *opt* está associado a apenas um operando. Se a condição de guarda for verdadeira, o operando correspondente é executado. Do contrário, esse operando é “pulado”. Veja um exemplo na [Figura 7-19](#).

O operador *loop* serve para representar que uma interação (operando) deve ser realizada zero ou mais vezes. Após o nome do operador, é possível (opcional) representar os limites mínimo e máximo para definir a quantidade de iterações. A sintaxe do operador *loop* é a seguinte: *'loop[''<minint> [, ''<maxint>] ''']'*. A [Figura 7-20](#) fornece um exemplo de diagrama de sequência no qual essa sintaxe é utilizada. Mais uma vez, os elementos dessa sintaxe que estão cercados por colchetes são opcionais, o que significa que podemos utilizar apenas o operador. Nessa sintaxe, temos:

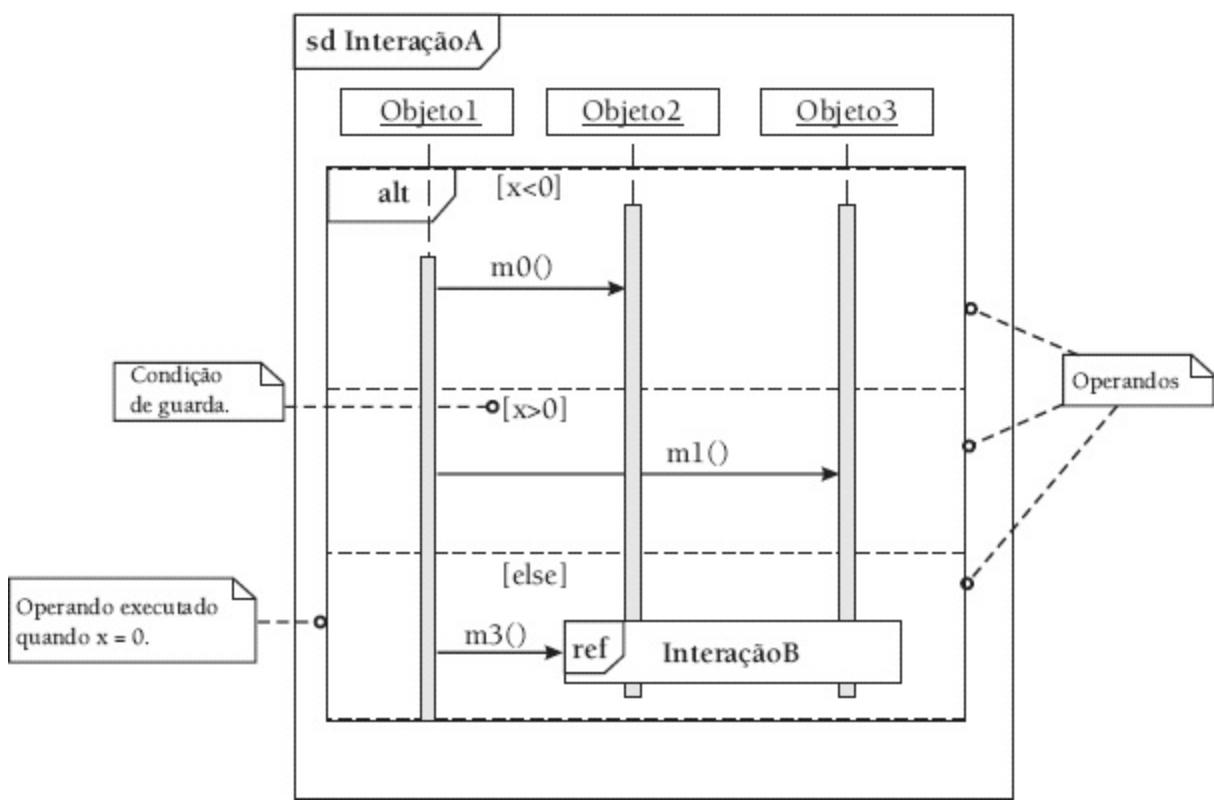


Figura 7-18: Utilização do operador de interação *alt*.

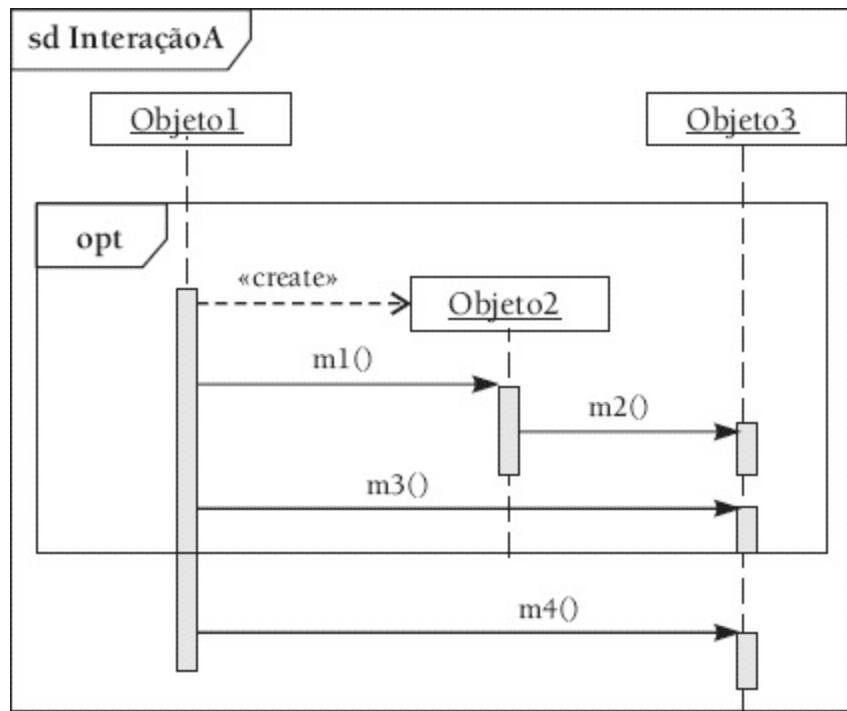


Figura 7-19: Utilização do operador de interação *opt*.

1. O limite mínimo <minint> é um número inteiro não negativo.
2. O limite máximo <maxint> pode ser representado de duas maneiras: (1) com um número inteiro maior ou igual a <minint>; (2) pelo símbolo ‘*’, que significa uma quantidade de iterações sem limite superior.

Na sintaxe do operador *loop*, se somente o componente <minint> é definido, isso significa que <minint> é igual a <maxint>. Por outro lado, se somente é utilizado o operador (sem os limites

mínimo e máximo), isso significa uma quantidade de iterações com limite inferior ou igual a zero e sem limite superior.

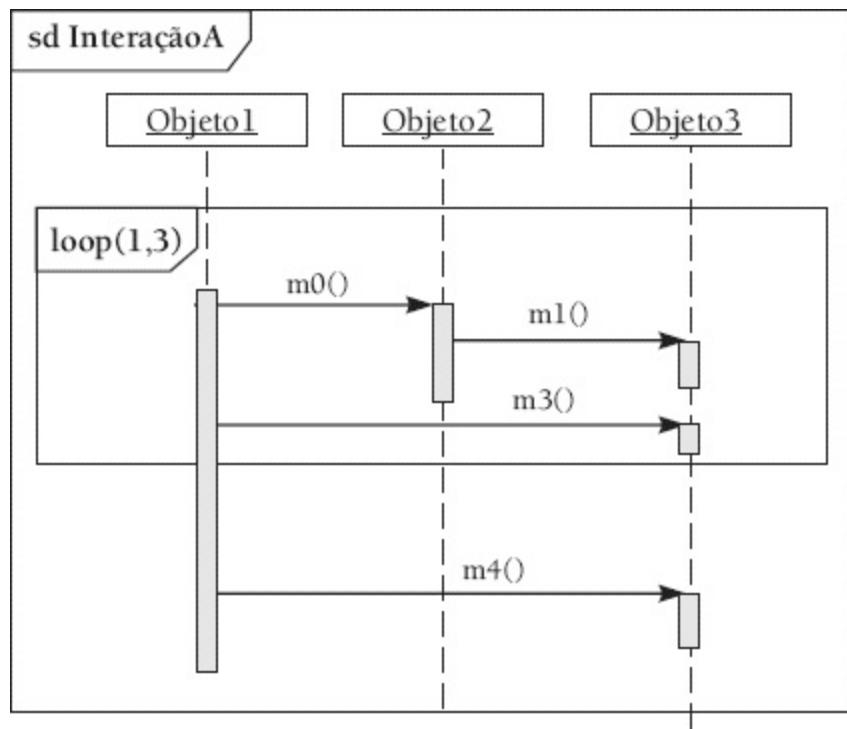


Figura 7-20: Utilização do operador de interação *loop*.

7.4.2 Diagrama de visão geral da interação

Outra abordagem para modularizar a construção de diagramas de interação introduzida pela UML 2.0 é o *diagrama de visão geral da interação* (*interaction overview diagram*). Primeiramente, note que os diagramas de interação preexistentes (o diagrama de sequência e o de comunicação) servem para representar as interações que ocorrem em um cenário de caso de uso. Por outro lado, como seu próprio nome deixa transparecer, esse novo diagrama introduzido na UML tem o objetivo de dar uma *visão geral* dos diversos cenários de um caso de uso. Esse diagrama é um tipo especial de *diagrama de atividade*. Estudamos o *diagrama de atividade* em mais detalhes no [Capítulo 10](#). O leitor é convidado a consultar aquele capítulo para detalhes acerca da notação do diagrama de atividade. Em nosso estudo de caso, que continuamos na [Seção 7.7](#), fornecemos um exemplo de utilização do diagrama de visão geral da interação.

7.5 Construção do modelo de interações

Nas [Seções 7.2, 7.3](#) e [7.4](#), descrevemos os principais elementos de notação dos diagramas de interação. Agora passamos a discutir detalhes acerca da construção do modelo de interações. Na [Seção 7.5.1](#), discutimos a relação entre os conceitos de *mensagem* e *responsabilidade*. Nas [Seções 7.5.2](#) e [7.5.3](#), apresentamos princípios de projeto de software orientado a objetos que são fundamentais para a construção do modelo de interações por meio da correta atribuição de responsabilidades. Finalmente, na [Seção 7.5.4](#) descrevemos um procedimento para construção do modelo de interações.

7.5.1 Responsabilidades e mensagens

Na [Seção 1.2.2](#), apresentamos o conceito de mensagem de forma conceitual. Na [Seção 5.4.3](#), por sua vez, mencionamos que a modelagem inicial de classes pode ser realizada por meio de técnicas de identificação baseadas em responsabilidades. Nesta seção, apresentamos a relação estreita que existe entre os conceitos de *responsabilidade* e de *mensagem* no contexto da modelagem de interações.

Quando um objeto precisa de ajuda para realizar alguma de suas responsabilidades durante a realização de um caso de uso, esse objeto deve enviar mensagens a outros objetos. Portanto, o fato de um objeto precisar de ajuda indica a necessidade de ele enviar mensagens para outros. Esse aspecto é fundamental na identificação de mensagens.

Por outro lado, uma mensagem implica a existência de uma operação no objeto receptor que será executada quando aquela mensagem for enviada. Portanto, na construção de diagramas de interação, quando o modelador especifica mensagens de um objeto a outro, ele está na verdade especificando operações que as classes devem ter.

Uma mensagem implica a existência de uma operação na classe do objeto receptor. A resposta ao recebimento de uma mensagem é a execução dessa operação.

Por exemplo, considere a [Figura 7-21](#), que ilustra um fragmento de diagrama de sequência com uma mensagem sendo enviada a um objeto da classe Usuário. A mensagem indica que essa classe define uma operação denominada validar. Além disso, essa operação possui dois argumentos, id e senha, ambos do tipo String. Esse exemplo envolve uma mensagem contida em um diagrama de sequência, mas o mesmo vale para diagramas de comunicação.

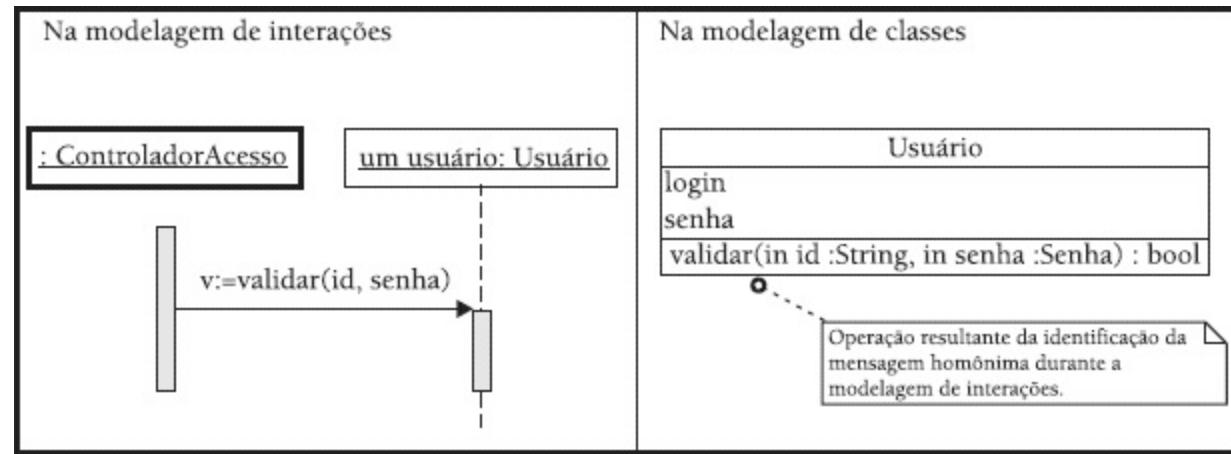


Figura 7-21: A classe Usuário possui uma operação validar, com dois argumentos: id e senha.

Pode-se concluir que a existência de responsabilidades com as quais um objeto não consegue cumprir sozinho implica indiretamente a necessidade de operações definidas em seus colaboradores. O modelador deve levar isso em consideração no momento da construção dos diagramas de interação. Sempre que uma mensagem for enviada a um objeto em um diagrama de interação, o modelador deve questionar se tal objeto tem condições de responder à mensagem sozinho ou se precisa enviar mensagens a outros objetos.

7.5.2 Coesão e acoplamento

De acordo com a seção anterior, quando definimos uma mensagem no modelo de interações, estamos atribuindo uma responsabilidade a uma classe. Por conta disso, podemos entender a modelagem de interações como um processo cujo objetivo final é decompor as responsabilidades do sistema e alocá-las a classes. Dado um conjunto de N responsabilidades de um sistema, uma possibilidade é criar uma única classe no sistema para assumir todas as N responsabilidades. Outra ideia é criar N classes no sistema, sendo atribuída a cada uma delas uma das N responsabilidades. Certamente as duas alternativas anteriores são absurdas do ponto de vista prático. Mas, entre esses dois extremos, há muitas maneiras possíveis de alocar responsabilidades. Como podemos saber quais delas são melhores que outras?

Conforme já mencionamos no [Capítulo 5](#), a tarefa de identificação de classes e de alocação de responsabilidades é bastante complexa, para a qual não há uma única maneira de realizar. O bom resultado dessa tarefa depende, entre outros fatores, da experiência do modelador e da correta aplicação de alguns princípios básicos de desenvolvimento de software. A *coesão* e o *acoplamento* são dois desses princípios. O acoplamento e a coesão servem como guias para a correta atribuição de responsabilidades a classes. A seguir, descrevemos esses importantes princípios de projeto orientado a objetos.

A **coesão** é uma medida do quanto fortemente relacionadas e focalizadas são as responsabilidades de uma classe. Durante o projeto das classes de um SSOO, é extremamente importante que o projetista procure construir soluções com alta coesão. Isso equivale a fazer com que as responsabilidades atribuídas a cada classe sejam altamente relacionadas entre si. Em outras palavras, o projetista precisa definir classes de tal forma que cada uma delas tenha alta coesão, para que capture uma única abstração e, por consequência, seja mais reutilizável.

Um sintoma típico de uma classe com baixa coesão (o que é ruim para a qualidade de um sistema) é o fato de a mesma apresentar dois ou mais grupos de atributos (ou de operações), sendo que há alto grau de correlação *dentro de cada grupo*, mas baixo grau de correlação *entre os grupos*. Essa característica é um indício de que, em um projeto de melhor qualidade, deveria haver duas ou mais classes, cada uma contendo os grupos de atributos (ou de operações) com alta correlação entre si.

Além de ser menos reutilizáveis, classes com baixa coesão normalmente são mais complexas do que deveriam ser. Por isso, tais classes são menos inteligíveis e de manutenção (modificação) mais complicada.

Como exemplo, considere a [Figura 7-22](#), que apresenta uma possível versão de projeto da classe Turma, na qual há cinco operações (detalhes acerca da sintaxe da UML para representação de operações são apresentados na [Seção 8.3](#)). Considere que duas dessas operações (`haChoqueHorarios` e `inscrever`) têm a ver com as responsabilidades intrínsecas ao conceito de turma: uma turma deve ser responsável (1) por registrar a inscrição de um aluno e (2) por verificar se possui choque de horários com outra turma. Por outro lado, considere que as três demais operações (`inserir`, `excluir` e `atualizar`) foram criadas para permitir refletir em um mecanismo de armazenamento persistente as alterações feitas sobre um objeto dessa classe. Por exemplo, quando um objeto Turma é criado e precisa ser armazenado em um mecanismo de persistência, uma mensagem pode ser enviada a ele para ativar a operação `inserir`. Repare que os propósitos desses dois grupos de operações são diferentes. No primeiro grupo, o propósito é permitir que a classe cumpra com responsabilidades relativas ao domínio ao qual pertence. No segundo, a meta é permitir a movimentação de dados acerca de uma turma de e para um mecanismo de armazenamento. Dentro de cada grupo, a similaridade (ou

correlação) entre operações é maior do que a obtida quando comparamos duas operações pertencentes a grupos distintos. O resultado é que a classe `Turma` assim projetada contém dois grupos de operações com pouca relação entre si, exceto pelo fato de serem operações aplicáveis ao mesmo conceito do domínio do problema (p. ex., uma turma). Esse é um sintoma de que a coesão da classe `Turma` não está em um nível adequado. Uma possível solução neste caso é refazer o projeto, dessa vez com a criação de duas classes, situação em que cada uma delas passa a conter um dos grupos de operações. Como consequência dessa mudança no projeto, obtemos duas classes, cada qual com coesão maior do que a classe `Turma` projetada originalmente.

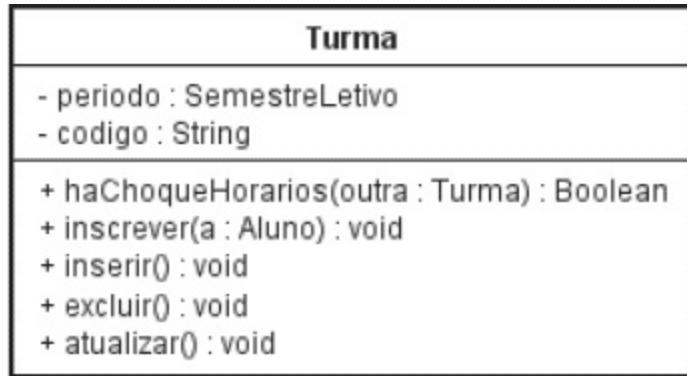


Figura 7-22: Exemplo de classe com coesão inadequada (i.e., coesão baixa)

N a [Seção 5.4.3.2](#), descrevemos o padrão tático do DDD denominado Serviço de Domínio. Mencionamos este padrão novamente por conta de sua relação com o princípio da coesão. De fato, o que normalmente leva um modelador a decidir por utilizar um Serviço de Domínio é justamente evitar que as classes cujos objetos estão envolvidos em uma determinada interação assumam uma responsabilidade que eventualmente resultaria na diminuição da coesão da solução de projeto. Uma alternativa é atribuir a um Serviço do Domínio essa responsabilidade que não se encaixa (i.e., não é adequada) nas classes envolvidas.

O **acoplamento** é uma medida do quanto fortemente uma classe está conectada a outras classes, tem conhecimento ou depende das mesmas. Uma classe com acoplamento fraco (baixo) não depende de muitas outras, o que é bom para a qualidade do projeto. Por outro lado, uma classe com acoplamento forte é menos inteligível isoladamente e menos reutilizável, exatamente por depender de várias outras classes. Além disso, uma classe com alto acoplamento é mais sensível a mudanças, quando é necessário modificar as classes da qual ela depende.

Pelo exposto anteriormente, podemos ver que a criação de modelos de projeto com alta coesão e baixo acoplamento deve ser um objetivo de qualquer projetista. Mas o que os princípios de coesão e acoplamento têm a ver com a modelagem de interações?

Na modelagem de interações, quando definimos uma mensagem, estamos criando uma dependência entre os objetos envolvidos. Isso é o mesmo que dizer que estamos aumentando o acoplamento entre os objetos em questão. Portanto, é necessário que o modelador fique atento para definir apenas mensagens que são realmente necessárias. Podemos analisar esse aspecto também sob a perspectiva do modelo de classes, da seguinte forma. Sempre que possível, devemos evitar o envio de mensagens que implique a criação de associações redundantes no modelo de classes. Isso porque a adição de uma associação entre duas classes aumenta o acoplamento entre as mesmas.

preciso evitar também a definição de módulos que têm baixa coesão.

Outro princípio de projeto que pode ser entendido como uma variante do princípio do acoplamento é conhecido como *Lei de Demeter* (tradução para *Law of Demeter*, LoD), também chamado de *princípio do conhecimento mínimo*. A Lei de Demeter impõe certas restrições acerca de quais são os objetos para os quais devem ser enviadas mensagens na implementação de uma operação definida em certa classe. Mais especificamente, esse princípio indica que, dentro de um método M contido em uma classe C, uma mensagem N somente deve ser enviada aos seguintes objetos:

- (a) ao próprio objeto remetente da mensagem N;
- (b) a um objeto recebido como parâmetro do método M;
- (d) a um objeto criado dentro do método;
- (e) a um elemento de uma coleção que é atributo da classe.

A intenção da LoD é manter o acoplamento em um nível aceitável e também evitar que um objeto tenha conhecimento das associações eventualmente existentes entre outros objetos. Por exemplo, considere que existam duas classes associadas, Venda e Pagamento, e que outro objeto necessita saber o valor do pagamento de uma venda. Uma alternativa é fazer com que esse objeto envie uma mensagem para o objeto Venda para conhecer seu objeto Pagamento associado e, em seguida, envie uma mensagem para Pagamento para saber seu valor. Uma alternativa melhor, segundo a Lei de Demeter, é fazer com que o objeto Venda responda diretamente o valor do pagamento. O objeto Venda fica, então, responsável por consultar seu pagamento e retornar o valor correspondente. Nessa última alternativa, o objeto que precisa desse valor somente necessita ter conhecimento da classe Venda.

Para finalizar esta seção, é importante notar que as medidas de coesão e de acoplamento aplicam-se também a diferentes níveis de abstração e artefatos de software. Nesta seção, descrevemos o seu uso no contexto das classes. Entretanto, essas medidas podem também ser aplicadas a pacotes (de classes), componentes e camadas de software, conforme discutimos no [Capítulo 10](#), quando descrevemos aspectos relativos à arquitetura de um SSOO. A descrição desses princípios também continua no [Capítulo 8](#), no qual apresentamos detalhes sobre a modelagem de classes na etapa de projeto. Também no [Capítulo 8](#), apresentamos outros princípios de projeto igualmente úteis na modelagem de interações.

7.5.3 Encapsulamento

Outro princípio de projeto que podemos utilizar durante a construção do modelo de interações para atribuir responsabilidades de forma correta é o encapsulamento. Descrevemos este princípio de uma perspectiva mais conceitual na [Seção 1.2.3](#). Quando o interpretamos na perspectiva do código-fonte de um SSOO, esse princípio recomenda que uma unidade de software oculte, por meio de uma *interface*, detalhes de implementação do seu estado e do seu comportamento. Qualquer acesso a informações internas da unidade deve ser realizado pela sua interface, que é composta por *operações*.

O princípio do encapsulamento está de acordo com o critério de decomposição de um sistema em partes menores denominado. Esse critério recebe o nome de *ocultamento da informação* (tradução para *Information Hiding*), e foi proposto por David Parnas em 1972. Sua definição é fornecida a seguir (PARNAS, 1972):

Todo módulo [...] é caracterizado pelo seu conhecimento de uma decisão de projeto que ele esconde de todos os outros [módulos]. Sua interface ou definição deve ser tal que revele o mínimo possível sobre seu funcionamento interno.

Nessa definição, David Parnas usou a expressão “módulo” porque, na época, o paradigma dominante era o da programação estruturada. Entretanto, com o passar do tempo, constatou-se que essa definição é igualmente aplicável a classes na orientação a objetos, assim como também vale para camadas de software, ou mesmo para subsistemas.

Uma vantagem de uma unidade de software cujo projeto esteja aderente ao princípio do encapsulamento é que a implementação dessa unidade pode ser alterada, sem que seus clientes necessitem de alterações. Como consequência, projetos aderentes ao princípio do encapsulamento possuem boa manutenibilidade, além de serem mais extensíveis e reusáveis.

Na modelagem de interações, surgem situações em que o projetista deve fazer uso do princípio do encapsulamento para atribuir responsabilidades corretamente. A seguir, apresentamos como exemplo duas dessas situações e a forma adequada de modelagem.

Como primeiro exemplo, considere a [Figura 7-23](#), que apresenta um fragmento de diagrama de sequência. Nesse diagrama, um objeto da classe RealizarInscricaoControlador recebe uma mensagem para registrar a inscrição de um aluno em uma turma. A questão aqui é de que forma esse objeto envia mensagens subsequentes para seus colaboradores de forma que possa cumprir com essa responsabilidade. A seguir, apresentamos duas soluções de projeto alternativas para essa questão e analisamos a diferença entre elas à luz do princípio do encapsulamento.

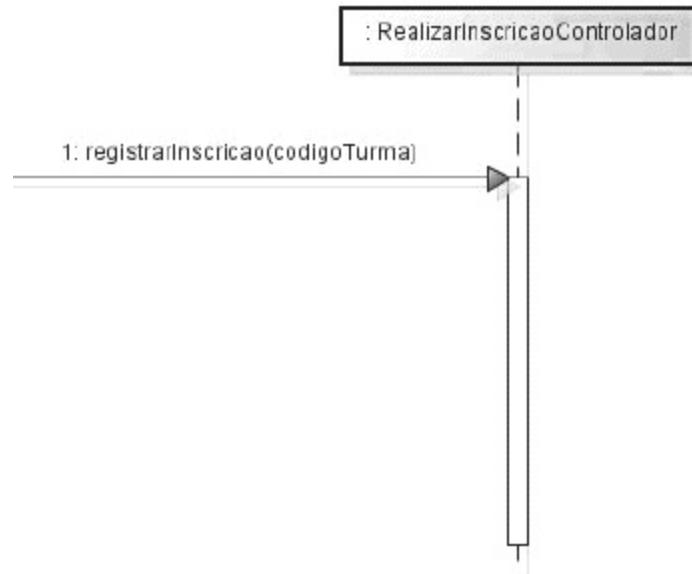


Figura 7-23: Mensagem enviada para controlador de caso de uso.

A [Figura 7-24](#) apresenta uma primeira solução. Repare que nessa solução o objeto RealizarInscricaoControlador obtém uma referência para o objeto Turma (mensagem 1.1), cria um objeto Inscricao (mensagem 1.2) e o associa ao objeto Turma (mensagem 1.3). Essa solução não está de acordo com o princípio do encapsulamento. Para entender porque, repare que o objeto RealizarInscricaoControlador desnecessariamente tem conhecimento da existência do objeto Inscricao, visto que a classe Turma já possui uma associação com Inscricao (veja a [Figura 5-49](#)) e poderia assumir a responsabilidade de criação deste objeto.

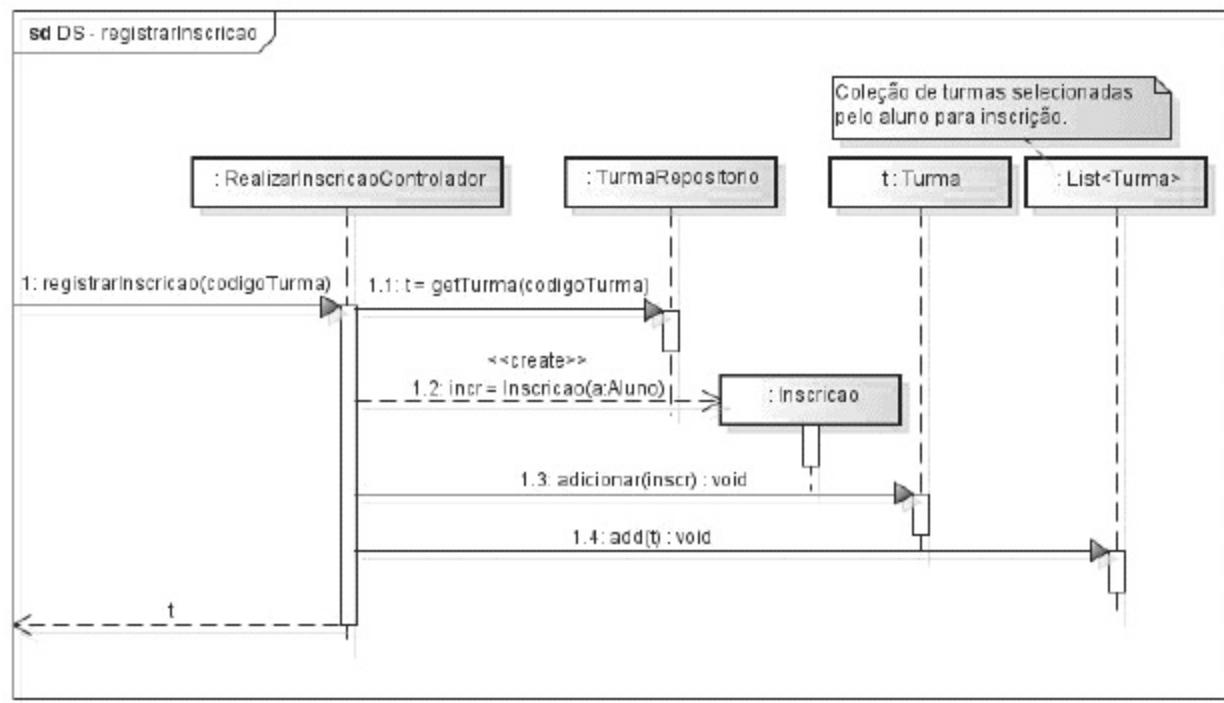


Figura 7-24: Interação que viola o princípio do encapsulamento.

Agora considere a [Figura 7-25](#). Diferente da solução anterior, o objeto `RealizarInscricaoControlador` repassa para o objeto `Turma` a responsabilidade de criação do objeto `Inscrição`. O efeito dessa mudança é que a forma pela qual o objeto `Inscrição` é criado não precisa ser conhecida pelo objeto `RealizarInscricaoControlador`; essa decisão fica escondida (ou encapsulada) na classe `Turma`. Repare que o acoplamento dessa segunda solução é também menor, visto que a classe `RealizarInscricaoControlador` não tem conhecimento da existência da classe `Inscrição`. Em geral, a aplicação adequada do princípio do encapsulamento leva a soluções de projeto com níveis de acoplamento mais baixos.

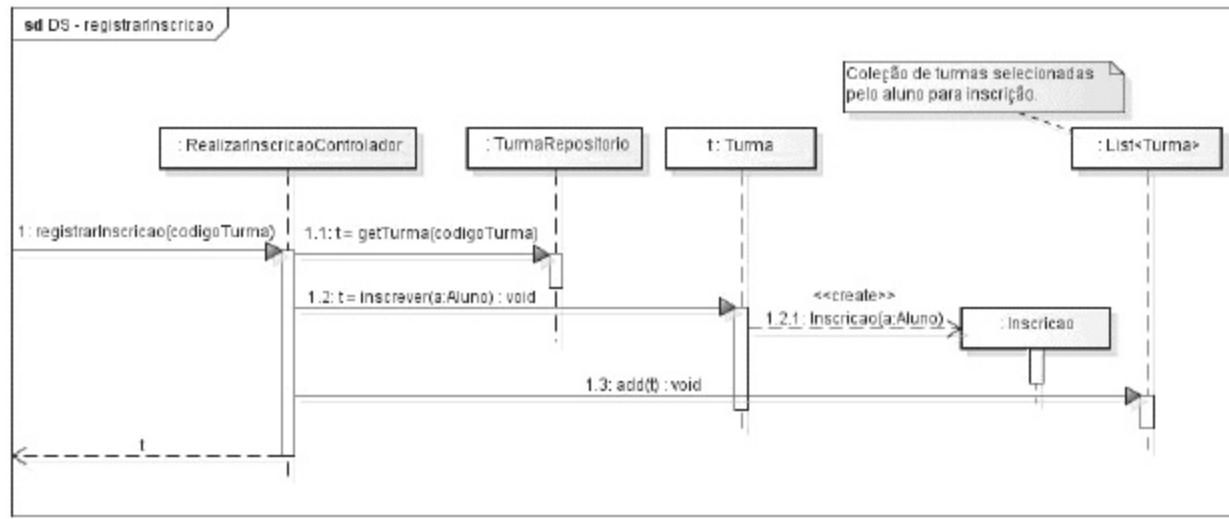


Figura 7-25: Interação aderente ao princípio do encapsulamento.

A segunda situação de modelagem que apresentamos é aquela na qual um objeto deve ser criado a partir de informações coletadas externamente ao sistema. Essas informações podem, por exemplo, ser fornecidas por um ator em um caso de uso. Na realização de um caso de uso, é comum atribuir a responsabilidade de criar objetos de entidade a um objeto de controle, que recebe os dados necessários à instanciação a partir de objetos de fronteira. A [Figura 7-26](#) ilustra essa situação, na qual um objeto da classe `RealizacaoPedidosServico` envia a mensagem `adicionarItemPedido` para o objeto `Pedido`.

O primeiro é um objeto de controle, e o segundo, do domínio. Repare que a mensagem `adicionarItemPedido` repassa informações coletadas pelo controlador

Outra situação de modelagem na qual uma responsabilidade de criação deve ser alocada é aquela que envolve relacionamentos de agregação ou composição ([Seção 5.2.2.7](#)). Nesse caso, o objeto-todo normalmente deve assumir a responsabilidade para criar suas partes. Portanto, em uma agregação (ou composição), é mais adequado que o objeto todo crie suas partes quando requisitado por outros objetos. A [Figura 7-26](#) também ilustra essa situação, na qual um objeto da classe Pedido envia uma mensagem de criação para dar origem a um objeto ItemPedido. Veja também a [Figura 5-39](#), que apresenta o diagrama de classes em que essas duas classes estão envolvidas.

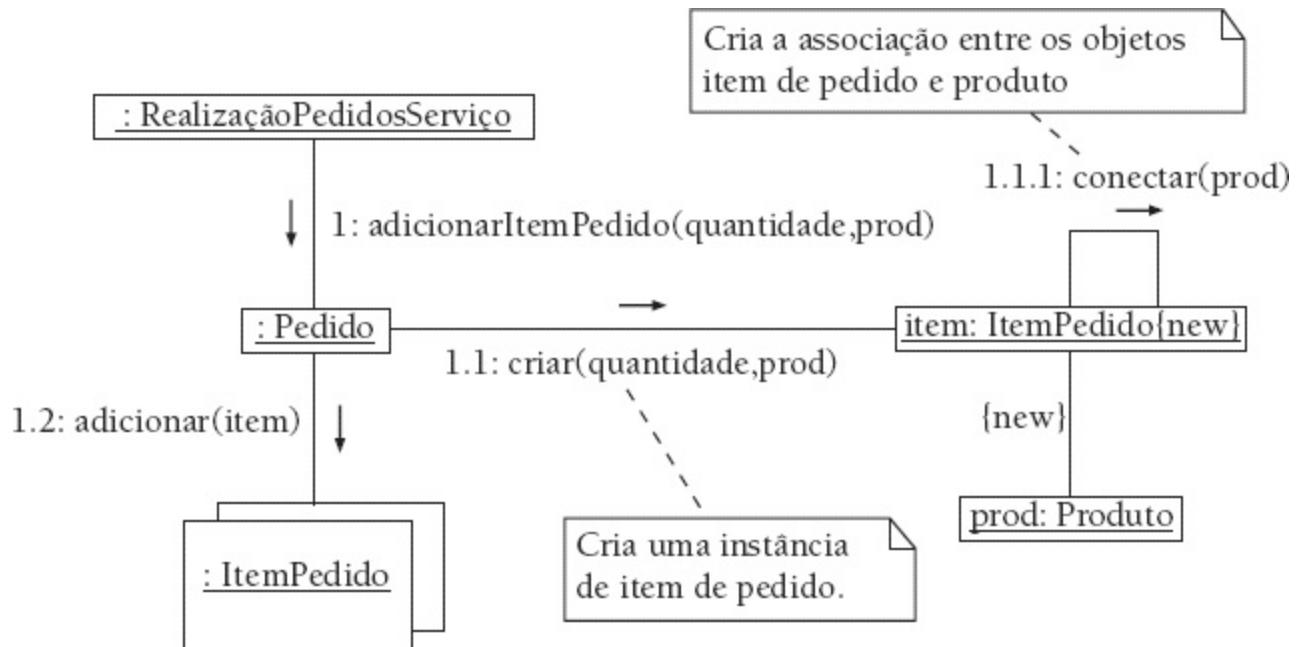


Figura 7-26: Em um relacionamento todo-parte, o todo é normalmente responsável por criar suas partes.

Para finalizar a discussão aqui apresentada sobre o princípio do encapsulamento, vamos relacioná-lo ao padrão tático Agregado, descrito na [Seção 5.4.3.2](#) no contexto da modelagem de interações. De fato, os agregados são manifestações da aplicação correta do princípio do encapsulamento, uma vez que, para interagir com objetos contidos em um agregado, é necessário enviar uma mensagem para sua raiz. Se voltarmos nossa atenção para o diagrama de comunicação apresentado na [Figura 7-26](#), vamos perceber que a classe Pedido é a raiz do agregado que envolve essa própria classe, além das classes ItemPedido e Produto. Repare também que o objeto da classe RealizacaoPedidosServico envia uma mensagem para a raiz, que por sua vez trata de interagir com os demais objetos do agregado.

7.5.4 Procedimento de construção do modelo de interações

Nesta seção, descrevemos um procedimento para construir modelos de interação. Esse procedimento genérico serve tanto para diagramas de sequência quanto para diagramas de comunicação, resguardando-se as diferenças de notação entre os dois. Entretanto, antes de entrarmos nos detalhes desse procedimento, é necessário que definamos os conceitos de *evento de sistema* e de *operação de sistema*.

7.5.4.1 Eventos de sistema e operações de sistema

Um **evento de sistema** é alguma ocorrência no ambiente do sistema respondida por alguma ação

desse sistema. No contexto de casos de uso, eventos de sistema correspondem às *ações do ator* no cenário de determinado caso de uso. Sendo assim, é relativamente fácil identificar eventos de sistemas em uma descrição de caso de uso: devemos procurar nessa descrição os eventos que correspondem a ações do ator. No caso particular em que o ator é um ser humano e existe uma interface gráfica para que o mesmo interaja com o sistema, os eventos do sistema são resultantes de ações desse ator sobre essa interface gráfica, que corresponde a objetos de fronteira. O conceito de evento de sistema não é novo. Na verdade, já era utilizado na metodologia de análise estruturada para a construção do *modelo ambiental* do sistema (YOURDON, 1990). Posteriormente esse conceito foi adaptado por Bertrand Meyer e Craig Larman ao paradigma da orientação a objetos (LARMAN, 2003; MEYER, 1997). Esse mesmo conceito é também descrito pelo professor Raul Wazlawick em WAZLAWICK (2011).

Em geral, as mensagens enviadas por um objeto de fronteira como consequência do acontecimento de algum evento de sistema resultam na execução de operações que devem ser alocadas na classe do objeto controlador do caso de uso em questão. Essas operações são denominadas *operações de sistema*. Por definição, uma operação de sistema é qualquer operação que deve ser executada para “recepçinar” um evento de sistema.

Na construção do modelo de classes de análise, vimos que é possível definir um objeto de controle para cada caso de uso (veja a [Seção 5.4.2.1](#)). Esse objeto fica então responsável por coordenar a realização desse caso de uso. Como o controlador tem essa responsabilidade de coordenação, todas as ações do ator resultam em alguma atividade realizada por esse objeto de controle. Portanto, as operações de sistemas identificadas em um caso de uso devem ser alocadas ao controlador desse caso de uso.

Figura 7-27: Protótipo da interface gráfica do caso de uso Fornecer Grade de Disponibilidades.

Para exemplificar os conceitos de evento de sistema de operação de sistema, considere a [Figura 7-27](#), na qual é apresentado o protótipo de um dos formulários de nosso estudo de caso, o SCA. Esse formulário corresponde ao caso de uso Fornecer Grade de Disponibilidades (veja a [Seção 4.7.3](#); veja também a [Figura 5-48](#)). Neste formulário, o professor pode configurar sua *grade de disponibilidade*, que corresponde a um agregado de disciplinas que ele deseja lecionar e de dias e horários em que está disponível para lecionar no próximo período letivo. A princípio, o professor deve fornecer sua matrícula e solicitar sua validação pelo sistema (atente para o botão “Validar Matrícula”). Evidentemente,

essa solicitação, que corresponde a um evento de sistema, desencadeia uma sequência de ações executadas pelos objetos componentes do SCA. Ou seja, essa solicitação do ator é um típico evento de sistema. Se a validação da matrícula do professor tiver sucesso, este pode montar sua grade (note as duas abas no formulário para fornecimento de disponibilidades). Ao final da montagem de sua grade, o professor pode solicitar ao sistema que este faça o registro da mesma. Em resumo, temos neste exemplo a seguinte lista de eventos de sistema (veja também a [Seção 7.7](#), que fornece mais detalhes das operações de sistema identificadas para este caso de uso do SCA):

- Solicitação de validação de matrícula de professor.
- Solicitação de adição de uma disciplina à grade.
- Solicitação de adição de um item de horário (dia, hora inicial e hora final) à grade.
- Solicitação de registro da grade.

Repare que, embora o exemplo acima esteja no contexto de um formulário (p. ex., de um componente da interface gráfica com o usuário), é importante notar que nem todo evento de sistema é originado em um objeto de fronteira correspondente a uma interface gráfica. Um evento de sistema corresponde a qualquer ocorrência na fronteira do sistema que o leve a reagir de alguma forma; essa ocorrência pode mesmo ser gerada por um ator que não seja um ser humano (p. ex., outro sistema ou um equipamento).

Uma vez identificados os eventos de sistema, a definição das operações de sistema correspondentes é relativamente fácil: dado um evento de sistema, definimos uma operação de sistema correspondente (ou seja, uma mensagem enviada ao objeto de controle) e *parâmetros* para essa operação (ver [Seção 8.3.1](#)) que permitam que as informações necessárias à execução da operação sejam passadas de um objeto a outro. Por exemplo, para os quatro eventos de sistema acima, o controlador do caso de uso em questão deve possuir as seguintes operações:

- validarProfessor(matrícula)
- adicionarDisciplina(nomeDisciplina)
- adicionarItemHorario(dia, horaInicial, horaFinal)
- registrarGradeDisponibilidade()

Obviamente, diversas outras operações são normalmente invocadas como consequência da chamada de uma única operação de sistema. Ou seja, uma vez que uma operação de sistema é identificada, devemos também identificar as operações que devem ser invocadas subsequentemente. A identificação correta dessas operações subsequentes e sua correta alocação aos objetos do sistema são tarefas complexas. Já a identificação das operações (ou equivalentemente, das mensagens) subsequentes depende muito da experiência dos projetistas envolvidos. Para essa tarefa, é necessário que esses projetistas apliquem corretamente diversos princípios da orientação a objetos (coesão, acoplamento, encapsulamento etc). É também fundamental que o projetista tenha conhecimento acerca de diversos padrões de projetos e de quais são os contextos de suas utilizações. O conhecimento acerca de como a aplicação deve estar organizada arquiteturalmente também é importante aqui, posto que uma operação de sistema pode resultar na invocação de operações em classes residentes em diferentes camadas do software. Entretanto, apesar de toda a complexidade envolvida, pelo menos o modelador tem um ponto de partida para começar a tarefa. Esse ponto de partida corresponde às operações de sistema.

Mas por que as operações de sistema são importantes para a modelagem de interações? Essas operações são importantes para essa atividade, porque as interações entre objetos de um sistema acontecem por conta do acontecimento de algum evento de sistema, que por sua vez está sempre associado a uma operação de sistema. Ou seja, um evento de sistema é alguma ação tomada por um ator que resulta em uma sequência de *mensagens* trocadas entre os objetos do sistema. Portanto, o ponto de partida para a modelagem de interações é a identificação das operações de sistema. Uma vez feita essa identificação, podemos desenhar diagramas de interação que modelam como os objetos colaboram entre si para produzir a resposta desejada a cada operação de sistema.

7.5.4.2 Procedimento para construção do modelo de interações

Após a discussão apresentada na [Seção 7.5.4.1](#), estamos agora em condições de apresentar um procedimento genérico para construção do modelo de interações de um determinado caso de uso. Os passos desse procedimento são apresentados a seguir.

1. Para o caso de uso em questão, identifique as operações *de sistema correspondentes*.
2. Para cada operação de sistema identificada:
 - a. Adicione o objeto de controle no diagrama e defina uma mensagem de chegada nesse objeto correspondente à operação de sistema.
 - b. Adicione outros objetos à medida que a sua participação for necessária no cenário selecionado.
 - c. Defina as mensagens necessárias entre o controlador e os demais objetos, assim como aquelas entre esses outros objetos.

Em relação ao passo 1, a definição das operações de sistema pode ser feita conforme descrito na [Seção 7.5.4.1](#). O passo 2a do procedimento é trivial, enquanto que os passos 2b e 2c são de longe os mais complexos. Esses dois últimos passos envolvem a correta aplicação de vários princípios de projeto orientado a objetos, como encapsulamento, acoplamento e coesão. A [Seção 7.7](#) apresenta exemplos de aplicação desse procedimento. Para finalizar esta seção apresentamos a seguir alguns comentários gerais acerca do procedimento de construção.

É possível notar no procedimento de construção apresentado acima que não recomendamos a adição do ator nem do objeto de fronteira, embora isso possa ser feito em um diagrama de sequência ou de comunicação (conforme descrito na [Seção 7.1](#)). A justificativa para isso é que a adição desses elementos não contribui significativamente para o entendimento de uma interação. Em vez disso, recomendamos iniciar a modelagem da interação com a representação do recebimento da mensagem para ativação da operação de sistema no controlador do caso de uso. Essa recomendação é consistente com a boa prática de separar a lógica do domínio e a lógica da interface com o usuário. Mais detalhes a respeito dessa separação são descritos na [Seção 11.1.2.1](#).

Durante a modelagem de classes de análise ([Capítulo 5](#)) são identificadas as classes conceituais (do domínio) que participam em cada caso de uso. Essas *classes do domínio* correspondem às entidades do mundo real envolvidas na tarefa do caso de uso se este fosse executado manualmente. Durante a modelagem de interações, objetos dessas classes participam da realização de um ou mais casos de uso. Por outro lado, durante a modelagem de interações, o projetista pode ter a necessidade de adicionar *classes de software* (ou seja, classes da aplicação que não têm correspondência no mundo real) que ajudem a organizar as tarefas a serem executadas. Essas classes de software

normalmente são necessárias para manter a coesão e o acoplamento das demais classes em um nível adequado. Craig Larman (2003) chama essas classes de *fabricações puras* (*pure fabrications*). Aqui, se encaixam algumas classes de fronteira, classes de controle (veja a [Seção 5.4.2.1](#)). Por exemplo, classes de acesso ao mecanismo de armazenamento, classes para autorização e autenticação etc. Como um exemplo mais específico, existe o padrão de projeto denominado DAO (*Data Access Object*), que define uma estratégia de acesso a informações em um banco de dados. Detalhamos esse padrão na [Seção 12.2.3](#). Durante a aplicação do procedimento descrito acima, o projetista deve fazer o máximo para construir diagramas de interação o mais inteligíveis possível. Por exemplo, é possível utilizar notas explicativas (ver [Seção 3.2](#)) para esclarecer algumas partes do diagrama de interação. Essas notas podem conter pseudocódigo ou mesmo texto livre. Outra estratégia que ajuda a construir um modelo de interações mais inteligível é utilizar os recursos de modularização (quadros, referências entre diagramas) que a UML 2.0 acrescentou. Descrevemos esses recursos de modularização na [Seção 7.4](#).

Outro aspecto importante na modelagem de interações diz respeito aos objetos de controle. O fato de as operações de sistema serem alocadas ao controlador pode levar a uma estrutura de classes bastante acoplada; no pior caso, o controlador pode ter conhecimento da existência de todas as classes participantes do caso de uso. Por conta disso, é bastante importante que o projetista se certifique de que esse controlador realiza *apenas coordenação*. Em particular, responsabilidades específicas do domínio devem ser atribuídas aos objetos de domínio (entidades), e não ao controlador. Além disso, sempre que for adequado, o projetista deve fazer, com base nos princípios de projeto, com que as classes de domínio enviem mensagens entre si. Dessa forma, o controlador envia uma mensagem para um objeto do domínio e este, por sua vez, dispara o envio de mensagens para outros objetos do domínio, o que evita que o controlador precise ter conhecimento desses últimos (i.e., fique acoplado aos mesmos).

7.6 Modelo de interações em um processo iterativo

Uma questão relevante é qual seria o momento adequado para começar a construir o modelo de interações. Alguns textos sobre modelagem de sistemas orientados a objetos indicam o início da modelagem de interações já na atividade de análise. Outros defendem o início de sua construção somente na etapa de projeto. O fato é que a distinção entre essas duas fases não é tão nítida na modelagem orientada a objetos, e a modelagem de interações pode ser realizada em ambas as fases. Na análise, podemos começar a construir o modelo de interações logo depois que uma primeira versão do modelo de casos de uso estiver pronta. Inicialmente, o modelo de interações pode ser utilizado para representar apenas os objetos participantes em cada caso de uso e com mensagens exibindo somente o nome da operação. Posteriormente (na etapa de projeto), esse modelo pode ser refinado, incluindo criação e destruição de objetos, detalhes sobre o tipo e assinatura completa de cada mensagem etc.

Os objetivos da modelagem de interações são: (1) obter informações adicionais para completar e aprimorar outros modelos (modelo de casos de uso e de classes); e (2) fornecer aos programadores uma visão detalhada dos objetos e mensagens envolvidos na realização dos casos de uso do sistema. De uma forma mais geral, é da natureza de um processo incremental e iterativo que os modelos evoluam em conjunto durante o desenvolvimento do sistema. Embora esses modelos representem visões distintas do sistema, eles são *interdependentes*. Os itens a seguir demonstram como o modelo

de interações se relaciona aos de casos de uso e de classes.

- *Modelo de casos de uso → modelo de interações.* Utilizamos os cenários extraídos do modelo de casos de uso como fonte de identificação de mensagens na modelagem de interações. É adequado que se verifique se cada cenário relevante para todos os casos de uso foi considerado na modelagem de interações.
- *Modelo de classes → modelo de interações.* Utilizamos informações provenientes do modelo de classes para construir modelos de interações (ver Seção 7.3.3). Em particular, algumas responsabilidades já podem ter sido definidas durante a modelagem de classes de análise. Isso é particularmente verdadeiro quando identificação dirigida a responsabilidades (ver Seção 5.4.3) foi utilizada durante a análise. Note também que a notação do diagrama de interações não precisa ser totalmente utilizada na fase de análise. Em particular, podemos definir diagramas de interação que apresentam as mensagens apenas com os seus nomes. Essa definição inicial pode ser refinada posteriormente com a utilização da sintaxe mais avançada.
- *Modelo de interações → modelo de casos de uso.* A partir do conhecimento adquirido com construção do modelo de interações, podemos aperfeiçoar e validar os cenários do modelo de casos de uso.
- *Modelo de interações → modelo de classes.* É pela construção de modelos de interações que informações necessárias para a alocação e o detalhamento de operações para classes surgem. A esse respeito, existem inclusive *ferramentas CASE* (ver Seção 2.6) que automaticamente alteram o diagrama de classes, adicionando uma operação à classe do objeto receptor de uma mensagem após o modelador adicionar essa mensagem em um diagrama de interação. Ou seja, nessas ferramentas, o diagrama de classes é completado de modo automático pela construção de um diagrama de sequência ou de comunicação. Por outro lado, durante a construção de um diagrama de interação, pode ser que algumas informações necessárias em certa classe ainda não existam, em virtude de não terem sido identificadas anteriormente. Portanto, além da identificação de operações, é provável que *novos atributos* sejam identificados durante a modelagem de interações. Outro elemento do modelo de classes que é comumente identificado ou validado pela construção do modelo de interações são as associações: uma mensagem implica a existência de uma associação (ou algum outro tipo de dependência; ver Seção 8.4.1) entre os objetos envolvidos. Finalmente, é também comum a situação em que identificamos *novas classes* durante a modelagem de interações, principalmente aquelas relacionadas ao domínio da solução do problema.
- *Modelo de interações → visões de classes participantes.* Este tópico pode parecer redundante quando consideramos o tópico anterior. Afinal, as visões de classes participantes (ver Seção 5.5.3) fazem parte do modelo de classes. Entretanto, a interdependência entre o modelo de interações e as VCP é importante o suficiente para descrevermos em um tópico em separado. No Capítulo 5, analisamos o conceito de VCP como um diagrama que apresenta as classes participantes da realização de certo caso de uso. Pois bem, uma VCP deve ser construída com base em informações obtidas a partir dos diversos diagramas de interação associados a um mesmo caso de uso. Em particular, as corretas associações do controlador de um caso de uso com os objetos do domínio que aparecem na VCP só podem ser validadas pela análise das mensagens identificadas durante a construção do modelo de interações. As VCPs também servem como um mecanismo adequado de modularização de

modelagem. Isso porque durante essa etapa de projeto surge a necessidade de definir diversas outras classes em um caso de uso, classes essas relacionadas ao espaço da solução do problema. Representar todas as classes de um sistema em um único diagrama de classes pode tornar-se impraticável em algumas situações. Nesses casos, uma alternativa é criar uma VCP para cada caso de uso e nela definir tanto as classes do espaço do problema quanto do espaço da solução. A [Seção 8.8](#) apresenta exemplos de VCP que resultam nessa abordagem, no contexto do SCA.

A [Figura 7-28](#) enfatiza a interdependência entre os artefatos de software produzidos pelos três modelos mencionados há pouco. As setas indicam que as atividades de modelagem alimentam umas as outras com informações para que cada modelo evolua.

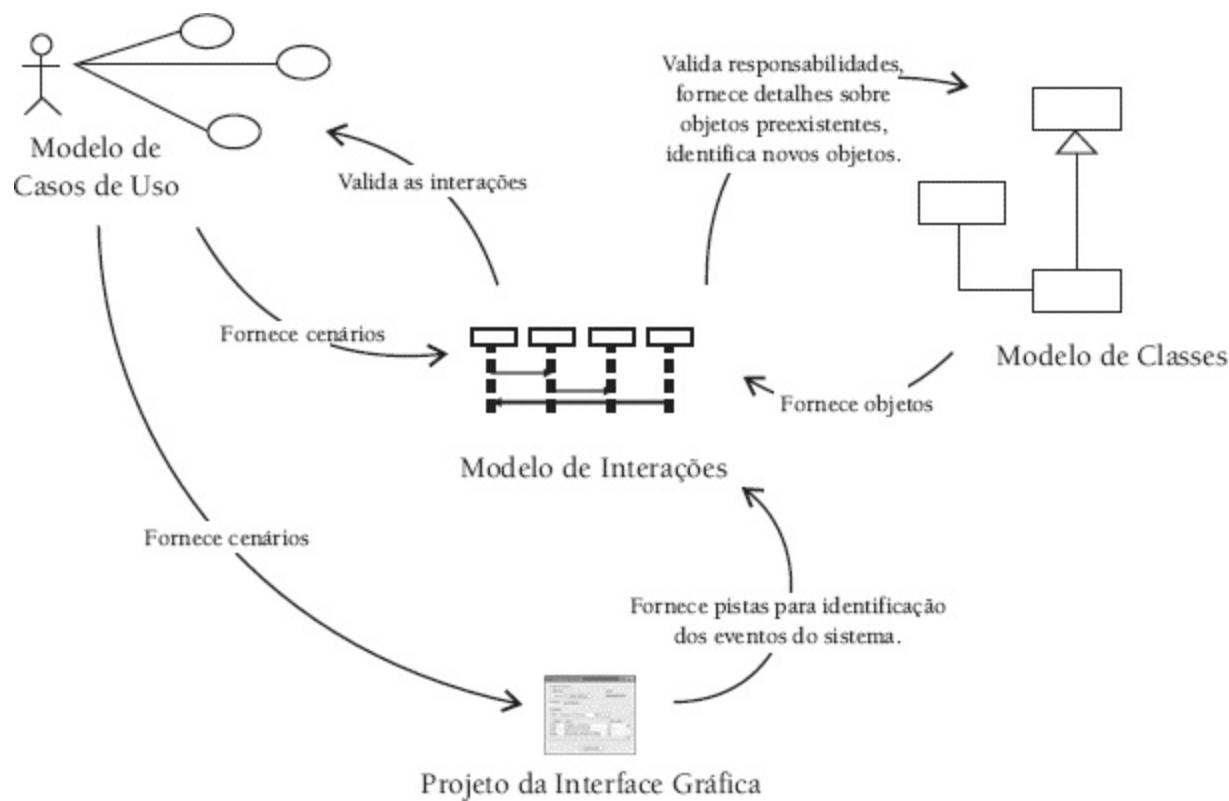


Figura 7-28: Interdependência entre os artefatos produzidos durante o desenvolvimento.

Como um comentário final nesta seção, é importante notar que a modelagem dinâmica de um sistema é uma tarefa complicada. Em um sistema complexo, há uma quantidade imensa de possíveis “caminhos” que sua execução pode tomar. Nesse cenário, é difícil escolher que comportamento cada classe deve assumir. Nessa situação, a melhor alternativa é trabalhar iterativamente: é adequado desenvolver o modelo de casos de uso e o modelo de classes iniciais; após isso, é possível desenvolver o modelo de interações e de estados (sobre este último, ver o [Capítulo 9](#)); finalmente, o projetista pode retornar aos modelos iniciais para verificar a consistência dos mesmos, internamente, e em relação aos demais modelos.

7.7 Estudo de caso

Esta seção apresenta uma parte do modelo de interações para o SCA. Na [Seção 5.7](#), mostramos diversos artefatos resultantes da modelagem de classe de análise desse sistema. Esses artefatos

servem de entrada para a construção de novos artefatos, apresentados aqui. Além disso, os artefatos descritos aqui geram informações para transformar o modelo de classes de análise no modelo de classes de projeto. De fato, a modelagem aqui apresentada está relacionada ao que apresentamos na [Seção 8.8](#), que descreve o efeito de incorporar as informações obtidas durante a modelagem de interações ao modelo de classes de análise, para obter o modelo de classes de projeto.

7.7.1 Operações de sistema

Por outro lado, na [Seção 7.5.4.1](#) vimos que operações de sistema correspondem ao ponto de partida para a modelagem de interações. Sendo assim, a lista a seguir apresenta as assinaturas das principais operações de sistema identificadas para cada um dos casos de uso considerados do SCA. (A [Seção 8.3.1](#) apresenta a sintaxe de especificação de assinaturas de operações que é utilizada na lista a seguir.)

- **Fornecer Grade de Disponibilidades**
 1. validarProfessor(matr: String): DisponibilidadesInfo
 2. adicionarDisciplina(nomeDisciplina: String): Disciplina
 3. adicionarItemHorario(dia: String, inicio: String, fim: String): void
 4. confirmarGradeDisponibilidade(): void
- **Realizar Inscrição**
 5. iniciarInscricao(matrAluno: String): List<Turma>
 6. registrarInscricao(codigoTurma: String): Turma
 7. confirmarInscricoes(): void
 8. adicionarEmListaEspera(codigoTurma: String): Integer
- **Lançar Avaliações**
 9. iniciarLancamento(matrProfessor: String): List<Turma>
 10. solicitarLancamento(codTurma: String): AvaliacaoInfo
 11. lancarAvaliacoes(avaliacoes: AvaliacaoInfo): void
 12. finalizarLancamento(): void

7.7.2 Observações gerais

A lista de operações apresentada acima serve como ponto de partida para a modelagem de interações dessa parte do SCA. A descrição da modelagem fornecida a seguir tem o propósito de esclarecer ao leitor o processo de raciocínio utilizado na construção de modelos de interações. Entretanto, para facilitar a descrição que se segue, apresentamos algumas observações gerais relevantes.

- A ordem em que cada caso de uso é considerado é irrelevante. Porém, é adequado considerar as operações de sistema de um mesmo caso de uso na ordem em que essas operações são executadas durante a realização do caso de uso correspondente. Sendo assim, a descrição que se segue apresenta detalhes da modelagem de interações dessas operações de sistema na mesma ordem em que elas aparecem na lista acima.
- Optamos por realizar a apresentação dos modelos de interações por meio de diagramas de sequência. Essa escolha é arbitrária (p. ex., a notação do diagrama de comunicação também poderia ter sido utilizada).
- Cada diagrama de sequência apresentado nesta Seção está embutido dentro de um *quadro de interação* (veja a [Seção 7.4.1](#)). Criamos um diagrama de sequência para cada operação de sistema. Do exterior de cada quadro chega uma mensagem para ativar a operação de

sistema no controlador do caso de uso correspondente.

- Nos diagramas apresentados nesta seção, podemos perceber o padrão de comunicação entre objetos que descrevemos neste capítulo: ao receber a mensagem para execução de uma operação de sistema, o controlador do caso de uso se comunica com os objetos do domínio envolvidos para produção da resposta à solicitação enviada.
- Por questões de simplicidade, optamos por não apresentar os índices (identificadores) das mensagens, posto que, nos diagramas de sequência, é possível inferir a ordem e o aninhamento das mensagens pela sua localização no diagrama.

7.7.3 Modelos de interações

7.7.3.1 Fornecer Grade de Disponibilidades

Agora estamos em condições de apresentar a modelagem de interações para o SCA. A [Figura 7-29](#) apresenta o diagrama de sequência para a operação de sistema `validarProfessor`. De acordo com a [Figura 7-29](#), podemos perceber que apenas uma operação de sistema pode ser complexa o suficiente para justificar a construção de um diagrama de interações exclusivamente para ela.

Ao receber a mensagem para invocar essa operação, tudo que o controlador conhece inicialmente é a matrícula do professor, para o qual devem ser registradas as informações sobre a grade de disponibilidade. Sendo assim, o controlador precisa obter uma referência para o objeto professor correspondente, o que faz com que ele envie uma solicitação ao objeto `ProfessorRepositorio`. A referência retornada pelo repositório pode ser nula, na situação em que o usuário forneceu valor que não corresponde a uma matrícula de professor. Por conta disso, usamos um *fragmento combinado* (veja a [Seção 7.4.1](#)) para definir mensagens enviadas apenas se a referência para professor for válida. No interior desse fragmento, encontramos o procedimento de criação de um objeto da classe `FichaDisponibilidades`. Essa classe não havia sido identificada durante a análise inicial do SCA (veja a [Seção 5.7](#)).

A melhor forma de pensar nesta classe é considerar a situação em que não existe um sistema de software para definição da grade de disponibilidades e, por conta disso, essas informações devam ser preenchidas pelo professor em um formulário (ou ficha) impresso. Nesse formulário, deveriam constar nome e da matrícula do professor, além de duas listas, descritas a seguir. A primeira é uma relação de itens de horário, para possibilitar que o professor selecione em quais dias e horários está disponível para ministrar aulas. A segunda contém as disciplinas que o professor está apto a lecionar, e tem o propósito de possibilitar ao professor informar que disciplinas deseja lecionar um determinado semestre letivo. Dessa forma, o propósito da classe `FichaDisponibilidades` é encapsular as informações que existiriam naquele formulário impresso.

A criação do objeto da classe `FichaDisponibilidades` não é uma tarefa trivial. Essa criação envolve a manipulação de objetos das classes `Professor`, `ItemHorario` e `Disciplina`. Isso nos levou à decisão de criar uma *fábrica* (veja a [Seção 5.4.3.2](#)) para assumir a responsabilidade de criação de um objeto `FichaDisponibilidades`. Denominada essa fábrica de `FichaDisponibilidadesFabrica`.

Voltando ao contexto do fragmento combinado, o controlador solicita ao repositório a referência ao professor e repassa essa referência para a fábrica, que cria o objeto `FichaDisponibilidades` correspondente. Finalmente, esse objeto é retornado como resposta à solicitação original.

Duas outras operações de sistema consideradas nesse caso de uso são `adicionarDisciplina` e `adicionarItemHorario`. As interações correspondentes são ilustradas na [Figura 7-30](#) e na [Figura 7-31](#), respectivamente, e são autoexplicativas.

A última operação de sistema que consideramos para o caso de uso `Fornecer Grade de Disponibilidades` é

confirmarGradeDisponibilidade. Aqui, tudo que o controlador faz é solicitar ao objeto GradeDisponibilidades que realize uma validação acerca de sua consistência interna: uma grade de disponibilidade deve ter sido definida com ao menos uma disciplina e ao menos um item de horário. Essa interação é ilustrada na [Figura 7-32](#), que não exibe as mensagens entre o objeto GradeDisponibilidades e sua coleções internas para realizar a validação correspondente.

Outro aspecto importante é que, juntamente com o modelo de casos de uso, o modelo de interações de um sistema fornece informações para atividade denominada projeto da interface gráfica. Conforme descrito no [Capítulo 6](#), não está no escopo deste livro a descrição da tarefa de projeto de interface gráfica. Ainda assim, apenas como exemplo, a [Figura 7-27](#) apresenta o protótipo da interface gráfica do caso de uso Fornecer Grade Disponibilidade.

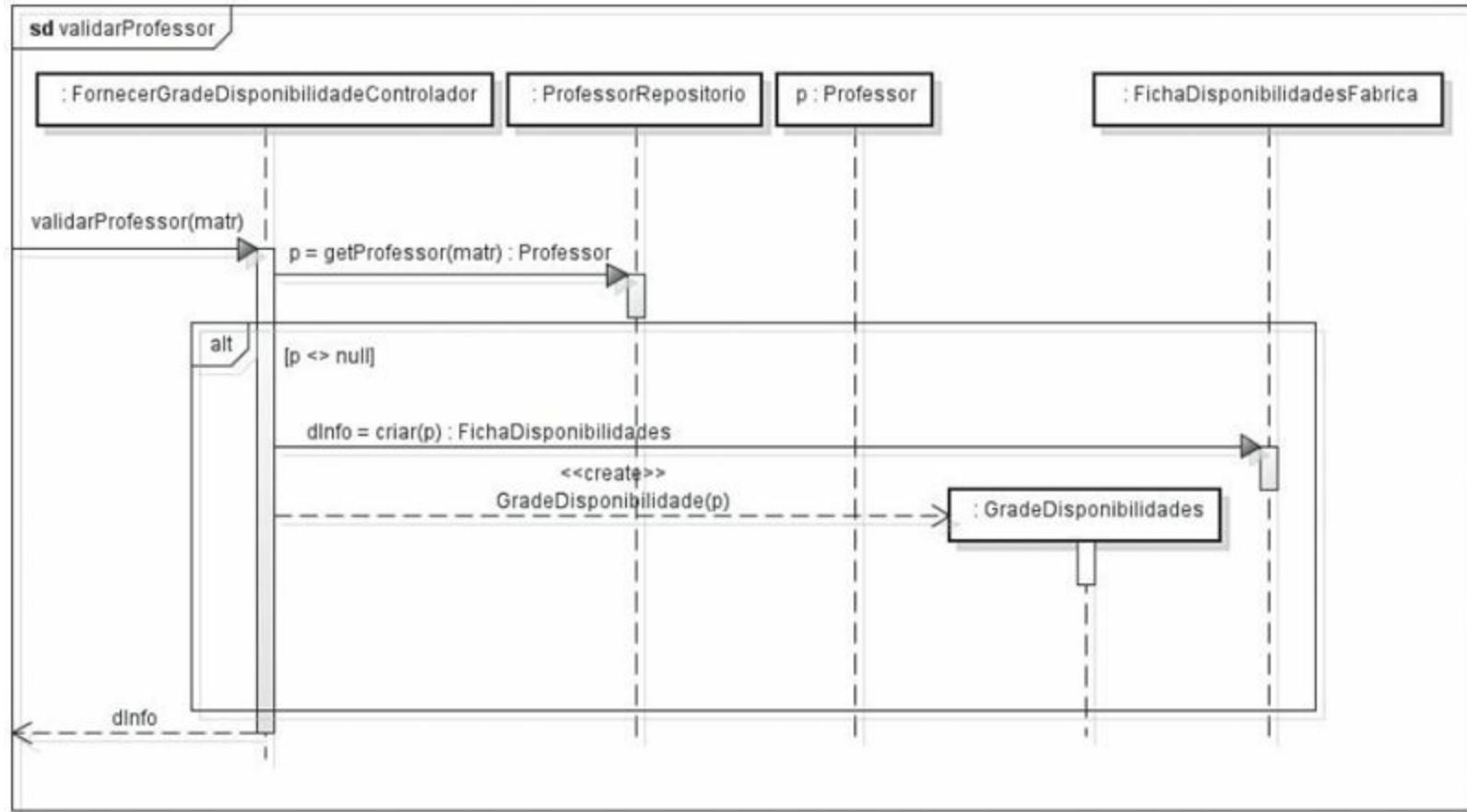


Figura 7-29: Interações resultantes da operação `validarProfessor`.

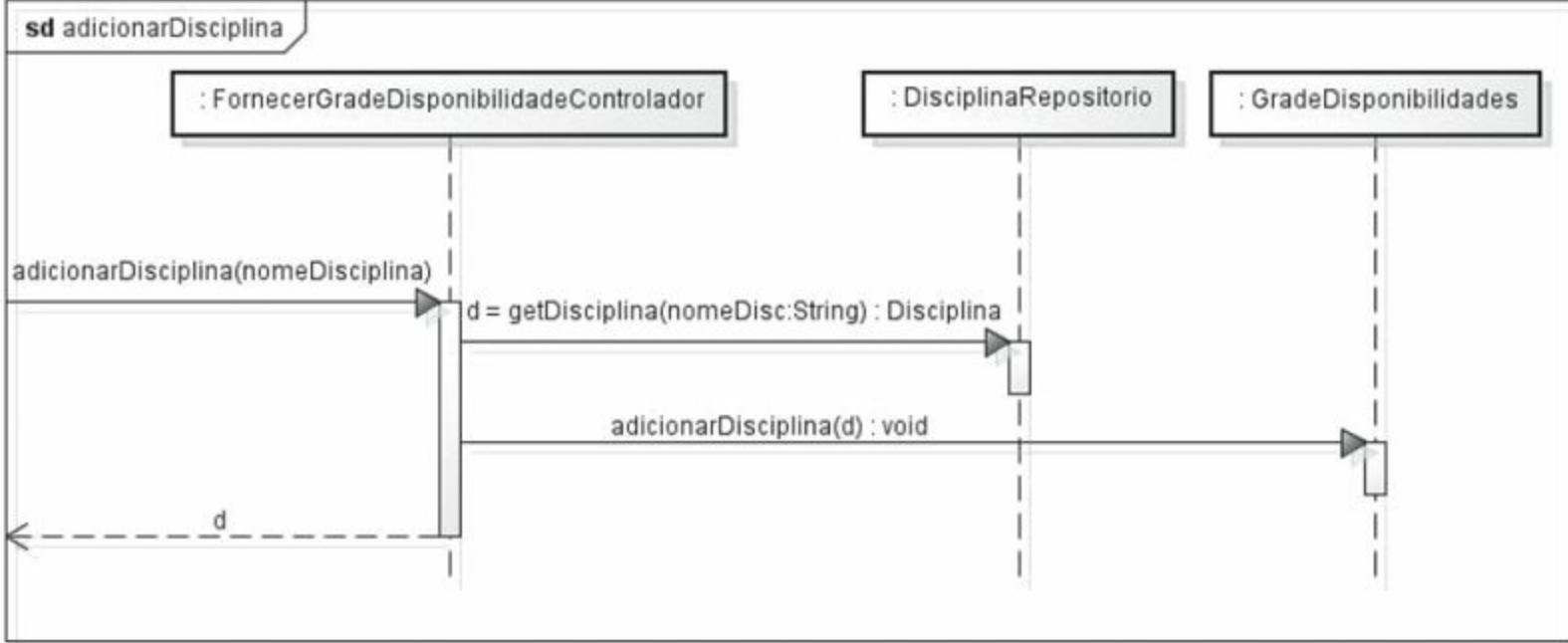


Figura 7-30: Interações resultantes da operação adicionarDisciplina.

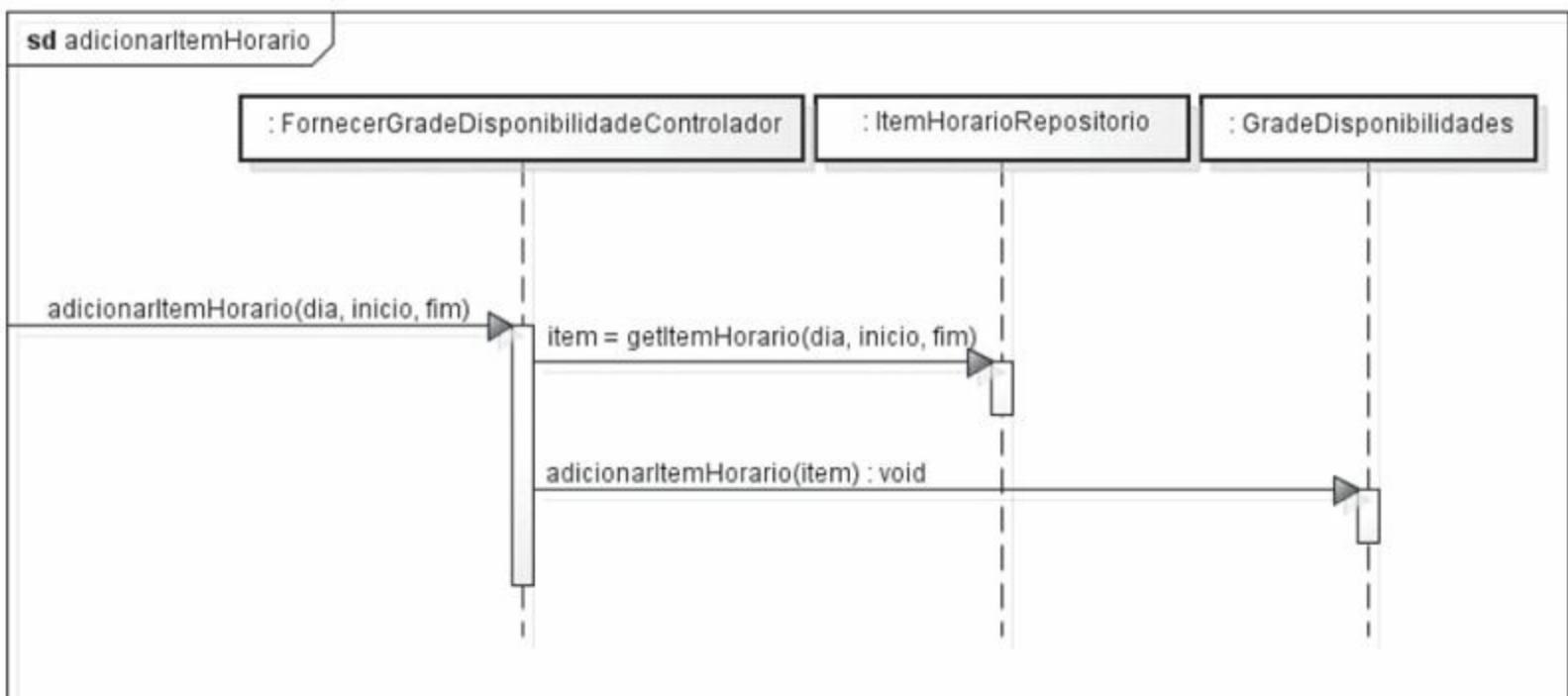


Figura 7-31: Interações resultantes da operação adicionarItemHorario.

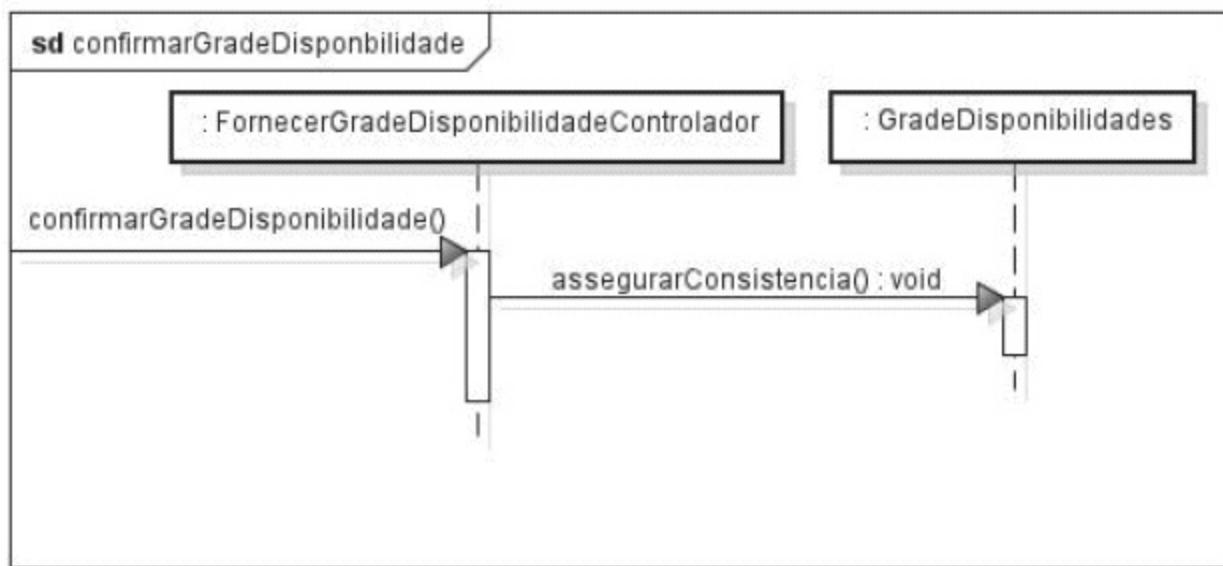


Figura 7-32: Interações resultantes da operação `confirmarGradeDisponibilidade`.

7.7.3.2 Realizar Inscrição

A primeira operação de sistema desse caso de uso é `iniciarInscricao`. O propósito dessa operação é produzir a lista de turmas em que é possível um determinado aluno realizar inscrição. (A descrição do conceito de *turmas possíveis* foi apresentada inicialmente na [Seção 5.4.3.2](#), quando descrevemos o padrão tático Serviço do Domínio.) A interação correspondente a essa operação de sistema é apresentada na [Figura 7-33](#). Na operação `iniciarInscricao`, o sistema deve apresentar uma lista de turmas para que o aluno selecione aquelas que deseja cursar no próximo semestre letivo. No entanto, essa lista não deve conter turmas de disciplinas que o aluno já cursou com aprovação; também não deve conter turmas de disciplinas para as quais o aluno não tem os pré-requisitos. Para formar essa lista, o controlador coordena a colaboração de diversos objetos do domínio, conforme descrevemos a seguir.

Para essa interação, decidimos criar um serviço do domínio (veja a [Seção 5.4.3.2](#)), representado pela classe `TurmasPossiveisServico`. A justificativa para criação dessa classe é livrar o controlador de manipular diversos objetos do domínio. O efeito disso é que a lógica para geração de turmas possíveis fica completamente encapsulada em classes do domínio.

Duas mensagens que surgiram durante a modelagem de interações resultantes da operação `iniciarInscricao` SÃO `getDisciplinasPossiveis` e `getTurmaAbertasPara`. A primeira foi definida em `Aluno` e atribui a essa classe a responsabilidade de gerar a lista de disciplinas possíveis para um aluno se inscrever. Repare que a classe `Aluno` cumpre essa responsabilidade ao interagir com a classe `HistoricoEscolar`, mas, por simplicidade, essa subinteração não é apresentada no diagrama da [Figura 7-33](#). Já a mensagem `getTurmaAbertasPara` atribui à classe `GradeHorarios` a responsabilidade de gerar uma lista de turmas abertas no semestre letivo corrente e correspondentes à lista de disciplinas predeterminada. Essa lista de turmas corresponde ao resultado da interação apresentada na [Figura 7-33](#).

Após ser apresentado à lista de turmas em que pode se inscrever, o aluno pode selecionar uma ou mais dessas turmas. A seleção de uma turma é representada pela operação de sistema `registrarInscricao`. O diagrama de sequência correspondente a essa interação é apresentado na [Figura 7-34](#). Uma observação digna de nota nesse diagrama é o uso de uma coleção temporária de objetos para armazenar as turmas que o aluno seleciona para inscrição.

A próxima operação de sistema, `confirmarInscricoes`, tem o propósito de validar a regra do negócio

RN01 (veja a [Seção 4.7.2](#)). A interação correspondente é apresentada no diagrama da [Figura 7-35](#). Para isso, criamos outro serviço do domínio, representado pela classe VerificarQtdCreditosServico. Esse serviço manipula objetos das classes Turma (p. ex., as turmas selecionadas para inscrição pelo aluno durante a realização do caso de uso) e Disciplina (posto que é esta classe que conhece a quantidade de créditos) para validar a regra do negócio mencionada acima.

Para finalizar esta Seção, mencionamos a operação de sistema adicionar EmListaEspera, identificada em um dos fluxos alternativos do caso de uso Realizar Inscrição (veja a [Seção 4.7.3](#)). A interação correspondente é apresentada no diagrama da [Figura 7-36](#). O propósito dessa operação é inserir o aluno na lista de espera pela abertura de alguma turma de determinada disciplina. Uma observação relevante aqui é o fato de o controlador manter uma referência para o objeto entre chamadas sucessivas de duas operações de sistemas. Em particular, uma referência para o objeto Aluno determinada na interação correspondente à operação iniciarInscricao é reusada na interação para adicionarEmListaEspera; veja a nota explicativa apresentada na [Figura 7-36](#).

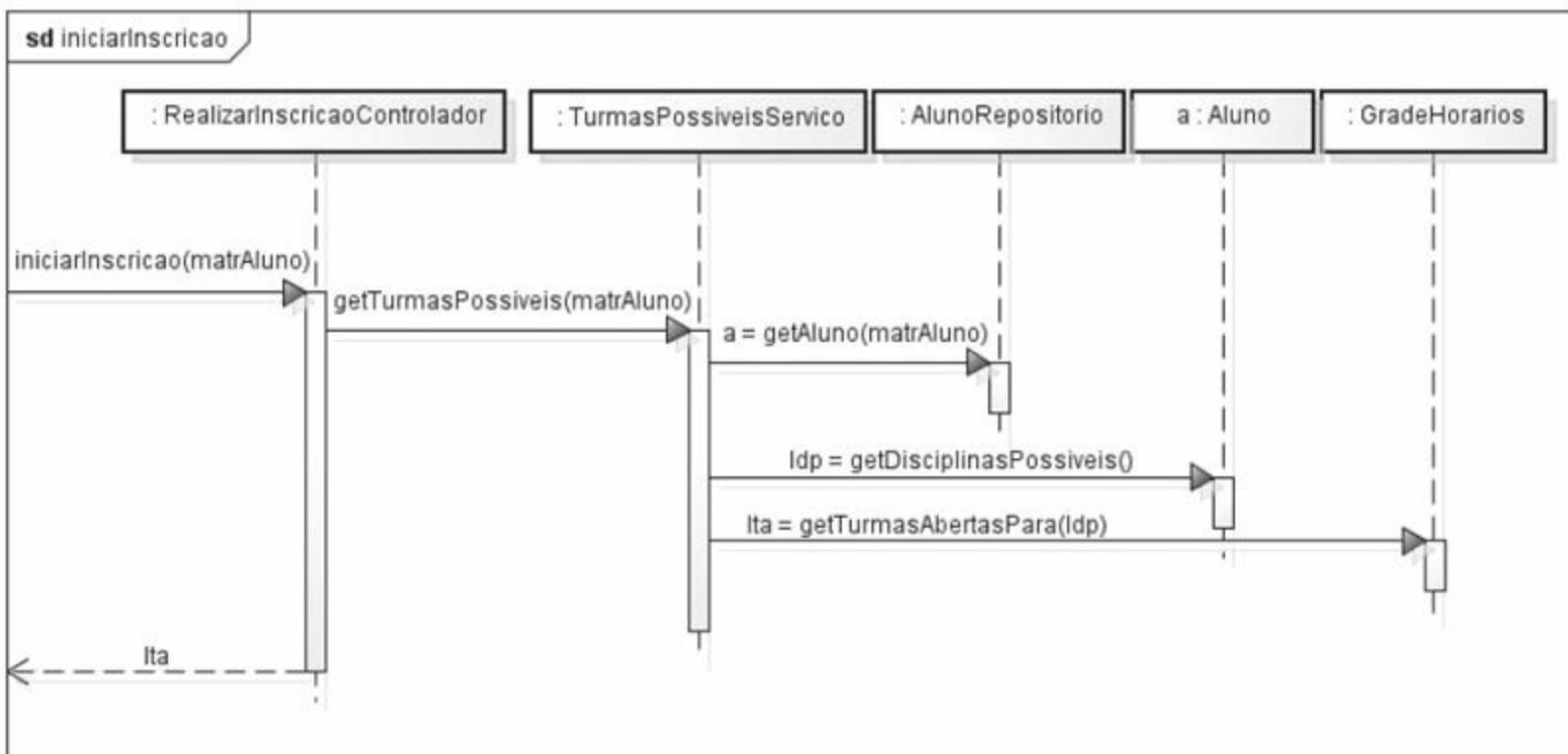


Figura 7-33: Interações resultantes da operação `iniciarInscricao`.

sd registrarInscricao

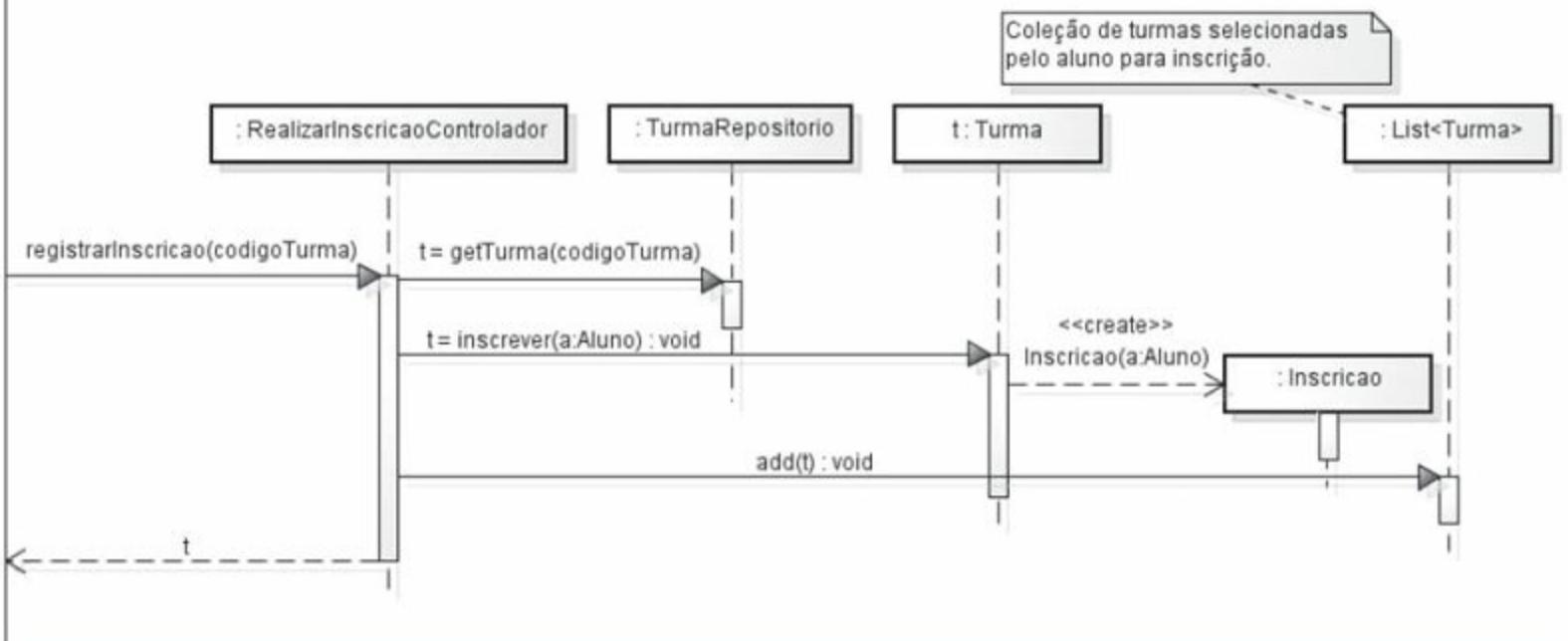


Figura 7-34: Interações resultantes da operação `registrarInscricao`.

sd confirmarInscricoes

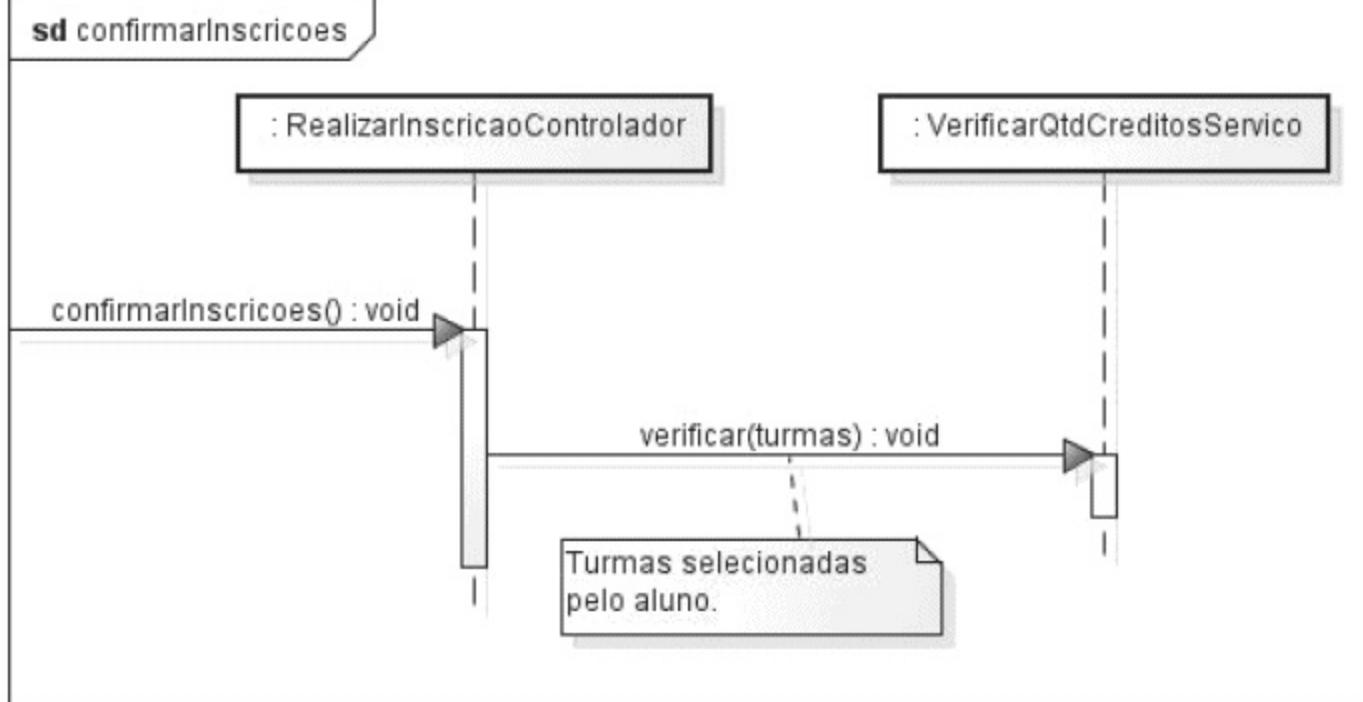


Figura 7-35: Interações resultantes da operação `confirmarInscricoes`.

sd adicionarEmListaEspera

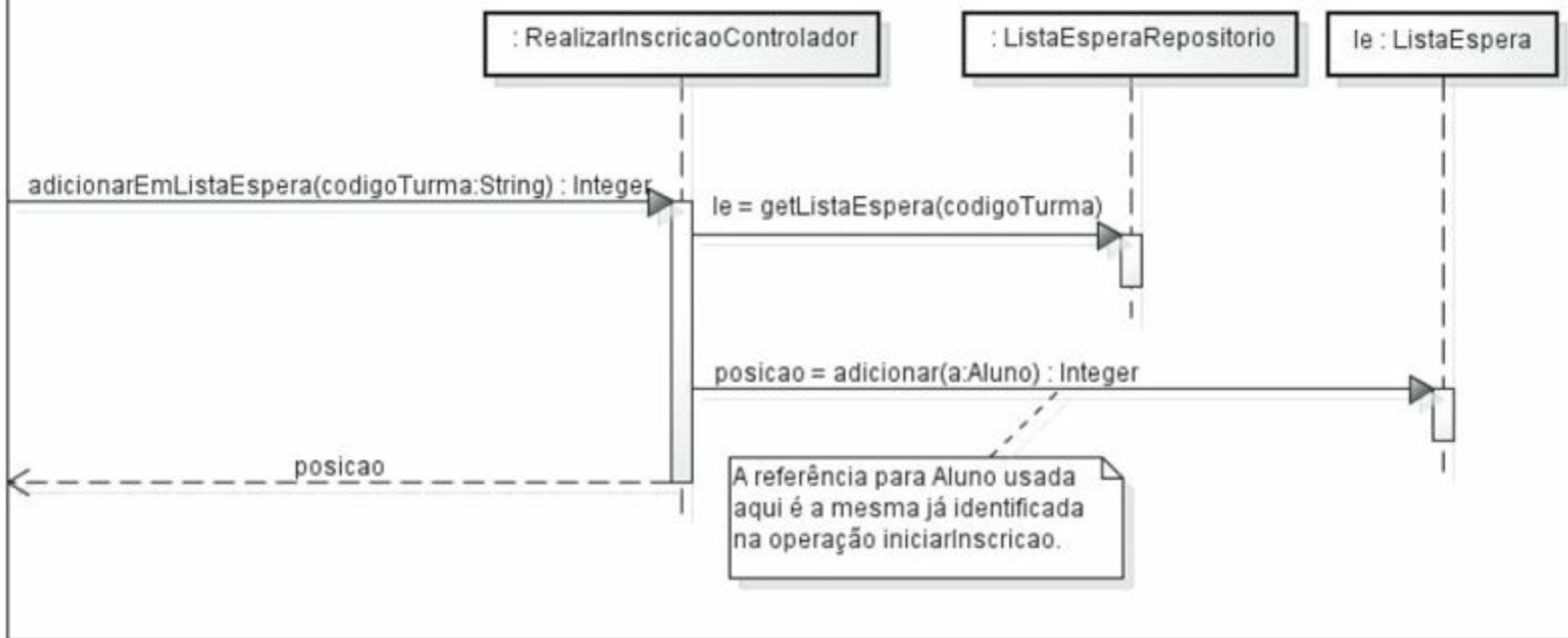


Figura 7-36: Interações resultantes da operação adicionarEmListaEspera.

7.7.3.3 Lançar Avaliações

No caso de uso Lançar Avaliações, começamos pela operação `iniciarLancamento`, cuja interação é apresentada no diagrama da [Figura 7-37](#). Nessa operação, tudo que o controlador faz é solicitar ao objeto `TurmaRepositorio` a lista de turmas alocada ao professor cuja matrícula é passada como argumento da mensagem.

A próxima operação de sistema que consideramos é `gerarFichaLancamento`, cuja interação é apresentada no diagrama da [Figura 7-38](#). Nessa operação, o sistema deve gerar o que denominamos *ficha de lançamento de avaliação* para uma determinada turma. Esse conceito é representado pela classe `FichaLancamentoAvaliacao`. Repare que esse objeto possui uma fábrica associada. O processo de raciocínio utilizado aqui é análogo ao já descrito na [Seção 7.7.3.1](#) no contexto da operação de sistema `validarProfessor` com relação à classe `FichaDisponibilidade`.

Na operação de sistema `lancarAvaliacoes`, o controlador recebe a ficha de lançamento preenchida e delega ao objeto `Turma` a responsabilidade de coordenar a atualização dos objetos do domínio correspondentes. Esse objeto `Turma` solicita a cada aluno inscrito o lançamento de sua avaliação. Neste ponto, ocorre outra delegação, pois cada objeto `Aluno` repassa a tarefa de lançamento para o seu objeto `HistoricoEscolar` associado. Essa interação é apresentada no diagrama da [Figura 7-39](#).

sd iniciarLancamento



Figura 7-37: Interações resultantes da operação iniciarLancamento.

sd gerarFichaLancamento

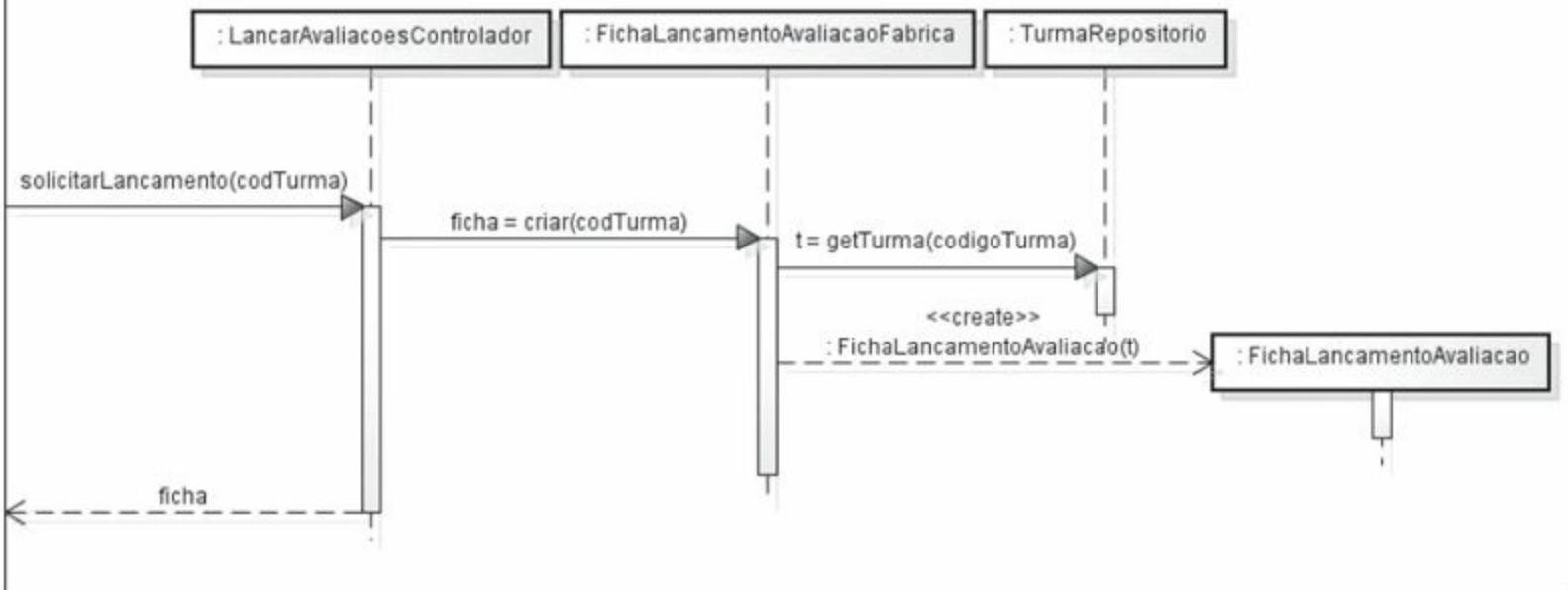


Figura 7-38: Interações resultantes da operação gerarFichaLancamento.

sd lancarAvaliacoes

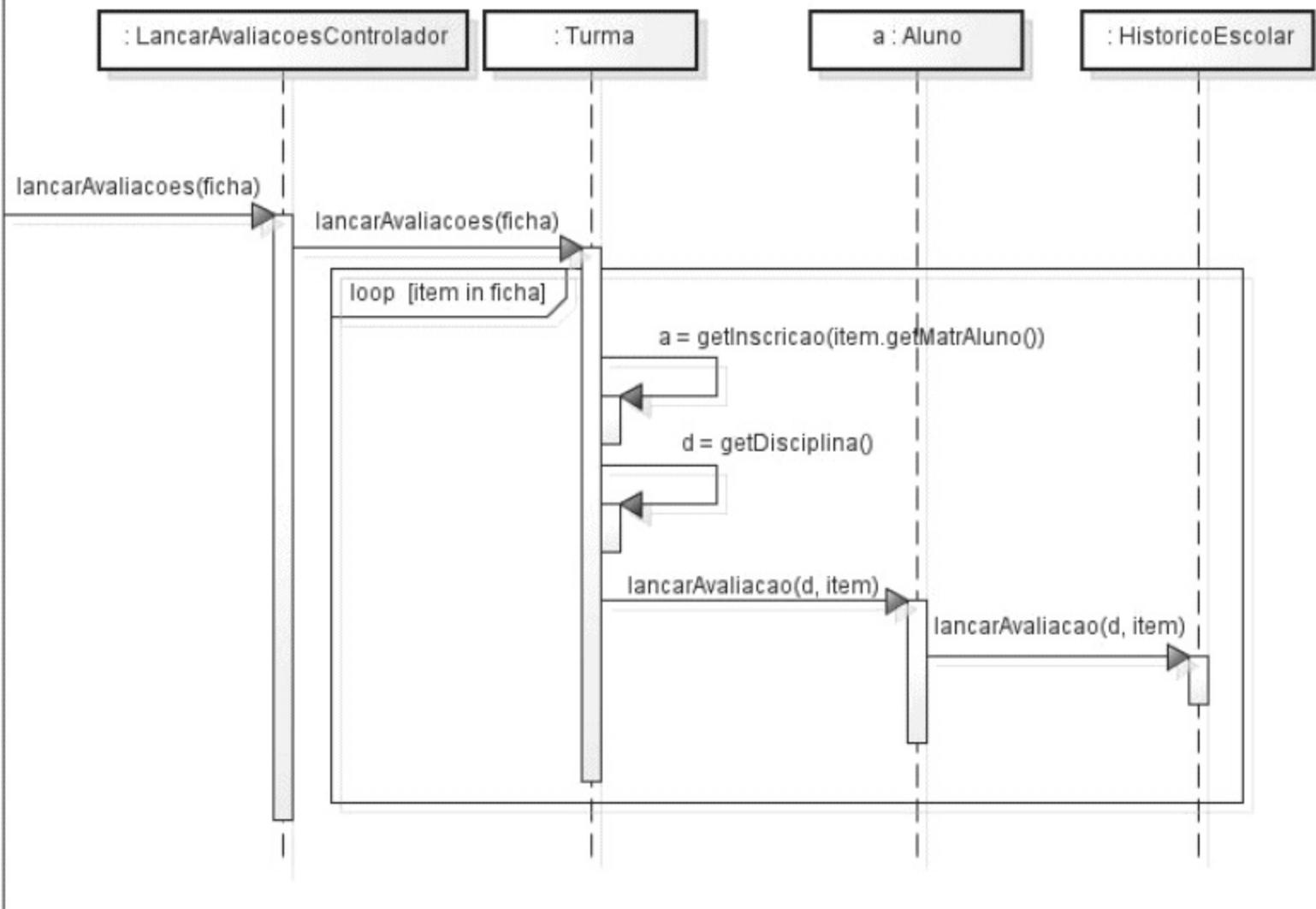


Figura 7-39: Interações resultantes da operação `lancarAvaliacoes`.

7.7.4 Visão geral das interações em um caso de uso

Na [Seção 7.4](#), mencionamos que a especificação da UML 2.0 introduziu um novo diagrama de interação, o diagrama de visão geral da interação. Como seu próprio nome diz, esse diagrama permite apresentar uma visão de alto nível de certa interação.

Para dar um exemplo desse diagrama no contexto do SCA, considere a [Figura 7-40](#), na qual apresentamos o diagrama de visão geral da interação para o caso de uso *Realizar Inscrição*.

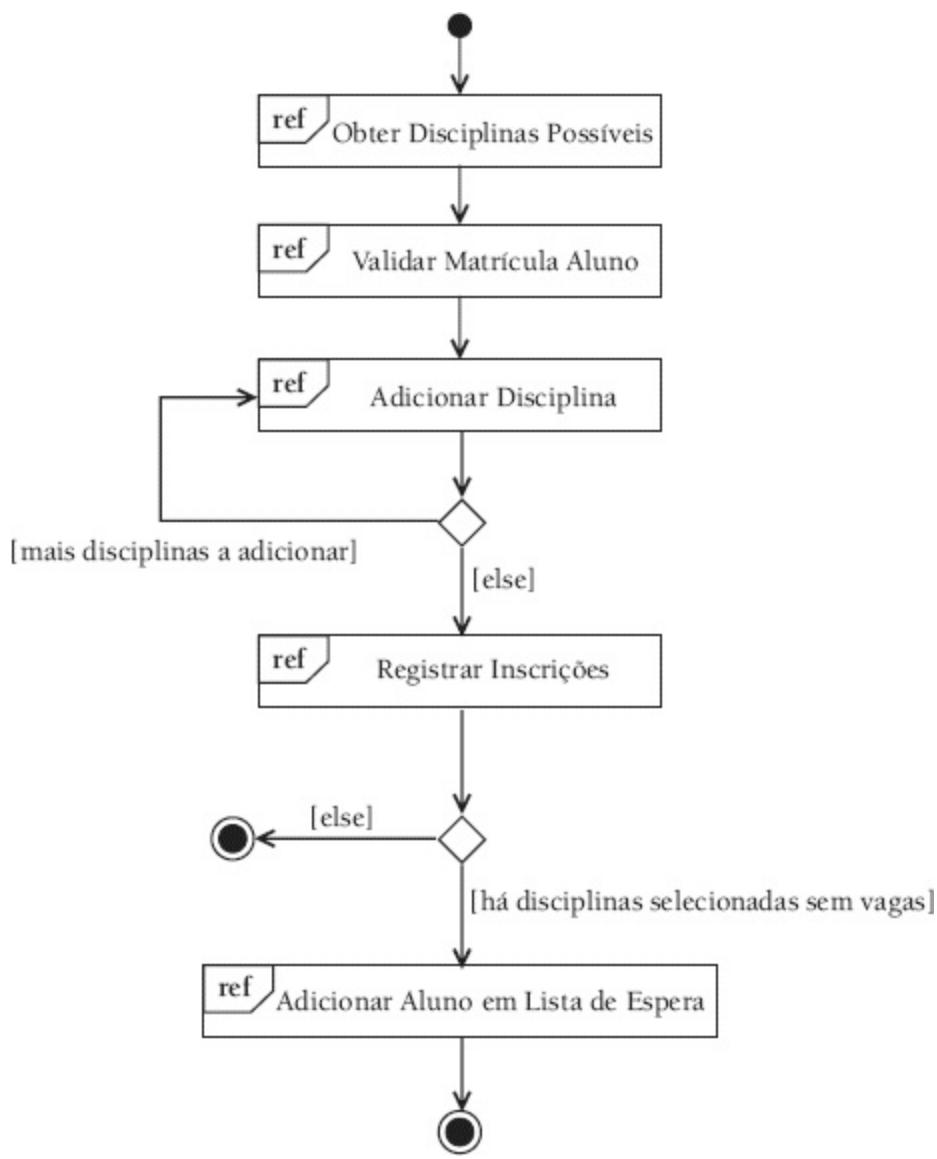


Figura 7-40: Visão geral da interação do caso de uso Realizar Inscrição.

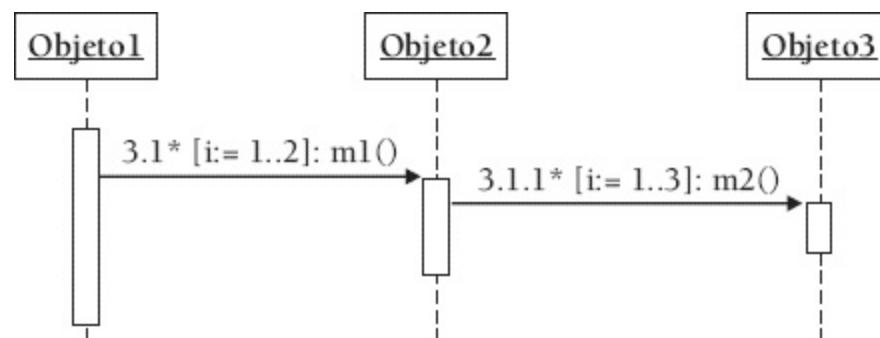
O diagrama de visão geral da interação é um tipo especial de diagrama de atividade. Para uma explicação mais detalhada da notação utilizada em diagramas de atividade, veja o [Capítulo 10](#).

► EXERCÍCIOS

7-1: Descreva a posição dos diagramas de interação no processo de desenvolvimento incremental e iterativo. Quando eles são utilizados? Para que são utilizados?

7-2: Ivar Jacobson, um dos proponentes da UML, disse uma vez: “Somente após a construção de diagramas de interação para os cenários de um caso de uso pode-se ter certeza de que todas as responsabilidades que os objetos devem cumprir foram identificadas.” Reflita sobre essa afirmativa. A construção dos diagramas de interação é realmente essencial para a definição das responsabilidades dos objetos participantes de um caso de uso? Justifique.

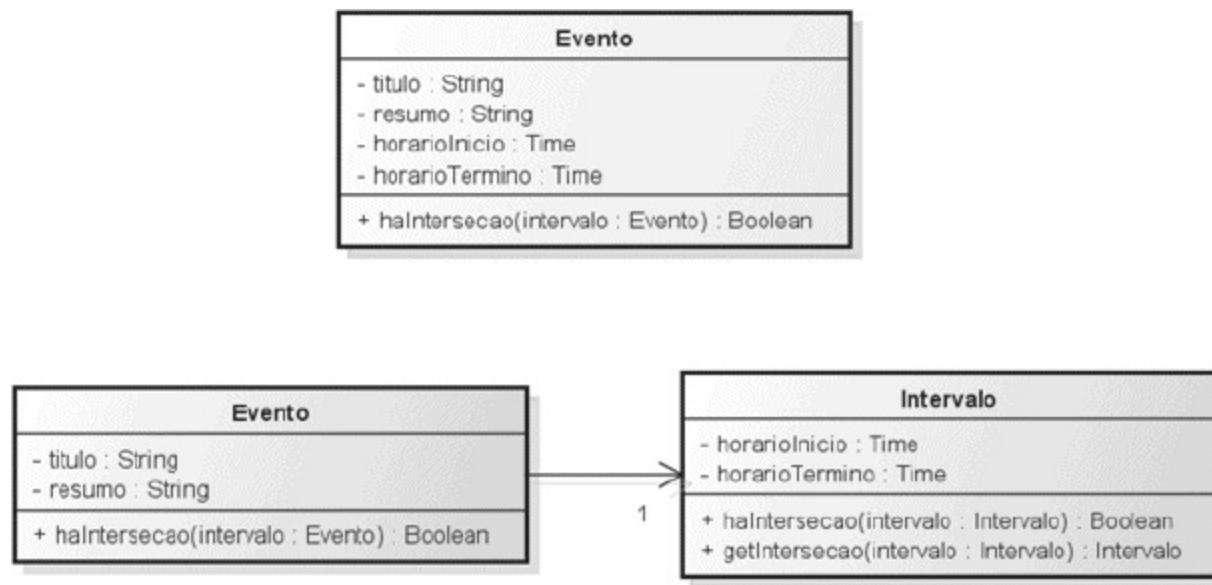
7-3: Considere o fragmento de diagrama de sequência a seguir. Determine a ordem na qual as mensagens m1 e m2 serão passadas.



7-4: Desenhe um diagrama de comunicação equivalente ao diagrama de sequência do exercício anterior.

7-5: Construa os diagramas de comunicação equivalentes aos diagramas de sequência apresentados na [Figura 7-24](#) e na [Figura 7-25](#).

7-6: Considere um SSOO que tem o objetivo de permitir a organização de eventos que podem ocorrer em um congresso científico. Exemplo de eventos são palestras, apresentações de artigo, mesas redondas etc. Cada evento possui características como título, resumo e horários de início e término de sua ocorrência. Nesse contexto, dois projetistas elaboraram projetos diferentes para representar o conceito de evento nesse sistema, conforme apresentado na figura a seguir. Apresente uma justificativa acerca de qual dos dois projetos abaixo possui maior coesão.



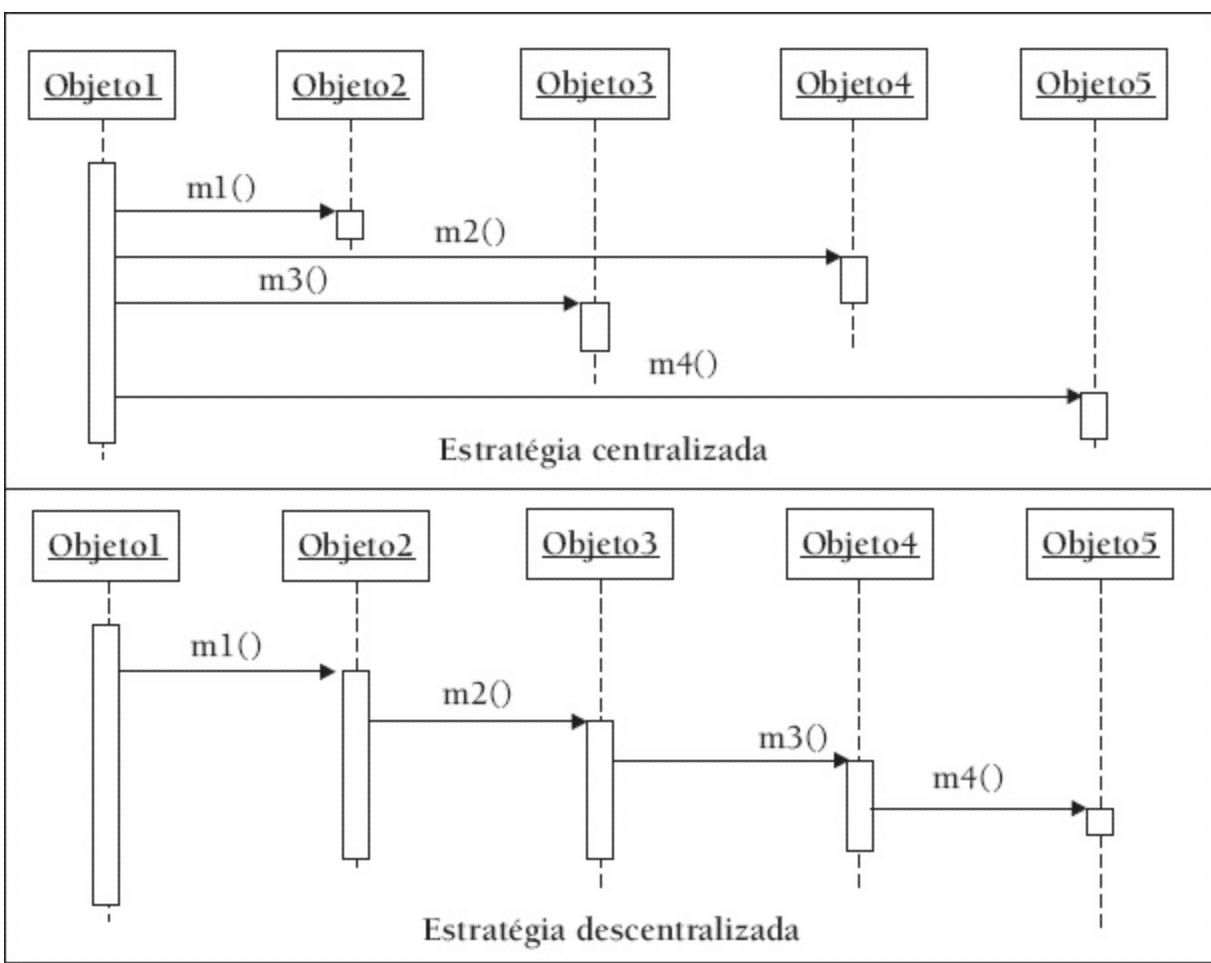
7-7: Considere a classe RelatorioOrcamentoConsolidado apresentada a seguir, que faz parte de um SSOO e cujo objetivo é gerar um relatório com o orçamento consolidado dos serviços a serem prestados a um cliente em uma empresa. Essa classe contém

operações para abrir uma conexão com o mecanismo de armazenamento persistente (para obter informações necessárias à geração do relatório), para a geração propriamente dita do relatório, para armazenar o relatório em um arquivo do sistema operacional e para enviar o relatório para impressão. O projeto dessa classe é adequado do ponto de vista do princípio da coesão? Forneça uma solução alternativa para o projeto dessa classe.

RelatorioOrcamentoConsolidado
abrirConexao() : Conexão gerarRelatorio() : OrcamentoConsolidado armazenarRelatorio() : void imprimirRelatorio() : void

7-8: Uma mensagem enviada a um objeto invoca a execução de uma de suas operações. Por outro lado, em linguagens de programação estruturadas (não orientadas a objetos) existe o conceito de *chamada de sub-rotinas*, em que sub-rotinas fazem chamadas a outras sub-rotinas dentro de um módulo de um programa. Discuta as semelhanças e as diferenças entre os conceitos de passagem de mensagem e de chamada de sub-rotinas.

7-9: De acordo com a divisão de responsabilidades pelos objetos de um sistema, a colaboração entre eles para a realização de um cenário pode ser classificada em um espectro que vai desde a forma *centralizada* até a forma *descentralizada*.² Na primeira forma de colaboração (centralizada), a inteligência do sistema está concentrada em um único objeto. Por outro lado, na forma descentralizada, a inteligência do sistema está mais uniformemente espalhada pelas classes. A figura a seguir apresenta de maneira esquemática as duas estratégias de colaboração, utilizando a notação vista para diagramas de sequência. Note que, na estratégia centralizada, há um objeto que controla os demais (Obj1). Já na estratégia descentralizada, há uma cadeia de *delegações* entre os objetos; não há um objeto central que “conhece” todos os demais. Cada objeto realiza uma parte da tarefa. Discuta as vantagens e desvantagens de cada uma dessas estratégias.



1. Para ser mais exato, um objeto pode enviar uma mensagem para si próprio por meio de uma *mensagem reflexiva*.

Modelagem de classes de projeto

A perfeição (no projeto) é alcançada, não quando não há nada mais para adicionar, mas quando não há nada mais para retirar.

— ERIC RAYMOND, THE CATHEDRAL AND THE BAZAAR

Conforme descrito no [Capítulo 6](#), é na etapa de projeto que são definidos os detalhes da solução para o problema relativo ao desenvolvimento de um SSOO. Nessa etapa, o objetivo é encontrar alternativas técnicas para que o sistema atenda aos requisitos funcionais, ao mesmo tempo em que respeite as restrições definidas pelos requisitos não funcionais definidos na etapa de análise.

Iniciamos o detalhamento das atividades da etapa de projeto de um SSOO no [Capítulo 7](#), com a apresentação de aspectos da modelagem de interações entre objetos. Por outro lado, naquele mesmo capítulo, mencionamos o fato de que a atividade de modelagem de interações gera informações para completar o modelo de classes construído na etapa de análise.

Neste capítulo, damos continuidade ao detalhamento das atividades da etapa de projeto de um SSOO. Na [Seção 8.1](#) apresentamos uma descrição dos mecanismos de reúso, bastante utilizados nesta etapa do desenvolvimento. A seguir, descrevemos algumas alterações pelas quais passam as classes e suas propriedades com o objetivo de transformar o modelo de classes de análise no modelo de classes de projeto. Em particular apresentamos notação adicional e transformações sobre atributos ([Seção 8.2](#)), operações ([Seção 8.3](#)) e associações ([Seção 8.4](#)). Na [Seção 8.5](#), voltamos a abordar o relacionamento de generalização, apresentando dessa vez outros elementos de notação e princípios de modelagem que normalmente não são usados na análise, mas que são importantes na construção do modelo classes de projeto. Nesta seção, descrevemos conceitos como classes abstratas, interfaces, polimorfismo. Além disso, voltamos a estudar o princípio do acoplamento, descrevendo formas de aplicação desse princípio. Na [Seção 8.6](#), apresentamos uma breve introdução a alguns padrões de projeto. Na [Seção 8.7](#), apresentamos uma discussão da modelagem de classes de projeto no contexto de um projeto de desenvolvimento iterativo. Finalmente, na [Seção 8.8](#), apresentamos a continuação da modelagem de nosso estudo de caso, o Sistema de Controle Acadêmico.

8.1 Reúso: padrões, frameworks, bibliotecas, componentes

Durante o projeto de um SSOO, os projetistas devem usar o máximo de sua experiência e criatividade para definir a solução mais adequada possível para o problema em questão. Quando a modelagem de um SSOO chega nessa etapa, fica claro que somente objetos das classes identificadas na análise não são suficientes para cumprir as responsabilidades definidas para esse sistema. Portanto, na etapa de projeto é necessário definir classes para implementar as funcionalidades do SSOO definidas na análise. Nesse contexto, diversos aspectos precisam ser levados em

consideração. Alguns exemplos de casos para os quais precisamos definir classes são:

- distribuição do sistema por diversos nós de processamento e consequente necessidade de comunicação entre esses nós,
- controle de segurança (i.e., autorização e autenticação de acesso),
- gerenciamento de transações,
- registro de ações realizadas (i.e., *logging*),
- acesso e armazenamento de informações persistentes,
- interação com usuários de diferentes naturezas (seres humanos, outros sistemas, dispositivos móveis etc),
- leitura de parâmetros de configuração a partir de arquivos no sistema operacional etc.

Cada um dos aspectos listados acima implica em responsabilidades adicionais do sistema para as quais o projetista pode definir novas classes. É também comum a **reutilização de classes e de soluções** preexistentes durante a etapa de projeto. Essas fontes de reutilização devem ser pesquisadas pelos projetistas para evitar o desenvolvimento (codificação) desnecessário. Fontes de reutilização são **padrões de projeto, bibliotecas de classes, frameworks e componentes**. Descrevemos essas fontes a seguir.

Um *padrão de projeto* é uma descrição da essência da solução para um problema que ocorre de forma recorrente no desenvolvimento de software. Um padrão de software descreve um punhado de objetos em colaboração para solucionar o problema recorrente. Padrões de software são normalmente identificados e catalogados por desenvolvedores de software experientes. O uso de padrões de software permite que desenvolvedores sejam mais produtivos, pois evita que estes desenvolvedores “reinventem a roda” durante o desenvolvimento: se existe algum padrão de software para um problema com o qual o desenvolvedor se depara, a solução apontada por esse padrão serve como ponto de partida. Descrevemos alguns exemplos de padrões de software nas [Seções 5.4.4](#) e [8.6](#). Além disso, outros padrões de projeto, como View Helper, DAO e Front Controller, que fazem parte dos padrões do catálogo *J2EE* (ALUR *et al.*, 2003), são descritos em outras partes deste livro, conforme resumido na [Seção 6.7](#).

Padrões de projeto fornecem uma forma de reúso abstrata, uma vez que eles descrevem a *essência da solução* para algum problema recorrente no desenvolvimento de software. Uma vez que é tomada a decisão de que a solução descrita por um determinado padrão de projeto é adequada em um determinado contexto, os desenvolvedores devem definir classes para implementar a solução. Duas formas mais concretas de reúso são os frameworks e as bibliotecas de classes, descritos a seguir.

Um **framework** é uma coleção de classes em colaboração, assim como em um padrão de projeto. Entretanto, há duas diferenças marcantes entre um framework e um padrão de projeto. Em primeiro lugar, um framework é muito maior (possui uma quantidade maior de classes) que o definido em um padrão de projeto. Além disso, um framework apresenta implementação; em um padrão, a solução apontada pelo mesmo deve ser implementada pelo desenvolvedor. Dois exemplos de frameworks de código aberto são o *Hibernate* (<http://www.hibernate.org>) e o *JUnit* (<http://www.junit.org>). O primeiro é um framework para persistência em um SGBD relacional para Java e .NET. O segundo é um framework para auxiliar o desenvolvedor a definir testes para sua aplicação. Outro exemplo de framework, dessa vez de propósito genérico, é o *Spring* (<http://projects.spring.io/spring-framework/>). Algumas funcionalidades importantes em aplicações OO modernas e fornecidas pelo Spring são: suporte para injeção de dependências, gerenciamento de transações, controle de

segurança, construção de interfaces em aplicações WEB, acesso a dados e definição de testes.

Finalmente, uma **biblioteca de classes** corresponde a uma coleção de classes que possuem algum objetivo específico. Bibliotecas são aglomerados de classes e de interfaces que publicam serviços por meio de uma interface de programação de aplicações (*Application Programming Interface - API*). Um exemplo de biblioteca é a Log4j (<http://logging.apache.org/log4j/>), usada para registrar ações realizadas durante a execução de um SSOO. As próprias plataformas de desenvolvimento como Java e .NET já fornecem diversas bibliotecas de classes reutilizáveis; no contexto da plataforma Java, RMI (veja a [Seção 11.2.3](#)) e JDBC (veja a [Seção 12.2.3](#)) são exemplos de especificações para as quais existem diversas implementações na forma de bibliotecas de classes. Outro exemplo de biblioteca nessa plataforma é a Joda-Time (www.joda.org/joda-time), utilizada para a manipulação de informações relacionadas a intervalos de tempo e datas em geral.

Tanto frameworks quanto bibliotecas são mecanismos de reúso utilizáveis durante o desenvolvimento de um SSOO. A diferença básica entre os dois está relacionada ao mecanismo denominado **inversão de controle**. Quando uma aplicação invoca uma operação em uma biblioteca, é a aplicação que está no controle. Por outro lado, se uma aplicação é implementação com o uso de algum framework, normalmente é o código desse framework que está no controle e que invoca partes do código desenvolvido pelos programadores do SSOO. Em um framework é implementada boa parte de algum aspecto de uma aplicação (como persistência de objetos ou interação com o usuário). Tudo o que o desenvolvedor deve fazer é completar a lógica já existente no framework com o comportamento específico necessário no SSOO. Ou seja, com o uso de um framework, o controle da execução (relativo a terminado aspecto da aplicação) é *invertido*, e passa da aplicação para o próprio framework. Por outro lado, frameworks e bibliotecas possuem características em comum. Uma delas é que ambos definem um API. Além disso, é comum implementar bibliotecas e frameworks por meio do uso de diversos padrões de projeto. Frameworks também fazem bastante uso de diversas bibliotecas em sua implementação. Além disso, é comum o uso de variados padrões de projeto tanto na implementação de frameworks quanto de bibliotecas. A [Figura 8-1](#) ilustra as relações existentes entre bibliotecas e frameworks.



Figura 8-1: Relação entre os conceitos de framework e de biblioteca.

Outro conceito relevante na fase de projeto e relacionado ao reúso é o de **componente**. A melhor maneira de entender os componentes de um software é com uma analogia com a engenharia elétrica. Para isso, considere um chip de memória. Essa peça é um tipo de componente eletrônico de um computador. Ele pode ser utilizado para construir um computador e também ser substituído por outro chip mais poderoso que possua a mesma especificação (interface). Da mesma forma, um componente de software é uma unidade que pode ser utilizada na construção de vários sistemas e que pode ser

substituída por outra unidade com a mesma funcionalidade. Sendo assim, podemos pensar em um sistema de software constituído de diversos componentes, elementos que existem a tempo de execução do sistema. Quando necessário, esses elementos podem ser substituídos por outros equivalentes (de mesma interface) e mais sofisticados. Além disso, o mesmo componente pode ser utilizado na construção de diversos sistemas de software. As tecnologias COM (Microsoft™), CORBA (OMG™) e Enterprise Java Beans (Oracle™) são exemplos de tecnologias baseadas em componentes. Além disso, os clientes podem configurar os componentes para modificar sua forma de funcionar. Um componente pode prover acesso aos seus serviços por meio de uma *interface* (ver [Seção 8.5.4](#)). Adicionalmente, um componente fornece *serviços* a outros componentes do sistema, mas também pode requisitar serviços. Componentes de software existem a tempo de execução do sistema. Quando construído segundo o paradigma da orientação a objetos, um componente é tipicamente composto de diversos objetos. Nesse caso, a interface do componente é constituída de um ou mais serviços que as classes dos referidos objetos implementam. Ou seja, um componente comprehende a colaboração de vários objetos.

Existe uma área ativa de pesquisa, conhecida como *desenvolvimento baseado em componentes* (*Component Based Development*, CBD), cujo objetivo é a construção de componentes que tenham utilidade em diversas situações, não só no sistema no qual eles foram construídos. Dessa forma, com o passar do tempo, o desenvolvimento de software passaria a ser cada vez mais baseado no *reúso* de componentes preexistentes do que na sua construção e teste. Isso colocaria os componentes de software no mesmo nível dos componentes das outras áreas da engenharia. A [Figura 8-2](#) ilustra de forma esquemática o processo de construção de componentes. De acordo com essa figura, no desenvolvimento de um sistema de software, diversas partes são organizadas para se tornarem componentes reutilizáveis. Esses componentes são catalogados e armazenados em um repositório e, posteriormente, podem ser reutilizados no desenvolvimento de outros sistemas de software.

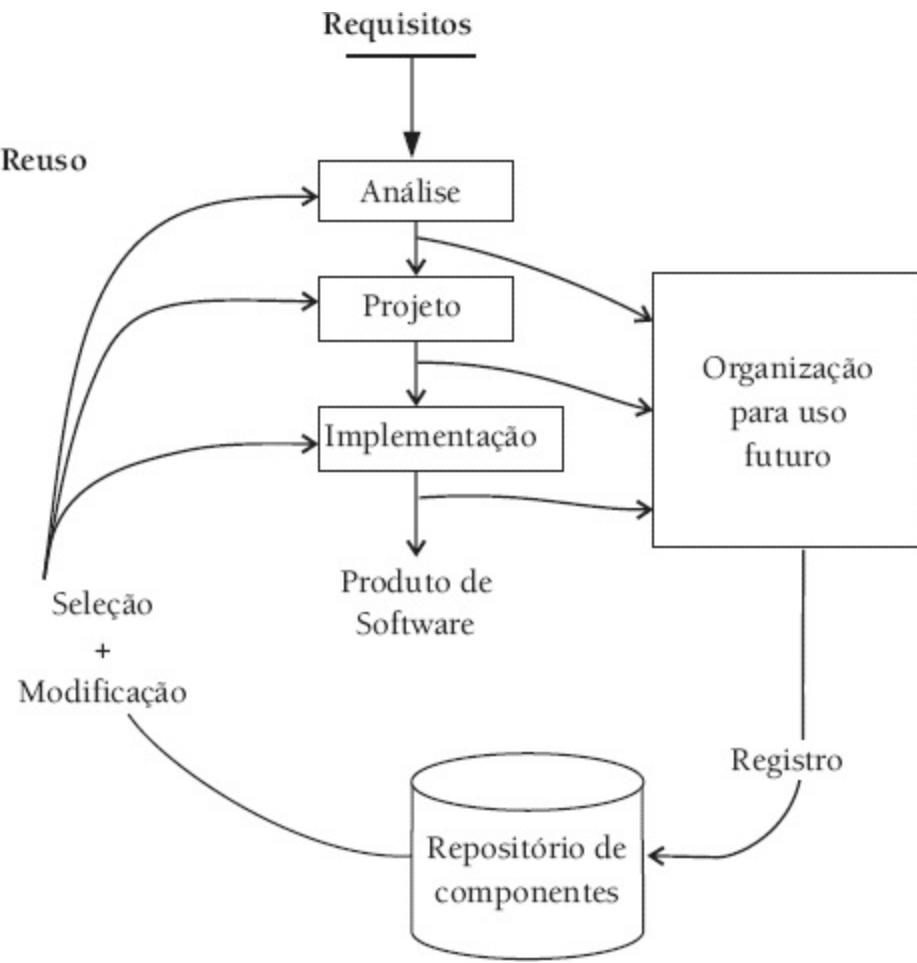


Figura 8-2: Reúso por meio de componentes de software.

8.2 Especificação de atributos

8.2.1 Notação da UML para atributos

No modelo de classes de análise, os atributos, quando representados, são somente pelo nome. No entanto, a sintaxe completa da UML para definição de um atributo é bem mais detalhada. Essa sintaxe, cujo uso é apropriado para a fase de especificação do modelo de classes, é a seguinte:

/|visibilidade nome: tipo = valor_inicial

Vamos descrever cada um dos elementos dessa sintaxe. (A [Figura 8-3](#) ilustra a utilização desses elementos para atributos da classe Cliente.)

A *visibilidade* de um atributo diz respeito ao seu nível de acesso. Ou seja, ela permite definir quais atributos de um objeto são acessíveis por outros objetos. A visibilidade de um atributo pode ser *pública*, *protegida*, *privativa* ou *de pacote*. A primeira é a visibilidade por omissão (se nenhuma visibilidade for especificada, essa é assumida). O símbolo e o significado de cada visibilidade são apresentados na [Tabela 8-1](#).

Cliente
#nome : String -dataNascimento : Data -telefone : String #/idade : int

```

#limiteCrédito : Moeda = 500.0
-quantidadeClientes : int
-idadeMédia : float

```

Figura 8-3: Especificação dos atributos da classe Cliente.

A propriedade de visibilidade serve para implementar o *encapsulamento* da estrutura interna da classe. Somente as propriedades realmente necessárias ao exterior da classe devem ser definidas com visibilidade pública. Todo o resto é “escondido” dentro da classe através das visibilidades protegida e privativa. Um atributo definido com a visibilidade protegida diz respeito ao conceito de herança entre classes. Esse conceito é descrito na [Seção 8.5](#).

Tabela 8-1: Possíveis visibilidades de um elemento (atributo ou operação)

Visibilidade	Símbolo	Significado
Pública	+	Qualquer objeto externo pode obter acesso ao elemento, desde que tenha uma referência para a classe em que o atributo está definido.
Protegida	#	O elemento protegido é visível para subclasses da classe em que este foi definido.
Privativa	-	O elemento privativo é invisível externamente para a classe em que este está definido.
Pacote	~	O elemento é visível a qualquer classe que pertence ao mesmo pacote no qual está definida a classe.

O elemento *nome* da sintaxe do atributo corresponde ao nome do atributo. Na verdade, somente esse elemento é obrigatório na sintaxe de declaração de um atributo.

O elemento *tipo* especifica o tipo do atributo. Esse elemento é dependente da linguagem de programação na qual a classe deve ser implementada. O tipo definido para um atributo pode ser definido pela utilização de um *tipo primitivo* da linguagem de programação a ser utilizada na implementação. Por exemplo, se estamos utilizando a linguagem Java, temos, dentre outros, os seguintes tipos primitivos: int, float e boolean. É também comum a utilização de *tipos abstratos de dados* (tradução para *abstract data type*, TAD) para declarar atributos. Um TAD é um tipo definido pelo usuário ou disponível na linguagem de programação que contém uma estrutura interna. Outra possibilidade é a utilização de algum tipo simples disponível na própria linguagem de programação. Por exemplo, na [Figura 8-3](#), os atributos matrícula e Nome são definidos como do tipo String. Além de String, temos (em Java): BigDecimal, Long e Date. Atributos de tipos simples dessa natureza são representados diretamente no comportamento de atributos em vez de ser representado como uma classe separada no diagrama de classes. Isso porque o diagrama ficaria muito carregado visualmente se essas classes simples fossem representadas por retângulos separados.

A maioria das linguagens de programação já fornece tipos abstratos de dados. No entanto, os desenvolvedores podem definir seus próprios tipos. Por exemplo, a [Figura 8-3](#) apresenta o atributo limiteCrédito como do tipo Moeda, um TAD para manipular valores monetários. Outros exemplos de TAD são listados a seguir. No DDD, eles são os denominados Objetos Valor (veja a [Seção 5.4.3.2](#)).

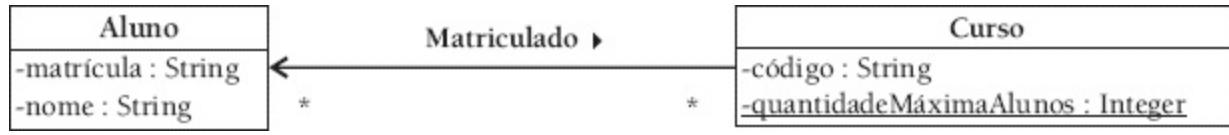
- Data (para processamento de datas em geral).
- Horário (para processamento de valores de tempo).
- Endereço (para armazenamento de endereços em geral).
- CódigoPostal, CPF e CNPJ (para armazenamento e validação desses valores).

Um *valor inicial* também pode ser declarado para o atributo. Dessa forma, sempre que um objeto de uma classe é instanciado, o valor inicial declarado é automaticamente definido para o atributo. Assim como o tipo, o valor inicial de um atributo também é dependente da linguagem de programação.

Pode-se definir um *atributo derivado* em uma classe. Um atributo é derivado quando o seu valor pode ser obtido a partir do valor de outro(s) atributo(s). Por exemplo, o valor da idade de uma pessoa pode ser obtido a partir de sua data de nascimento. Um atributo derivado é representado com uma barra inclinada à esquerda. Atributos derivados normalmente são definidos por questões de desempenho. Por exemplo, considere a classe Cliente da [Figura 8-3](#). Em vez de calcular a idade média de toda a coleção de clientes a cada vez que este valor é necessário, definimos o atributo derivado `idadeMédia` e um método seletor correspondente.

Outra característica de um atributo é seu *escopo*. Cada objeto tem um valor para cada atributo definido em sua classe. Esse é o caso mais comum, e diz-se que o atributo tem *escopo de objeto*. No entanto, pode haver atributos que tenham *escopo de classe*, ou seja, que armazenam valor comum a todos os objetos da classe. Esse tipo de atributo é definido por um sublinhado na sintaxe da UML. Um atributo que tenha escopo de classe é também denominado *atributo estático*. Por exemplo, a classe Cliente da [Figura 8-3](#) contém o atributo `quantidadeClientes` para armazenar a quantidade de instâncias criadas a partir dessa classe. Há apenas um valor desse atributo para todos os objetos da classe Cliente. O atributo `idadeMédia` também é estático.

Uma possível aplicação dos atributos estáticos acontece durante a implementação de regras de negócio. Por exemplo, considere a [Figura 8-4](#), em que a classe Curso possui um atributo estático `quantidadeMáximaAlunos`. Este atributo pode ser utilizado para implementação da seguinte regra de negócio: *Em um curso podem estar matriculados até 30 alunos*.



[Figura 8-4:](#) Atributos estáticos podem ser utilizados para implementar regras de negócio.

8.3 Especificação de operações

Na descrição do modelo de interações, declaramos que as operações de uma classe são identificadas em consequência da identificação de mensagens enviadas de um objeto a outro. No detalhamento dessas operações, devemos considerar diversos aspectos: seu nome, lista de parâmetros, tipo de cada parâmetro, tipo de retorno. Nesta seção, descrevemos a notação fornecida pela UML para realizar esse refinamento de operações. Descrevemos também alguns princípios e dicas recomendados durante essa atividade.

8.3.1 Notação da UML para operações

As operações de uma classe correspondem a algum processamento realizado por essa classe. Em termos de implementação, uma operação é uma rotina associada a uma classe. Conforme visto no [Capítulo 5](#), a construção do modelo de classes de análise permite a identificação de algumas operações. Durante a construção do modelo de interações, essas operações são validadas, e várias

outras são identificadas. Todas essas operações devem ser adicionadas ao diagrama de classes e devem ser documentadas com a definição de sua *assinatura*. A sintaxe definida na UML para a assinatura de uma operação é a seguinte:

visibilidade nome(parâmetros): tipo-retorno {propriedades}

Cliente
<pre>+obterNome() : String +definirNome(in umNome : String) +obterDataNascimento() : Data +definirDataNascimento(in umaData : Data) +obterTelefone() : String +definirTelefone(in umTelefone : String) +obterLimiteCrédito() : Moeda +definirLimiteCrédito(in umLimiteCrédito : float) +obterIdade() : int <u>+obterQuantidadeClientes() : int</u> <u>+obterIdadeMédia() : float</u></pre>

Figura 8-5: Especificação das operações da classe Cliente.

O elemento *nome* corresponde ao nome dado à operação. A *visibilidade* usa a mesma simbologia das visibilidades para atributos. Se uma operação é pública, ela pode ser ativada com a passagem de uma mensagem para o objeto. Se a é protegida, ela só é visível para a classe e para seus descendentes (o conceito de *descendente* de uma classe é descrito no [Capítulo 9](#)). Finalmente, se a operação é privativa, somente objetos da própria classe podem executá-la. Os *parâmetros* de uma operação correspondem às informações que ela recebe quando é executada. Normalmente, essas informações são fornecidas pelo objeto remetente da mensagem que requisita a execução da operação no objeto receptor. Uma operação pode ter zero ou mais *parâmetros*, que são separados por vírgulas. Cada parâmetro da lista tem a seguinte sintaxe:

direção nome-parâmetro: tipo-parâmetro

O elemento *direção* serve para definir se o parâmetro pode ou não ser modificado pela operação. Por meio desse elemento, o modelador pode definir se o parâmetro é de entrada, saída ou ambos. Se a direção for omitida, o valor assumido pelo parâmetro é *in*. Os significados das direções são fornecidos na [Tabela 8-2](#). Note que a direção *in* corresponde à passagem de parâmetros por valor, enquanto as direções *out* e *inout* correspondem à passagem de parâmetros por referência. Essas formas de passagem de parâmetros são comuns à maioria das linguagens de programação, resguardadas as diferenças sintáticas de cada uma.

Tabela 8-2: Possíveis direções na definição de um parâmetro

Direção	Significado definido pela UML
In	Parâmetro de entrada: não pode ser modificado pela operação. Serve somente como informação para o objeto receptor.
Out	Parâmetro de saída: pode ser modificado pela operação para fornecer alguma informação ao objeto remetente.
inout	Parâmetro de entrada que pode ser modificado.

O elemento *nome-parâmetro* corresponde ao nome do parâmetro. Cada parâmetro deve ter um nome único dentro da assinatura da operação. O elemento *tipo-parâmetro* corresponde ao tipo do parâmetro e é dependente da linguagem de programação. O elemento *tipo-retorno* representa o tipo de dados do valor retornado por uma operação. Esse tipo depende da linguagem de programação. Assim como atributos, as operações também possuem um *escopo*. Uma operação pode ter escopos *de classe* ou *de instância*. Uma operação que tem escopo de classe processa atributos estáticos. Se essa operação tem escopo de classe, ela é também chamada de ***operação estática***. Uma operação que tenha escopo de instância indica que ela processa atributos que têm escopo de instância.

8.3.2 Dicas práticas

Recomenda-se atribuir um nome a uma operação de forma que este lembre o resultado da operação e tenha a forma geral <verbo>+<complemento>. Além disso, o desenvolvedor deve se lembrar de que uma operação é um serviço que a classe presta. Por consequência, esse nome deve ser definido a partir da perspectiva externa, e não interna. Ele deve também deixar claro o que a operação produz como resultado, e não como ela faz isso. Por exemplo, considere a [Figura 8-5](#). Essa figura apresenta uma operação de nome `obterIdadeMédia()`. Esse nome é mais apropriado do que o nome `calcularIdadeMédia()`, por exemplo.

Note que todas as operações que correspondem a mensagens de um objeto a outro em um diagrama de interação (ver [Seção 7.5.1](#)) devem ter visibilidade pública ou de pacote. Isso porque, para um objeto enviar uma mensagem a outro, a operação no objeto receptor deve ser conhecida pelo objeto remetente da mensagem.

O desenvolvedor deve definir operações com o mínimo de parâmetros possível. Uma razão para isso é que quanto menos parâmetros uma operação tiver, mais fácil é a utilização da mesma. Além disso, quanto menor a quantidade de parâmetros de uma operação, menor é a dependência entre as classes envolvidas. É também adequado que o modelador defina, para cada parâmetro, seu valor assumido (se existir).

Da mesma forma que a lista de parâmetros de uma operação deve ser a mínima possível, assim também acontece com a lista de operações que podem modificar o estado de um objeto de certa classe. Dessa forma, o projetista deve avaliar cuidadosamente quais atributos de certa classe podem ser alterados e quais operações são realmente necessárias para realizar a alteração.

Assim como para tipos de atributos e de parâmetros, o tipo de retorno também pode ser um TAD. Nem todas as operações retornam um valor, portanto, esse elemento nem sempre é especificado.

8.3.3 Projeto por contrato

Além de todas as informações descritas no diagrama de classes, o modelador deve também especificar o *objetivo* e o *funcionamento* de cada operação. A descrição do funcionamento, uma descrição da lógica da operação, é útil quando o algoritmo da operação for complexo. O diagrama de atividades da UML, descrito no [Capítulo 10](#), é uma boa ferramenta para realizar essa descrição do funcionamento.

Outra forma de documentar uma operação é com a técnica de *projeto por contrato* (tradução para *Design by Contract, DbC*), proposta por Bertrand Meyer, o criador da linguagem de programação orientada a objetos *Eiffel* (MEYER, 1997). O objetivo da aplicação dessa técnica é obter uma especificação mais rigorosa das operações de uma classe (ou interface) que seja útil para a implementação, o teste e o uso das mesmas. Meyer utiliza o termo *fornecedor* para denotar o

desenvolvedor, e *consumidor* para o usuário. Se uma classe A utiliza os serviços de uma classe B, então B é fornecedor e A é o consumidor. A ideia básica do projeto por contrato é criar um contrato entre o *desenvolvedor* e os consumidores de uma operação. Para cada parte envolvida no contrato, há obrigações e benefícios. Para um resumo, veja a [Tabela 8-3](#).

No projeto por contrato há dois elementos, as *invariantes* e as *asserções*. Uma asserção, por sua vez, pode ser de dois tipos: *precondição* ou *pós-condição*. Vamos descrever dois elementos componentes da técnica DbC nos próximos parágrafos.

Uma *precondição* é uma condição atrelada a uma operação e que deve ser satisfeita pelo consumidor ao requisitar a execução da operação ao fornecedor. A cada operação, é adicionado um conjunto de precondições. Como exemplo de precondição, podemos citar a faixa de valores possíveis para certo parâmetro de uma operação. Além das precondições, para cada operação, também são definidas *pós-condições*. Considere que $C_{\text{pré}}$ e $C_{\text{pós}}$ são dois conjuntos de precondições e de pós-condições, respectivamente, associados a certa operação. O fornecedor deve garantir que cada pós-condição pertencente a $C_{\text{pós}}$ é satisfeita após a execução da operação, contanto que as precondições pertencentes a $C_{\text{pré}}$ tenham sido satisfeitas pelo consumidor quando da chamada da operação. Se pelo menos uma das precondições em $C_{\text{pré}}$ não forem garantidas pelo consumidor, a execução correta da operação não é garantida pelo fornecedor.

Outro elemento da técnica de DbC é a *invariante*. A cada classe pode ser associada a um conjunto de invariantes (note a diferença em relação às asserções, que são associadas a cada operação). Uma invariante de uma classe C é uma condição que deve obrigatoriamente ser satisfeita após quaisquer mudanças realizadas no estado de algum objeto de C. O objetivo de uma invariante é garantir que o objeto permanece em um *estado válido* após qualquer modificação aplicada sobre ele.

Tabela 8-3: Obrigações de benefícios associados a consumidores e a fornecedores no DbC

	Obrigações	Benefícios
Consumidor	Satisfazer precondições	Resultado satisfaz pós-condições e invariantes
Fornecedor	Satisfazer precondições e invariantes	Implementação pode confiar na validade das precondições

A técnica DbC pode ser aplicada formalmente com o uso da Linguagem de Restrição de Objetos (ver [Seção 3.6](#)) para definir as asserções de cada operação e invariantes de cada classe. Dois outros exemplos de linguagens de especificação formais são a VDM (*Vienna Development Method*) e a Z.

A definição de asserções e invariantes influencia diretamente na implementação da classe em questão. Mais especificamente, essa influência diz respeito à forma de validar os valores dos parâmetros de suas operações e de seus atributos. Com relação a isso, um recurso bastante comum em linguagens de programação OO, e que pode ser utilizado para implementar asserções e invariantes, é a *exceção*. Uma exceção normalmente corresponde a uma classe predefinida na linguagem que pode ser utilizada para notificar alguma situação de erro durante o processamento de uma operação. Quando uma situação de exceção ocorre durante uma operação, um objeto de alguma classe de exceção correspondente ao erro acontecido pode ser criado e retorna a região de código que fez a chamada à operação. Particularmente no caso dos parâmetros de operações e dos atributos de uma classe, um exemplo de situação de exceção é o fato de ser atribuído a algum desses elementos

um valor inválido. O [Quadro 8-1](#) apresenta, novamente com a sintaxe da linguagem Java, um exemplo de validação de invariantes da classe Aluno com o uso de exceções. Note que essa operação (na verdade, um dos *construtores* da classe) utiliza a exceção chamada `IllegalArgumentException` para notificar o chamador de que houve algum erro na passagem dos argumentos para a operação. De qualquer forma, é importante que o projetista defina: (1) a faixa de valores válidos para cada parâmetro de operação e para o atributo em uma classe; e (2) que exceções devem ser disparadas quando a verificação desses valores identificar algum valor inadequado.

Quadro 8-1: Exemplo de validação de invariantes com o uso de exceções

```
// Invariante: sexo in {"M", "F"}  
// Invariante: nome é uma cadeia de caracteres não vazia  
// Invariante: matrícula é uma cadeia de caracteres não vazia  
public class Aluno {  
    private String nome;  
    private String matrícula;  
    private Char sexo;  
    ...  
    public Aluno(final String nome, final String matrícula, final Char sexo) {  
        if(nome == null || nome.equals("")) {  
            throw new IllegalArgumentException("Valor inválido para nome.");  
        }  
        if(matrícula == null || matrícula.equals("")) {  
            throw new IllegalArgumentException("Valor inválido para matrícula.");  
        }  
        if(sexo == null || (!sexo.equals("M") && !sexo.equals("F"))) {  
            throw new IllegalArgumentException("Valor inválido para sexo.");  
        }  
        this.nome = nome;  
        this.matrícula = matrícula;  
        this.sexo = sexo;  
    }  
    ...  
}
```

[8.3.4 Operações de criação e destruição de objetos](#)

Duas operações utilizadas frequentemente nas interações entre objetos são a *criação* (instanciação) e a *destruição* de objetos. Essas operações correspondem às mensagens de criação e destruição de objetos, que descrevemos na [Seção 7.2.4](#). Uma operação de criação serve para instanciar um objeto. Operações de destruição normalmente não têm parâmetros e servem para liberar alguma quantidade de memória que tenha sido alocada dinamicamente na instanciação.

Normalmente há diversas operações de criação definidas em uma classe. Os parâmetros de uma operação de criação são utilizados para iniciar um ou mais atributos do objeto. Para definir os parâmetros de uma operação de criação, considere o seguinte: se o valor do atributo é essencial para o significado do objeto, é mais adequado iniciar seu valor logo no momento da criação do objeto. Se esse for o caso, uma das operações de criação deve ter um parâmetro que sirva de valor inicial para o atributo essencial. Diz-se também que operações de criação são utilizadas para definir o *estado inicial* de um objeto. O *estado* de um objeto é definido como o conjunto de valores de seus atributos em um dado momento. A modelagem de estados de um objeto é mais bem descrita no [Capítulo 9](#).

A forma de utilização de métodos de criação e destruição varia de uma linguagem de programação para outra. Por exemplo, na linguagem *Delphi* os métodos de criação (*Create*) e de destruição (*Destroy*) são invocados explicitamente. A linguagem *C++* fornece operadores para a criação e destruição de objetos (*new* e *delete*). Já as linguagens *C#* e *Java* possibilitam a criação de objeto através de um operador (*new*) e possuem um mecanismo automático de remoção (destruição) de objetos, quando esses não são mais necessários à aplicação.

8.3.5 Seletores e modificadores

Pelo princípio do *encapsulamento* (ver [Seção 1.2.3.1](#)), qualquer acesso a informações internas a uma classe deve ser realizado por meio de sua *interface*. A interface de uma classe é definida como o conjunto de suas operações de visibilidade pública.

Para assegurar o encapsulamento da classe, o projetista deve definir todos os seus atributos com visibilidade privativa ou protegida. A seguir, precisa definir operações para obter acesso e para modificar o valor de cada atributo. Essas operações são denominadas *seletores* e *modificadores*. (Também recebem o nome comum de *métodos de acesso*.) Um *seletor* retorna o valor de um atributo, enquanto um *modificador* altera o valor de um atributo.

Como convenção, o nome de um seletor é formado com o prefixo “obter” seguido do nome do atributo. Analogamente, o nome de um modificador é construído com o prefixo “definir” seguido do nome do atributo. Outra convenção comum é utilizar o nome do atributo para nomear tanto seu seletor quanto seu modificador. Como exemplo, considere novamente a [Figura 8-5](#), que apresenta seletores e modificadores para os atributos da classe *Cliente*. Repare que cada seletor possui um tipo de retorno. Já com os modificadores isso não acontece, embora cada um possua um parâmetro de mesmo tipo do atributo correspondente.

Nem sempre faz sentido ter um modificador para um atributo. Isso acontece quando o valor desse atributo é consultado, mas não alterado. Voltando ao exemplo da [Figura 8-5](#), os atributos estáticos só possuem seletores, pois seus valores dependem do número de objetos da classe.

8.3.6 Outras operações típicas

Durante o projeto de uma classe, diversos métodos devem ser definidos. Um exemplo disso são os *métodos de ligação*. Como o próprio nome deixa transparecer, esse tipo de método serve para definir ligações (conexões) entre objetos a tempo de execução do SSOO. Considere o [Quadro 8-2](#), que apresenta a implementação parcial de uma classe em linguagem Java para representar turmas. Note que essa classe possui um atributo que corresponde a uma coleção de objetos da classe *Inscricao*. Além disso, note que os métodos de ligação *inscrever* e *cancelarInscricao* estão definidos nessa classe. Esses métodos servem para criar e destruir conexões entre os objetos das classes *Turma* e *Inscricao* na ocasião da execução do sistema.

Quadro 8-2: Operações para manutenção de associações

```
public class Turma {  
    private Set<Inscricao> inscricoes = new HashSet<Inscricao>();  
    ...  
    public void inscrever(Aluno aluno) {  
        Inscricao inscrição = criarInscricao(aluno);  
        this.inscricoes.add(inscrição);  
    }  
}
```

```

}
public boolean cancelarInscricao(Aluno aluno) {
    Inscricao inscricao = obterInscricao(aluno);
    return this.inscricoes.remove(inscricao);
}
public Set<Inscricao> getInscricoes() {
    return Collections.unmodifiableSet(this.inscricoes);
}
}

```

Além dos métodos de ligação, outros métodos comumente encontrados para calcular valores derivados de atributos da classe são os de formatação e os de conversão e cópia de objetos. Além disso, alguns métodos que definimos em uma classe devem ter uma operação inversa óbvia. Alguns exemplos: *habilitar* versus *desabilitar*; *tornarVisível* versus *tornarInvisível*; *adicionar* versus *remover*; *depositar* versus *sacar* etc. De qualquer modo, devemos sempre validar a necessidade de cada método com relação ao modelo de interações (ver o [Capítulo 7](#)).

8.4 Especificação de associações

A especificação de relacionamentos (associações, agregações e composições) entre objetos é uma das atividades da etapa de projeto. Nesta seção, descrevemos alguns detalhes a serem adicionados para refinar os relacionamentos de associação identificados na análise. Aqui definimos o conceito de dependência, sua notação e de que forma esse conceito pode ser utilizado no refinamento de associações. Apresentamos também o conceito de navegabilidade de uma associação, que por sua vez está relacionado à modelagem de interações ([Capítulo 7](#)).

8.4.1 O conceito de dependência

No modelo de classes de análise, relacionamentos entre objetos são normalmente definidos apenas com o uso da associação (ou como um de seus casos especiais, a agregação ou a composição). Entretanto, o fato de existir uma associação entre duas classes implica a existência de uma *dependência* entre as mesmas. Uma dependência entre classes indica que uma classe depende dos serviços fornecidos pela outra.

No modelo de análise, é suficiente para o modelador identificar a existência de associações entre classes, que é uma forma de dependência. Mas, na fase de especificação do modelo de classes, essa dependência precisa ser mais bem definida pelo projetista, uma vez que a mesma tem influência na forma utilizada para implementar as classes envolvidas. Vamos descrever, então, detalhes das possíveis formas de dependência. Para isso, considere duas classes, A e B, onde A depende de B. Vários tipos de dependência entre essas duas classes podem existir. Esses tipos são descritos a seguir.

- Dependência por atributo:** A possui um atributo cujo tipo é B. A dependência por atributo é também chamada de *dependência estrutural*.
- Dependência por variável global:** A utiliza uma variável global cujo tipo é B.
- Dependência por variável local:** A possui alguma operação cuja implementação utiliza uma variável local de tipo B.

4. Dependência por parâmetro: A possui pelo menos uma operação, que possui pelo menos um parâmetro, cujo tipo é B.

Se analisarmos os tipos de dependência descritos anteriormente, podemos constatar que uma dependência realmente indica que uma classe depende dos serviços fornecidos por uma outra classe. Por exemplo, em uma dependência por parâmetro, possivelmente a implementação da operação em A invoca uma ou mais operações definidas em B.

A notação da UML para representar uma dependência no diagrama de classes é de uma seta tracejada ligando as classes envolvidas. A seta sai da classe dependente e chega na classe da qual depende. A [Figura 8-6](#) ilustra a notação da UML para dependências.



Figura 8-6: Notação para o relacionamento de dependência.

Ainda com respeito à notação, as dependências podem ser estereotipadas para indicar o tipo de dependência existente entre duas classes. A tabela a seguir exibe os estereótipos predefinidos na UML que podem ser utilizados em dependências.

Tabela 8-4: Estereótipos da UML para rotular relacionamentos de dependência

Estereótipo	Tipo de dependência
<<global>>	Por variável global
<<local>>	Por variável local
<<parameter>>	Por parâmetro

A [Figura 8-7](#) ilustra exemplos de dependências entre classes. Os parâmetros de oper1 e oper2 fazem com que ClasseA seja dependente de ClasseB e de ClasseC.

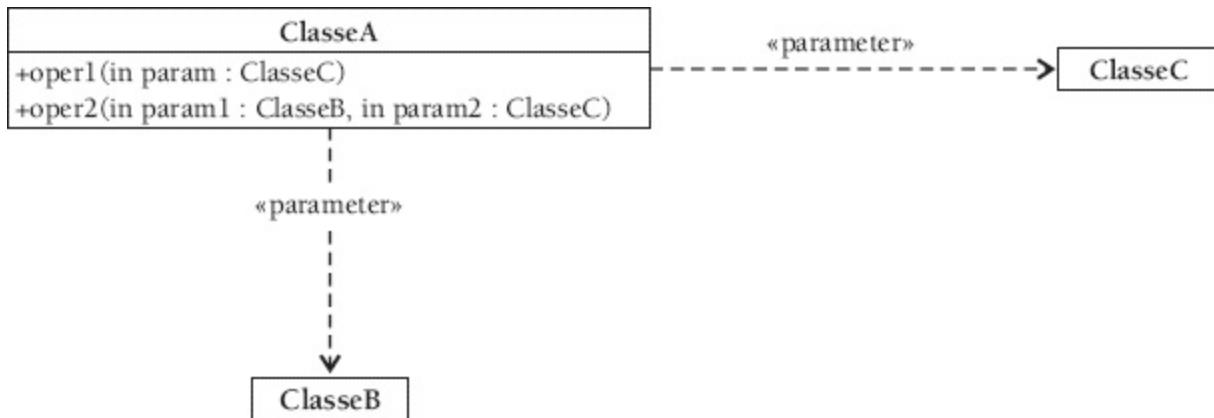


Figura 8-7: Exemplo de uso de dependências em um diagrama de classes.

Agora que definimos o conceito de dependência e vimos sua notação, é importante analisarmos como esse conceito influencia o refinamento de associações provenientes do modelo de classes de análise. Fazemos isso na próxima seção.

8.4.2 Transformação de associações em dependências

No modelo de classes de análise, o modelador especifica uma ou mais associações entre uma classe e as demais apresentadas no modelo. Na passagem do modelo de classes de análise para o de especificação, o modelador deve estudar cada associação para identificar se ela pode ser transformada em dependências dos tipos 2, 3 ou 4 descritos na [Seção 8.4.1](#). A razão para essa transformação é aumentar o encapsulamento e diminuir o encapsulamento das classes: a dependência por atributo (tipo 1), que corresponde ao modo de implementar associações, torna as classes envolvidas mais dependentes umas das outras. Quanto menos dependências estruturais houver no modelo de classes, maior será a qualidade do projeto (do ponto de vista do encapsulamento e do acoplamento das classes constituintes).

Certamente não há um critério predefinido que o modelador pode utilizar para decidir as associações que continuam e as que devem ser transformadas em dependências não estruturais. Essa decisão depende de vários fatores e varia de caso para caso. No entanto, de um modo geral, as associações que ligam classes de entidade permanecem como associações no modelo de especificação. Isso porque associações entre classes de entidade normalmente devem ser mantidas de forma persistente (ver o [Capítulo 12](#)). Além disso, associações entre controladores e classes de entidade são bastante suscetíveis a serem transformadas em dependências não estruturais (as de tipo 2, 3 ou 4). Entretanto, note que, se um controlador utilizar um objeto de entidade muito frequentemente, talvez seja mais adequado manter a associação por meio de uma dependência estrutural, por motivos de desempenho.

8.4.3 Navegabilidade de associações

As associações (assim como agregações e composições) podem ser classificadas em *bidirecionais* e *unidirecionais*. Uma associação bidirecional indica que há um conhecimento mútuo entre os objetos associados. Ou seja, se um diagrama de classes exibe uma associação entre duas classes C_1 e C_2 , então as duas assertivas a seguir são verdadeiras:

1. Cada objeto de C_1 conhece todos os objetos de C_2 aos quais está associado.
2. Cada objeto de C_2 conhece todos os objetos de C_1 aos quais está associado.

Graficamente, uma associação unidirecional é representada adicionando-se um sentido à associação. A classe para a qual o sentido aponta é aquela cujos objetos *não* possuem visibilidade dos objetos da outra classe. O diagrama da [Figura 8-8](#) ilustra o uso de uma associação unidirecional. Esse diagrama indica que objetos da classe Pedido possuem, cada um, uma referência para um objeto Cliente. Por outro lado, um objeto de Cliente não tem referências para os objetos associados em Pedido.

Associação unidirecional. Cada objeto da classe Pedido “conhece” o seu objeto correspondente na classe Cliente.

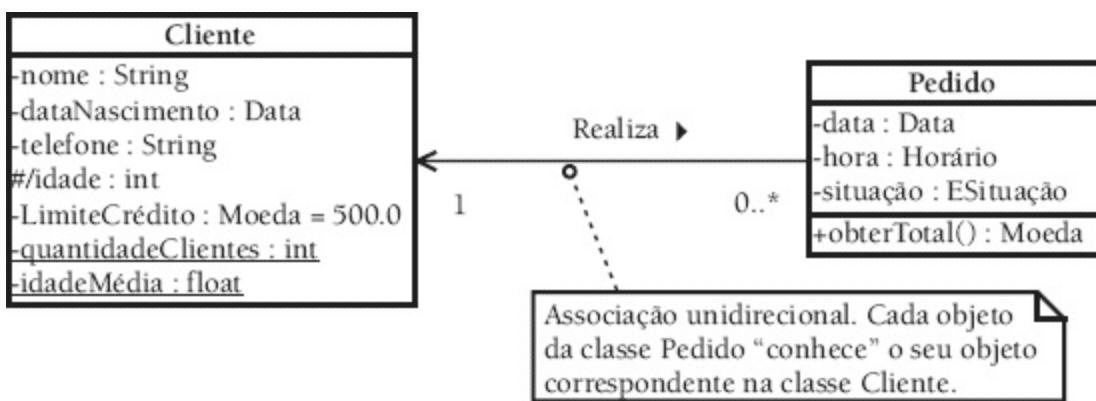


Figura 8-8: Exemplo de associação unidirecional.

Durante a construção do modelo de classes de análise, associações são normalmente consideradas “navegáveis” em ambos os sentidos, ou seja, as associações são bidirecionais. No modelo de classes de projeto, o modelador deve refinar a navegabilidade de todas as associações. Pode ser que algumas associações precisem permanecer bidirecionais. No entanto, se não houver essa necessidade, recomenda-se transformar a associação em unidirecional. Isso porque uma associação bidirecional é mais difícil de implementar e de manter do que uma associação unidirecional correspondente. Para entender isso, basta ver que o *acoplamento* (ver [Seção 7.5.2](#)) entre as classes envolvidas na associação é maior quando a navegabilidade é bidirecional. Portanto, um dos objetivos a alcançar durante o projeto é identificar quais navegabilidades são realmente necessárias.

A definição da navegabilidade se dá em função da troca de mensagens ocorridas nas interações entre objetos do sistema (ver [Capítulo 7](#)). Mais especificamente, devemos analisar os *fluxos das mensagens* entre objetos no diagrama de interações. Dados dois objetos associados, A e B, se há pelo menos uma mensagem de A para B em alguma interação, então a associação deve ser navegável de A para B. Da mesma forma, se existir pelo menos uma mensagem de B para A, então a associação deve ser navegável de B para A.

A definição do sentido da navegabilidade é feita em função das mensagens identificadas na modelagem de interação. Em particular, se um objeto do tipo A envia uma mensagem para outro do tipo B, então deve haver uma navegabilidade no sentido de A para B no diagrama de classes.

Mesmo em situações em que a navegabilidade de uma associação precise ser bidirecional, é possível implementar apenas um de seus sentidos. Por exemplo, na [Figura 8-9](#), considere que é necessário saber de que corridas certo cavalo participou e também os cavalos que participaram de certa corrida. Essas duas necessidades indicam uma associação bidirecional. Entretanto, se a coleção de cavalos não for tão grande, pode-se definir uma associação unidirecional no sentido de Corrida para Cavalo. Isso satisfaz a segunda necessidade. Para fazer o mesmo com a primeira, basta percorrer a coleção de corridas e, para cada uma, verificar se o cavalo em questão participou da mesma. Note que essa solução só se justifica se o tempo necessário para percorrer a coleção de corridas não tiver um grande impacto no desempenho. De um modo geral, se o desempenho for um fator crítico, é melhor implementar a associação nos dois sentidos.

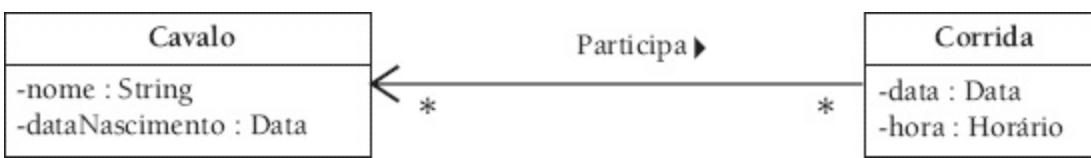


Figura 8-9: É possível substituir uma associação bidirecional por uma unidirecional.

8.4.4 Implementação de associações

O mapeamento de uma associação para código-fonte em alguma linguagem de programação (i.e., sua implementação) depende tanto de sua navegabilidade quanto da conectividade correspondente. Nesta seção, apresentamos as diversas situações possíveis e sua respectiva forma de mapeamento.

Para associações de conectividade *um para um*, a implementação é trivial. Suponha que existam duas classes ligadas por uma associação de conectividade um para um, C_a e C_b . Considere ainda que a navegabilidade definida é no sentido de C_a para C_b . Nesse caso, é definido um atributo do tipo C_b na classe C_a . Se a navegabilidade for bidirecional, o procedimento anterior pode ser aplicado para as duas classes. Portanto, em associações um-para-um, não há necessidade de um refinamento adicional.

Como um exemplo, considere a [Figura 8-10](#) e o [Quadro 8-3](#), que apresentam, respectivamente, um fragmento de diagrama de classes e um esboço de implementação (na sintaxe de linguagem Java) da classe Professor. Note que há um atributo nessa classe cujo tipo é GradeDisponibilidades. Dessa forma, cada objeto da classe Professor possui uma referência para um objeto da classe GradeDisponibilidades, o que é consistente com o modelo na [Figura 8-10](#).



Figura 8-10: Exemplo de associação de conectividade um para um.

Quadro 8-3: Implementação de associação de conectividade um para um

```

public class Professor {
    private GradeDisponibilidades grade;
    ...
}

```

Para o caso das associações de conectividade *um para muitos* ou *muitos para muitos*, o modelador pode decidir representar detalhes adicionais no modelo de classes de projeto, com o objetivo de esclarecer como tais associações devem ser implementadas. Uma razão para a representação desses detalhes adicionais pode ser o fato de que o sistema esteja sendo desenvolvido com o uso de uma ferramenta CASE (ver [Seção 2.6](#)) que gera códigos automaticamente a partir do diagrama de projeto. Nessa situação, o objetivo do modelador no detalhamento pode ser o de especificar a forma de implementação que a ferramenta deve utilizar. O detalhamento de associações de conectividade *um-para-muitos* ou *muitos-para-muitos* se baseia em classes que representam *coleções* de elementos. Vamos, então, descrever o conceito de *classe parametrizada*, que permite a representação de coleções em um diagrama de classes de especificação.

Uma coleção pode ser representada em um diagrama de classes por uma *classe parametrizada*. Uma classe parametrizada é aquela utilizada para definir outras classes. Ela possui operações ou

atributos cuja definição é feita em função de um ou mais parâmetros. Uma aplicação do conceito de classe parametrizada é na definição de classes que representam coleções. Independentemente do tipo de elementos que uma coleção possui, ela precisa possuir operações para adicionar um novo elemento, remover um elemento, encontrar um elemento etc. Sendo assim, uma coleção pode ser definida a partir de uma classe parametrizada, em que o parâmetro é o tipo do elemento da coleção.

Há duas notações possíveis na UML para representar uma classe parametrizada em um diagrama de classes. A [Figura 8-11](#) é um exemplo das duas representações possíveis para a classe parametrizada denominada Coleção.



Figura 8-11: Notações possíveis na UML para uma classe parametrizada.

Uma questão interessante diz respeito a qual é a correspondência do conceito de classe parametrizada em linguagens de programação. Na linguagem Java, por exemplo, classes parametrizadas são representadas pelo mecanismo de *tipos genéricos* (termo original: *Java Generics*). A linguagem C++ também possui uma forma de implementar classes parametrizadas, com o conceito de *templates*. A linguagem C# é outra que fornece o mecanismo de classes parametrizadas. Como um exemplo, considere o fragmento de classe apresentado no [Quadro 8-4](#), no qual utilizamos a sintaxe da linguagem Java. Esse fragmento apresenta um atributo definido como um tipo genérico, préRequisitos. Nesse caso particular, esse atributo representa uma coleção de objetos cujos elementos são da classe Disciplina.

Quadro 8-4: Classe parametrizada na sintaxe da linguagem Java

```
public class Disciplina {  
    private Set<Disciplina> préRequisitos;  
    ...  
}
```

Conforme declaramos há pouco, o conceito de classe parametrizada pode ser utilizado para detalhar de que forma é possível refinar uma associação de conectividade um para muitos ou muitos para muitos. Analisamos esse aspecto a seguir.

Em uma associação de conectividade “um para muitos”, a forma de refiná-la (e, consequentemente, de implementá-la) depende da navegabilidade definida. Há dois casos a considerar. No primeiro caso, a navegabilidade aponta para o lado “muitos” da associação. No segundo caso, ela aponta para o lado “um” da associação. Vamos agora analisar cada um desses casos.

Para refinar uma associação de conectividade um para muitos em que a navegabilidade é do lado “um” para o lado “muitos”, a ideia básica é definir uma classe parametrizada cujo parâmetro é a classe correspondente ao lado “muitos” da associação. Como exemplo, a [Figura 8-12](#) ilustra três versões de um mesmo fragmento de diagrama de classes para representar o fato de que um objeto da classe ListaEspera possui associação com diversos (muitos) objetos da classes Aluno. A parte (a) ilustra a versão menos detalhada (apenas a navegabilidade é especificada). As partes (b) e (c) da [Figura 8-](#)

12 ilustram duas representações mais detalhadas. Essas representações são equivalentes e utilizam o conceito de classe parametrizada. Conforme mencionamos na [Seção 7.1.5](#), considera-se que a classe parametrizada correspondente à coleção possui operações que permitem adicionar, remover e obter acesso a seus elementos. As linguagens de programação orientadas a objetos fornecem diversas classes cujos objetos podem ser utilizados como contêineres. Note também que a utilização da classe parametrizada resultou na transformação de uma associação um-para-muitos em uma associação um-para-um.

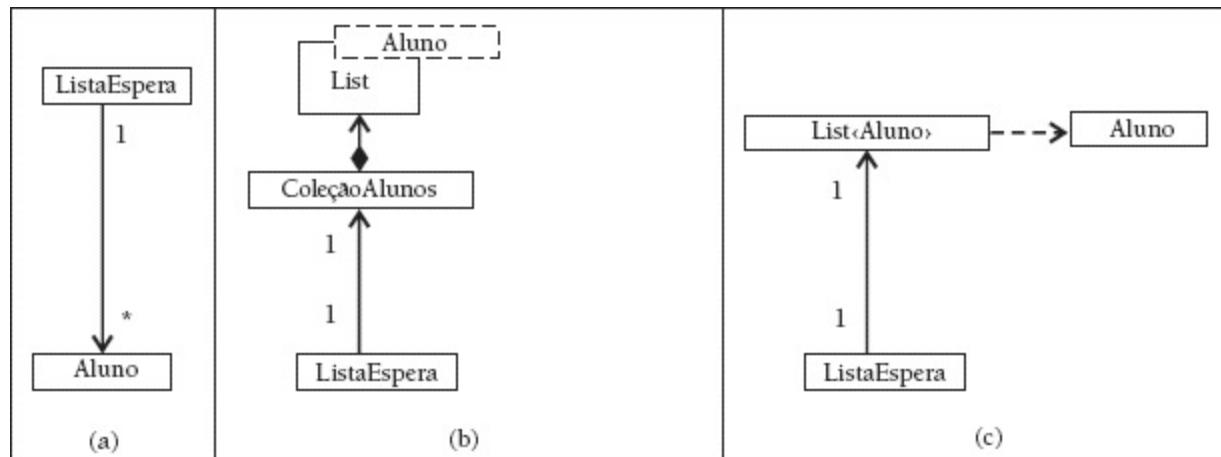


Figura 8-12: Formas alternativas para representar uma associação um para muitos.

Para entender como o diagrama da [Figura 8-12](#) pode ser implementado, considere o [Quadro 8-5](#). Esse quadro apresenta um trecho da implementação da classe `ListaEspera` em linguagem Java. Note a presença do atributo denominado `alunos`, uma lista parametrizada de objetos da classe `Aluno` (p. ex., um contêiner de alunos). Nesse exemplo, utilizamos a interface `List` presente na linguagem API Collections da linguagem Java. Também apresentamos os protótipos dos métodos `adicionarAluno` e `removerAluno`, que servem para manipular a coleção representada pelo atributo `alunos` (ver [Seção 8.3.6](#)).

Quadro 8-5: Implementação de associação um para muitos e de naveabilidade para o lado “muitos”

```
public class ListaEspera {
    private List<Aluno> alunos;
    ...
    public boolean adicionarAluno(Aluno a){
        ...
    }
    public boolean removerAluno(Aluno a) {
        ...
    }
}
```

O segundo caso possível de refinamento de uma associação um para muitos é aquele em que a mesma tem naveabilidade apontando para o lado “um”. Nesse caso, a utilização de uma coleção (como no primeiro caso) não é necessária. Quando a naveabilidade somente aponta para o lado “um”, tudo o que deve ser feito é criar um atributo de referência na classe correspondente ao lado “muitos”. O tipo desse atributo corresponde à classe que está no lado “um” da associação. Por exemplo, o [Quadro 8-6](#) apresenta um exemplo de trecho de código resultante da aplicação dessa regra. Nesse quadro, temos a classe `Turma`, que possui uma associação com a classe `Disciplina`. A

navegabilidade dessa associação é de Turma para Disciplina (ou seja, apontando para o lado “um”). Portanto, como mostra essa figura, criamos um atributo na classe Turma para referenciar o objeto Disciplina ao qual a primeira está associada.

Quadro 8-6: Implementação de associação muitos para um

```
public class Turma {  
    private Disciplina disciplina;  
    ...  
}
```

A utilização do conceito de classe parametrizada no detalhamento de uma associação cuja conectividade é *muitos para muitos* é bastante semelhante ao caso das associações um para muitos. Considere o exemplo da [Figura 8-13a](#), que ilustra uma associação entre as classes Disciplina e GradeDisponibilidades. Essa figura também informa que a associação é navegável no sentido de GradeDisponibilidades para Disciplina. A [Figura 8-13b](#) apresenta o detalhamento do diagrama apresentado na [Figura 8-13a](#). Note que novamente uma classe parametrizada é utilizada no refinamento da associação.

Além do refinamento das associações comuns que descrevemos anteriormente, há também o caso das classes associativas (ver [Seção 5.2.2.4](#)). No refinamento de classes associativas, a solução mais comum é criar uma classe para substituir a associativa. Uma vez feito isso, o problema se transforma no refinamento de associações um-para-muitos ou muitos-para-muitos. Como exemplo, a [Figura 8-14](#) ilustra dois diagramas de classes. O diagrama da direita corresponde a um refinamento de projeto do diagrama da esquerda.

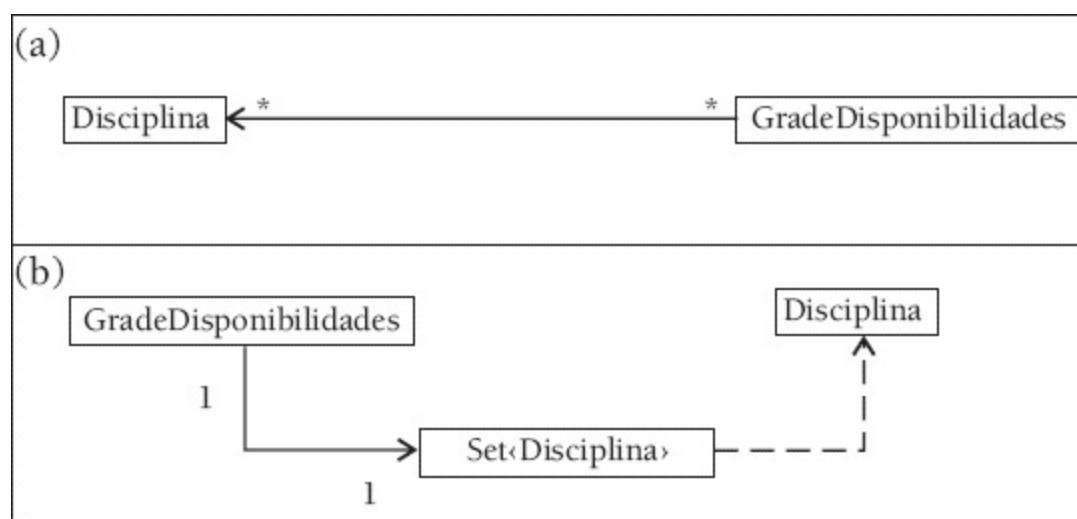


Figura 8-13: Associação muitos para muitos entre Disciplina e GradeDisponibilidades.

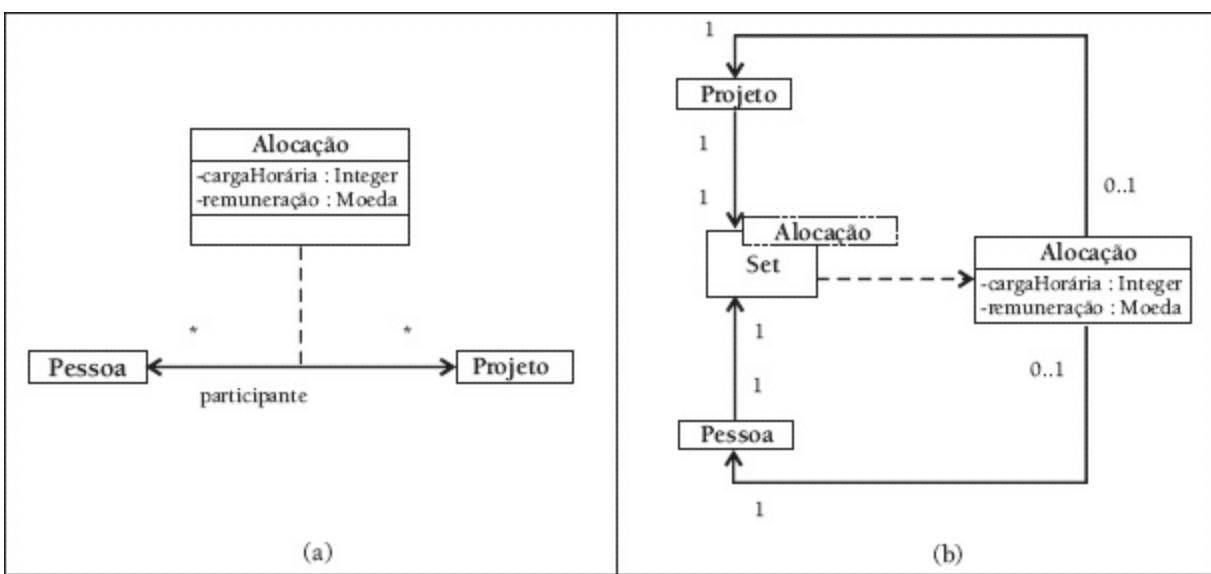


Figura 8-14: Refinamento de projeto de uma classe associativa.

8.5 Herança

Na [Seção 5.2.3](#), estudamos o relacionamento de herança no contexto do modelo de classes de análise. Na etapa de projeto, há diversos aspectos relevantes quanto ao refinamento desse relacionamento entre classes. Descrevemos esses aspectos nesta seção.

8.5.1 Tipos de herança

Há diversas categorizações que podem ser feitas a respeito de conceito de herança. Uma delas é com relação à quantidade de superclasses que certa classe pode ter. De fato, uma classe pode ter mais de uma superclasse. Essa situação corresponde ao conceito de *herança múltipla*. Esse conceito é ilustrado na [Figura 8-15](#). Este diagrama representa um carro anfíbio, que possui características tanto de um carro quanto de um barco. Se uma classe tem apenas uma superclasse, então estamos diante da *herança simples*.

O uso de herança múltipla deve ser evitado. Um dos motivos para isso é que esse tipo de herança é difícil de entender. Além disso, algumas linguagens de programação não possuem construções para dar suporte à implementação da herança múltipla. Java, C# e Smalltalk, por exemplo, são linguagens orientadas a objetos que não dão suporte direto ao conceito de herança múltipla. Por fim, a maioria das linguagens de programação mais modernas possui uma alternativa para o uso de herança múltipla, as *interfaces*, que descrevemos na [Seção 8.5.4](#).

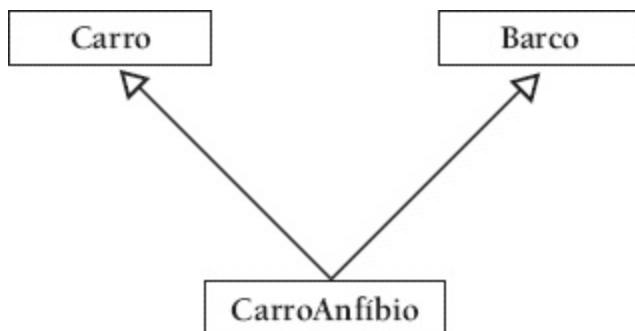


Figura 8-15: Exemplo de herança múltipla.

Outro tipo de categorização do conceito de herança é com relação à forma de reutilização envolvida. Há duas formas de reutilização por herança em LPOO modernas (Java, C# etc.): *herança de implementação* e *de interface*. Para estabelecer a diferença entre esses dois tipos, devemos primeiramente definir o conceito de *serviço*. Um serviço é composto de sua *especificação* e de seu *método*. A *especificação* define o *que* o serviço realiza, além de documentar as informações necessárias para que isso aconteça e seu resultado. Já o *método* corresponde ao modo de realizar um serviço. Dada uma especificação de um serviço, podem existir diversos métodos para realizá-lo. Na herança de implementação, uma subclasse herda métodos de um “ancestral”. Já na herança de interface, ela herda as especificações definidas na interface de um “ancestral” e se compromete a implementar essa interface. A herança de implementação é bastante intuitiva; é fácil entender que uma subclasse herda os métodos de qualquer operação definida em seu ancestral. Entretanto, o conceito de herança de interface não é tão intuitivo. Para esclarecer esse tipo de herança, nas próximas seções introduzimos os conceitos de *classe abstrata* e de *interface*.

8.5.2 Classes abstratas

É comum a existência de uma classe se justificar pelo fato de ser possível gerar instâncias da mesma. Classes que geram instâncias são chamadas de *concretas*. No entanto, podem existir classes que não geram instâncias (objetos) diretamente. Essas classes são normalmente utilizadas para organizar e simplificar uma hierarquia de herança. Portanto, classes abstratas só existem em hierarquias. Propriedades comuns a diversas classes podem ser organizadas e definidas em uma única classe abstrata da qual as primeiras herdam.

Na notação da UML, uma classe abstrata é representada com o seu nome em *itálico*. A [Figura 8-16](#) apresenta um exemplo de classe abstrata que serve como superclasse para duas outras classes. Uma notação alternativa é utilizar a etiqueta `{abstract}` (ver [Seção 3.3](#)) dentro do compartimento do nome da classe.

Esta é uma classe abstrata. Note o itálico no seu nome.

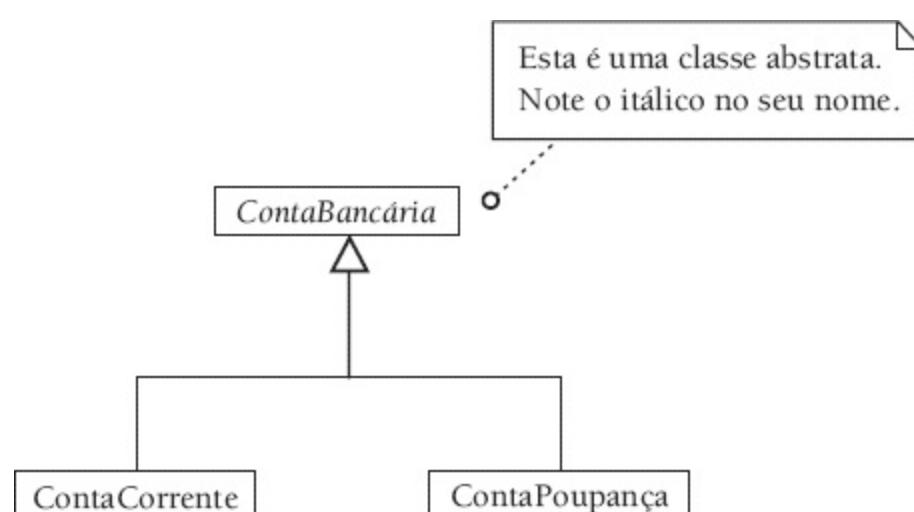


Figura 8-16: Exemplo de classe abstrata.

Subclasses de uma classe abstrata também podem ser abstratas, mas a hierarquia deve terminar em uma ou mais classes concretas. Ou seja, não faz sentido existir uma hierarquia de herança com uma classe abstrata na base dessa hierarquia.

Além de não poder ser diretamente instanciada, outra característica associada ao conceito de

classe abstrata é o fato de ela possuir ao menos uma *operação abstrata*. Na terminologia da [Seção 8.5.1](#), uma operação abstrata corresponde à *especificação* de um serviço que a classe deve fornecer (sem *método*). Uma classe qualquer pode possuir tanto operações abstratas quanto concretas (ou seja, operações que possuem implementação). Entretanto, uma classe que possui pelo menos uma operação abstrata é, por definição, abstrata. Assim como acontece com qualquer operação pública, uma operação abstrata definida em uma classe também é herdada por suas subclasses. Quando uma subclasse herda uma operação abstrata e não fornece uma implementação para a mesma, esta classe também será abstrata (pois, por herança, passará a possuir uma operação abstrata). Por outro lado, se esta classe fornecer uma implementação para quaisquer operações abstratas herdadas, e ela própria não definir operações abstratas, esta classe será concreta.

Graficamente, uma operação abstrata é representada em *itálico* no diagrama de classes, segundo a notação definida pela UML. O uso da etiqueta `{abstract}` também é uma alternativa. Como exemplo, considere a [Figura 8-17](#). Nesta figura, a classe *FiguraGeométrica* é abstrata, pois possui uma operação abstrata, *desenhar* (note que a assinatura dessa operação está declarada em itálico). As classes *Círculo* e *Quadrado* são concretas, pois fornecem implementação para a operação abstrata herdada.

Operação abstrata. Note o itálico em sua assinatura. Subclasses devem implementar o comportamento desta operação para serem concretas.

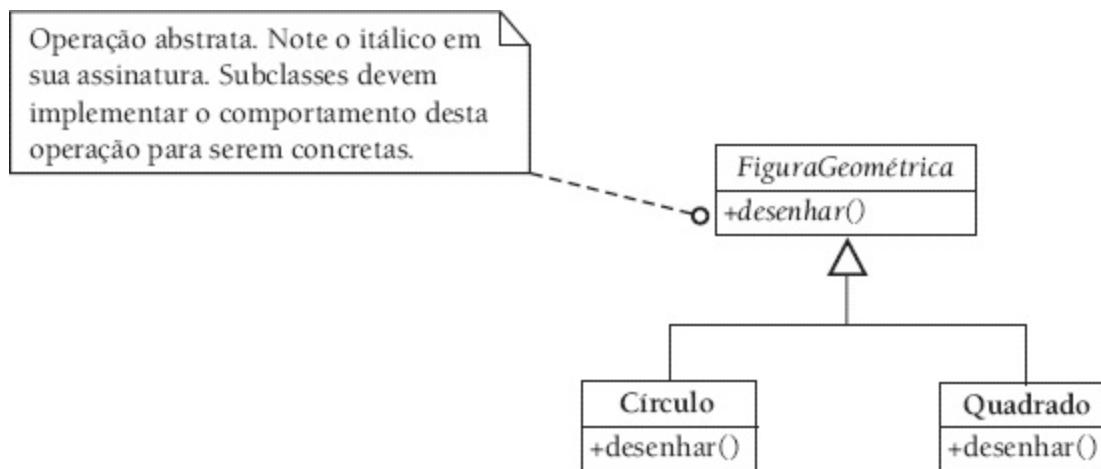


Figura 8-17: Herança de operação abstrata.

8.5.3 Operações polimórficas

Uma subclasse herda todas as propriedades de sua superclasse que tenham visibilidade pública ou protegida. Isso quer dizer que é possível a situação em que um objeto da subclasse recebe uma mensagem para que uma operação pública ou protegida definida em sua superclasse seja executada. Entretanto, pode ser que o comportamento de alguma operação herdada tenha que ser diferente para a subclasse. Nesse caso, a subclasse deve redefinir o comportamento da operação. Note que a assinatura da operação é reutilizada pela subclasse; tudo o que ela faz é implementar o novo comportamento desejado para a operação. O resultado disso é que a subclasse e a superclasse agora compartilham a mesma assinatura de certa operação, mas cada uma delas possui um *método* (forma de implementação) particular para essa operação.

Por definição, *operações polimórficas* são operações de mesma assinatura definidas em diversos níveis de uma hierarquia de herança e que possuem métodos diferentes. Essas operações devem ter a assinatura repetida na(s) subclasse(s) para enfatizar que elas estão sendo redefinidas (somente a

assinatura é herdada, mas não a implementação).

Como exemplo, considere as classes Funcionário e Vendedor, esta última subclasse da primeira. Em Funcionário, existe um método obterPagamento que retorna o valor do pagamento do funcionário. Suponha que o método para obter o valor do pagamento de um vendedor *não* seja o mesmo para obter o de um funcionário. Nesse caso, não faz sentido Vendedor herdar a implementação dessa operação. Em vez disso, Vendedor deve *redefinir* a operação obterPagamento. Diz-se, então, que essa operação é polimórfica. A Figura 8-18 ilustra graficamente esse exemplo. A assinatura da operação obterPagamento é repetida em Vendedor. (As notas explicativas [ver Seção 3.2] apresentam as expressões de cálculo do pagamento utilizadas em uma classe e na outra.) Note que, como as operações têm a mesma assinatura, a mensagem utilizada para obter o valor do pagamento de um funcionário ou de um vendedor é a mesma.

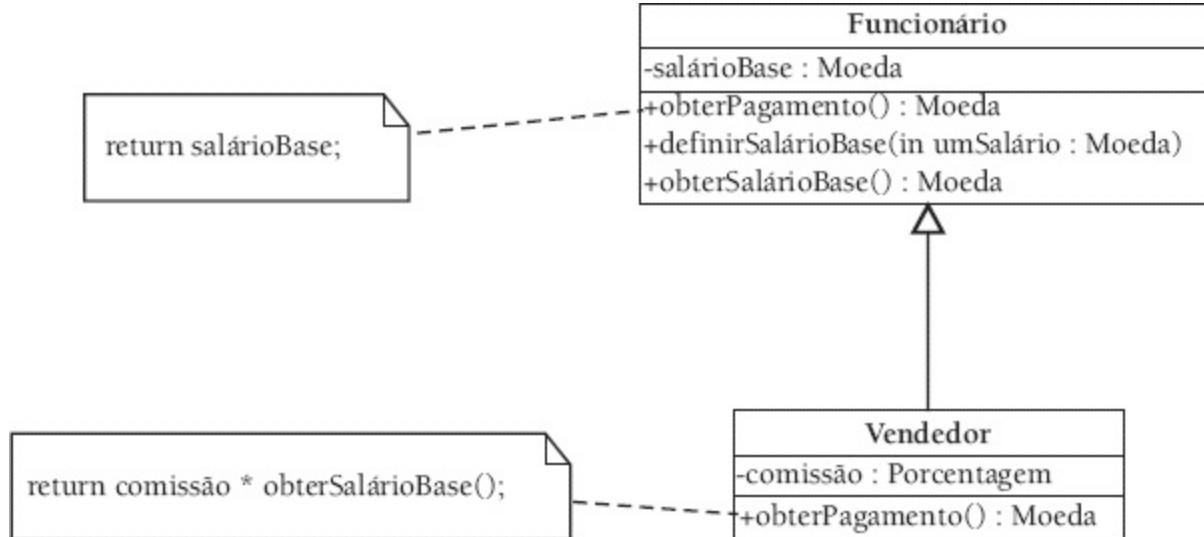


Figura 8-18: Definição de operações polimórficas.

Operações polimórficas também podem existir em classes abstratas. A Figura 8-19 ilustra um exemplo dessa situação. Nesse exemplo, há uma classe abstrata denominada ContaBancária. Essa classe possui duas subclasses, ContaCorrente e ContaPoupança. Note também que a operação aplicarJuros em ContaBancária também é abstrata (pois sua assinatura está declarada em itálico). Isso significa que as subclasses devem fornecer uma implementação a essa operação para que também não se tornem abstratas. Portanto, tanto ContaCorrente quanto ContaPoupança implementam a operação herdada.

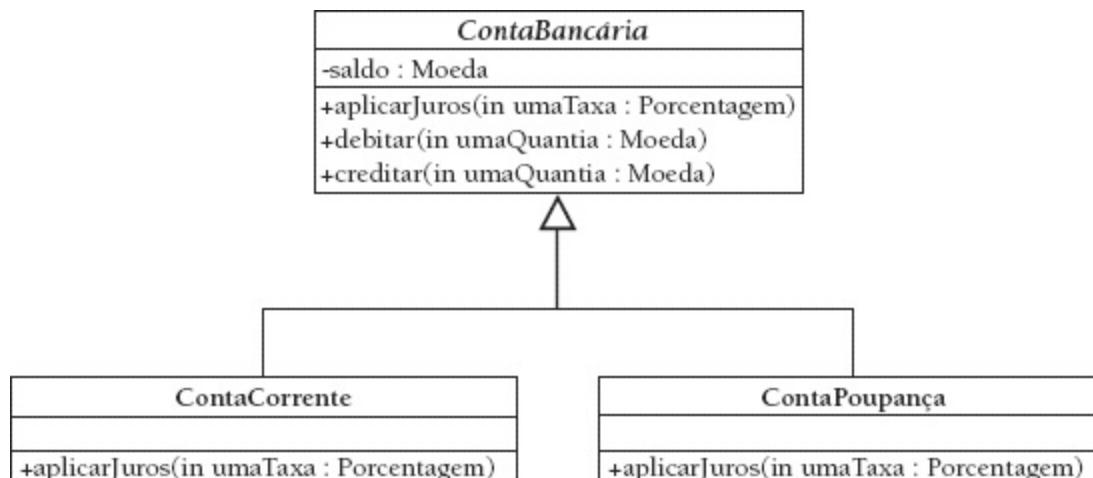


Figura 8-19: Operações polimórficas também podem existir em classes abstratas.

Operações polimórficas são importantes, porque ajudam na implementação do *princípio do polimorfismo*, no qual objetos que pertencem a classes diferentes respondem à mesma mensagem. (O princípio do polimorfismo é apresentado na [Seção 1.2.3.2.](#)) A utilização de operações polimórficas tem o objetivo de garantir que as subclasses compartilhem uma mesma operação, mas com métodos diferentes. Em outras palavras, se duas ou mais subclasses de uma mesma classe implementam a mesma operação polimórfica, o remetente da mensagem não precisa saber qual a verdadeira classe de cada objeto receptor, pois eles aceitam a mesma mensagem. A diferença é que o método que implementa a operação a ser executada é diferente em cada subclasse. Como exemplo disso, considere o trecho de código apresentado no [Quadro 8-7](#). Nesse exemplo há uma coleção cuja definição indica que ela armazena objetos da classe ContaBancária. A princípio, esse exemplo parece incoerente com a definição de classe abstrata: como pode existir uma coleção cujos elementos são objetos de uma classe na qual é impossível existir instâncias? A resposta para a aparente incoerência é que, embora não possa haver instâncias diretas da classe ContaBancária, pode haver instâncias *indiretas* da mesma. Note que instâncias das classes ContaCorrente e ContaPoupanca são indiretamente instâncias da classe ContaBancária. Portanto, a coleção denominada contasBancárias pode conter objetos das subclasses de ContaBancária. Agora note a região de código destacada (em negrito) no [Quadro 8-7](#). Essa região corresponde a uma iteração que percorre todos os objetos na coleção contasBancárias e, para cada um deles, invoca a execução da operação aplicarJuros. Como essa coleção contém objetos de ambas as subclasses de ContaBancária, os métodos invocados serão diferentes, em função da verdadeira classe do objeto receptor da mensagem. De qualquer modo, o fato importante nesse exemplo é que há uma região de código (a que está destacada no [Quadro 8-7](#)) que envia a mensagem para objetos de classes diferentes, *sem saber qual é a verdadeira classe dos objetos receptores* (ContaCorrente ou ContaPoupanca). Essa é a essência do princípio do polimorfismo.

Quadro 8-7:Operações polimórficas ajudam na implementação do *polimorfismo*

```

ContaCorrente cc; ContaPoupanca cp;
...
List<ContaBancária>contasBancárias;
... contasBancárias.add(cc); contasBancárias.add(cp);
...
for(ContaBancária conta: contasBancárias) {
    conta.aplicarJuros(0.05);
}
...

```

8.5.4 Interfaces

De acordo com a definição de um dicionário, uma interface é um dispositivo ou um sistema que entidades não relacionadas utilizam para se comunicar. No contexto do desenvolvimento de sistemas orientados a objetos, quando estamos construindo uma comunidade de entidades (objetos) que interagem entre si, uma interface representa a “cara” que uma dessas entidades mostra à outra, ou seja, que serviços ela fornece. Uma entidade pode, é claro, ter muitas interfaces, mostrando as “caras diferentes” a diferentes membros de comunidade de entidades.

Considere o exemplo a seguir para entender o conceito de interface. Suponha que exista uma classe Bicicleta, que pertence a uma hierarquia de classes (veículos). A classe Bicicleta define o que uma bicicleta pode fazer em termos do comportamento de um veículo. No entanto, um objeto bicicleta

pode interagir com o mundo de outras formas. Por exemplo, uma bicicleta poderia ser um dos produtos manipulados por um sistema de vendas. Esse sistema provavelmente não necessita do comportamento relativo a um veículo da classe Bicicleta, apenas que objetos dessa classe se comportem como produtos que possam ser vendidos e que forneçam certas informações (p. ex., preço de venda, número de registro etc.). Ou seja, para ser manipulado pelo sistema de vendas, um objeto da classe bicicleta deve estar em concordância com o *protocolo ou contrato de comportamento* predefinido. Este protocolo compreende um conjunto não vazio de *assinaturas de operações*. Cada assinatura apresenta o nome da operação, juntamente com a definição de seus eventuais parâmetros e tipo de retorno. O protocolo de interação não contém a implementação das operações nele definidas. Em uma linguagem de programação orientada a objetos, esses protocolos são denominados *interfaces*.

Quanto ao exemplo do sistema de vendas, ele pode interagir com os objetos a serem vendidos (produtos) por intermédio de uma interface; não há necessidade desse sistema conhecer as verdadeiras classes dos objetos com os quais interage. Dessa forma, objetos da classe Bicicleta ou da classe Geladeira podem ser tratados de forma indistinta pelo sistema de vendas. De uma forma geral, qualquer classe que implemente (forneca implementação) para as operações definidas na interface que o sistema de vendas espera pode representar o papel de produto. Além disso, se por um lado uma interface é usada para definir um *contrato de comportamento*, por outro o mesmo contrato de comportamento pode ser implementado por diferentes classes. A consequência disso é que não há a necessidade de definirmos relacionamentos diretos entre classes que, em situações normais, não estariam relacionadas. Em vez disso, podemos definir uma interface para que essas classes se comuniquem. É possível a partir disso estabelecer os seguintes objetivos do conceito de interface:

1. Capturar semelhanças entre classes não relacionadas sem forçar relacionamentos entre elas.
2. Declarar operações que uma ou mais classes devem implementar.
3. Revelar as operações de um objeto, sem revelar a sua classe.
4. Facilitar o desacoplamento entre elementos de um sistema.

Uma interface corresponde a um conjunto de especificações de *serviços* (ver [Seção 8.5.1](#)) fornecidos por um *classificador*. Um classificador é conceito da UML usado para denotar uma *classe*, um *subsistema* ou um *componente*. (Estes dois últimos elementos são descritos nas [Seções 11.1](#) e [11.2.2](#), respectivamente.) Um classificador possui comportamento, ou seja, implementa um conjunto de operações. Por meio dessas operações, um classificador pode fornecer serviços para outros classificadores. Ele também pode utilizar serviços destes últimos. Um classificador tem a opção de implementar várias interfaces, assim como uma interface pode ser implementada por vários classificadores. Em uma interface, não há métodos associados às especificações. O classificar é que fornece métodos para as especificações das classes que ele implementa.

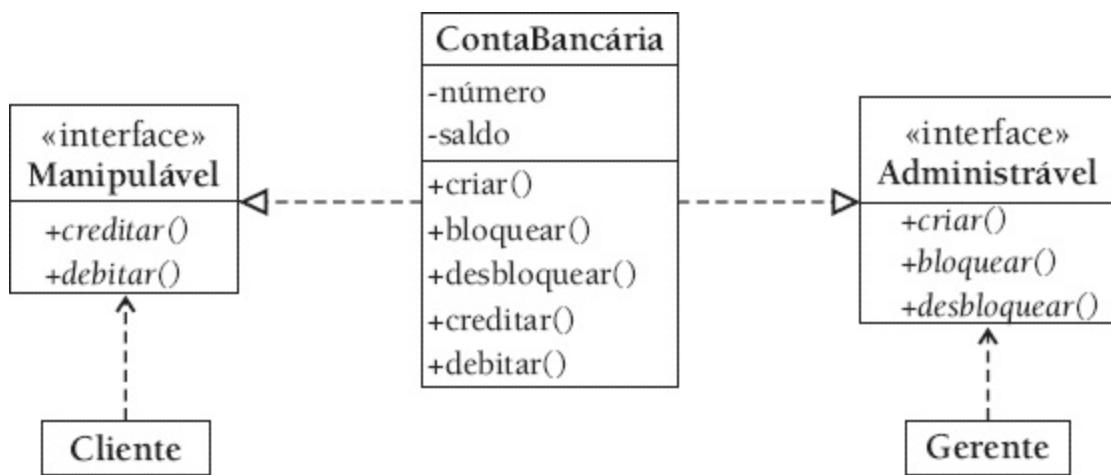


Figura 8-20: A classe ContaBancária implementa duas interfaces: Manipulável e Administrável.

A UML define duas notações equivalentes para representar graficamente uma interface. (Os exemplos a seguir são fornecidos em relação a classes, mas a notação vale para qualquer classificador.) A primeira notação para uma interface é a mesma definida para classes em que são exibidas as operações que a interface define. Entretanto, o compartimento dos atributos fica sempre vazio, pois uma interface não possui atributos. Além disso, no compartimento do nome da interface deve aparecer o estereótipo `<<interface>>`. Como exemplo, a Figura 8-20 utiliza a primeira notação e indica que **ContaBancária** realiza as interfaces **Manipulável** e **Administrável**. A primeira interface (**Manipulável**) especifica as operações **creditar()** e **debitar()**. A segunda interface (**Administrável**) especifica as operações **criar()**, **bloquear()** e **desbloquear()**. Para as classes **Cliente** e **Gerente**, que utilizam as interfaces que **ContaBancária** realiza, **tudo se passa como se estivessem utilizando a própria classe**. A diferença é que essas classes somente visualizam as operações que fazem sentido para elas. Além disso, qualquer outra classe que implemente as interfaces **Manipulável** e **Administrável** pode substituir a classe **ContaBancária** sem modificações nas demais classes. A segunda notação possível para uma interface na UML é por meio de um segmento de reta com um pequeno círculo em um dos extremos e ligado à classe que a realiza (implementa) no outro extremo. O nome da interface é posicionado próximo ao extremo do segmento que contém o pequeno círculo. Nesta notação mais simplificada, as operações da interface não são ilustradas. As classes clientes estão conectadas à interface por um relacionamento de dependência. A Figura 8-21 exibe essa segunda forma de representação de interfaces. Note que, em ambas as formas de representação de interfaces, a linha de dependência parte de um classificador e termina no símbolo de uma das interfaces de outro classificador.

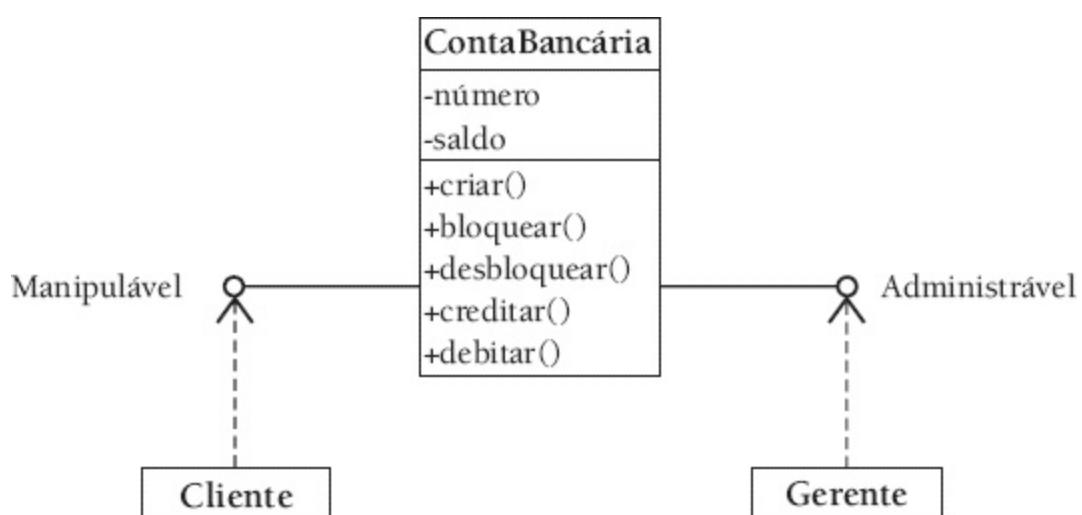


Figura 8-21: Exemplo da forma simplificada para representar interfaces.

Conforme mencionado há pouco, uma classe pode realizar (implementar) várias interfaces. Nesse caso, ela deve fornecer implementações (métodos) para as operações cujas assinaturas (especificações) são declaradas nas interfaces. Por exemplo, a classe ContaBancária fornece implementações para as operações declaradas nas interfaces Manipulável e Administrável. Também é possível que a mesma interface seja realizada por várias classes. Nesse caso, essas classes devem implementar as operações da interface que realizam. Por exemplo, se uma interface I é realizada por três classificadores A, B e C e declara as operações op1(), op2(), op3() e op4(), então essas operações devem estar implementadas em alguns dos classificadores.

Algumas linguagens de programação possuem em sua sintaxe suporte direto ao conceito de interface, mas outras não. As linguagens Java e C# apresentam o conceito de interface em suas sintaxes. A linguagem C++, por outro lado, não possui explicitamente o conceito de interface. Mesmo assim, nessa linguagem, interfaces podem ser definidas indiretamente por classes abstratas em que todas as operações são abstratas e não há atributos.

Juntamente com o conceito de classes abstratas, interfaces permitem alcançar o *acoplamento abstrato*. Descrevemos esse conceito na próxima Seção.

8.5.5 Acoplamentos concreto e abstrato

Na [Seção 7.5.2](#), descrevemos o conceito de *acoplamento*. Vimos que esse conceito corresponde a uma medida da *dependência* existente entre classes de um sistema. Vimos também que devemos manter o acoplamento em um nível mínimo possível, com o objetivo de diminuir as dependências na estrutura de classes correspondente. Note a utilização do termo “mínimo possível” na frase anterior. O acoplamento é necessário quando um objeto precisa solicitar serviços (interagir) de outro com o envio de mensagens. Nesse caso, o objeto remetente precisa ter uma *referência* para o receptor.

Uma forma de um objeto (remetente) ter uma referência para outro (receptor) é fazer com que o remetente *tenha conhecimento direto da classe do receptor*. Esse tipo de dependência corresponde ao chamado *acoplamento concreto*. Esse nome surge do fato de que há duas *classes concretas* (ver [Seção 8.5.2](#)) envolvidas. Entretanto, há outra forma de dependência que permite que um objeto remetente envie uma mensagem para um receptor *sem ter conhecimento da verdadeira classe desse último*; essa forma de dependência corresponde ao *acoplamento abstrato*. Mas como o acoplamento abstrato pode ser possível? Para entendermos isso, vamos analisar novamente o conceito de *interface* que descrevemos na [Seção 8.5.4](#). Acompanhe essa análise a seguir, juntamente com a [Figura 8-22](#), que ilustra esquematicamente os acoplamentos concreto e abstrato.

Uma interface pode ser interpretada como um nível de *indireção*, pelo qual os classificadores podem utilizar os serviços uns dos outros. Para entender isso, considere dois classificadores, Ca e Cb1. Suponha que Ca utiliza serviços de Cb1 por meio de uma interface ICb. Se um novo classificador Cb2 que implementa a mesma interface ICb foi desenvolvido, ele pode substituir Cb1 sem que Ca precise sofrer modificações quando essa substituição ocorrer. Para entender isso, note que Ca utiliza os serviços implementados em Cb1 por intermédio da interface ICb. Ou seja, Ca não tem uma referência direta para uma instância de Cb. Com efeito, o uso de interfaces entre classificadores torna o sistema mais flexível a mudanças. Os classificadores fornecedores de algum serviço podem ser substituídos sem causar modificações nos classificadores clientes, desde que a interação ocorra por uma interface. Essa é a essência do acoplamento abstrato. Interfaces permitem encapsular comportamento, ocultando qual a classe de um objeto que está realizando uma tarefa

específica. Isso é possível porque a classe que implementa uma interface é obrigada a seguir o protocolo definido nesta última.

Conforme também podemos inferir da [Figura 8-22](#), o acoplamento abstrato pode ser alcançado não só com o uso de interfaces, mas também com *classes abstratas* (ver [Seção 8.5.2](#)). A razão disso é que o conceito de interface guarda diversas semelhanças com o conceito de classe abstrata, conforme descrevemos a seguir. Em primeiro lugar, da mesma forma que uma classe abstrata, uma interface não gera instâncias. Além disso, tanto interfaces quantos classes abstratas possuem um conjunto de operações abstratas. Note, entretanto, que há diferenças entre esses dois conceitos. Ao contrário de uma classe abstrata, uma interface não pode conter qualquer estrutura interna (ou seja, não pode ter atributos nem associações). Além disso, todas as operações de uma interface só possuem especificações; a implementação (método) de cada operação não é definida na interface.

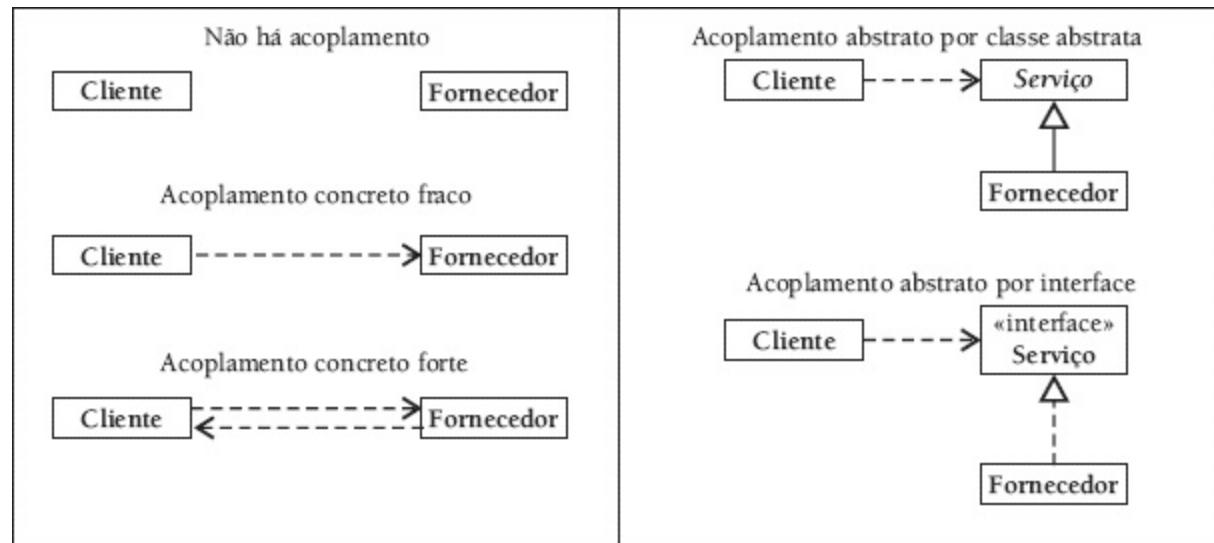


Figura 8-22: Formas de acoplamento entre classificadores.

Uma interface (ou uma classe abstrata) descreve uma parte do comportamento externamente visível de um conjunto de objetos, possivelmente de classes diferentes. O acoplamento abstrato entre elementos (classificadores) que podemos alcançar com o uso de uma interface ou de uma classe abstrata é muito importante na construção de sistemas orientados a objetos de qualidade. Quando bem utilizado, esse tipo de acoplamento aumenta a flexibilidade e a capacidade de reutilização do sistema, assim como a qualidade do código-fonte. Isso porque o acoplamento abstrato nos permite isolar um conjunto de serviços de certa aplicação que são instáveis, no sentido de terem alto potencial e de precisarem de modificações em sua implementação no futuro. Se detectamos algum conjunto de serviços com essa característica de instabilidade, podemos isolá-lo por trás de uma interface (ou classe abstrata). Os clientes desse conjunto de serviços têm acesso a eles por intermédio da interface, sem precisarem conhecer diretamente as classes que os implementam. Se a implementação de algum serviço precisar ser alterada, não precisaremos modificar os clientes do serviço. Nos padrões de projeto que descrevemos na [Seção 8.6](#), o acoplamento abstrato é um princípio fundamental.

8.5.6 Reúso por delegação

Na [Seção 8.5.1](#), descrevemos duas formas de reúso por herança: *reúso de interface* e *reúso de comportamento*. Este último se baseia na noção de que subclasses herdam *comportamento* de sua superclasse. Por exemplo, na [Figura 8-23](#), quando um objeto da classe ContaCorrente recebe uma mensagem para executar a operação debitar, ele não tem como atender a essa mensagem só com os recursos de sua classe. Ele utiliza então a operação herdada da superclasse. A herança de comportamento é um mecanismo fácil de implementar; tudo que o desenvolvedor deve fazer na definição de uma subclass é adicionar novas propriedades às existentes na superclasse, ou redefinir as propriedades herdadas para se adequarem ao seu comportamento. No entanto, a herança por comportamento tem a desvantagem potencial de expor as subclasses aos detalhes de sua superclasse, violando assim o *princípio do encapsulamento* (ver [Seção 1.2.3.1](#)).

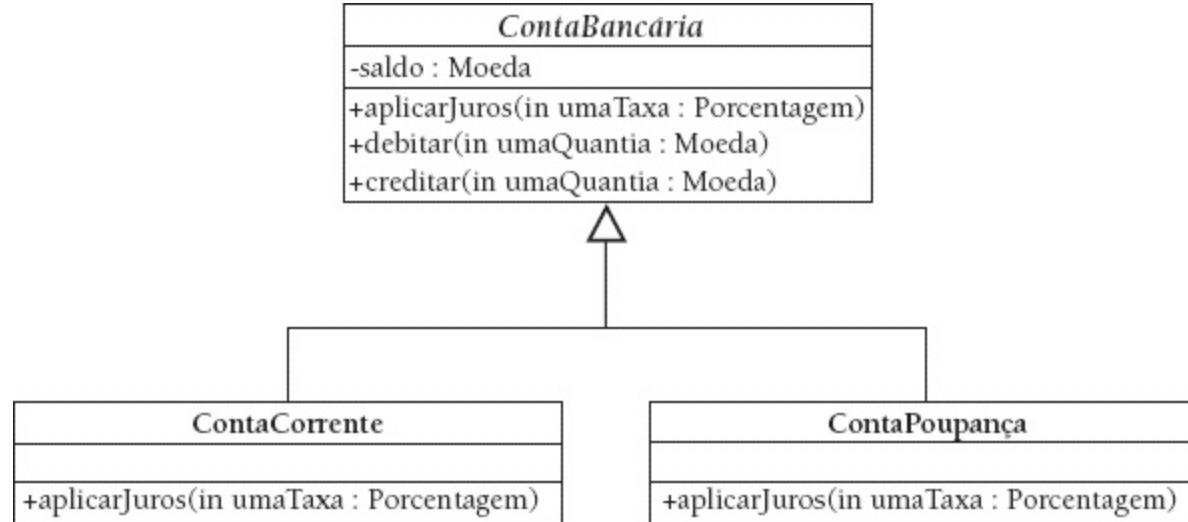


Figura 8-23: Redefinição de operações abstratas herdadas da superclasse.

O reúso de comportamento pode ocorrer não só pela herança entre classes, mas também pelo mecanismo de *delegação*. No reúso por delegação, sempre que um objeto não pode realizar uma tarefa (ou parte dela) por si próprio, ele delega a realização da mesma a outro(s) objeto(s). Nesse sentido, podemos afirmar que o objeto *reúsa* as operações dos objetos para os quais ele delega responsabilidades.

Como um exemplo que serve de comparativo para as duas estratégias de reúso de comportamento, a [Figura 8-24](#) ilustra a definição de uma classe Pilha segundo duas alternativas. Na primeira alternativa, a definição é feita pelo reúso por herança: a classe Pilha é definida com uma subclass de Vetor. Note que a classe Pilha herda também operações que não fazem sentido para objetos dessa classe (p. ex., elementos não podem ser inseridos em qualquer posição de uma pilha, assim como ocorre em um vetor). Na segunda alternativa, a definição é feita pela delegação. Um objeto Pilha agrega (tem como componente) um objeto Vetor. As operações na classe Pilha são implementadas pela delegação de mensagens para o objeto Vetor. Dessa forma, os clientes da classe Pilha somente têm acesso às operações definidas nessa classe. Isso ocorre porque o objeto Vetor está encapsulado na implementação da classe Pilha.

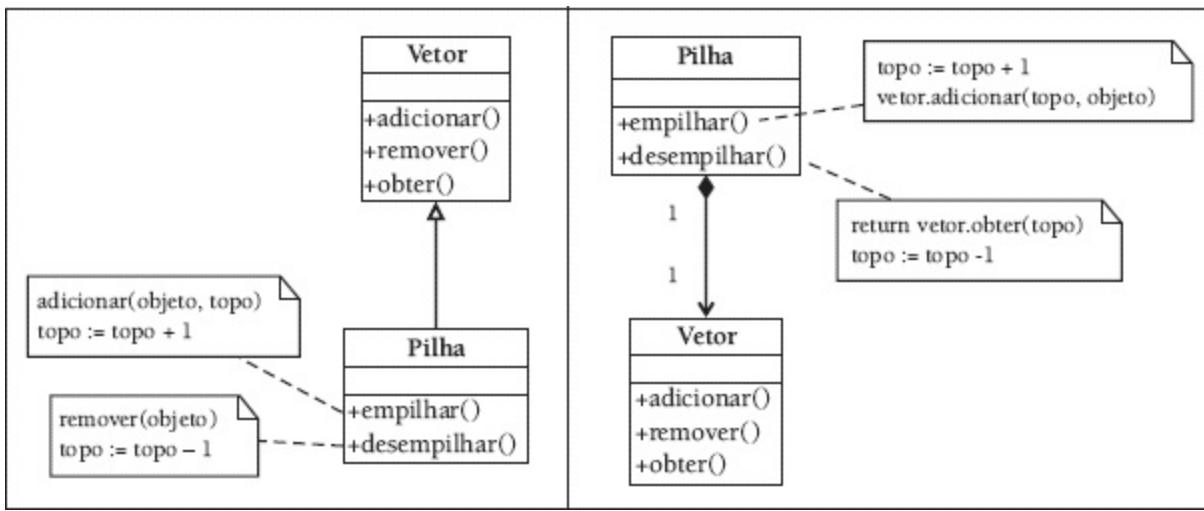


Figura 8-24: Duas definições da classe Pilha reutilizando a classe Vetor.

A delegação é mais genérica que a herança entre classes, pois um objeto pode reutilizar o comportamento de outro sem que o primeiro precise ser uma subclasse do segundo. Outra vantagem da delegação sobre a herança é que o compartilhamento de comportamento e o reúso podem ser realizados em tempo de execução. Na herança de classes, o reúso é especificado estaticamente. No entanto, a delegação pode apresentar perda de desempenho quando comparada a uma solução que use herança. Isso porque a delegação implica cruzar a fronteira de um objeto a outro para enviar a mensagem de delegação. Concluindo, há vantagens e desvantagens tanto no reúso por heranças e por delegação, e a utilização de uma ou outra depende da consideração de diversos fatores. De uma forma geral, *não* se recomenda utilizar herança nas seguintes situações:

- Para representar papéis de uma superclasse.
- Quando a subclasse for herdar propriedades que não se aplicam a ela.
- Quando um objeto de uma subclasse pode se transformar em um objeto de outra subclasse. P exemplo, um objeto da classe Cliente se transforma em um objeto da classe Funcionário.

8.5.7 Classificação dinâmica

Na maioria das situações em que a herança deve ser utilizada, temos o caso da *classificação estática*. Segundo essa classificação, desde o momento em que um objeto é instanciado até o momento em que é destruído, sua classe permanece a mesma. Entretanto, um problema com o qual o modelador pode se deparar na especificação e implementação de um relacionamento de herança é a *classificação dinâmica*, também chamada de *metamorfose*. Para entender esse conceito, considere uma empresa em que há empregados e clientes. Pode ser que uma pessoa, em determinado momento, seja apenas cliente; depois pode ser que ela passe a ser também empregada da empresa. A seguir, essa pessoa é desligada da empresa, continuando a ser cliente. Ou seja, um mesmo objeto pode pertencer a diferentes classes durante a sua vida (daí o nome *metamorfose*). Mais que isso, um mesmo objeto pode pertencer a múltiplas classes simultaneamente (p. ex., uma pessoa que é, ao mesmo tempo, cliente e funcionário da empresa).

As linguagens de programação orientadas a objetos clássicas (C++, Java, as linguagens .NET, Smalltalk) não dão suporte direto à implementação da classificação dinâmica. Nessas linguagens, se um objeto é instanciado como sendo de uma classe, ele não pode pertencer posteriormente a outra classe.

Uma solução parcial para o problema da classificação dinâmica é definir todas as possíveis subclasses em uma determinada situação. Utilizando essa solução, o problema da empresa descrito há pouco seria modelado como uma hierarquia de classes em que haveria as subclasses Empregado (objetos que são só empregados), Cliente (objetos que são só clientes) e EmpregadoCliente (objetos que são tanto clientes quanto empregados). Essa solução não resolve todo o problema, pois pode ser que um objeto mude de classe. Além disso, considere que o conceito de gerente tenha de ser adicionado à hierarquia de herança. Isso elevaria o número de subclasses de três para cinco (fica como exercício para o leitor verificar esse fato), tornando o modelo ainda mais complexo.

Uma melhor solução para o problema da classificação dinâmica é utilizar a técnica de delegação descrita na [Seção 8.5.6](#). Por meio dela, o relacionamento de herança existente entre cada subclass e a superclasse é substituído por uma composição de objetos. Como um exemplo dessa solução, considere a [Figura 8-25](#). A parte esquerda da figura exibe um fragmento de diagrama de classes de análise que apresenta uma hierarquia de classes. A parte direita da figura apresenta uma possível reestruturação feita na especificação para solucionar o problema da classificação dinâmica. Note que, no diagrama reestruturado, uma pessoa pode: (1) não apresentar comportamento nem de cliente nem de empregado; (2) apresentar comportamento somente de um cliente; (3) apenas de um empregado; ou (4) apresentar comportamento de ambos.

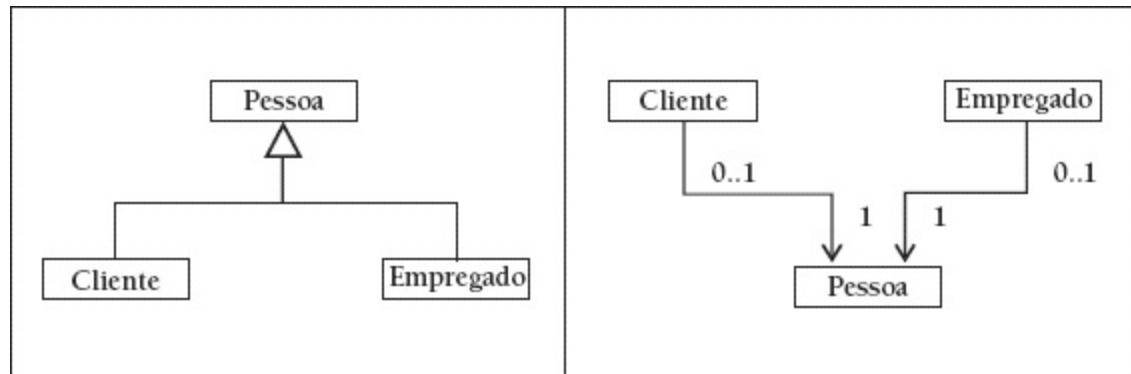


Figura 8-25: Possível solução para o problema da classificação dinâmica.

8.6 Padrões de projeto

É da natureza do desenvolvimento de software o fato de que os mesmos problemas tendem a acontecer diversas vezes. Na [Seção 5.4.4](#), descrevemos alguns padrões de análise, ou seja, padrões de software úteis para serem utilizados em problemas de modelagem recorrentes na fase de análise do desenvolvimento de um SSOO. Outra categoria de padrões de software extremamente útil são os padrões de projeto (tradução para *design patterns*). O texto clássico a respeito desse assunto é o livro de Eric Gamma e seus colaboradores (GAMMA *et al.*, 2000). Esses autores são popularmente conhecidos como equipe GoF (sigla de *Gang of Four*, por serem quatro autores). Nesse livro, os autores catalogaram 23 padrões de projeto. Os padrões de projeto descritos pela equipe GoF foram divididos em três categorias, descritas a seguir.

1. **Criacionais:** procuram separar a operação de uma aplicação de como os seus objetos são criados.
2. **Estruturais:** proveem generalidade para que a estrutura da solução possa ser estendida no futuro.

3. **Comportamentais**: utilizam herança para distribuir o comportamento entre subclasses, ou agregação e composição para construir comportamento complexo a partir de componentes mais simples.

Os 23 padrões documentados pela equipe GoF são enumerados na [Tabela 8-5](#).

Uma descrição detalhada de todos os padrões enumerados mais adiante está fora do escopo deste livro. Para um estudo mais detalhado sobre o assunto, o leitor pode consultar o texto de Gamma (2000). Outra referência importante é a que descreve os padrões de projeto do catálogo J2EE (ALUR *et al.*, 2003). Por outro lado, consideramos importante, mesmo em um livro introdutório como este, dar uma visão geral de um assunto tão importante para o desenvolvimento de um SSOO. Para isso, descrevemos nas próximas seções três dos padrões de projeto representativos da categoria GoF. São eles: Composite, Observer, Strategy, Factory Method, Mediator e Façade.

Tabela 8-5: Padrões de projeto documentados pelo catálogo GoF

Criacionais	Estruturais	Comportamentais
Abstract Factory Builder Factory Method Prototype Singleton	Adapter Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interpreter Iterator Mediator Memento Observer State Strategy Template Method Visitor

Um padrão de projeto corresponde a um esboço de uma solução reusável para um problema comumente encontrado em um contexto particular. Estudar padrões é uma maneira eficaz de aprender com a experiência de outros.

8.6.1 Composite

O padrão Composite é um dos 23 catalogados no livro de Eric Gamma e de seus colaboradores. O problema que esse padrão considera é o seguinte: como definir uma relação hierárquica entre objetos de tal forma que tanto o objeto todo quanto os objetos parte sejam equivalentes em certos aspectos?

O padrão Composite pode ser utilizado para solucionar o problema de representar uma hierarquia de composição recursiva entre entidades. Esse problema é conhecido como “problema da explosão de partes” (*parts explosion problem*). Seu exemplo clássico é o da montagem de peças: uma peça é composta de diversas outras, e essas peças componentes são elas próprias compostas, e assim por diante.

A [Figura 8-26](#) exibe a estrutura do padrão Composite (só apresentadas somente as classes, sem atributos nem operações). O componente é uma superclasse. Há dois tipos de componente, folha e composto. Uma folha não possui componentes, enquanto um composto possui. A associação entre composto e componente é que representa a hierarquia de composição recursiva: um composto possui diversos componentes, que podem ser ou folhas ou outros compostos.

O padrão GoF Composite representa uma hierarquia de composição recursiva: um composto possui diversos componentes, que podem ser ou folhas ou outros compostos.

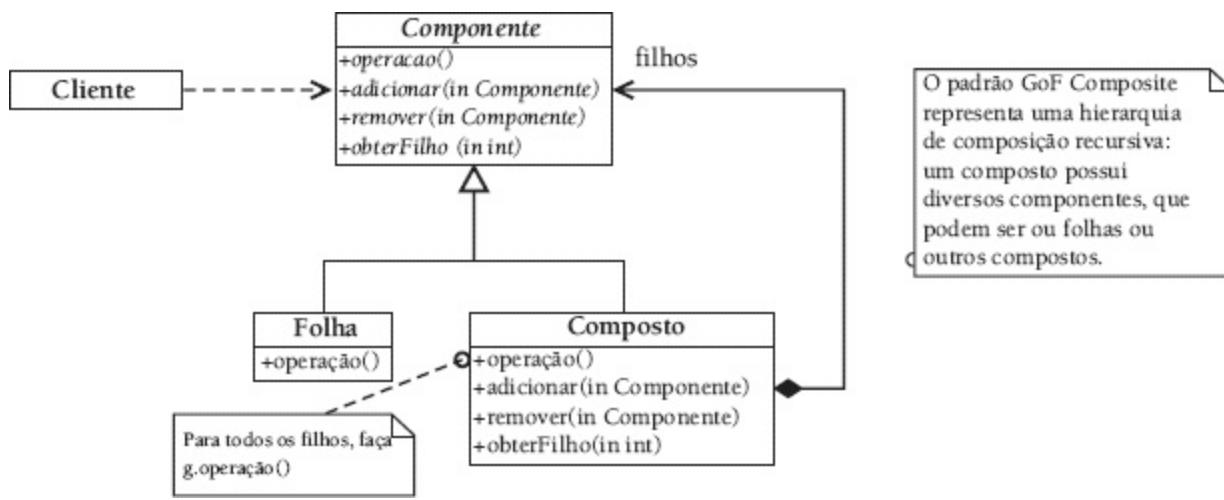


Figura 8-26: O padrão Composite.

8.6.2 Observer

O objetivo desse padrão é definir de forma flexível uma dependência um-para-muitos entre objetos. Ou seja, o padrão Observer é útil quando há um objeto central e diversos outros objetos que *dependem* do primeiro. A dependência aqui é no sentido de que, se houver alguma modificação no estado do objeto central, os objetos dependentes devem ser notificados. A preocupação aqui é com respeito ao acoplamento: necessitamos que o objeto central seja capaz de enviar mensagens de notificação aos seus dependentes sem, no entanto, conhecê-los diretamente. Em outras palavras, desejamos que haja um acoplamento fraco entre os objetos dependentes e o objeto central.

A solução que o padrão Observer descreve para o problema é fazer com que os objetos dependentes implementem uma interface (ver [Seção 8.5.4](#)) comum para propiciar o acoplamento abstrato entre cada dependente e o objeto central. Além disso, o objeto central deve manter uma lista de referências para seus dependentes. Quando o estado do objeto central é modificado, os dependentes são comunicados por intermédio dessa interface. Tudo o que o objeto central precisa saber é que existem objetos que implementam essa interface. No entanto, o objeto central não precisa ter conhecimento de quais são as classes dos objetos dependentes.

Quadro 8-8: Interface Observer

```
interface Observer {
    void update(Observable t, Object o);
}
```

A interface que o objeto central assume que seus dependentes implementam é apresentada no [Quadro 8-8](#) (utilizamos a sintaxe da linguagem Java). Essa interface possui uma única operação, *update*. A classe *Observable* é uma superclasse do objeto central. Ao ser modificado, esse objeto notifica a seus dependentes chamando a operação *update* e passando a si próprio como primeiro argumento. Dessa forma, cada observador (objeto dependente) pode consultar o objeto central para saber o que foi modificado neste último.

A razão para a necessidade de a classe do objeto central ser uma subclasse de *Observable* é que essa última fornece duas funcionalidades importantes para suas subclasses: (1) a de *notificação* dos dependentes e (2) a de *manutenção* desses dependentes. Em particular, a classe *Observable* em Java contém as seguintes operações, para manutenção da lista de objetos dependentes:

```
public void addObserver(Observer obs)
```

Adiciona um novo objeto na lista de objetos dependentes.

```
public void deleteObserver(Observer obs)
```

Remove um objeto da lista de objetos dependentes.

Note que essas operações são definidas em termos da interface Observer. Isso significa que, para um objeto ser inserido na lista de dependentes (observadores) do objeto central, basta o primeiro implementar a interface Observer. A [Figura 8-27](#) dá uma visão geral da estrutura do padrão Observer.

Uma aplicação prática do padrão de projeto Observer ocorre durante a implementação da comunicação entre camadas de software (ver [Seção 11.1.1](#)). Esse padrão é útil nesse caso por permitir que uma camada de software mais genérica possa enviar sinais para uma camada menos genérica. O elemento pertencente à camada mais genérica, representado pelo objeto central, pode estar associado a diversos objetos que dele dependem e que estão localizados na camada mais específica, que são representados como objetos dependentes.

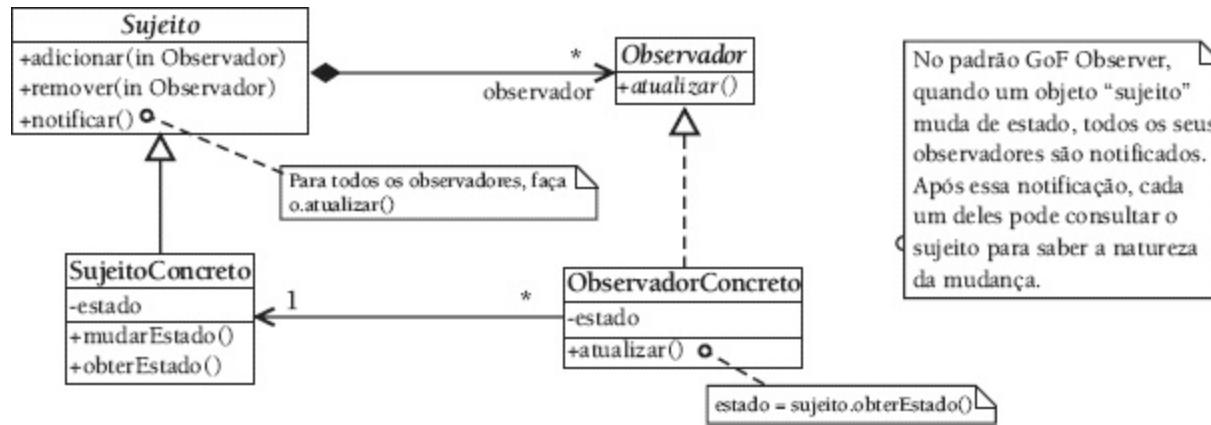


Figura 8-27: Estrutura do padrão Observer.

8.6.3 Strategy

O padrão Strategy tem o objetivo de encapsular diferentes algoritmos para realização de alguma tarefa computacional por trás de uma interface e permitir que a região de código cliente dessa tarefa possa utilizar qualquer um desses algoritmos sem precisar ser modificada. Podemos também interpretar o padrão Strategy como uma forma de desacoplar uma região de código cliente de uma tarefa das diferentes maneiras de implementar essa tarefa.

Como exemplo do padrão Strategy, considere novamente nosso estudo de caso representado pelo Sistema de Controle Acadêmico (ver [Seção 4.7](#)). A instituição de ensino na qual o SCA está para ser implantado utiliza uma forma de calcular o grau final de um aluno em uma disciplina cursada. Esse grau é uma letra: A, B, C, D ou E. Além disso, ele é calculado a partir de notas atribuídas a avaliações. O projetista do SCA identificou que, atualmente, cada nota varia na faixa de 0 a 10. No entanto, percebeu também que é comum a coordenação modificar a estratégia (algoritmo) de atribuição de graus a partir da notas das avaliações. Esse projetista definiu o diagrama de classes da [Figura 8-28](#), correspondente a uma estrutura de classes para cálculo do grau. Nesse diagrama, nessa solução, Avaliacao é uma classe que representa o registro do aproveitamento (em termos de notas e frequência) obtido por um aluno em uma turma. Além disso, a operação *calcular* retorna o grau do

aluno.

A solução proposta pelo projetista foi inspirada pelo padrão de projeto Strategy. Na Figura 8-28, temos uma hierarquia de classes. A raiz dessa hierarquia é uma classe abstrata. Essa classe possui uma operação abstrata, *calcular*. As duas subclasses fornecem implementação para a operação abstrata que herdam de sua superclasse. Uma região de código da aplicação fica responsável por: (1) identificar qual é a estratégia vigente para cálculo de grau; e (2) instanciar o objeto da subclasse correspondente. Uma vez que esse objeto está instanciado, a região de código cliente pode, por intermédio da interface, enviar a mensagem que ativa a execução do método *calcular*.

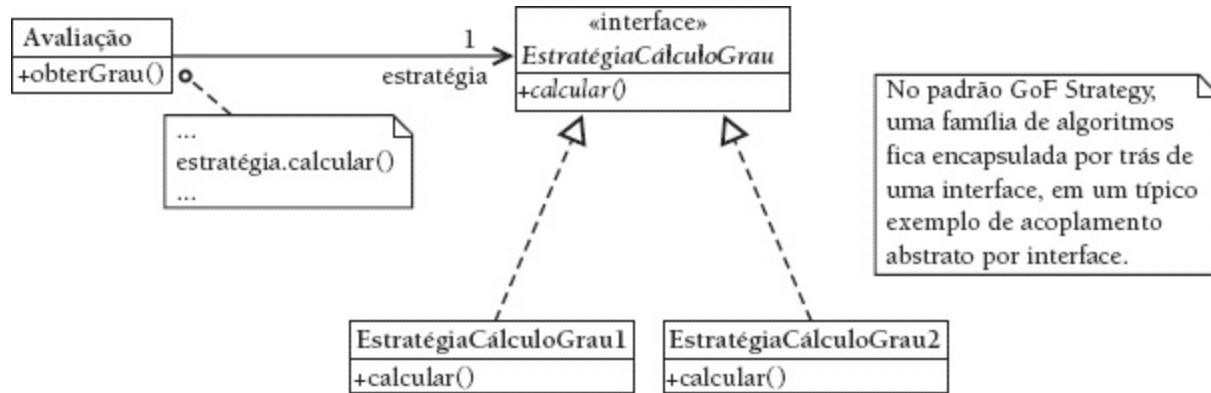


Figura 8-28: Exemplo do padrão de projeto Strategy.

A grande vantagem dessa solução está na flexibilidade resultante. Para entender isso, note que a região de código cliente não precisa ser alterada junto com a forma de cálculo de grau. Tudo o que a região de código cliente conhece acerca da região de código fornecedora do serviço (nesse caso, o cálculo do grau do aluno em uma disciplina) é que existe uma operação chamada *calcular*, que realiza o cálculo. A região de código cliente não sabe nem precisa saber qual o objeto específico que está fornecendo uma implementação dessa operação.

É importante notar também que uma solução alternativa equivalente à utilização de uma classe abstrata seria utilizar uma *interface* (ver Seção 8.5.4). Nessa solução, as classes *EstratégiaCálculoGrau1* e *EstratégiaCálculoGrau2* iriam implementar a *interface*. Em ambos os casos, entretanto, o *acoplamento abstrato* (ver Seção 8.5.5) é mantido entre os elementos envolvidos.

8.6.4 Factory Method

Quando uma determinada parte de um SSOO precisa dos serviços de um objeto, essa parte envia mensagem para este último. No entanto, para que um objeto receba mensagens, ele deve existir: não faz o menor sentido falar em “envio de mensagens” para um objeto que não existe. Para um objeto existir, ele deve ser *instanciado* a partir de uma classe. Instanciar uma classe significa criar um objeto dela. Fisicamente falando, isso significa: (1) alocar espaço em memória para armazenar o objeto; e (2) iniciar o seu estado (ou seja, os valores de seus atributos). Após a instanciação, o objeto pode prover serviços para outros objetos.

A operação de instanciação de um objeto pode ser bastante complexa. Além disso, pode ser que não seja adequado fazer com que uma região de código que necessite de um serviço tenha uma referência direta para a classe que o fornece. Uma razão para isso pode ser o fato de que a forma de implementar tal serviço é instável, no sentido de precisar ser alterada no futuro. Se a região de código instanciar diretamente a classe que lhe fornece determinado serviço, essa região fica

acoplada definitivamente a essa classe fornecedora e a sua forma específica de prover o serviço requerido.

Uma forma de resolver o problema descrito no parágrafo anterior é com uma *fábrica de objetos*, que corresponde a uma classe cuja responsabilidade é criar quando requisitada. O fato é que essa fábrica retorna o objeto criado na forma de uma referência para uma *superclasse abstrata* (ver Seção 8.5.2) ou para uma *interface* (ver Seção 8.5.4). Dessa maneira, a região cliente, que previamente criava o objeto para lhe prover o serviço, agora fica dependente desse serviço através de um *acoplamento abstrato* (ver Seção 8.5.5). Com efeito, a classe que realmente fornece o serviço requerido pode ser alterada sem que a região de código cliente precise de modificação. Em essência, essa é a solução que o padrão Factory Method descreve. Em outras palavras, o objetivo do padrão Factory Method é definir uma interface para instanciar objetos. Esse é um dos padrões criacionais definidos pelo GoF.

Note que a fábrica de objetos precisa fazer referência às classes concretas cujos objetos fornecem o serviço requerido, o que é um exemplo de *acoplamento concreto*. Entretanto, essa referência fica localizada em uma região de código particular (na fábrica de objeto) e é consequentemente aceitável. Note que quaisquer mudanças nas classes que fornecem o serviço em questão ficam localizadas na fábrica de objetos e não afetam os clientes.

Outro aspecto importante sobre o padrão Factory Method é que sua utilização adiciona complexidade ao projeto. Na verdade, essa é uma tônica na maioria dos padrões de projeto: eles adicionam *flexibilidade* à custa de acrescentar também *complexidade* à solução. No caso particular do padrão Factory Method, a decisão por utilizar o mesmo deve levar em consideração o fato de a classe que fornece o serviço ser realmente suscetível a mudanças futuras. Do contrário, a referência (instanciação) direta da classe que fornece o serviço (em vez de utilização do padrão Factory Method) é uma alternativa mais viável.

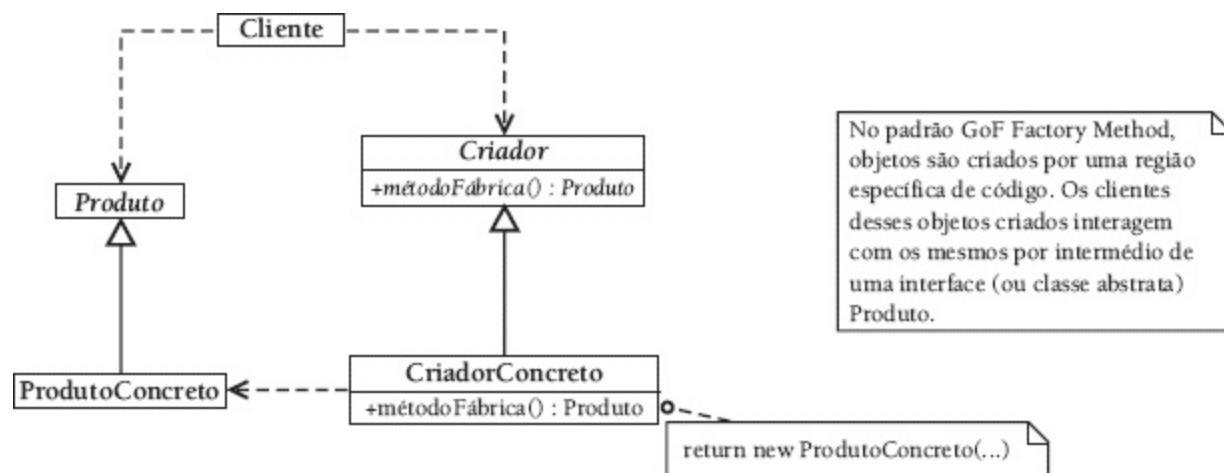


Figura 8-29: Estrutura do padrão Factory Method.

Outro padrão GoF de criação comumente utilizado é o chamado *Abstract Factory*. Esse padrão é uma extensão do Factory Method para criação de uma família de objetos relacionados.

8.6.5 Mediator

O padrão de projeto Mediator permite a um grupo de objetos interagir, ao mesmo tempo em que mantém um acoplamento fraco entre os componentes desse grupo. A solução proposta pelo padrão Mediator para alcançar esse objetivo é definir um objeto, o *mediador*, para encapsular interações da

seguinte forma: o resultado da interação de um subgrupo de objeto é passado a outro subgrupo pelo mediador. Dessa forma, os subgrupos não precisam ter conhecimento da existência um do outro e podem variar de maneira independente. Os objetos de controle, cuja especificação descrevemos na Seção 8.1.3, são exemplos de mediadores.

8.6.6 Façade

No [Capítulo 11](#), descrevemos aspectos relativos à divisão de um SSOO em subsistemas. Nessa divisão, é necessário definir as interfaces de comunicação (ou interação) entre os subsistemas resultantes. Nesse contexto, quando dois subsistemas se comunicam, podemos dizer que há um cliente e um fornecedor. O subsistema cliente requisita algum serviço, e o subsistema fornecedor é o que provê esse serviço. O padrão Façade procura resolver o seguinte problema: *como definir uma interface de alto nível que torna um subsistema mais fácil de ser utilizado?* Em outras palavras, de que maneira podemos definir uma interface de comunicação mínima possível entre os subsistemas cliente e fornecedor?

A solução fornecida por este padrão é a seguinte: criar uma *fachada* para o subsistema fornecedor, de tal forma que o subsistema cliente se comunique com o primeiro por intermédio desta fachada. A vantagem dessa solução é que o cliente conhece apenas o necessário e suficiente em relação à complexidade do fornecedor. Toda a complexidade adicional desse último fica “escondida” por trás da interface de comunicação.

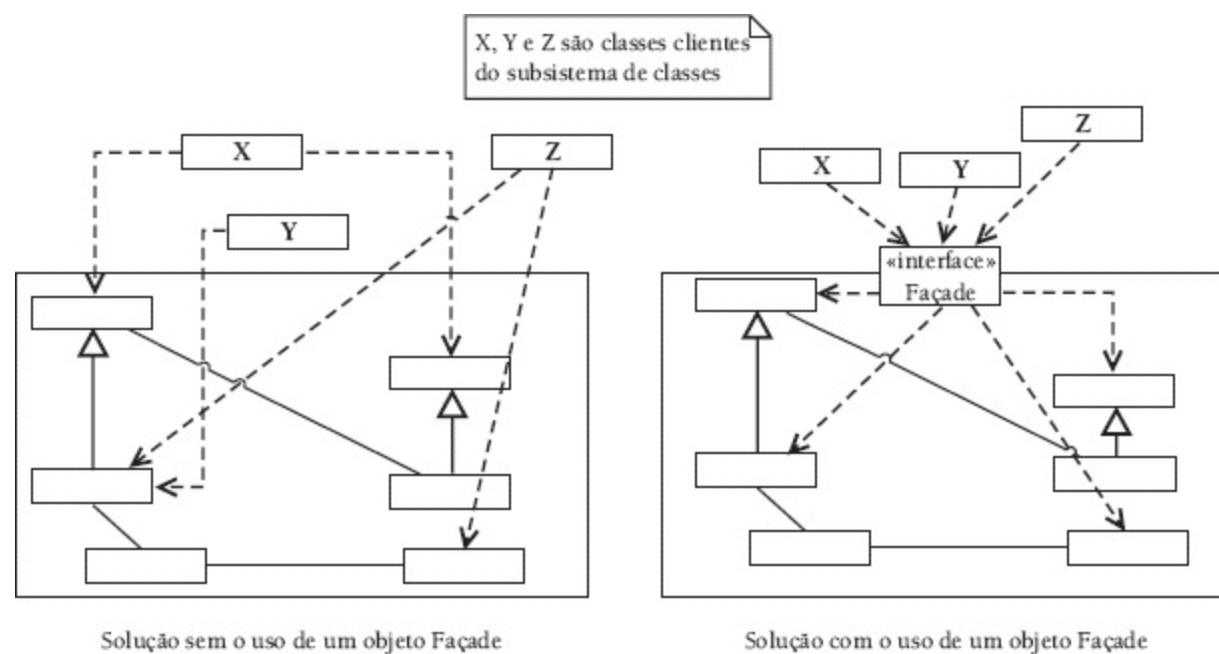


Figura 8-30: Estrutura do padrão Façade.

A implementação do padrão Façade é relativamente simples. Consiste em definir uma ou mais classes que implementam a interface de comunicação necessária e suficiente. O subsistema fornecedor fica encapsulado por essa interface.

8.7 Modelo de classes de projeto em um processo iterativo

Em um processo de desenvolvimento iterativo, o modelo de classes de projeto é construído em

diversas iterações. Esse modelo tende a ser construído **em paralelo** com o *modelo de interações* (ver [Capítulo 7](#)). Isso porque o modelo de interações fornece informações para completar o modelo de classes em seu estágio de análise a fim de levá-lo ao estágio de projeto. Em particular, uma mensagem que identificamos durante a construção dos modelos de interações sempre corresponde a uma operação na classe do objeto destinatário dessa mensagem. Para cada mensagem identificada, a assinatura da operação correspondente deve ser especificada no modelo de classes de projeto.

Na [Seção 5.5.3](#), apresentamos o conceito de Visão de Classes Participantes (VCP) como um diagrama de classes que participam de um determinado caso de uso. VCPs podem começar a ser construídas na etapa de análise, mas são tão mais importantes na etapa de projeto. Isso porque, nesta etapa, novas classes são adicionadas ao sistema para contemplar aspectos relativos à solução do problema. Dessa forma o uso de VCPs serve como um mecanismo de modularização para a construção do modelo de classes de projeto: em vez de criar apenas um diagrama de classes para o sistema todo, podemos criar uma VCP para cada caso de uso. Nessa VCP, são apresentadas as classes do domínio que participam do caso de uso, assim como quaisquer outras cujo propósito é dar suporte a sua execução. Como exemplo dessa abordagem é apresentado na [Seção 8.8](#), na qual apresentamos VCPs para três casos de uso do SCA.

Conforme descrevemos na [Seção 10.2.3](#), o diagrama de atividades pode ser utilizado na construção do modelo de classes de projeto, com o objetivo de especificar alguma operação cuja lógica é um tanto mais complexa. No entanto, é importante notar que, em um SSOO, operações são normalmente simples, o que dispensa o detalhamento de sua lógica interna. No mais das vezes, essas operações têm dez instruções ou menos. Algumas vezes essas operações têm uma ou duas instruções. Além disso, o modelo de interações fornece informações suficientes para a implementação correta da lógica de funcionamento de uma operação. Em suma, responsabilidades e operações que não são nem simples nem claras sugerem que a classe correspondente está mal projetada. Da mesma forma, classes com uma quantidade excessiva de operações (mais de uma dúzia) também são altamente suspeitas. Aliás, clareza e simplicidade devem ser perseguidas desde a fase de análise, durante a identificação das classes iniciais. Lembre-se da modelagem CRC, em que cada cartão possui um tamanho fixo, forçando o modelador a projetar classes simples. Os mesmos princípios de clareza e simplicidade também devem ser perseguidos durante a modelagem de interações, pois essa atividade gera informações para a completa especificação das operações.

O conceito de reúso é fundamental durante a construção do modelo de classes de projeto. Os padrões de projeto são mecanismos de reúso de soluções para problemas recorrentes no desenvolvimento de software. Outra forma de reúso que deve ser considerada na etapa de projeto, não somente para a modelagem de classes, é o uso de bibliotecas de classes e de *frameworks*. Na [Seção 8.1](#), apresentamos uma pequena introdução a esses mecanismos de reúso.

É adequado dizer que as atividades de especificação (projeto) e de implementação são intimamente entrelaçadas. Nas palavras de Philippe Krutchen, um reconhecido arquiteto de software: “*Eu codifico para melhor entender o que estou projetando.*” Essa sinergia entre o projeto e codificação de um SSOO se justifica pelo fato de que, ao ser forçado a moldar certa decisão de projeto em uma implementação concreta e sem ambiguidades, o desenvolvedor valida a completa utilidade das classes que projetou. Note outra declaração de dois outros respeitados tecnologistas, Don Roberts e Ralph Johnson: “*As pessoas desenvolvem abstrações por meio da generalização aplicada a exemplos concretos. Cada tentativa de determinar abstrações corretas no papel, sem realmente desenvolver um sistema, está fadada ao insucesso.*” Há ainda outra citação, dessa vez do lendário Fred Brooks: “*Planeje para jogar uma [versão do sistema] fora; isso acontecerá de*

qualquer maneira." Todas essas declarações carregam a mesma mensagem, em essência: quando uma parte do sistema é implementada, isso resulta em informações para melhorar a qualidade do projeto do qual partiu a implementação. Nesse contexto, a construção de *protótipos* (ver [Seção 2.5](#)) para validar as decisões de projeto é importante.

Uma dúvida frequente diz respeito ao nível de detalhe na especificação dos diagramas de classes de projeto. Há diversos fatores que influenciam a resposta para essa questão. Em primeiro lugar, devemos perceber que um modelo que não esteja em conformidade com o sistema não tem serventia. Por outro lado, quanto mais detalhes os modelos que construímos possuem, mais difícil é mantê-los em sincronia com o código-fonte, pois este último é muito instável. Se a equipe de desenvolvimento tiver à disposição ferramentas CASE que permitam realizar engenharias direta e reversa (ver [Seção 2.6](#)), a construção de modelos mais detalhados é possível. Isso porque as próprias ferramentas podem atualizar os modelos existentes. Entretanto, se este não for o caso, a equipe corre o risco de perder tanto tempo mantendo os modelos atualizados quanto leva para realmente implementar o sistema. Portanto, o bom senso também vale para decidir o nível de detalhamento que deve ser adotado na construção, não só dos diagramas de classes, mas de qualquer modelo de um SSOO.

8.8 Estudo de caso

Esta seção continua a apresentação da modelagem do Sistema de Controle Acadêmico e apresenta visões de classes participantes (veja a [Seção 5.5.3](#)) para os três casos de uso já considerados em outras seções em que este estudo de caso foi abordado:

- Fornecer Grade de Disponibilidades ([Figura 8-32](#))
- Realizar Inscrição ([Figura 8-33](#))
- Lançar Avaliações ([Figura 8-34](#))

Repare que, embora essas VCPs sejam apresentadas neste capítulo, a sua construção de fato ocorreu *em paralelo* com os modelos de interações apresentados na [Seção 7.7](#). Isso porque, quando uma mensagem é definida em um modelo de interações, é também necessário definir uma operação na classe do objeto receptor dessa mensagem.

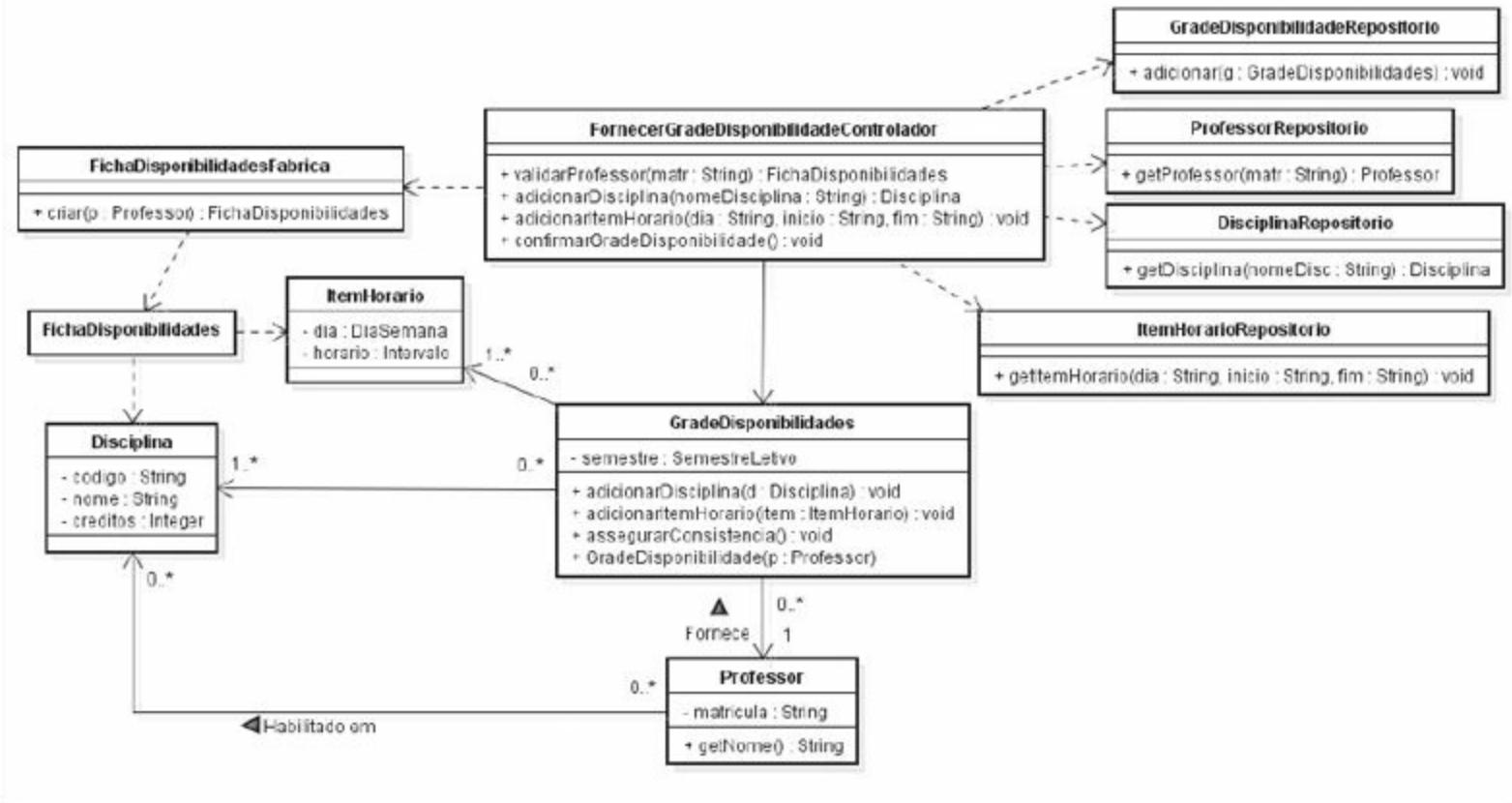


Figura 8-31: VCP para o caso de uso Fornecer Grade de Disponibilidade.

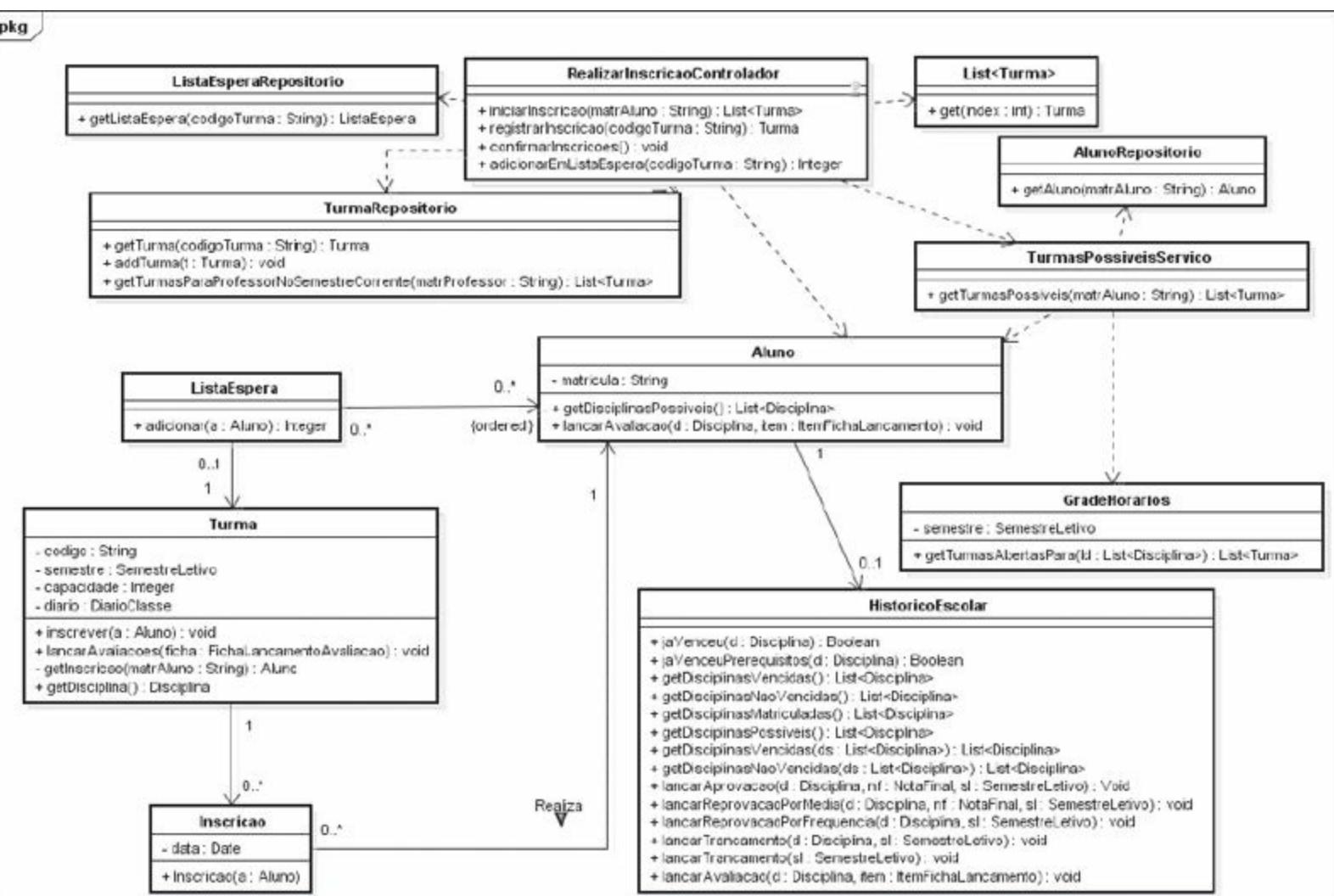


Figura 8-32: VCP para o caso de uso Realizar Inscrição.

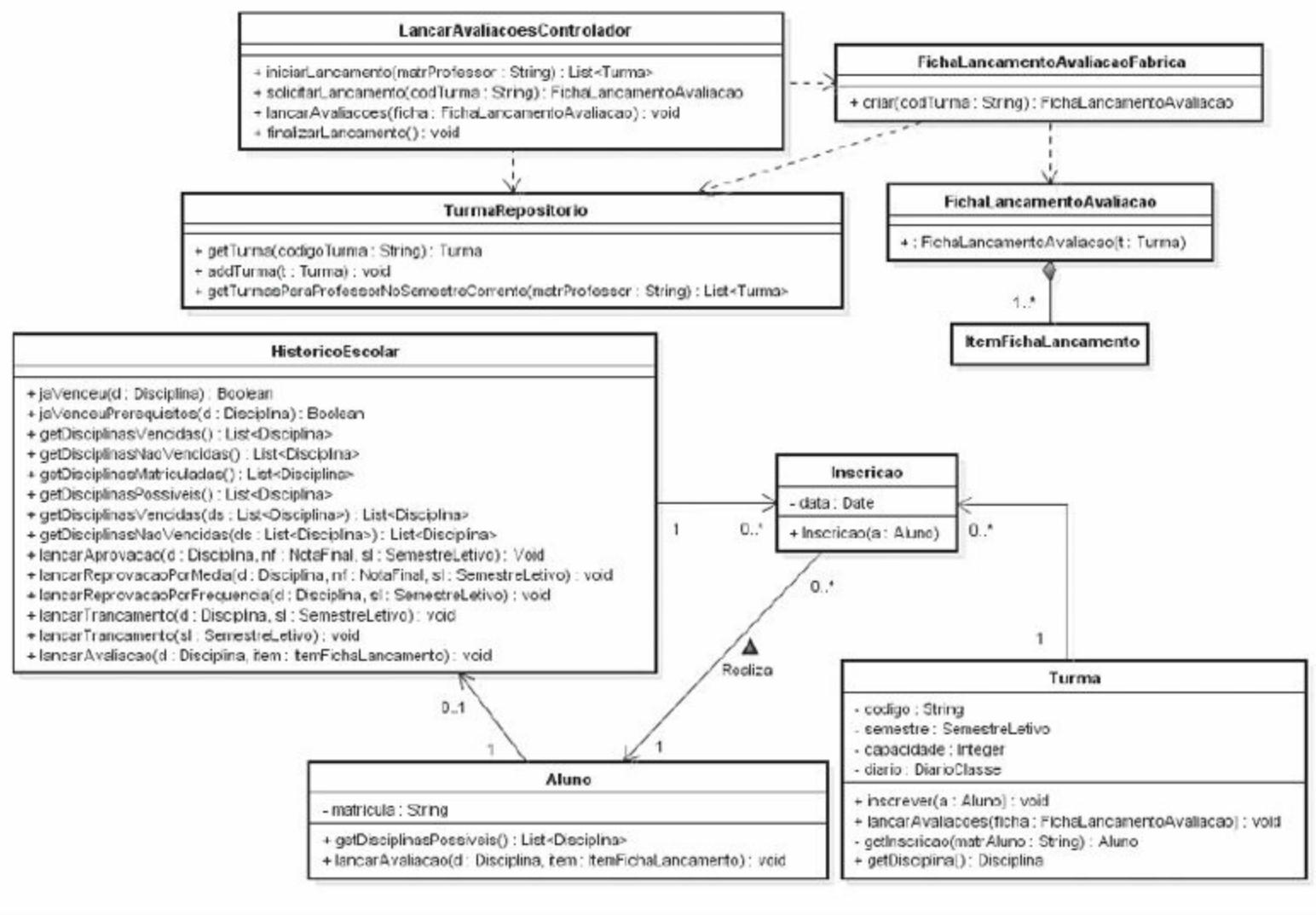


Figura 8-33: VCP para o caso de uso Lançar Avaliações.

► EXERCÍCIOS

8-1: No [Capítulo 7](#), descrevemos os conceitos de coesão e acoplamento. Qual é a relação desses conceitos com a qualidade resultante da modelagem de classes de projeto?

8-2: Este capítulo apresenta o relacionamento de dependência e descreve diversos tipos de dependência que podem existir (por atributo, por variável local etc.). Seja uma classe A que possui pelo menos uma operação cujo tipo de retorno é a classe B. Que tipo(s) de dependência pode haver entre A e B?

8-3: Em cada um dos itens a seguir, discuta qual tipo de relacionamento é mais adequado (associação, agregação, composição). Desenhe o diagrama de classes correspondente, indicando as multiplicidades. Especifique, ainda, atributos e possíveis restrições.

- a) Um País possui uma Capital.
- b) Uma Empresa é subsidiária de diversas outras Empresas.
- c) Uma Pessoa à mesa de jantar está usando uma Faca.
- d) Um Polígono é composto por um conjunto ordenado de Segmentos.
- e) Um Estudante acompanha uma Disciplina com um Professor.
- f) Uma Caixa contém Garrafas.
- g) Um Livro contém Capítulos.
- h) Um Arquivo contém Registros.

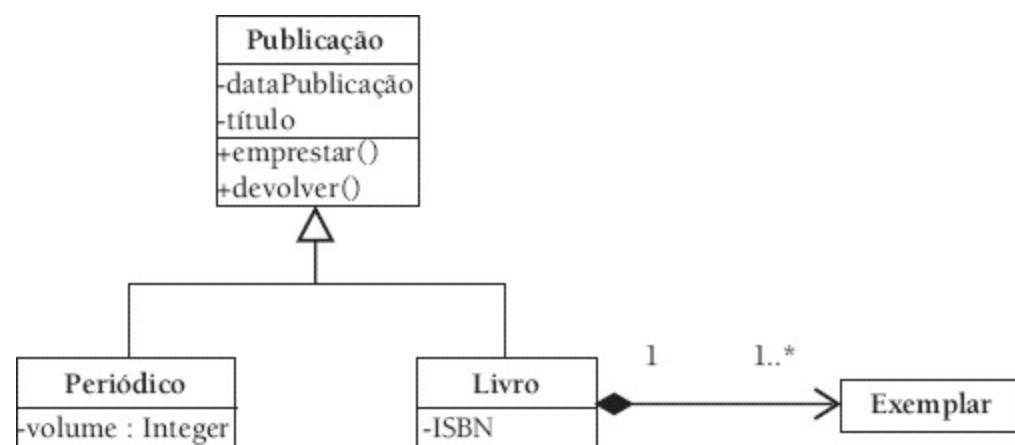
8-4: Para cada um dos itens do exercício anterior, defina refinamentos sobre as associações existentes, considerando diversas possibilidades de naveabilidade entre tais associações.

8-5: Forme uma hierarquia de classes a partir dos seguintes conceitos: pessoa, empregado, instrutor e estagiário.

8-6: Em cada um dos itens abaixo, desenhe o diagrama de classes correspondente, indicando as multiplicidades. Especifique, ainda, possíveis restrições que se apliquem.

- a) Uma pessoa, como programador, utiliza uma linguagem de programação.
- b) Um objeto de desenho pode ser um texto, um gráfico ou um grupo de objetos.
- c) Modem, teclado e impressora são dispositivos de entrada e saída.
- d) Um banco de dados contém tabelas de sistema e tabelas de usuário. Uma tabela de sistema mantém informações sobre uma ou várias tabelas de usuário. Uma tabela contém registros.
- e) Um Item pode ser atômico ou composto. Um item composto possui dois ou mais Itens.

8-7: O que há de errado com o diagrama de classes a seguir? Construa uma nova versão deste diagrama, consertando os erros identificados.

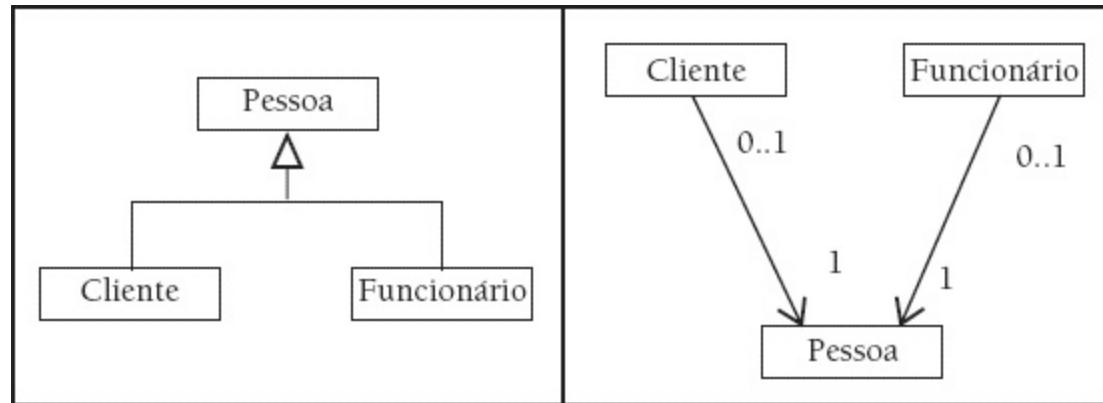


8-8: Crie um diagrama de classes de projeto para representar os conceitos de círculo e de elipse como classes. Defina uma operação desenhar em ambas as classes. Dica: crie uma superclasse abstrata para organizar a hierarquia.

8-9: Considere as classes Rádio e Relógio a seguir. Qual o problema em definir uma classe RádioRelógio como subclasse das duas classes (herança múltipla)? Defina uma solução, utilizando o mecanismo de delegação, para substituir a solução por herança múltipla. Defina outra solução através de interfaces.

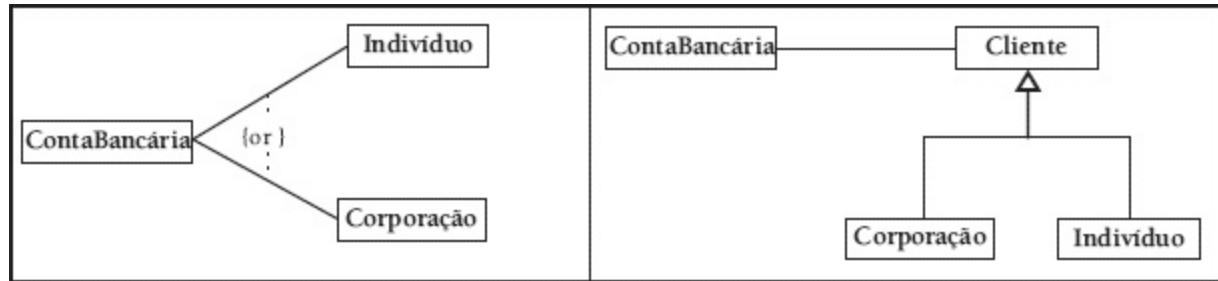


8-10: No desenvolvimento de um sistema de software comercial, um modelador precisou representar as classes clientes e funcionários de uma empresa. Em vez de optar pela solução do diagrama à esquerda, ele optou pela solução representada à direita na figura a seguir. Discuta as vantagens e desvantagens de cada uma das duas soluções. Reflita sobre a situação em que um cliente pode também ser um funcionário da empresa. E se uma classe gerente tivesse de ser modelada?

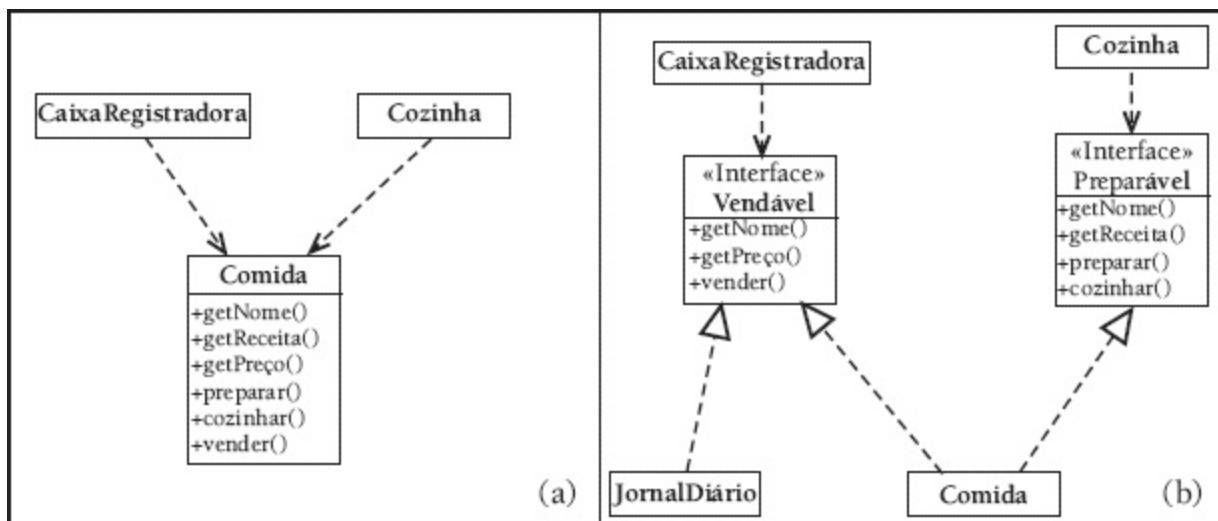


8-11: Considere os diagramas de classes de análise fornecidos nos itens (a) e (b) a seguir, ambos de acordo com a notação da UML. Esses diagramas desejam representar o fato de que uma conta bancária pode estar associada a uma pessoa, que pode ser ou uma pessoa física (representada pela classe *Individuo*), ou uma pessoa jurídica

(representada pela classe *Corporação*). Uma dessas duas soluções é melhor que a outra? Se sim, qual delas e em que sentido?



8-12: Considere uma aplicação para um bar-café. Nessa aplicação, considere a existência de uma classe que representa um comestível qualquer vendido pelo bar-café: *Comida*. Considere ainda duas outras classes nessa aplicação, *Cozinha* e *CaixaRegistradora*. A classe cozinha manipula objeto da classe Comida para montar pratos. Já a classe CaixaRegistradora manipula objeto comida para registrar a venda dos mesmos e cobrar por eles. Portanto, essas duas classes dependem dos serviços fornecidos pela classe Comida. Em um primeiro modelo dessa aplicação, o modelador fez com que as classes Cozinha e CaixaRegistradora dependessem diretamente da classe Comida, conforme a [Figura \(a\)](#). No entanto, conforme o desenvolvimento foi se evoluindo, o modelador identificou um novo requisito na aplicação: agora era preciso registrar a venda de coisas não comestíveis. Por exemplo: o café-bar passou a vender jornais diários. Para atender ao novo requisito, o modelador criou duas novas interfaces, *Vendável* e *Comestível*, conforme está na [Figura \(b\)](#). Discuta a decisão de projeto do modelador. Você achou a decisão adequada? Que princípios de projeto levaram o modelador a tomar tal decisão?



8-13: Você está desenvolvendo uma aplicação para uma empresa que vende componentes de computador. Atualmente, você está construindo uma hierarquia de classes para representar os diferentes tipos de componentes (você nomeou essas

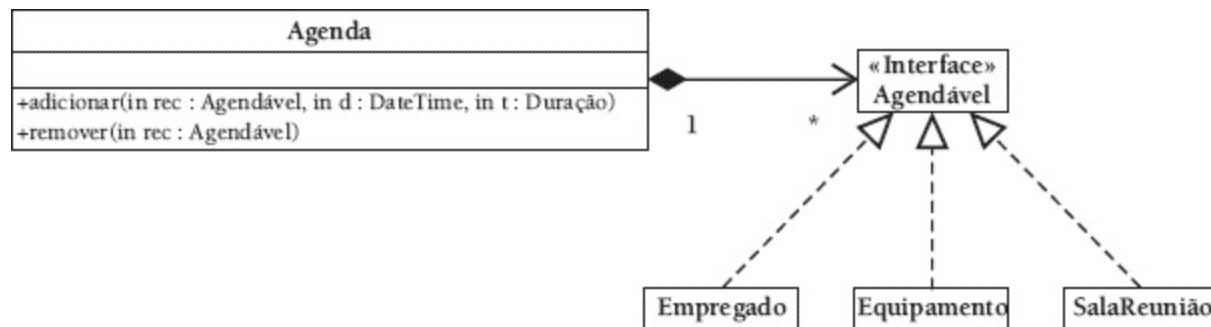
classes como *Processador*, *DiscoRígido* e *CD-ROM*). Essa hierarquia contém também uma superclasse abstrata denominada *Componente*, da qual são derivadas as demais classes. Por outro lado, um amigo seu já desenvolveu um sistema semelhante e lhe passou classes para representar discos rígidos e CPUs (as classes dele são denominadas *HardDisk* e *CPU* respectivamente). De que maneira você pode construir sua hierarquia de classes aproveitando (reutilizando) as classes fornecidas pelo seu amigo sem precisar modificá-las? Que padrão de software poderia ser empregado para auxiliar na definição da estrutura de classes a fim de representar a situação discutida anteriormente? Utilizando esse padrão, forneça um fragmento de diagrama de classes que represente essa situação.

8-14: Suponha que você precise criar em uma aplicação um objeto que, uma vez instanciado, não deve ter seu estado alterado sob hipótese alguma. Descreva uma solução para esse problema. Pode pensar em possíveis aplicações para essa situação. Dica: procure por *Immutable Pattern* na Internet.

8-15: Suponha que você precise ter uma classe C em sua aplicação para a qual a forma de acesso é diferenciada, no seguinte sentido: há um conjunto de classes que têm permissão para alterar o estado dos objetos da classe C; há também outro conjunto de classes que, embora façam uso dos serviços fornecidos por objetos da classe C, não têm permissão para modificar o estado desses objetos. Defina uma solução de projeto para esse problema. Dica: utilize o conceito de interface.

8-16: Imagine que um projetista está especificando um SSOO para gerenciar conferências. Uma conferência possui diversos recursos cuja alocação precisa ser agendada. Ou seja, para certo evento em uma conferência, diversos recursos são alocados. Os recursos são de diversas naturezas: funcionários, equipamentos, salas de apresentação.

Suponha que você é esse projetista. Que solução de projeto poderia ser definida para esse problema?



8-17: Analise as relações existentes entre o conceito de interface e os princípios de

polimorfismo e de encapsulamento da orientação a objetos.

8-18: Defina as diferenças e semelhanças existentes entre uma classe abstrata e uma interface. Qual é a diferença entre uma classe concreta que herda de uma classe abstrata e a realização de uma interface.

Modelagem de estados

Todos os adultos um dia foram crianças, mas poucos se lembram disso.
— ANTOINE DE SAINT-EXUPÉRY, O PEQUENO PRÍNCIPE

Objetos do mundo real se encontram em estados particulares a cada momento. Por exemplo, uma jarra está *cheia* de líquido; uma pessoa está *cansada*. Da mesma forma, cada objeto participante de um sistema de software orientado a objetos se encontra em um estado particular. Um objeto muda de estado quando acontece algum *evento* interno ou externo ao sistema. Quando um objeto muda de um estado para outro, diz-se que ele realizou uma *transição* entre estados. Os estados e as transições de estado de um objeto constituem o seu *ciclo de vida*. No momento de sua transição de um estado para outro, um objeto normalmente realiza determinadas *ações* dentro do sistema. Cada objeto pode passar por um número finito de estados durante a sua vida. Quando um objeto transita de um estado para outro, significa que o sistema no qual ele está inserido também está mudando de estado.

Pela análise das *transições* entre *estados* dos objetos de um sistema de software, é possível prever todas as possíveis *operações* realizadas, em função de *eventos* que podem ocorrer. O diagrama da UML utilizado para realizar essa análise é o *diagrama de transição de estado* (DTE). Esse diagrama permite descrever o ciclo de vida de objetos de uma classe, os eventos que causam a transição de um estado para outro e a realização de operações resultantes.

A notação e a semântica iniciais do DTE foram propostas por David Harel (1987) em seu trabalho com máquinas de estados finitos (*finite state machines*). A ideia básica de Harel foi definir uma máquina com uma quantidade fixa de estados (daí o nome máquina de estados finitos). A máquina pode receber eventos, e cada evento pode ocasionar a transição de um estado para outro. Posteriormente, Jim Rumbaugh (um dos “três amigos”) e outros (Mealy e Moore) estenderam a proposta inicial de Harel com várias outras notações e o diagrama de transições de estado foi incorporado à UML.

Na modelagem de sistemas orientados a objetos, a *modelagem dinâmica* descreve o comportamento dinâmico dos objetos durante a execução do sistema. No [Capítulo 7](#), descrevemos dois diagramas da UML para realizar modelagem dinâmica. Utiliza-se também o DTE para obter uma visão dinâmica do sistema. No entanto, diferentemente dos diagramas de interação (que descrevem o comportamento de objetos de classes diferentes), um DTE descreve o comportamento dos objetos de uma única classe. Os diagramas de transição de estado não são definidos para todas as classes de um sistema, mas apenas para aquelas que possuem um número definido de estados conhecidos, e quando o comportamento das classes de objetos é afetado e modificado pelos diferentes estados. Nessas classes, um DTE pode ser utilizado para enfatizar os eventos que resultam em mudanças de estado.

Neste Capítulo apresentamos detalhes da modelagem de estados dos objetos de um SSOO. Na [Seção 9.1](#), apresentamos os elementos de notação do diagrama de transição de estados especificados na UML. Na [Seção 9.2](#), descrevemos técnicas para identificação dos elementos de um diagrama de

estados. Na [Seção 9.3](#), apresentamos um procedimento para construção dos DTEs. Na [Seção 9.4](#), apresentamos de que forma a modelagem de estados se insere em um processo de desenvolvimento. Finalmente, na [Seção 9.5](#), damos continuidade à modelagem de nosso estudo de caso, o SCA.

9.1 Diagrama de transição de estado

A UML tem um conjunto rico de notações para desenhar um DTE. Alguns elementos básicos de um diagrama de transição de estados são os *estados* e as *transições*. Associados a estas últimas estão os conceitos de *evento*, *ação* e *atividade*. Um DTE pode também conter elementos menos utilizados, mas às vezes úteis, como *transições internas*, *estados aninhados* e *estados concorrentes*. A seguir, a notação e a semântica básicas dos diagramas de estado são descritas. Para essa descrição, o DTE da [Figura 9-1](#) é utilizado como exemplo. Esta figura ilustra um DTE para a classe ContaBancária.

9.1.1 Estados

Um *estado* é uma situação na vida de um objeto durante a qual ele satisfaz alguma condição ou realiza alguma atividade. Cada estado de um objeto é normalmente determinado pelos valores dos seus atributos e (ou) pelas suas ligações com outros objetos. Por exemplo: “O atributo *reservado* deste objeto livro tem valor *verdadeiro*”. Outro exemplo: “Uma conta bancária passa para o *vermelho* quando o seu saldo fica *negativo*”.

A notação UML para representar um estado é um retângulo com as bordas arredondadas. Em sua forma mais simples, o retângulo de um estado possui um único compartimento (conforme ilustrado na [Figura 9-2](#)). O estado inicial é representado como um círculo preenchido e indica o estado de um objeto quando ele é criado. Só pode haver um estado inicial em um DTE. Essa restrição serve para definir a partir de que ponto um DTE deve começar a ser lido.¹ O estado final é representado como um círculo “eclipsado” (ver [Figura 9-2](#)) e indica o fim do ciclo de vida de um objeto. Este estado é opcional e pode haver mais de um estado final em um DTE.

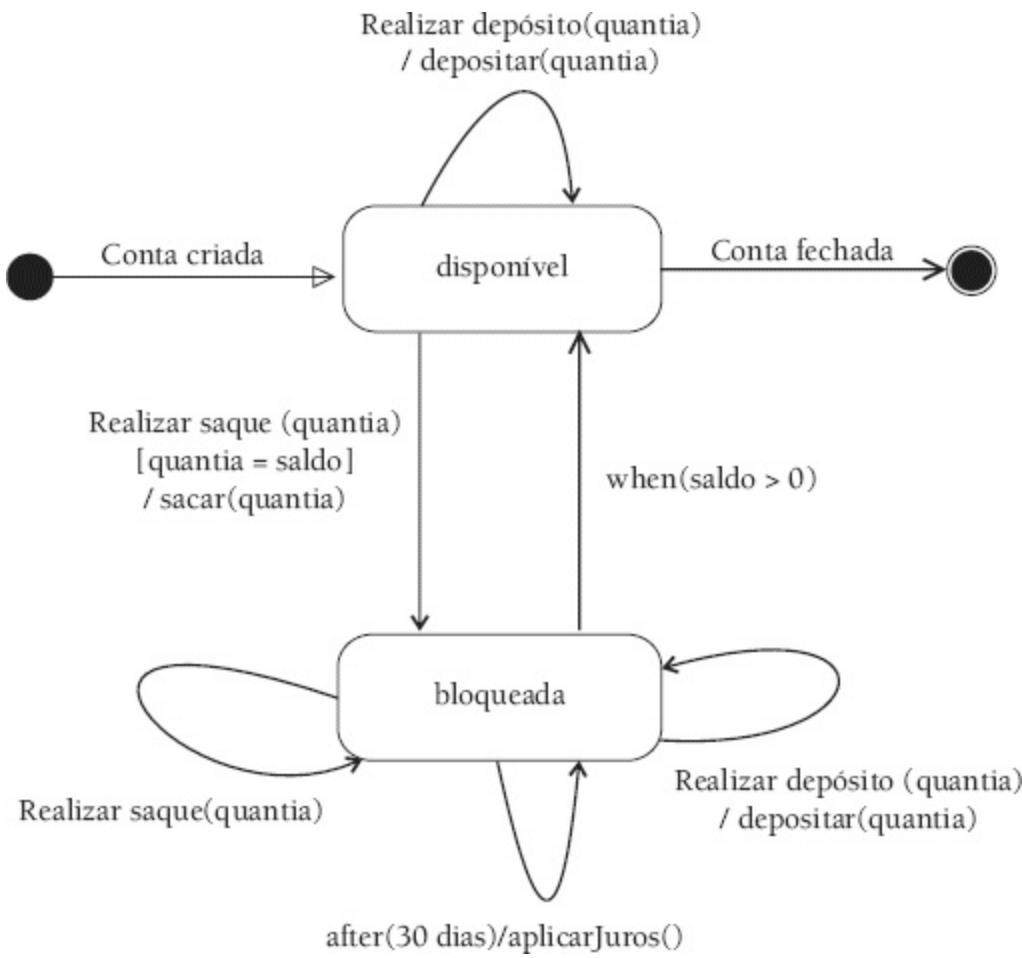


Figura 9-1: Exemplo de DTE para a classe ContaBancária.



Figura 9-2: Elementos gráficos básicos de um DTE.

9.1.2 Transições

Os estados estão associados a outros pelas *transições*. Uma transição é mostrada como uma linha conectando estados, com uma seta apontando para um dos estados. Quando ocorre uma transição entre estados, diz-se que a transição foi *disparada*. Note que, em uma transição, o estado subsequente pode ser igual ao estado original. Uma transição pode ser rotulada com uma expressão. A seguir, é apresentada a forma geral dessa expressão. Seus detalhes são descritos nas seções a seguir.

evento (lista-parâmetros) [guarda] / ação

9.1.3 Eventos

Uma transição possui um *evento* associado. Um evento é algo que acontece em algum ponto no tempo e que *pode* modificar o estado de um objeto. Alguns exemplos de eventos são listados a seguir:

Tabela 9-1: Exemplos de nomes de eventos em sistemas de software e em processos de negócio

Em sistemas de software	Em processos de negócio
1 Mouse pressionado	1 Pedido realizado
2 Disco inserido	2 Fatura paga 3 Cheque devolvido

Um evento pode conter uma *lista de parâmetros* (veja o elemento lista-parâmetros na sintaxe anterior), que são passados para fornecer informações úteis ao objeto receptor de evento. A lista de parâmetros é especificada entre os parâmetros. Por exemplo, na expressão Mouse Pressionado (local), o parâmetro local indica em que posição da tela o mouse foi pressionado.

Os eventos relevantes a uma classe de objetos podem ser classificados em três tipos. Estes tipos são descritos a seguir.

1. **Evento de chamada:** *recebimento de uma mensagem de outro objeto*. Pode-se pensar neste tipo de evento como uma solicitação de serviço de um objeto a outro.
2. **Evento de sinal:** *recebimento de um sinal*. Este é um tipo especial do evento anterior. Neste evento, o objeto recebe um sinal de outro objeto que pode fazê-lo mudar de estado. A diferença básica entre o evento de sinal e o evento de chamada é que neste último o objeto que envia a mensagem fica esperando a execução da mesma. No evento de sinal, o objeto remetente continua o seu processamento após ter enviado o sinal. O evento de sinal raramente é utilizado.
3. **Evento temporal:** *passagem de um intervalo de tempo predefinido*. Em vez de receber uma mensagem específica, um objeto pode interpretar a passagem de um certo intervalo de tempo como sendo um evento.² Um evento temporal é especificado com a cláusula *after* seguida de um parâmetro que especifica um intervalo de tempo. Por exemplo, a expressão *after(30 segundos)* indica que a transição correspondente será disparada trinta segundos após o objeto ter entrado no estado atual.
4. **Evento de mudança:** *uma condição que se torna verdadeira*. O fato de determinada condição se tornar verdadeira também pode ser visto como um evento. Este evento é representado por uma expressão de valor lógico (*verdadeiro* ou *falso*) e é especificado utilizando-se a cláusula *when*. Por exemplo, a expressão *when(saldo > 0)* significa que a transição é disparada quando o valor do atributo *saldo* for positivo (ver [Figura 9-1](#)). Eventos temporais (descritos no item anterior) também podem ser definidos utilizando-se a cláusula *when*. Alguns exemplos: *when(data = 13/07/2002)*, *when(horário = 00:00h)* etc.

Considere a [Figura 9-1](#). Há uma *transição reflexiva* (ou seja, uma transição que sai de um estado e entra no mesmo estado) quando a conta bancária está bloqueada. Essa transição possui um evento temporal, indicando que trinta dias após a conta ter entrado nesse estado, a operação *aplicarJuros()* é executada. A [Figura 9-1](#) também apresenta um exemplo de evento de mudança na transição de *bloqueada* para *disponível*. Esse evento passa um parâmetro (*saldo > 0*) correspondente a uma expressão condicional. Quando essa expressão for verdadeira, a transição é disparada.

A ocorrência de um evento relevante pode ocasionar outro evento. A [Figura 9-3](#) ilustra esquematicamente dois diagramas de estados. O primeiro exibe a forma normal de representação de um evento: quando Evento ocorre, o objeto passa de Estado 1 para Estado 2. A mesma transição acontece no segundo DTE. Entretanto, além da transição de estados, o evento OutroEvento (relevante

a *objetoAlvo*) é disparado.

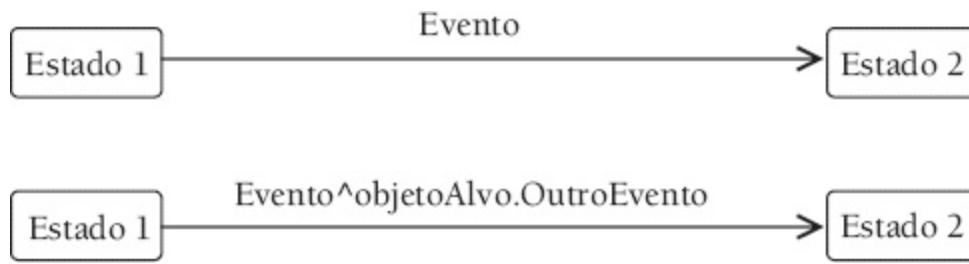


Figura 9-3: Ocorrência de evento ativando outro evento.

9.1.4 Condição de guarda

Uma transição pode ter também uma *condição de guarda*. Uma *condição de guarda*, ou simplesmente *guarda*, é uma expressão de valor lógico, da mesma forma que a utilizada para diagramas de interação (ver [Seção 7.1.1.2.2](#)). A condição de guarda de uma transição pode ser definida utilizando-se parâmetros passados no evento e também atributos e referências a ligações da classe em questão. Além disso, uma condição também pode testar o valor de um estado.

Uma transição na qual foi definida uma condição de guarda é disparada somente se o evento associado ocorre e a condição de guarda é verdadeira. Se uma transição não tiver uma condição de guarda, ela sempre será disparada quando o evento ocorrer. É importante não confundir a condição de guarda de uma transição com um evento de mudança (que também é definido com uma expressão de valor lógico). A expressão condicional de uma condição de guarda é sempre apresentada entre colchetes. Ao contrário, a expressão condicional nos eventos de mudança são parâmetros da cláusula *when*.

Como exemplo, a [Figura 9-1](#) ilustra uma transição entre os estados *disponível* e *bloqueada*. Essa expressão possui uma condição de guarda, *[quantia = saldo]*, onde *quantia* é um parâmetro recebido e *saldo* é um atributo da classe *ContaBancária*. Quando o evento *Realizar saque* ocorre, a transição somente ocorrerá se a condição de guarda for verdadeira. Se não for o caso, o evento é ignorado pelo objeto.

9.1.5 Ações

Ao transitar de um estado para outro, um objeto pode realizar uma ou mais ações. Uma ação é uma expressão que pode ser definida em termo dos atributos, das operações ou das associações da classe. Os parâmetros do evento também podem ser utilizados. Uma ação pode corresponder igualmente à execução de uma operação. A ação é representada na linha da transição e deve ser precedida por uma barra inclinada para a direita (símbolo “/”). Note que a ação associada a uma transição é executada somente se a transição for disparada. Uma ação pode gerar outros eventos. Esses eventos podem ser relevantes para o próprio objeto ou para outros.

Pode haver ações especiais que são executadas sempre que um objeto passa para um determinado estado (não importa por qual transição) ou quando ele sai de um estado (também não importando por qual transição). Essas ações especiais são descritas na [Seção 9.1.8](#).

9.1.6 Atividades

Semelhante a uma ação, uma atividade é algo que é executado pelo objeto. No entanto, uma atividade pode ser interrompida (considera-se que o tempo de execução de uma ação é tão insignificante que

ela não pode ser interrompida). Por exemplo, enquanto a atividade estiver em execução, pode acontecer um evento que a interrompa e cause uma mudança de estado. Outra diferença entre ação e atividade é que uma atividade sempre está associada a um estado (ao contrário, uma ação está associada a uma transição). A forma de declarar atividades é descrita na [Seção 9.1.8](#).

9.1.7 Ponto de junção

Em algumas situações, o próximo estado de um objeto varia de acordo com o valor da condição de guarda. Se o valor da condição de guarda for verdadeiro, o objeto vai para um estado E1; se for falso, o objeto vai para outro estado E2. É como se a transição tivesse bifurcações, e cada transição de saída da bifurcação tivesse uma condição de guarda. Essa situação pode ser representada em um DTE por um *ponto de junção*.

Um ponto de junção é desenhado como um losango³ em que chegam uma ou mais transições (provenientes de estados diferentes) e de onde partem uma ou mais transições. A cada transição de saída do ponto de junção está associada uma condição de guarda. A transição que o objeto segue é aquela para a qual a condição de guarda é verdadeira. A [Figura 9-4](#) apresenta de forma esquemática um fragmento de DTE que ilustra a utilização de um ponto de junção. Desse ponto, partem quatro transições. Cada uma dessas transições está associada a uma condição de guarda e a uma ação. As ações são opcionais. É importante enfatizar que, quando uma das condições é verdadeira, todas as demais são falsas.

Pontos de junção permitem que duas ou mais transições compartilhem uma “trajetória de transições”. Na [Figura 9-4](#), por exemplo, as transições que saem dos estados E0 e E1 compartilham o mesmo caminho de transições a partir do ponto de junção. De uma forma geral, pode haver um número ilimitado de transições saindo de um ponto de junção. Além disso, nada impede que o estado no qual chega uma transição que saiu de um ponto de junção seja o mesmo do qual saiu uma transição que chegou a esse ponto de junção. Pode haver também uma transição de saída que esteja rotulada com a cláusula `else`. Isso significa que, se todas as outras condições de guarda forem falsas, a transição correspondente à cláusula `else` será disparada.

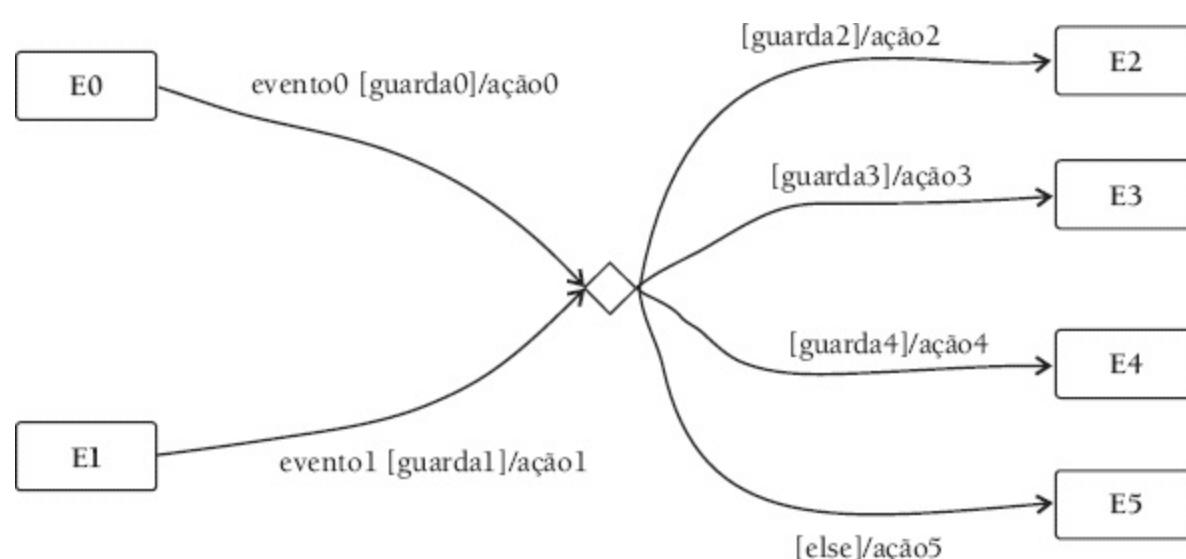


Figura 9-4: Utilização de pontos de junção em um DTE.

9.1.8 Cláusulas entry, exit e do

No comportamento adicional de um retângulo de estado, podem-se especificar ações ou atividades a

serem executadas.

1. A cláusula entry pode ser usada para especificar uma ação a ser realizada no momento em que o objeto entra em um estado. A ação dessa cláusula é sempre executada, independentemente do estado do qual o objeto veio (é como se a ação especificada estivesse associada a todas as transições de entrada no estado).
2. A cláusula exit serve para declarar ações que são executadas sempre que o objeto sai de um estado. Da mesma forma que a cláusula entry, sua ação é sempre executada, independentemente do estado para o qual o objeto vai (é como se a ação especificada estivesse associada a todas as transições de saída do estado).
3. A cláusula do serve para definir alguma atividade a ser executada quando o objeto passa para um determinado estado. (Note a diferença em relação à cláusula entry, que serve para especificar uma ação em vez de uma atividade.)

As cláusulas predefinidas (entry, exit, do) são especificadas no interior do retângulo do estado e têm a seguinte sintaxe:

evento / [ação | atividade]

A [Figura 9-5](#) ilustra de forma esquemática um DTE no qual foi especificada a cláusula do. Note que a inexistência de um evento na transição de Estado 2 para Estado 3 indica que, assim que a atividade2 for finalizada, a transição ocorrerá (ela é automática). Por outro lado, a transição de Estado 3 para Estado 1 pode ocorrer antes de a atividade3 terminar, bastando para isso que o Evento B ocorra antes do término da atividade.

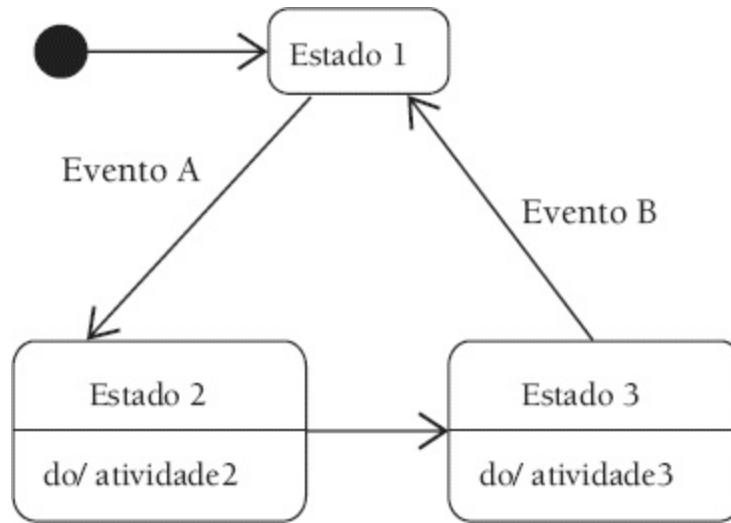


Figura 9-5: Exemplo de utilização da cláusula do.

A [Figura 9-6](#), adaptada da especificação da UML (OMG, 2001), ilustra um estado no qual são declaradas as cláusulas entry e exit. A leitura desse estado indica que a ação definirEco(cInvisivel) é executada sempre que o objeto (provavelmente um objeto de fronteira) entra nesse estado. O mesmo acontece com a ação definirEco(cVisivel) quando o objeto sai do estado.

```

entry/ definirEco(cInvisível)
caractere(c)/ tratarCaractere(c)
ajuda/ exibirAjuda(invisível)
exit/ definirEco(cVisível)

```

Figura 9-6: Exemplo de estado no qual foram definidas as cláusulas *entry*, *exit* e transições internas.

A [Figura 9-7](#) é outro exemplo que ilustra o uso das ações *entry* e *exit*. O diagrama de estados apresentado representa os estados de uma lâmpada (*ligada* ou *desligada*) e os eventos relevantes (*ligar* e *desligar*). À direita desta figura, é exibida a ordem de execução das operações quando a lâmpada está *ligada*, e os seguintes eventos ocorrem sucessivamente: *desligar*, *desligar*, *ligar*.

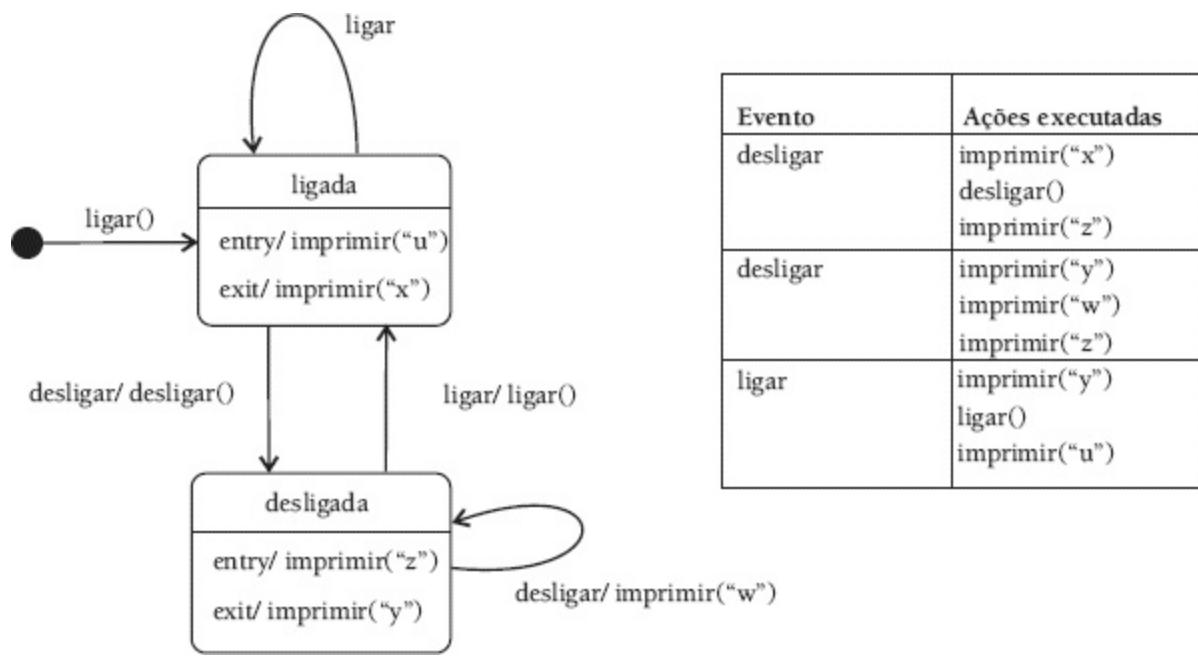


Figura 9-7: Utilização das cláusulas *entry* e *exit*.

9.1.9 Transições internas

Além das ações predefinidas *entry* e *exit*, zero ou mais *transições internas* também podem ser especificadas. Uma transição interna é uma transição que não faz o objeto mudar de estado. O objetivo das transições internas é fazer com que uma ação ou atividade seja executada sem que uma mudança de estados ocorra. A sintaxe de uma transição interna é a mesma utilizada para cláusulas *entry*, *exit* e *do*. Como exemplo, considere novamente a [Figura 9-6](#). No DTE dessa figura há duas transições internas: *caractere* e *ajuda*. Quando algum dos eventos associados a essas transições ocorre, a ação correspondente é executada, mas o objeto permanece no estado *Digitando senha*.

De forma geral, a declaração das cláusulas *entry*, *exit* e *do* e das transições internas pode ocorrer em qualquer ordem dentro do retângulo do estado. No entanto, recomenda-se utilizar a seguinte ordem de declaração: cláusula *entry*, cláusula *do*, transições internas e cláusula *exit*.

Um ponto importante é a ordem na qual as ações predefinidas, e as transições internas são executadas. Ações de saída (*exit*) precedem ações externas associadas a uma transição para fora do estado. Ações de entrada (*entry*) sucedem ações externas associadas a uma transição para o estado. Em uma transição reflexiva, a ordem de execução é a seguinte: ação *exit*, ação da transição reflexiva,

ação entry e, por fim, as demais ações ou atividades.

9.1.10 Exemplo

Como um exemplo mais completo dos elementos básicos de um DTE aqui descritos, considere a [Figura 9-8](#), um DTE para um relógio despertador, que apresenta um exemplo de utilização da cláusula *do*. Esse despertador funciona da seguinte forma: quando o evento Armar alarme ocorre, o parâmetro horário do evento informa o horário no qual o despertador deve tocar o alarme. O despertador fica esperando até que esse horário chegue. Isso ocorre quando a expressão de valor lógico definida no evento temporal `when(horário Definido = agora())` se torna verdadeira.⁴ A ocorrência desse evento acarreta a transição de esperando para tocando alarme. O estado tocando alarme possui uma cláusula *entry*, na qual a operação `tocarAlarme` é executada. Essa operação toca o alarme por dez segundos. Agora, note a transição de tocando alarme para esperando. Essa transição não possui nem evento nem condição de guarda. De fato, essa transição é automaticamente disparada quando a ação especificada na cláusula *entry* é finalizada. Isso serve para exemplificar que, de uma forma geral, uma associação que não possui um evento associado é disparada logo após as ações internas ao estado tiverem sido executadas. A ação associada a essa transição é executada, fazendo com que o novo horário de alarme seja definido (o alarme toca durante dez segundos a cada cinco minutos).⁵

Note que o estado esperando possui uma cláusula *entry* que incrementa um contador (*n* é um atributo privativo da classe Despertador). Esse contador serve para definir quantas vezes o alarme será tocado. Neste exemplo, o alarme será tocado três vezes se não for desarmado antes disso. A qualquer momento pode ocorrer um evento Desarmar alarme, que faz com que o alarme seja desarmado.

9.1.11 Estados aninhados

Podem existir estados aninhados dentro de outro estado. Um estado que contém diversos outros é dito *composto*. Todos os estados dentro de um estado composto herdam qualquer transição deste último. Isso significa que o uso de estados compostos geralmente torna um DTE mais legível.

A [Figura 9-9](#) apresenta um exemplo de DTE com um estado composto, armado. Esse estado possui dois estados aninhados, esperando e tocando alarme. Compare com o DTE equivalente, porém menos legível, da [Figura 9-8](#). Quando ocorre a transição de desarmado para armado, o estado aninhado esperando é ativado. A qualquer momento em que ocorrer o evento de chamada Desarmar alarme ou o evento temporal `when(n>3)`, não importando em que estado aninhado o objeto esteja, o novo estado será desarmado.

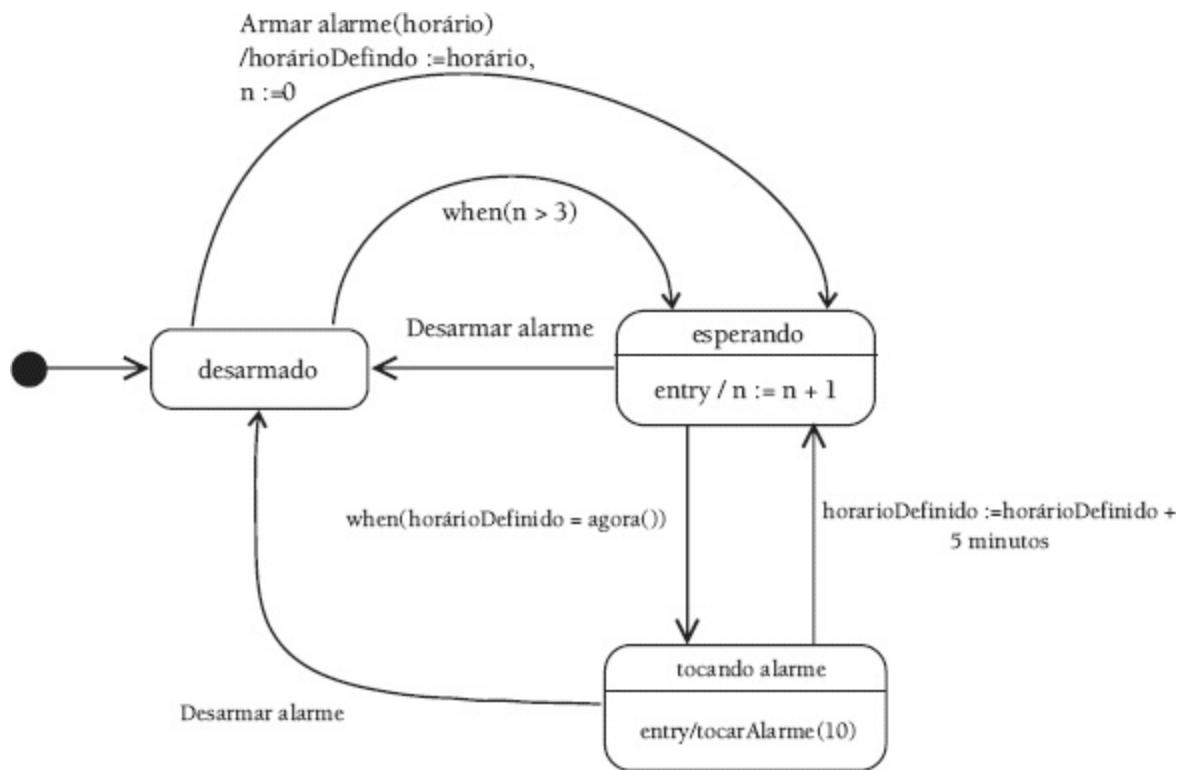


Figura 9-8: Diagrama de estados para a classe *Despertador*.

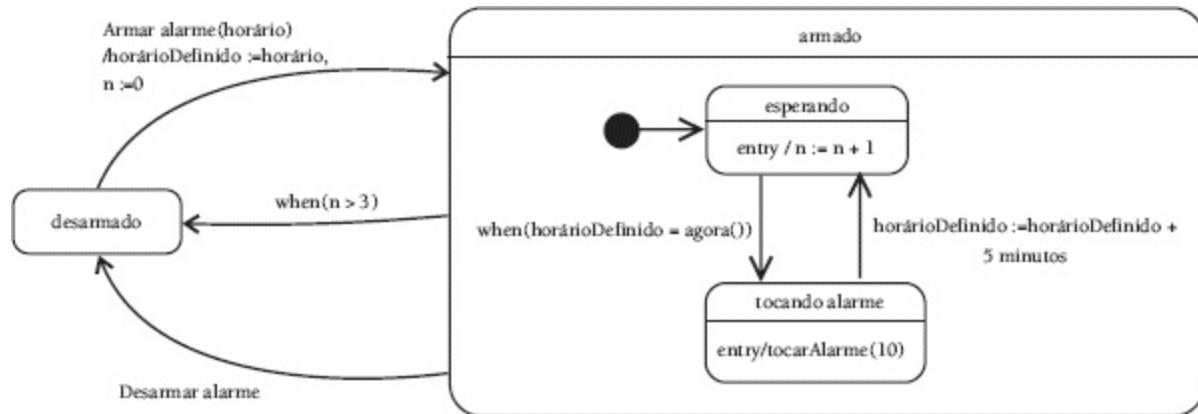


Figura 9-9: Exemplo de DTE com um estado composto (*armado*).

9.1.12 Estados concorrentes

Um estado concorrente é um tipo especial de estado composto. Um objeto em um estado concorrente pode, na verdade, se encontrar em dois ou mais estados independentes.

O diagrama de estados da [Figura 9-10](#) ilustra dois conjuntos de estados independentes da classe Refrigerador. Um objeto dessa classe pode estar com a porta aberta ou fechada. Independentemente disso, esse objeto pode estar com o motor ligado ou desligado. O estado composto torna mais simples a representação desses dois conjuntos de estados. Sem um estado composto, haveria um total de transições igual à cardinalidade do produto cartesiano dos dois conjuntos de estados.

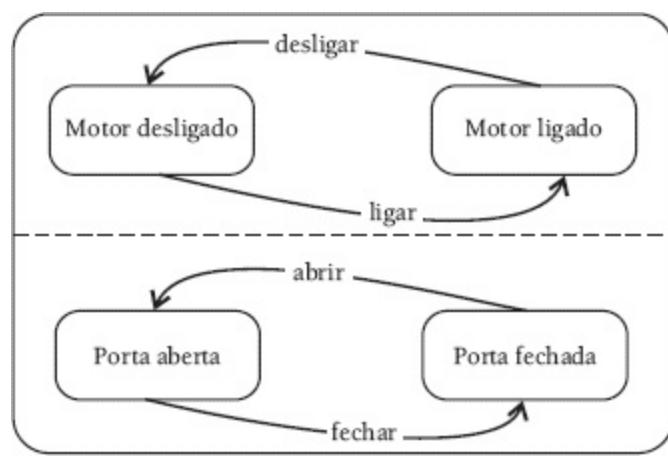


Figura 9-10: Estado composto para a classe Refrigerador.

9.2 Identificação dos elementos de um diagrama de estados

Os estados podem ser vistos como uma abstração dos atributos e associações de um objeto. A seguir temos alguns exemplos (os nomes em itálico são possíveis estados).

- Um professor está *licenciado* quando não está ministrando curso algum durante o semestre.
- Um tanque está *na reserva* quando o valor do nível de combustível está abaixo de 20%.
- Um pedido está *atendido* quando todos os seus itens estão atendidos.

Um bom ponto de partida para identificar estados de um objeto é analisar os possíveis valores de seus atributos e as ligações que ele pode realizar com outros objetos. No entanto, a existência de atributos ou ligações não é suficiente para justificar a criação de um DTE para uma classe. O comportamento de objetos dessa classe deve depender de tais atributos ou ligações.

Em relação a transições, devemos identificar os eventos que podem gerá-las. Além disso, é preciso examinar também se há algum fator que condicione o disparo da transição. Se existir, esse fator deve ser modelado como uma condição de guarda da transição. Já que as transições dependem de eventos para ocorrer, devem-se identificar esses eventos primeiramente. Um bom ponto de partida é a descrição dos casos de uso.

Os eventos encontrados na descrição dos casos de uso são externos ao sistema. Contudo, uma transição pode também ser disparada por um evento *interno* ao sistema (ver [Seção 9.1.2](#)). Para identificar os eventos internos relevantes a um objeto, analise os diagramas de interação. Esses diagramas contêm mensagens sendo trocadas entre objetos, e mensagens nada mais são do que solicitações de um objeto a outro. Essas solicitações podem ser vistas como eventos. De uma forma geral, cada operação com visibilidade pública de uma classe pode ser vista como um evento em potencial.

Outra fonte para identificação de eventos relevantes é analisar as regras de negócio definidas para o sistema (ver [Seção 4.5.1](#)). Normalmente essas regras possuem informações sobre condições limites que permitem identificar estados e transições. Por exemplo, vamos analisar alguns exemplos de regras de negócio.

- “Um cliente do banco não pode retirar mais de R\$ 1.000 por dia de sua conta”. Há potencialmente dois estados para uma conta bancária: *disponível* e *bloqueada*. Além disso, a transição de um estado para outro depende do atributo *saldo* (p. ex.) da conta bancária.

- “Os pedidos para um cliente não especial devem ser pagos antecipadamente”.

O cliente pode estar no estado de cliente especial e cliente comum.

- “*O número máximo de alunos por curso é igual a 30*”. Se houver uma classe *Curso* e uma classe *Aluno* no diagrama de classes, essa regra de negócio indica que os estados (*aberto* ou *fechado*) de um objeto *Curso* dependem da quantidade de ligações desse objeto com objetos de *Aluno*.

9.3 Construção de diagramas de transições de estados

Para sistemas bastante simples, a definição dos estados de todos os objetos não é tão trabalhosa. No entanto, a quantidade de estados possíveis de todos os objetos de um sistema complexo é grande. Isso torna a construção de um DTE para o sistema inteiro uma tarefa bastante complexa.

Por exemplo, considere, sem perda de generalidade, que todos os objetos de um sistema têm o mesmo número de estados: três estados cada um. Para um sistema de dois objetos, a quantidade de estados possíveis do sistema é 9 (3×3). Para um sistema de 4 objetos, a quantidade de estados possíveis do sistema passa para 81 ($3 \times 3 \times 3 \times 3$). Ou seja, o crescimento dos estados de um sistema se dá exponencialmente com o número de objetos, o que torna impraticável a construção de um único diagrama para todo o sistema.

Para resolver o problema da explosão exponencial de estados, os diagramas de estados são desenhados por classe. Geralmente, cada classe é simples o suficiente para que o diagrama de estados correspondente seja compreensível. Note, contudo, que essa solução de dividir a modelagem de estados por classes do sistema tem a desvantagem de dificultar a visualização do estado do sistema como um todo. Essa desvantagem é parcialmente compensada pela construção de diagramas de interação ([Capítulo 7](#)).

Nem todas as classes de um sistema precisam de um DTE. Um diagrama de estados é desenhado somente para classes que exibem um comportamento dinâmico relevante. A relevância do comportamento de uma classe depende muito de cada situação em particular. No entanto, objetos cujo histórico precisa ser rastreado pelo sistema são típicos para se construir um diagrama de estados. Por exemplo, considere um objeto *Pedido*. Esse objeto é criado quando o cliente realiza um pedido. O crédito deve ser aprovado para que o pedido seja *aceito*. Se o crédito é negado, o pedido é retornado ao cliente para ser modificado. Se é aceito, o pedido é *confirmado*. Após ser atendido (ou seja, todos os itens desse pedido foram providenciados), esse pedido é enviado ao sistema de distribuição para ser *entregue*. Depois disso, por alguma razão, o pedido pode ser *devolvido* pelo cliente. Esse histórico de estados de um pedido é relevante para o sistema (p. ex., só os pedidos atendidos podem ser entregues aos clientes). Esse é um caso típico de classe cuja definição de um diagrama de estados ajudaria a esclarecer seu comportamento.

Um objeto de entidade pode perdurar durante várias execuções do sistema. Consequentemente, esse objeto pode participar da realização de diversos casos de uso do sistema. Diagramas de estados são úteis para capturar o comportamento de *um* objeto de entidade durante *vários* casos de uso. Entretanto, note que não apenas as classes de entidade são candidatas a precisar de um DTE. As classes de controle e da fronteira (ver [Seção 5.4.2](#)) frequentemente necessitam de diagramas de estados para melhor esclarecer o seu comportamento.

Uma vez selecionadas as classes para cada uma das quais se deve construir um diagrama de estados, acompanhe essa sequência de passos:

1. Identifique os estados relevantes para a classe.
2. Identifique os eventos relevantes aos objetos de uma classe. Para cada evento, identifique qual transição que ele ocasiona.
3. Para cada estado, identifique as transições possíveis quando um evento relevante ocorre.
4. Para cada estado, identifique os eventos internos e as ações correspondentes relevantes.
5. Para cada transição, verifique se há fatores que influenciam o seu disparo. Se esse for o caso, uma condição de guarda deve ser definida. Verifique também se alguma ação deve ser executada quando uma transição é disparada (ações).
6. Para cada condição de guarda e para cada ação, identifique os atributos e as ligações que estão envolvidos. Pode ser que esses atributos ou ligações ainda não existam. Nesse caso, eles devem ser adicionados ao modelo de classes.
7. Defina o estado inicial e os eventuais estados finais.
8. Desenhe o diagrama de estados. Procure posicionar os estados de tal forma que o ciclo de vida do objeto possa ser visualizado de cima para baixo e da esquerda para a direita.

É importante notar que a construção de diagramas de estados de um sistema frequentemente leva à descoberta de novos atributos para uma classe, principalmente atributos que servem de abstrações para estados. Além disso, esse processo de construção permite identificar novas operações na classe, pois os objetos precisam reagir aos eventos que eles recebem, e essas reações são realizadas mediante operações.

9.4 Modelagem de estados no processo de desenvolvimento

A modelagem de estados é uma atividade que deve começar na fase de análise, em paralelo à construção nos diagramas de interação e nos diagramas de classes. Utilizando informações geradas nessa construção, os diagramas de estados podem ser construídos para as classes cujos estados são considerados relevantes.

O diagrama de classes fornece informações para a construção do diagrama de estados (ver [Seção 9.2](#)), pois é no primeiro que as classes com estados relevantes são identificadas. Por outro lado, durante a construção do diagrama de estados para uma classe, novos atributos e novas operações podem surgir. Essas novas propriedades devem ser adicionadas ao modelo de classes. Dessa forma, assim como vimos que é o caso da modelagem de interações, a modelagem de estados é uma atividade que tem a característica de retroalimentar as atividades de modelagem das quais obtém informações.

Deve-se notar também que o comportamento de um objeto varia em função do estado no qual ele se encontra. Em vista disso, pode haver a necessidade de atualizar a descrição de um ou mais métodos da classe desse objeto para indicar como as operações correspondentes se comportam em função de cada estado. Por exemplo, o comportamento da operação `sacar()` da classe ContaBancária ([Figura 9-1](#)) varia em função do estado no qual essa classe se encontra (saques não podem ser realizados em uma conta bloqueada).

Outra característica representada em um DTE e que resulta em alterações no modelo de classes é o *estado inicial* do objeto, porque o valor do estado inicial normalmente é identificado durante a modelagem dos estados da classe. O método de instanciação (construtor) da classe em questão deve definir o estado inicial de um objeto. Normalmente, informações para definição do estado inicial de um objeto são passadas como argumentos para o construtor. Por exemplo, a [Figura 9-11](#) ilustra um fragmento de declaração da classe Lâmpada em linguagem Java. Essa classe apresenta um método de instanciação que define o valor do atributo estado, responsável por manter a informação do estado atual do objeto.

```
public class Lâmpada
{
    .....
    private String estado;
    public Lâmpada () { estado = "desligada";}
    .....
}
```

Figura 9-11: O estado inicial é definido na operação de instanciação.

Também é importante notar a diferença de objetivo com o uso do diagrama de interações e do diagrama de transições de estados. Enquanto um diagrama de estados representa o comportamento de *um objeto*, os diagramas de interação (estudados no [Capítulo 7](#)) são utilizados para capturar o comportamento de *vários objetos* em um (cenário de) caso de uso. No entanto, para capturar o comportamento de vários objetos em casos de uso diferentes, a melhor alternativa são os diagramas de atividade, descritos no [Capítulo 10](#).

9.5 Estudo de caso

Esta seção continua a modelagem do Sistema de Controle Acadêmico. Das classes desse sistema, uma com certeza precisa de um DTE: Turma. Foram identificados os seguintes estados para essa classe:

Aberta	A turma está aberta para receber inscrições de alunos, até a sua quantidade máxima. O professor, as salas e os horários foram alocados.
Lotada	A turma alcançou sua capacidade máxima em relação à quantidade de alunos inscritos.
Fechada	A turma está totalmente definida (os alunos estão inscritos).
Cancelada	A turma é cancelada. O evento de cancelamento pode acontecer a qualquer momento do ciclo de vida de uma turma.

Em relação ao estado Lotada, o leitor deve se lembrar de que há uma regra de negócio do sistema que define a quantidade máxima de alunos. Essa regra tem identificador RN02, e sua declaração se encontra na [Seção 4.7.2](#).

Os eventos relevantes para objetos da classe Turma são a inscrição de um aluno, assim como as operações que realizam abertura, cancelamento e fechamento de uma turma. Além disso, o atributo

que armazena a quantidade atual de alunos inscritos deve ser definido para monitorar quando essa quantidade atingir o valor máximo.

O DTE para a classe Turma está ilustrado na [Figura 9-2](#).

O coordenador pode fechar a turma sem que a quantidade máxima de inserção tenha sido atingida.

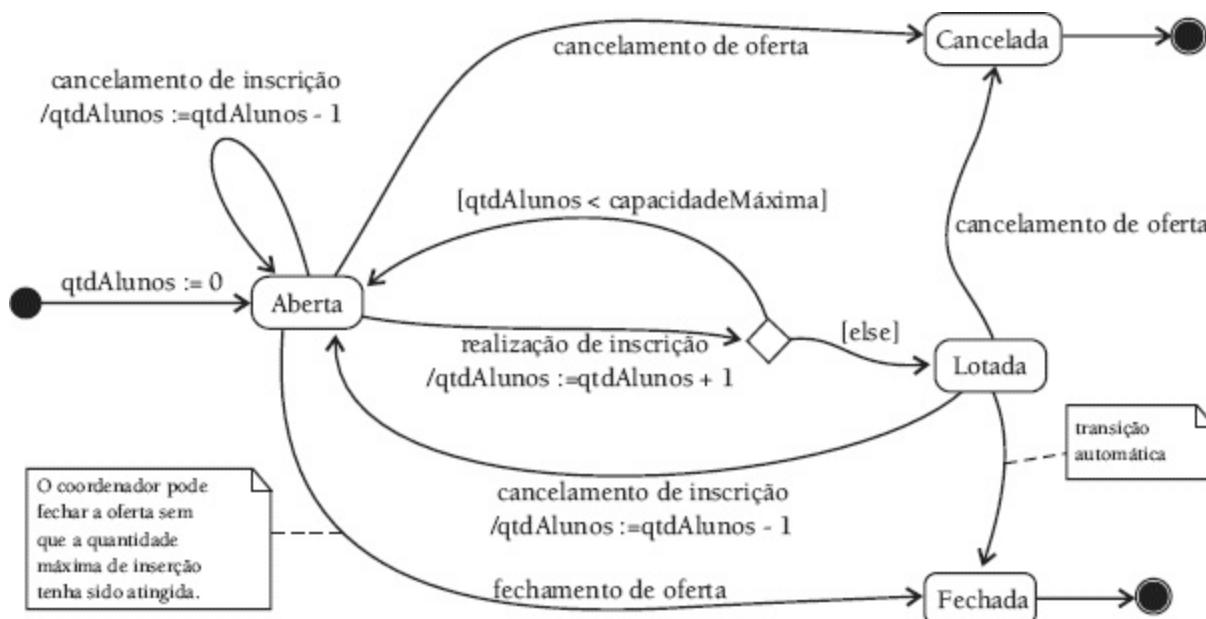


Figura 9-12: DTE para a classe Turma.

Conforme descrito na [Seção 9.4](#), a construção de um DTE para uma classe pode resultar na definição de novas propriedades nessa classe. De fato, a classe Turma deve ser atualizada com as informações obtidas na construção de seu DTE. A [Figura 9-13](#) ilustra a classe atualizada com base nas informações obtidas a partir de seu DTE.

Turma
-ano : Integer -semestre : Integer -qtdAlunos : Integer -capacidadeMáxima : Integer -status
+cancelar() +fechar() +abrir() -lotar()

Figura 9-13: Definição atualizada da classe Turma, após a construção de seu DTE.

► EXERCÍCIOS

9-1: Descreva a posição do diagrama de estados no processo de desenvolvimento incremental e iterativo. Quando são utilizados e para quê?

9-2: Modele por meio de um diagrama de estados a seguinte situação: em uma

máquina de encher garrafas de refrigerante passam diversas garrafas. Uma garrafa entra inicialmente vazia no equipamento. A partir de um determinado momento, ela começa a ser preenchida com refrigerante. Ela permanece nesse estado (sendo preenchida) até que, eventualmente, esteja cheia de refrigerante. Nesse momento, o equipamento sela a garrafa com uma tampinha, e assim a garrafa passa para o estado de “lacrada”. Uma garrafa vazia não deve ser lacrada pelo equipamento. Além disso, uma garrafa cheia de refrigerante não deve receber mais esse líquido.

9-3: Construa um diagrama de estados considerando o seguinte “ciclo de vida” de um paciente de hospital. O paciente entra no hospital, vítima de um acidente de carro. Ele é encaminhado para a emergência. Após uma bateria de exames, esse paciente é operado. Alguns dias depois, o paciente é movido da grande emergência do hospital para a enfermaria, pois não corre mais perigo de vida. Depois de passar por um período de observação na enfermaria, o paciente recebe alta médica.

9-4: Construa um DTE para a classe Pedido, mencionada na [Seção 9.3](#).

9-5: Construa um diagrama de estados para uma classe Mensagem, que representa uma mensagem de correio eletrônico. Como dica, considere os estados apresentados a seguir.

- a. **Recebida:** este é o estado inicial. A mensagem acabou de entrar na caixa de correio e permanece nesse estado até ser lida.
- b. **Lida:** a mensagem é lida pelo usuário.
- c. **Respondida:** o usuário responde à mensagem.
- d. **Na lixeira:** usuário remove a mensagem da caixa de correio.

9-6: Construa um diagrama de estados para um aparelho de secretária eletrônica. Como dica, considere os estados apresentados a seguir. Utilize estados compostos se for necessário.

- a. **Registrando recados:** alguém faz uma chamada para o aparelho de telefone ao qual a secretária está conectada, e a chamada não é atendida. A secretária, então, apresenta a mensagem gravada pelo usuário do aparelho e registra o recado deixado pela pessoa que fez a chamada telefônica.
- b. **Esperando:** a secretária está ociosa esperando ser ativada.
- c. **Revendo recados:** algum usuário da secretária requisitou que o aparelho apresentasse os recados gravados.

9-7: Considere o DTE para a classe Turma apresentado na [Seção 9.5](#). Suponha que as turmas tenham também um número mínimo de alunos para que possam ser fechadas. Ou seja, se esse número mínimo for N , não pode haver turmas com uma quantidade

de alunos inscritos menor que N. Modifique o DTE de Turma para contemplar essa nova restrição.

1. Na verdade, conforme descrito mais adiante, em um DTE com estados aninhados, pode haver mais de um estado inicial.
2. Para o leitor familiarizado com a terminologia da Análise Estruturada, esse evento é equivalente ao *evento temporal*.
3. Uma notação alternativa é utilizar um pequeno círculo, semelhante à representação de um estado inicial.
4. Considere que a função agora() informe o horário corrente. Essa função pode ser uma operação da classe ou uma função do sistema operacional.
5. Na verdade, uma transição não precisa ter ação ou um evento associado. Nessa situação, a transição é disparada quando as ações internas ao estado terminam. Essas transições são denominadas *transições automáticas*. Se houver várias transições automáticas partindo de um estado, obrigatoriamente cada uma deve ter uma condição de guarda, e as condições devem ser mutuamente exclusivas.

Modelagem de atividades

Qualquer um pode escrever código que um computador pode entender. Bons programadores escrevem código que seres humanos podem entender.
— MARTIN FOWLER

Há diversos diagramas da UML que descrevem os aspectos dinâmicos de um sistema. Um desses diagramas, o de estados, descreve como um sistema responde a eventos de uma maneira que é dependente do seu estado (ver [Capítulo 10](#)). Outros dois diagramas relativos a aspectos dinâmicos são os de interação (ver [Capítulo 7](#)). Outro artefato nessa categoria é o *diagrama de atividade*, que descrevemos neste Capítulo. Na [Seção 10.1](#), apresentamos os elementos de notação desse diagrama. Na [Seção 10.2](#), descrevemos em que situações esse diagrama pode ser utilizado em um processo de desenvolvimento. Finalmente, na [Seção 10.3](#) apresentamos a continuação da modelagem do SCA com relação a esse diagrama.

10.1 Diagrama de atividade

Um diagrama de atividade é um tipo especial de diagrama de estados, em que são representados os estados de uma atividade em vez dos estados de um objeto. Ao contrário dos diagramas de estados, que são orientados a eventos, diagramas de atividade são orientados a fluxos de controle.

O leitor que trabalhe com computação há mais de uma década deve se lembrar de uma ferramenta bastante utilizada no passado: o *fluxograma*. Na verdade, o diagrama de atividade pode ser visto como uma extensão dos fluxogramas. Além de possuir toda a semântica existente em um fluxograma (com notação ligeiramente diferente), o diagrama de atividade possui notação para representar ações concorrentes (paralelas), juntamente com a sua sincronização.

Os elementos de um diagrama de atividade podem ser divididos em dois grupos: os utilizados para representar fluxos de controle sequenciais e os utilizados para representar fluxos de controle paralelos. Esses elementos são ilustrados na [Figura 10-1](#) e descritos nas seções a seguir.

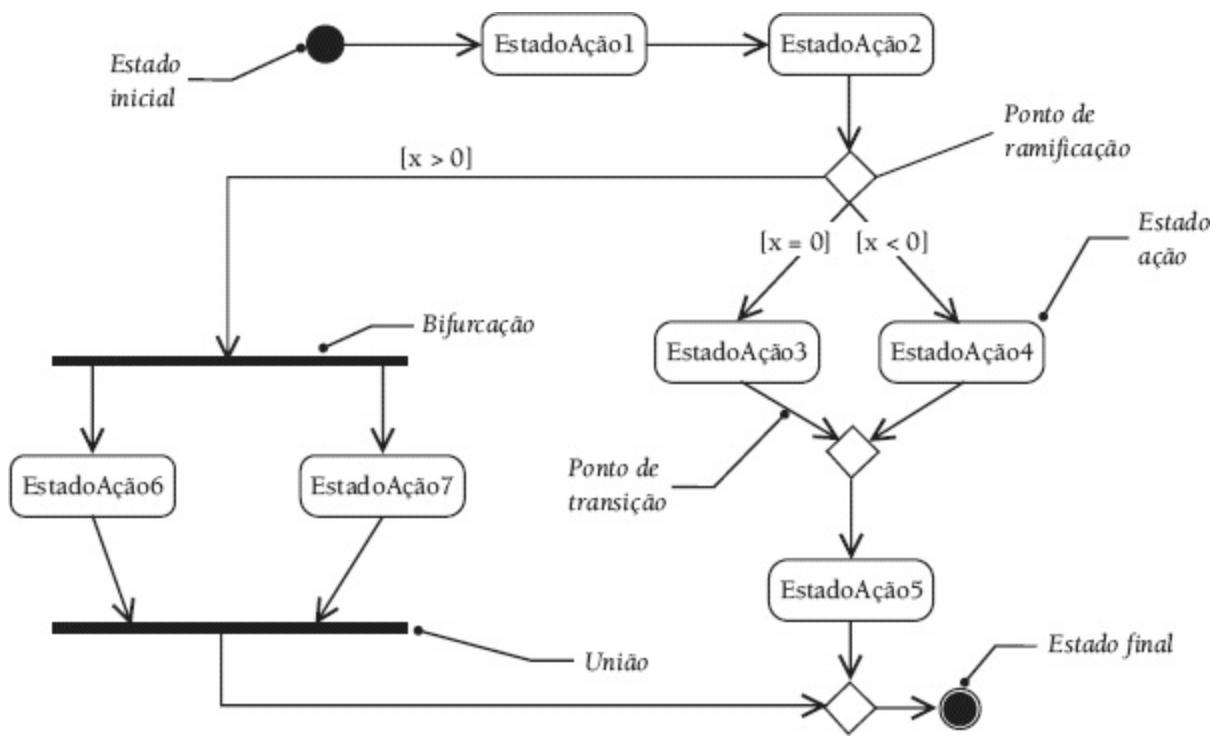


Figura 10-1: Elementos de um diagrama de atividade.

10.1.1 Fluxo de controle sequencial

Os elementos de um diagrama de atividade usados no controle sequencial são listados e descritos a seguir.

1. Estado ação
2. Estado atividade
3. Estados inicial e final e condição *de guarda*
4. Transição de término
5. Pontos de ramificação e de união

Um diagrama de atividade exibe os passos de uma computação. Cada estado corresponde a um dos passos da computação, em que o sistema está realizando algo. Um estado em um diagrama de atividade pode ser um *estado atividade* ou um *estado ação*. O primeiro leva um certo tempo para ser finalizado. Já o segundo é realizado instantaneamente.

Assim como no diagrama de estados, um diagrama de atividade deve ter um *estado inicial*; ele pode ter também vários *estados finais* e *guardas* associados a transições. Um diagrama de atividade pode não ter estado final, o que significa que o processo ou procedimento sendo modelado é cíclico.

Uma *transição de término* liga um estado a outro. Essa transição significa o término de um passo e o consequente início do outro. Observe que, em vez de ser disparada pela ocorrência de um evento (como nos diagramas de estados), essa transição é disparada pelo término de um estado de ação.

Um *ponto de ramificação* possui uma única transição de entrada e várias de saída. Para cada transição de saída, há uma condição de guarda associada. Quando o fluxo de controle chega a um ponto de ramificação, uma e somente uma das condições de guarda deve ser verdadeira. Pode haver uma transição rotulada com a condição de guarda especial *[else]*, o que significa que, se todas as demais condições de guarda foram avaliadas como falsas, a transição associada a essa guarda

especial é disparada. Um *ponto de união* (*rendezvous*) reúne diversas transições que, direta ou indiretamente, têm um ponto de ramificação em comum.

10.1.2 Fluxo de controle paralelo

Um diagrama de atividade pode conter fluxos de controle paralelos. Isso significa que é possível haver dois ou mais fluxos de controle sendo executados simultaneamente em um diagrama de atividades. Para sincronizar dois ou mais fluxos paralelos, as *barras de sincronização* são utilizadas. Há dois tipos de barra de sincronização: as *de bifurcação* (*fork*) e as *de junção* (*join*).

Uma barra de bifurcação recebe uma transição de entrada e cria dois ou mais fluxos de controle paralelos. A partir desse momento, cada um dos fluxos criados é executado independentemente e em paralelo com os demais.

Uma barra de junção recebe duas ou mais transições de entrada e une os fluxos de controle em um único fluxo. Essa barra tem o objetivo de sincronizar fluxos de controle paralelos criados anteriormente no diagrama. As transições de saída da barra de junção somente são disparadas quando *todas* as transições de entrada tiverem sido disparadas.

10.1.2.1 Raias de natação

Algumas vezes, as atividades de um processo podem ser distribuídas por vários agentes que o executarão. Isso normalmente ocorre em processos de negócio de uma organização, onde uma mesma tarefa é executada por diversas pessoas ou departamentos. Nesses casos, o processo pode ser representado em um diagrama de atividade com o uso de *raiias de natação* (tradução para *swim lanes*). As raias de natação dividem o diagrama de atividade em *compartimentos*. Cada compartimento contém atividades que são realizadas por uma agente específico.

A [Figura 10-2](#) ilustra a utilização de raias de natação. Nesse diagrama de atividades, há três compartimentos: Segurado, Seguradora e Oficina. Note que as atividades podem passar de uma raia para outra. Além disso, as entidades de cada compartimento podem estar realizando atividades em paralelo.

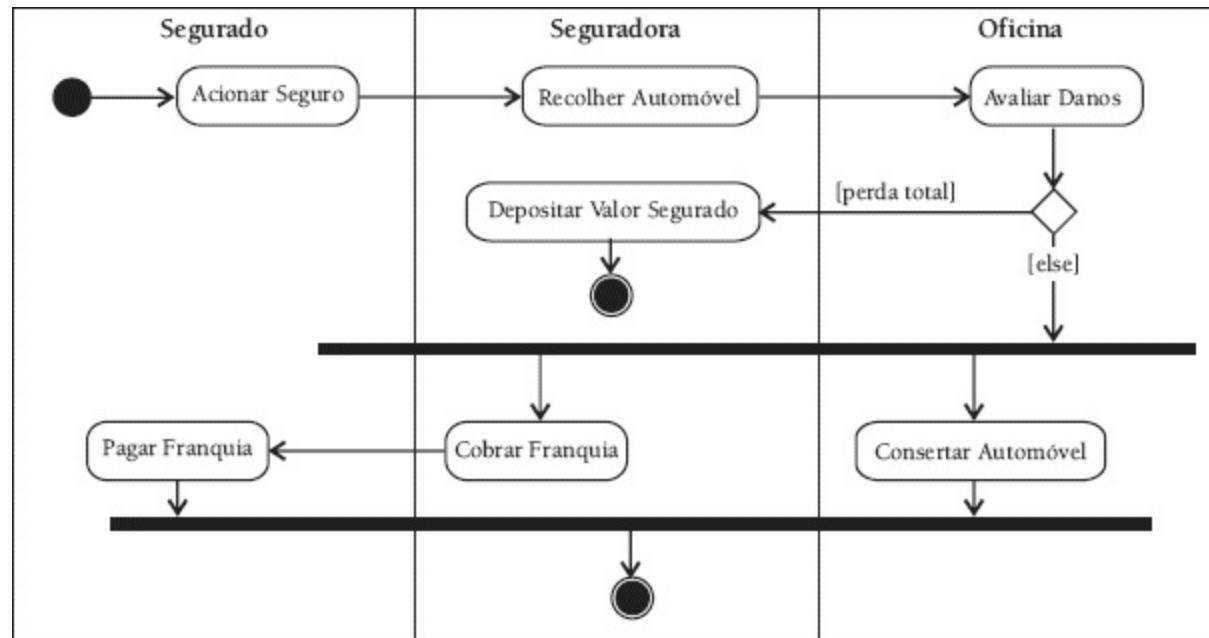


Figura 10-2: Raias de natação.

Tanto o diagrama de interação (ver [Capítulo 7](#)) quanto o de atividade são usados para modelar o comportamento do sistema. O diagrama de atividade também pode representar interações entre objetos. Entretanto, enquanto o diagrama de atividade mostra o fluxo de controle sem fazer referência às mensagens trocadas entre os objetos do sistema, o diagrama de interação exibe esses objetos e a troca de mensagens entre eles.

10.2 Diagrama de atividade no processo de desenvolvimento iterativo

Diagramas de atividade não são frequentemente utilizados na prática. Quando Seus usos, quando isso acontece, são descritos nesta seção.

Antes de passar a essa descrição, é importante notar que desenvolvedores de sistemas *orientados a objetos* devem avaliar cuidadosamente a utilidade da construção de diagramas de atividade que não seja para um dos objetivos citados nesta seção. Isso é particularmente verdadeiro para a construção de diagramas de atividade para o fluxo de controle do sistema como um todo. Lembre-se: na orientação a objetos, o sistema é dividido em objetos, e não em módulos funcionais, como na Análise Estruturada (utilizando-se o Diagrama de Fluxos de Dados, DFD). Não que haja algo de errado em utilizar as ferramentas da Análise Estruturada. O fato é que devemos sempre tentar utilizar as ferramentas certas para construir os modelos certos.

10.2.1 Modelagem dos processos do negócio

O processo de modelagem também é um processo de entendimento. Ou seja, algumas vezes o desenvolvedor constrói modelos para entender melhor determinado problema. Nesse caso, o enfoque está em entender o comportamento do sistema no decorrer de diversos casos de uso. Ou seja, como determinados casos de uso do sistema se relacionam no decorrer do tempo.

10.2.2 Modelagem da lógica de um caso de uso

A realização de um caso de uso requer que alguma computação seja realizada. Essa computação pode ser dividida em atividades. Além disso, na descrição de um caso de uso, não há uma sintaxe clara para indicar decisões, iterações e passos executados em paralelo. É comum recorrer a frases do tipo “O passo P ocorre até que a condição C seja verdadeira”, ou “Se C ocorrer, vá para o passo P”.

Nessas situações, é interessante complementar a descrição do caso de uso com um diagrama de atividade. Os fluxos principal, alternativo e de exceção podem ser representados em um único diagrama de atividade. Entretanto, note que o diagrama de atividades deve ser utilizado para complementar a descrição de um caso de uso, e não para substituí-la.

A descrição textual de um caso de uso fornece informações sobre o fluxo de eventos gerado no momento de sua realização. Portanto, para identificar atividades, pode-se examinar todos os fluxos (principal, alternativo e de exceção) de cada caso de uso. Note, entretanto, que casos de uso são descritos na perspectiva dos atores, enquanto diagramas de atividade especificam as atividades internas ao sistema.

10.2.3 Modelagem da lógica de uma operação complexa

Quando um sistema de software é adequadamente decomposto em seus objetos constituintes e as responsabilidades de cada objeto estão bem definidas, a maioria das operações é bastante simples. Essas operações não necessitam de modelagem gráfica para serem entendidas. No entanto, em alguns casos, notadamente quando uma operação de uma classe de controle (ver [Seção 5.4.2.3](#)) implementa uma regra de negócio (ver [Seção 4.5.1](#)), pode haver a necessidade de se descrever a lógica dessa operação ou da própria regra de negócio. Diagramas de atividade também podem ser utilizados com esse objetivo.

10.3 Estudo de caso

Esta seção ilustra a aplicação do diagrama de atividade no Sistema de Controle Acadêmico. O exemplo apresentado aplica-se à descrição da lógica de funcionamento do caso de uso Realizar Inscrição. A [Figura 10-3](#) ilustra o diagrama de atividades que define essa lógica de funcionamento. Note que esse diagrama realça as atividades do caso de uso que têm potencial para serem realizadas em paralelo.

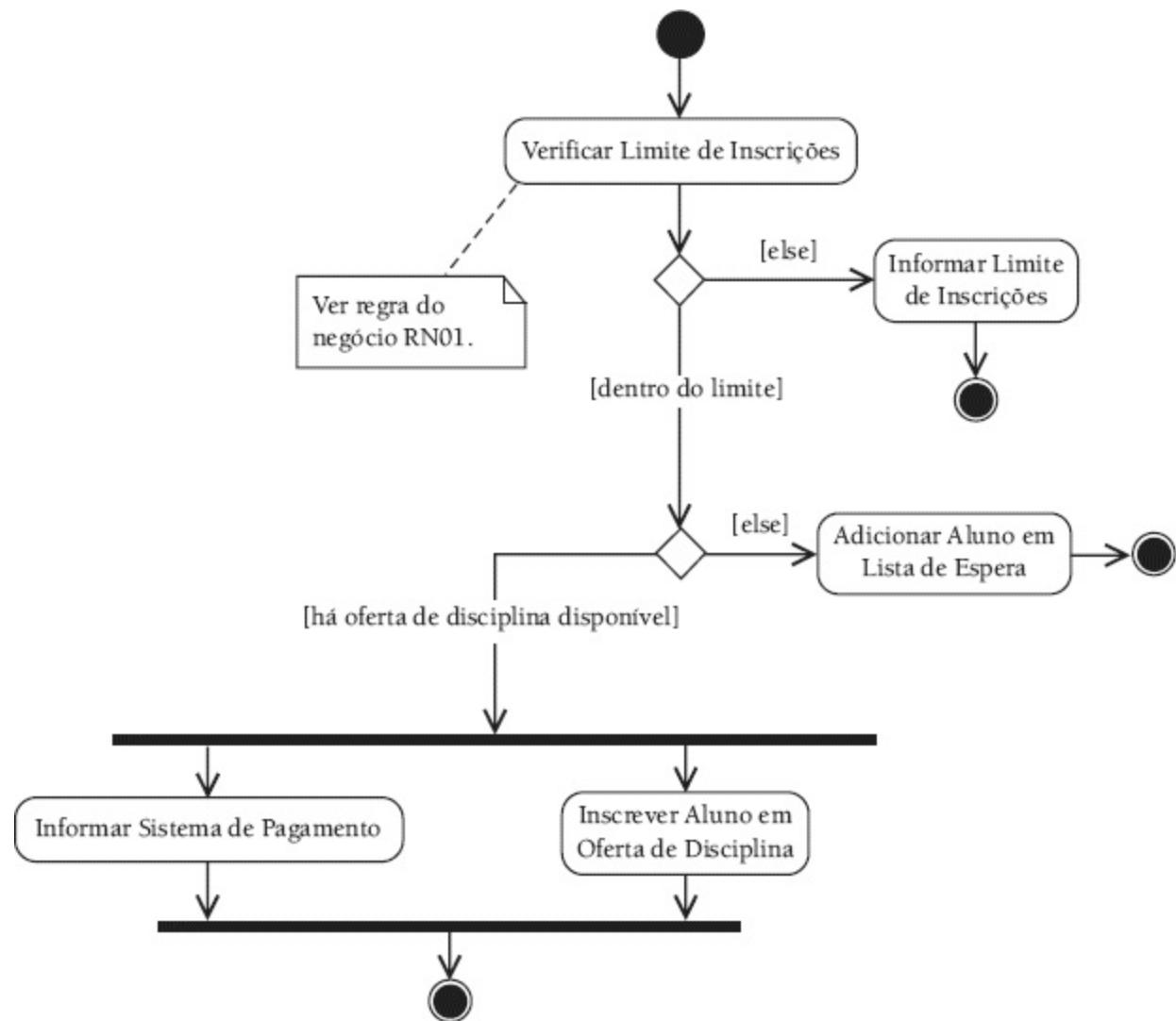


Figura 10-3: Diagrama de atividade para o caso de uso Realizar Inscrição.

Também podemos utilizar os diagramas de atividade para documentar regras de negócio. A [Figura](#)

10-4 ilustra esse uso. O diagrama de atividade dessa figura representa a lógica de implementação da regra de negócio Política de Avaliação de Alunos (RN06), apresentada na Seção 4.7.2.

Outra possibilidade de uso do diagrama de atividades para a modelagem do SCA é apresentada na Figura 10.5. Nessa figura, representamos o fluxo dos processos de negócios principais do SCA (ver Seção 10.2.1).

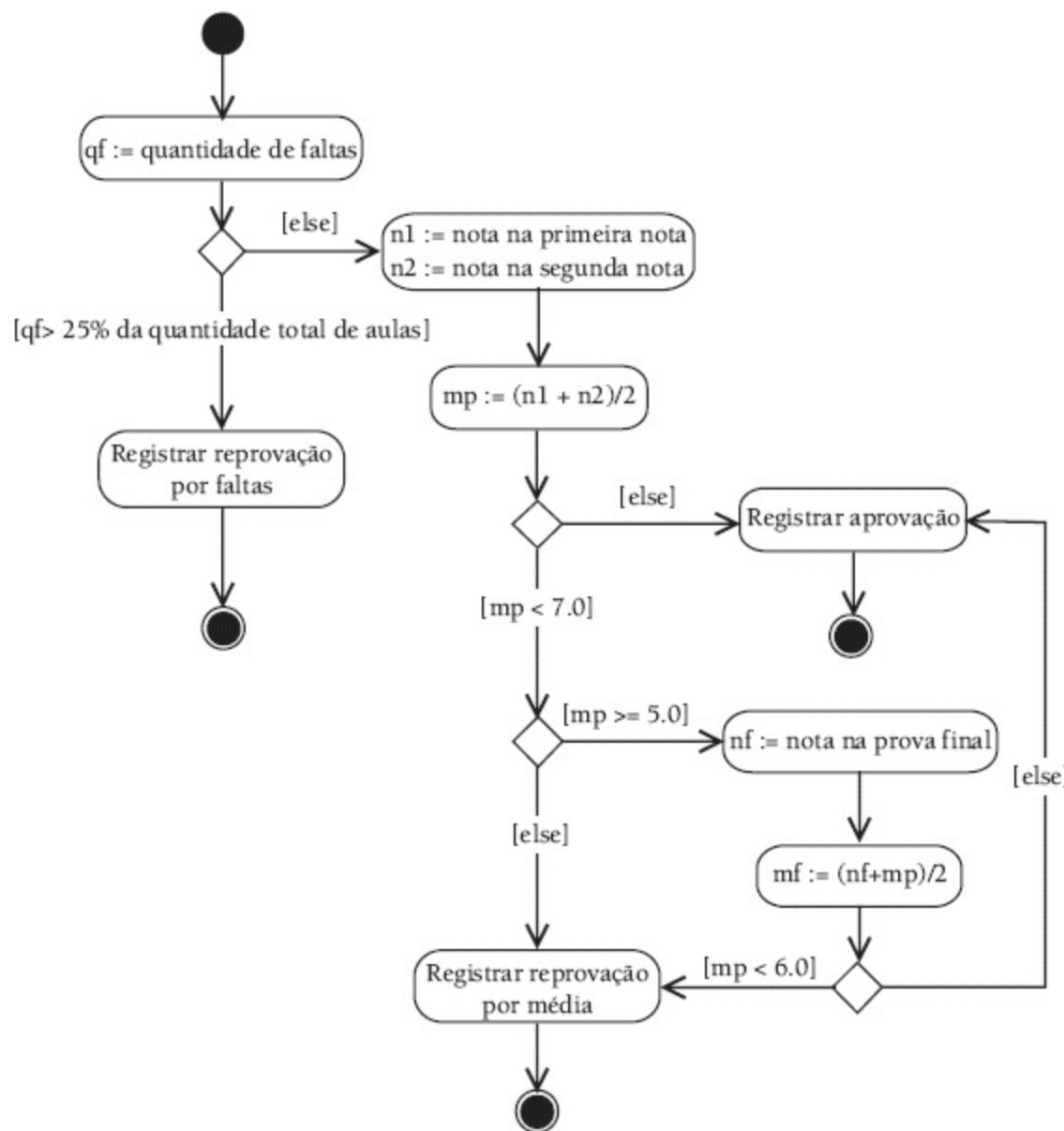


Figura 10-4: Diagrama de atividade para a regra de negócio RN06.

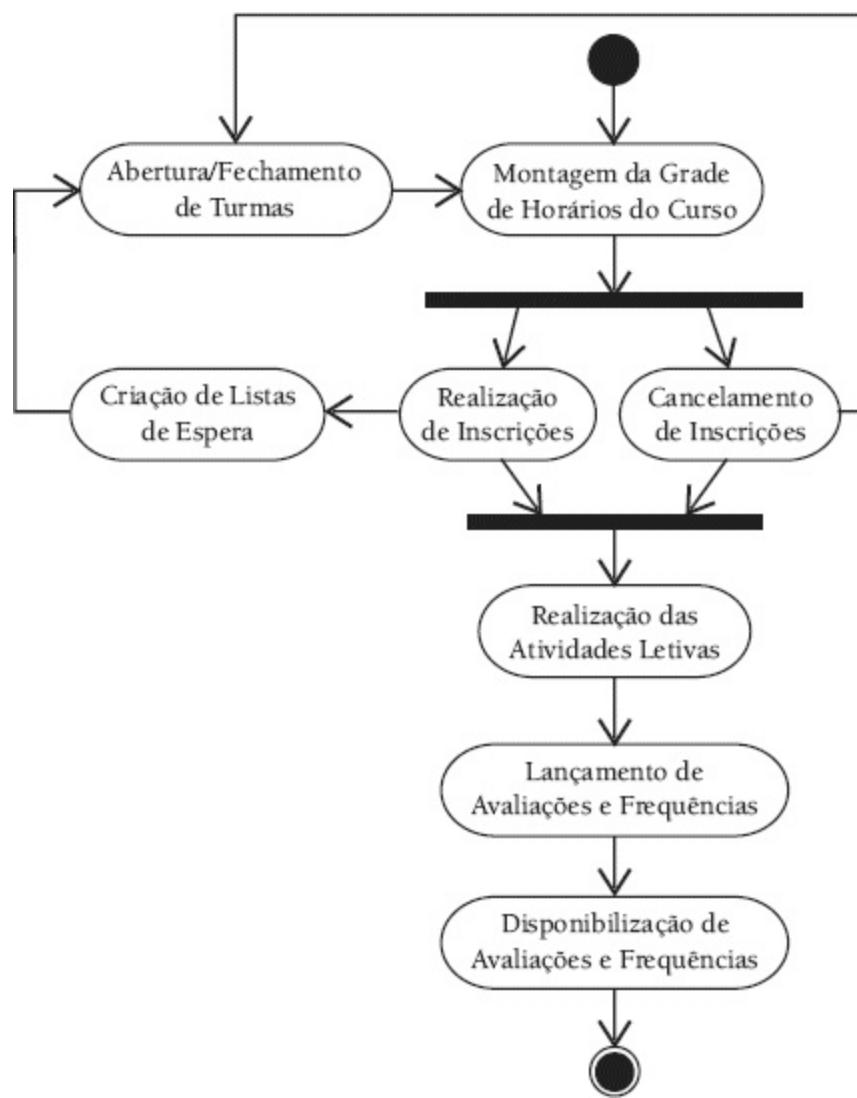


Figura 10-5: Fluxo de processos correspondentes ao caso de uso Abrir Turma.

► EXERCÍCIOS

10-1: Descreva a posição do diagrama de atividade no processo de desenvolvimento incremental e iterativo. Quando eles são utilizados e para quê?

10-2: Construa um diagrama de atividades para o seguinte processo de negócio: a autorização do pagamento tem início após um pedido ter sido feito pelo cliente. Ao mesmo tempo, a disponibilidade para cada um dos itens do pedido é verificada pelo depósito. Se a quantidade requisitada de um determinado item existe em estoque, tal quantidade é associada ao pedido. Caso contrário, somente a quantidade disponível no momento é associada ao pedido. O pedido é enviado pelo depósito ao cliente quando todos os itens estiverem associados e o pagamento estiver autorizado. O pedido será cancelado se a ordem de pagamento não tiver sido autorizada.

Arquitetura do sistema

Nada que é visto, é visto de uma vez e por completo.

— EUCLIDES

Um sistema orientado a objetos (SSOO) é composto de objetos que interagem entre si por meio do envio de mensagens com o objetivo de executar as tarefas desse sistema. Cada um desses objetos se comporta de acordo com a definição de sua classe. Por outra perspectiva, um SSOO também pode ser visto como um conjunto de *subsistemas* que o compõem. A definição dos subsistemas de um SSOO é feita no *projeto da arquitetura*, ou *projeto arquitetural*. Essa atividade é importante, porque define de que forma o sistema se divide em partes e quais são as interfaces entre elas. Além disso, há diversas vantagens em dividir um SSOO em subsistemas: produzir unidades menores de desenvolvimento; maximizar o reúso no nível de subsistemas componentes; ajuda a gerenciar a complexidade no desenvolvimento.

Atualmente, não há uma definição universal quanto ao que significa *arquitetura de software*. De acordo com o documento de especificação da UML (OMG, 2001), a definição desse termo é a seguinte: *[É] a estrutura organizacional do software. Uma arquitetura pode ser recursivamente decomposta em partes que interagem através de interfaces. Relacionamentos conectam as partes e restrições que se aplicam ao agrupamento das partes.* O termo *recursivamente* nessa definição indica que um sistema é composto de partes, sendo que as próprias partes são também sistemas relativamente independentes que cooperam entre si para realizar as tarefas do sistema.

Há dois aspectos relativos à arquitetura que devem ser definidos no *projeto arquitetural*. O primeiro aspecto está relacionado à definição de como o sistema é decomposto em diversos subsistemas e como as suas classes são dispostas pelos diversos subsistemas. Essa é a chamada *arquitetura lógica* do sistema. O segundo aspecto importante na decomposição de um sistema em seus subsistemas é definir como estes últimos devem ser dispostos *fisicamente* quando o sistema tiver de ser implantado. Ou seja, se houver diversos nós de processamento para o sistema ser executado, é importante definir em que nó cada subsistema estará posicionado e com que outros nós ele deve se comunicar. Esse segundo aspecto tem a ver com a disposição física das partes do sistema, o que é o domínio da *arquitetura física*. As decisões tomadas relativamente ao dois aspectos descritos acima para a definição da arquitetura de software influenciam diretamente na forma como um SSOO irá atender a seus *requisitos não funcionais* (ver [Seção 2.1.1](#)).

Neste capítulo, descrevemos os aspectos relativos à arquitetura de um sistema de software orientado a objetos. Aqui analisamos conceitos como subsistemas, partições, camadas, sistemas distribuídos, concorrência, nós de processamento e componentes. A [Seção 11.1](#) explica aspectos relativos à arquitetura lógica, enquanto que a [Seção 11.2](#) apresenta uma discussão acerca da arquitetura física. Finalmente, a [Seção 11.3](#) descreve o projeto arquitetural no contexto de um processo de desenvolvimento iterativo e incremental.

11.1 Arquitetura lógica

Chamamos de *arquitetura lógica* à organização das classes de um SSOO em *subsistemas*. Mas o que é um subsistema? Um sistema de software orientado a objetos, como todo sistema, pode ser subdividido em subsistemas, cada um dos quais corresponde a um aglomerado de classes e de interfaces. Um subsistema provê serviços para outros por meio de sua *interface*, que corresponde a um conjunto de serviços que ele provê. Cada subsistema provê ou utiliza serviços de outros subsistemas.

Uma visão gráfica dos diversos componentes de um SSOO pode ser representada por um *diagrama de subsistemas*. Lembremos que descrevemos os *pacotes* na [Seção 3.5](#) como um mecanismo de agrupamento geral da UML, que pode ser utilizado para agrupar vários artefatos de um modelo. Os pacotes podem ser utilizados para agrupar um tipo especial de artefato, *classes*. Pois bem, na notação da UML, um subsistema é representado por um pacote com o estereótipo `<<subsystem>>`. Veja o exemplo da [Figura 11-1](#). Um diagrama de subsistemas é aquele no qual os subsistemas de um SSOO são representados com a notação gráfica de pacotes. O fato de um subsistema utilizar os serviços fornecidos por outro é representado por um *relacionamento de dependência* (ver [Seção 8.4.1](#)).

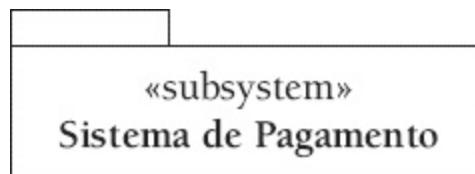


Figura 11-1: Notação da UML para um subsistema.

Durante o desenvolvimento de um SSOO, seus subsistemas devem ser identificados, juntamente com as interfaces entre eles. Cada classe do sistema é então alocada aos subsistemas. Uma vez feito isso, esses subsistemas podem ser desenvolvidos quase que independentemente uns dos outros. A seguir, são descritas algumas dicas que podem ser utilizadas para realizar a alocação de classes a subsistemas.

- O modelo de classes de domínio (ver [Capítulo 5](#)) fornece o ponto de partida para a definição dos subsistemas (pelo menos para os subsistemas relativos às classes de domínio). Isso porque podemos agrupar as classes desse modelo de acordo com o seguinte critério: primeiramente, identificamos as classes mais importantes do modelo de domínio. A seguir, para cada uma dessas classes, criamos um subsistema. Outras classes menos importantes e relacionadas a uma classe considerada importante são posicionadas no subsistema correspondente a esta última. Por exemplo, um sistema de vendas pela Internet pode ter sido decomposto nos subsistemas: Clientes, Pedidos e Entregas pelo fato de existirem classes de mesmo nome que são consideradas as mais importantes. A classe ItemPedido (menos importante) provavelmente será alocada ao subsistema Pedidos. Por outro lado, um subsistema provável para a classe Transportadora (que representa uma empresa que realiza entregas de pedidos) é o denominado Entregas.
- O princípio do *acoplamento* (ver [Seção 7.5.2](#)) pode ser aplicado para definir os subsistemas em um SSOO. Entre subsistemas, devemos manter o acoplamento baixo. Isso equivale a dizer que subsistemas devem ser *minimamente acoplados*. Para minimizar a dependência (e

o acoplamento), é preciso manter no menor patamar possível a quantidade de associações entre classes que estão definidas em diferentes subsistemas.

- O princípio de **coesão** (ver Seção 7.5.2) também pode ser aplicado: dentro de cada subsistema, devemos manter a coesão alta. Isso equivale a dizer que subsistemas devem ser *maximamente coesos*. A dependência entre os elementos *dentro* de um subsistema deve ser máxima; ou seja, deve haver mais associações entre elementos dentro de um subsistema do que entre elementos pertencentes a subsistemas diferentes.
- Dependências cíclicas entre subsistemas devem ser evitadas. Uma *dependência cílica* entre subsistemas P_1 , P_2 e P_3 existe quando P_1 depende de P_2 , que depende de P_3 , que depende de P_1 . Dessa forma, as dependências formam um ciclo: $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$. O raciocínio também vale para mais de três subsistemas. Uma alternativa para eliminar ciclos é quebrar um subsistema do ciclo em dois ou mais. Outra solução é combinar dois ou mais subsistemas do ciclo em um único (p. ex., aglutinar P_3 e P_1).
- Uma classe deve ser alocada em um único subsistema. O subsistema que define a classe deve mostrar todas as propriedades da mesma (nome, atributos e operações etc.). Outros subsistemas que fazem referência a essa classe podem utilizar sua notação simplificada. Isso evita que haja definições inconsistentes de uma mesma classe em diferentes subsistemas.

11.1.1 Conceito de camada de software

Uma camada de software (ou simplesmente camada) é uma coleção de unidades de software (como classes ou componentes) que podem ser executadas ou acessadas. As camadas representam diferentes níveis de abstração. Dessa forma, um SSOO é representado por uma pilha de camadas de software que se comunicam entre si. As camadas inferiores representam serviços cada vez mais genéricos (que podem ser utilizados em diversos sistemas), enquanto as superiores representam serviços cada vez mais específicos ao sistema em questão.

A divisão de um sistema de software em camadas permite que ele se torne mais portável e modificável. Uma mudança em uma camada mais baixa (ou seja, mais genérica) que não afete a sua interface não implicará mudanças nas camadas mais altas. E vice-versa: uma mudança em uma camada mais alta (ou seja, mais específica) que não implica a criação de um novo serviço em uma camada mais baixa não afetará esta última. Nesta seção, enfocamos a arquitetura cliente-servidor em camadas (em detrimento da arquitetura ponto a ponto), por ser mais utilizada.

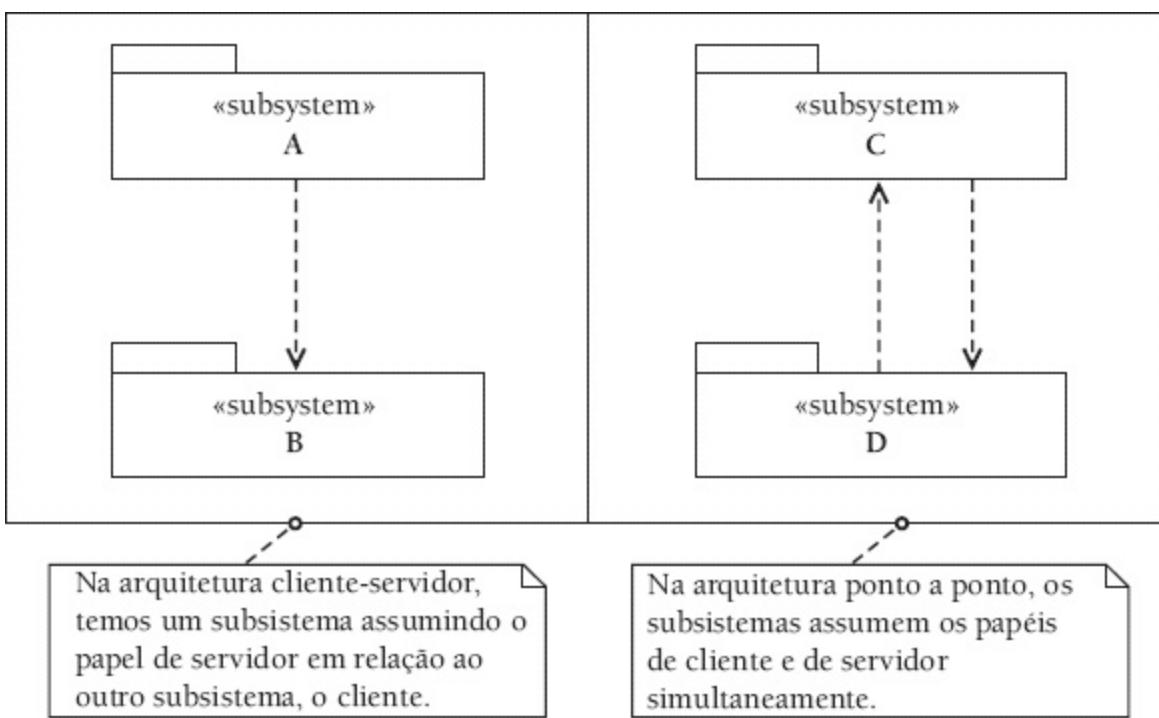


Figura 11-2: Arquiteturas cliente-servidor e ponto a ponto.

Um SSOO projetado em camadas pode ter uma *arquitetura aberta* ou uma *arquitetura fechada*. Em uma arquitetura fechada, um componente de uma camada de certo nível somente pode utilizar os serviços de componentes da sua própria camada ou da imediatamente inferior. Já na arquitetura aberta, uma camada em certo nível pode utilizar os serviços de qualquer camada inferior. Observe a Figura 11-3. Na maioria dos casos práticos encontramos sistemas construídos pelo uso de uma arquitetura aberta. A arquitetura aberta é também conhecida como *relaxada* ou *transparente*.

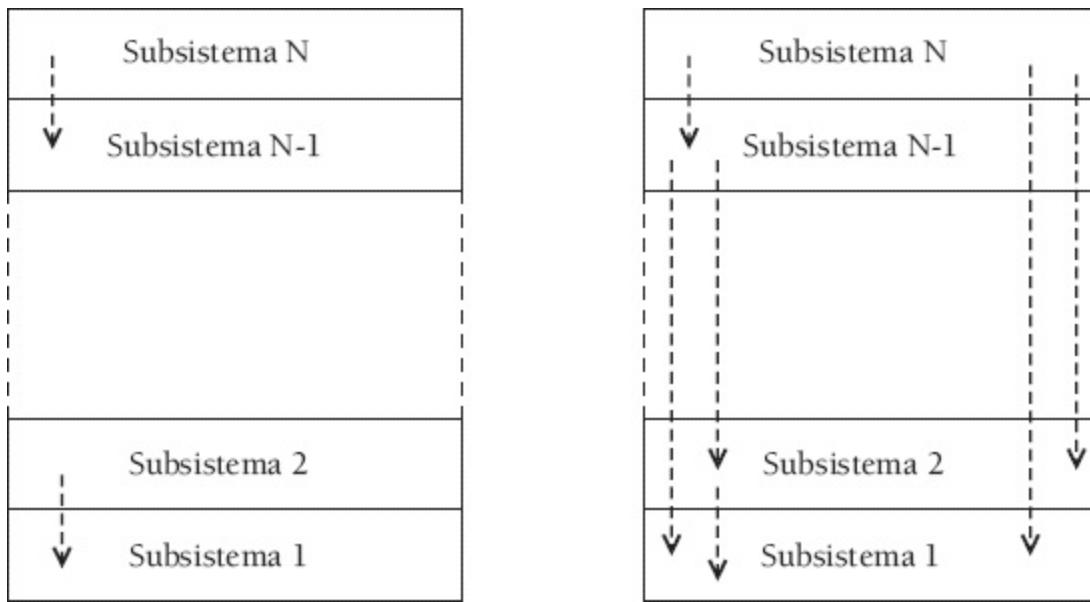


Figura 11-3: Arquiteturas abertas e fechadas.

11.1.2 Camadas típicas de um sistema de informação

A nomenclatura utilizada pela literatura para nomear as camadas típicas de um sistema de informação está longe de uma padronização. Entretanto, uma divisão tipicamente encontrada para as camadas lógicas de um SSOO é a que separa o sistema nas seguintes camadas: *apresentação*, *aplicação*,

domínio e *infraestrutura*. Nessa lista de nomes, da esquerda para a direita, temos camadas cada vez mais genéricas. Também da esquerda para a direita temos a ordem de dependência entre as camadas; por exemplo, a camada da apresentação depende (requisita serviços) da camada de aplicação, mas não o contrário.

A seguir, descrevemos as camadas tipicamente encontradas em sistemas de informação. Acompanhe pelo esquema da [Figura 11-4](#). (Note que, embora a notação para pacotes e para camadas seja a mesma, os significados desses conceitos são diferentes.)

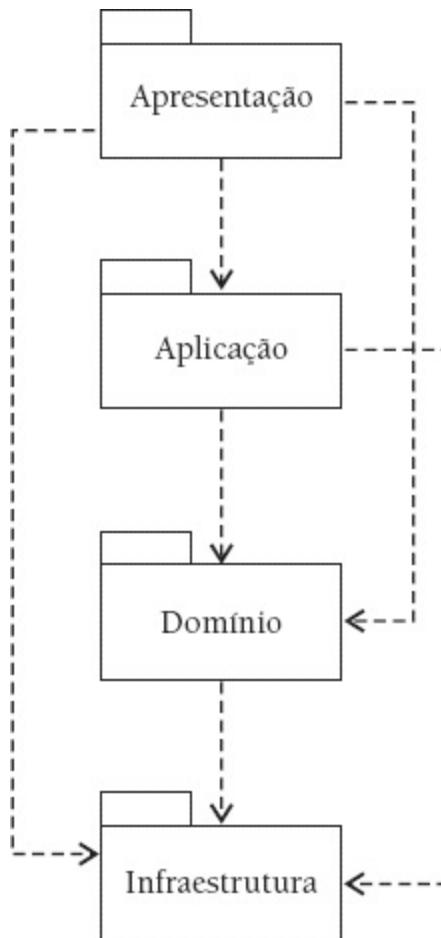


Figura 11-4: Camadas lógicas típicas de um sistema de informações.

11.1.2.1 Camada da apresentação

Essa camada também é conhecida como *camada de interface com o usuário*. A *camada da apresentação* é composta de classes que constituem a funcionalidade para visualização dos dados pelos usuários e interface com outros sistemas. As classes de fronteira se encontram nessa camada. Exemplos de camadas de apresentação: um sistema de menus baseados em texto; uma interface gráfica construída em algum ambiente de programação. Os objetos de fronteira que interagem com usuários são alocados nesta camada.

Na camada da apresentação, idealmente não deve existir inteligência, ou seja, a finalidade dessa camada é servir de mapeador de representações. Isso quer dizer que, quando uma informação é gerada por camadas subjacentes e repassada para a camada de apresentação, o que essa camada tem que fazer é formatar essa informação da maneira mais adequada possível para ser apresentada. Além disso, quando um evento de sistema (veja a [Seção 7.5.4](#)) é gerado nessa camada, o que acontece é que uma operação de sistema deve ser invocada na camada de aplicação para atender àquela

requisição (veja a descrição desta camada mais adiante).

A camada da apresentação deve apenas servir como um ponto de captação de informações a partir do ambiente, ou de apresentação de informações que o sistema processou. Portanto, não devemos atribuir às classes dessa camada responsabilidades relativas à lógica do negócio. A única inteligência que essas classes devem ter é a que lhes permite realizar a comunicação com o ambiente do sistema. Esse tipo de lógica por vezes é denominado ***lógica da apresentação***.

A construção de uma camada de apresentação leve (i.e., uma cuja única lógica que implementa seja a relativa à apresentação) é um objetivo a ser alcançado durante o projeto arquitetural. Há diversas razões para manter a separação entre as lógicas da apresentação e do negócio, conforme descrição a seguir.

- Se o sistema tiver que ser implantado em outro ambiente, as modificações resultantes de seu funcionamento propriamente dito (i.e., excluindo as interações com o novo ambiente) seriam mínimas, o que aumenta a portabilidade da aplicação. Por exemplo, considere a situação em que uma aplicação foi inicialmente construída com uma interface gráfica para a WEB e posteriormente tenha surgido um requisito para migrar essa aplicação para uso em algum dispositivo móvel.
- O sistema pode dar suporte a diversas formas de interação com seu ambiente (p. ex., uma interface gráfica e outra interface via um terminal de comandos).
- Finalmente, essa separação resulta em uma melhor *coesão* (ver [Seção 7.5.2](#)) da arquitetura do sistema.

A seguir enumeramos as principais formas de interação com o ambiente que podem existir em um sistema de software. Cada uma dessas formas resulta na necessidade de implementação de uma camada de apresentação.

1. **Cientes WEB.** Essa é a forma mais comum de interação em aplicações WEB. As classes da camada da apresentação geram páginas HTML dinâmicas, ou seja, páginas criadas em função de estímulos que o sistema recebe de seu ambiente. Aplicações ricas também se encaixam nessa categoria, em que mecanismos de comunicação assíncrona entre o cliente e o servidor são utilizados para permitir a atualização apenas de partes da interface gráfica com o usuário. Além disso, aplicações ricas são construídas por meio de tecnologias que propiciam uma melhor experiência para o usuário do ponto de vista de usabilidade (p. ex., AJAX).
2. **Cientes (dispositivos) móveis.** É cada vez mais comum o uso da tecnologia móvel em todas as áreas de negócio. Telefones celulares e computadores manuais são cada dia mais frequentes e interativos. Para esse tipo de interação com o ambiente, normalmente são necessárias classes de fronteira que implementem algum protocolo específico com o ambiente. Um exemplo é a WML (*Wireless Markup Language*).
3. **Cientes stand-alone.** É possível também que a aplicação tenha que fornecer uma interface gráfica por meio de formulários, com botões, menus etc. Nesse caso, é recomendável que os desenvolvedores pesquisem os recursos fornecidos pelo ambiente de programação sendo utilizado. Muitos desses ambientes fornecem componentes para construção de interfaces gráficas. Exemplo disso são o Swing/JFC da plataforma Java e o Windows Forms da plataforma .NET™.

4. Serviços WEB. Outra tecnologia cujo uso está se tornando bastante comum é a de serviços WEB (*WEB services*). Um serviço WEB é uma forma de permitir que uma aplicação forneça seus serviços (funcionalidades) pela Internet.

Nos casos 1 e 3, as classes da camada da apresentação correspondem à interface como seres humanos. Uma atividade importante de um processo de desenvolvimento que envolve essas classes é o *projeto da interface gráfica com o usuário*. Nessa atividade, são especificados aspectos de aparência e disposição dos componentes gráficos, assim como de navegação pelos diversos formulários da aplicação. Conforme mencionamos na [Seção 6.5](#), o projeto de interface gráfica com o usuário está fora do escopo deste livro.

Nos casos 2 e 4, o sistema deve se comunicar com outro sistema de software ou com um dispositivo (equipamento). Como exemplo, em nosso estudo de caso do Sistema de Controle Acadêmico (ver [Seção 4.7](#)) temos um ator denominado Sistema de Faturamento, correspondente a um sistema externo. Nesse estudo de caso, o SCA deve se comunicar com o Sistema de Faturamento, um sistema externo. Assim sendo, podemos criar uma classe denominada *SistemaFaturamento* para encapsular o comportamento de interação com o sistema externo. Entretanto, é também possível a situação em que a interação com um sistema externo é tão complexa que não seja adequado encapsulá-la em uma única classe. Portanto, é necessário criar um *subsistema* para representar a comunicação com o sistema externo. Além disso, esse subsistema agrupa todas as classes necessárias nessa comunicação e provê uma *interface* (ver [Seção 8.5.4](#)) a partir da qual seus serviços podem ser obtidos. A interação do restante da aplicação com um subsistema desse tipo normalmente é feita com o uso do padrão de projeto *Façade* (ver Seção 8.7.6).

11.1.2.2 Camada da aplicação

Essa camada também é conhecida como *camada de serviço*. A **camada da aplicação** é a que serve de intermediária entre os vários componentes da camada de apresentação (p. ex., telas da interface gráfica e interfaces de voz, interfaces por linha de comando etc) e a lógica contida nos objetos do negócio. Esta camada traduz as mensagens que recebe da camada da aplicação em mensagens compreendidas pelos objetos do domínio. É também responsabilidade dessa camada o controle da navegação do usuário de uma janela a outra, quando a interação ocorrer por meio de uma interface gráfica.

A camada da aplicação nem sempre é definida em sistemas multicamadas. A utilização dessa camada é normalmente necessária quando o sistema é complexo, com muitos casos de uso (dezenas ou até centenas deles). Os objetos de controle são alocados a essa camada.

Na [Seção 7.5.4](#), descrevemos o conceito de operação de sistema. Essas operações são alocadas em classes da camada da aplicação. Geralmente, há uma classe da camada da aplicação para cada caso de uso do sistema. Em cada uma dessas classes, cada operação de sistema direciona (por meio do envio de mensagens) os objetos do negócio na realização das funcionalidades do sistema. As classes de controle (veja a [Seção 5.4.2.3](#)) são a contrapartida, na etapa de análise, das classes que residem na camada da aplicação.

Idealmente a camada da aplicação não deve conter qualquer lógica relativa à validação de regras do negócio. Em vez disso, essa camada apenas delega tarefas para os objetos da camada do domínio. Classes dessa camada têm a responsabilidade de coordenar a *interação* entre outros objetos, principalmente os da camada do domínio. Alguns exemplos dos comportamentos cuja *coordenação*

(e não a *realização* propriamente dita) é responsabilidade das classes da camada da aplicação são repasse de informações para a camada da apresentação, autenticação de usuários, controle de acesso, controle de transações etc.

Um tipo especial de controlador é o *de caso de uso*, ao qual já fizemos menção na [Seção 5.4.2.3](#). Esse objeto é responsável pela coordenação da realização de um caso de uso em particular. Mais especificamente, as seguintes responsabilidades são esperadas de um típico controlador de caso de uso:

- Coordenar a realização de um caso de uso específico.
- Servir como canal de comunicação entre objetos das camadas da apresentação e do domínio
- Manipular exceções provenientes da camada do domínio.

A definição incorreta das responsabilidades de um controlador de caso de uso pode aumentar substancialmente o *acoplamento* (ver [Seção 7.5.2](#)) do sistema. Para evitar o alto acoplamento, ele deve *delegar* a realização de tarefas para outros objetos, particularmente os da camada do domínio. Essa abordagem resulta em um projeto mais flexível, controlável e reutilizável.

11.1.2.3 Camada do domínio

Também é chamada de *camada do negócio*. A **camada do domínio** é aquela onde se encontram a maioria dos objetos identificados durante a análise de domínio (ver [Seção 2.1.2](#)). Esta camada recebe requisições provenientes da camada da aplicação. Os objetos nesta camada normalmente são independentes da aplicação (o que significa que podem ser reusados em diferentes aplicações dentro de uma corporação). Essa camada é responsável por validações das *regras de negócio* (ver [Seção 4.5.1](#)), assim como de dados provenientes da camada de apresentação (por intermédio da camada de aplicação).

As classes residentes na camada do domínio são aquelas que representam os conceitos do domínio da aplicação que o sistema deve processar. Essas classes representam as informações produzidas durante a realização dos processos do negócio, assim como as regras de negócio que direcionam a manipulação dessas informações. Elas também possuem uma denominação alternativa, *classes do negócio*.

Na [Seção 5.4.3.2](#), descrevemos os padrões táticos do DDD. Esses padrões táticos são aplicáveis a classes que residem na camada de domínio. Por exemplo, no contexto do SCA, algumas entidades encontradas na camada do domínio desse sistema são Aluno, Disciplina, Turma, Professor.

Ainda com relação aos padrões táticos do DDD, uma observação importante é relativa aos *serviços do domínio* (*domain services*). Conforme descrevemos na [Seção 11.1.2.2](#), os controladores de caso de uso são as classes que compõem a camada da aplicação e são também denominados *classes de serviço*, o que, por conta da semelhança de nomes, pode gerar uma confusão entre estas e os serviços do domínio. Na verdade, a distinção é bastante simples: serviços do domínio, por definição, podem conter lógica do domínio, ao contrário das classes da camada de serviço.

Um aspecto importante a ser considerado durante a implementação da camada de domínio é tentar isolar a sua lógica de aspectos técnicos. Esse isolamento diz respeito a implementar a lógica dessa camada de tal forma que suas classes não fiquem dependentes de bibliotecas ou frameworks específicos utilizados na implementação das demais camadas. Nesse contexto, um artifício que pode ser usado para alcançar este isolamento é o acoplamento abstrato (veja a [Seção 8.5.5](#)), conforme

descrito a seguir. Quando uma classe do domínio precisar dos serviços contidos em uma classe de outra camada, em vez de invocar esses serviços diretamente a um objeto desta classe, isso pode ser feito por meio de uma interface ou de uma classe abstrata, que por sua vez é implementada por uma classe residente em outra camada. O **princípio da inversão de dependência** (*dependency inversion principle*, DIP) é outro nome utilizado nesse contexto. Esse princípio preconiza que tanto módulo de baixo quanto de alto nível de abstração devem depender de uma abstração (ou seja, de uma interface ou de uma classe abstrata), que é exatamente o que ocorre no acoplamento abstrato.

Um efeito colateral benéfico de se alcançar o isolamento da camada de domínio diz respeito a sua **testabilidade**. Se essa camada for corretamente isolada, é possível avaliar a parte mais importante da lógica do sistema (que, por definição, está localizada na camada de domínio) por meio de testes automatizados. A automação de teste em um sistema orientado a objetos corresponde a criar classes de teste para testar as classes do sistema propriamente ditas. Nessas classes de testes são definidos métodos que invocam métodos das classes do sistema e realizam a comparação entre o valor esperado e o valor efetivamente retornado. Esse procedimento serve para complementar e, em alguns casos substituir, o procedimento de teste manual do sistema, em que a equipe de testadores executa a aplicação, realiza as entradas de dados e verifica se o resultado produzido pelo software é coerente. Esse é um processo maçante e que requer uma quantidade significativa de tempo da equipe envolvida. A criação de testes automatizados permite a execução de vários casos de testes de forma relativamente rápida. Além disso, a bateria de testes automatizados que vai se formando gradativamente pode ser executada a cada atividade de manutenção (corretiva ou evolutiva) no software, com a finalidade de verificar se essa alteração gerou efeitos colaterais negativos em outras partes do sistema. Em seu livro intitulado *Refatoração*, Martin Fowler usa a expressão “rede de segurança” para fazer alusão aos testes automatizados, fazendo uma correspondência às redes de segurança que os acrobatas usam para realizar suas manobras (FOWLER, 2004). A analogia apresentada por esse autor é adequada, uma vez que a repetibilidade associada aos testes provê segurança à equipe de desenvolvedores quando há a necessidade de alterar o código-fonte da aplicação. Nesse contexto, o uso de algum framework da família xUnit é recomendado para agilizar a criação de classes para realizar testes automatizados. Um exemplo representativo dessa família para a plataforma Java é o JUnit, disponível em <http://junit.org/>.

11.1.2.4 Camada de infraestrutura

Essa camada é também conhecida como **camada de serviços técnicos**. A **camada de infraestrutura** é o lugar onde são encontrados serviços genéricos e úteis para uma gama bastante vasta de aplicações. Como o próprio nome diz, essa camada fornece serviços relacionados às tecnologias usadas implementação da aplicação. Exemplos de serviços fornecidos por esta camada são autenticação e autorização, controle de transações, registro de operações do sistema (*logging*), manipulação de arquivos, estruturas de dados (vetores de caracteres, mapas, listas etc.), classes utilitárias, entre outros.

Além dos serviços listados acima, a camada de infraestrutura também deve conter classes cujo objetivo é permitir que o sistema se comunique com outros sistemas para realizar tarefas ou adquirir informações (p. ex., acesso a um SGBD ou a serviços WEB). Em particular, a **camada de persistência**, que descrevemos na [Seção 12.2](#), é um serviço (subcamada) da camada de infraestrutura.

Conforme ilustrado na [Figura 11-4](#), as camadas superiores são dependentes (ou seja, requisitam

serviços) da camada de infraestrutura. Por exemplo, a implementação da camada da apresentação pode fazer uso de algum framework MVC (veja a [Seção 11.1.3](#)).

11.1.2.5 Discussão

Uma característica da divisão de um sistema de informação em camadas descrita anteriormente é que as camadas mais altas (superiores) devem depender das camadas mais baixas (inferiores), e não o contrário. Essa disposição ajuda a gerenciar a complexidade por meio da divisão do sistema em partes menos complexas que o todo. Também incentiva o reúso, porque as camadas inferiores são projetadas para serem independentes das camadas superiores. Finalmente, o acoplamento entre camadas é mantido no nível mínimo possível. Note, entretanto, que há casos em que uma camada inferior precisa enviar algum sinal para algum elemento de alguma camada superior (veja a discussão sobre o padrão Observer no fim desta seção).

Outra característica importante da divisão de um sistema de informação em camadas descrita anteriormente é que cada uma delas possui um conjunto específico de responsabilidades. De particular importância é a separação entre a apresentação das informações (que é feita pela camada de apresentação) e o processamento das mesmas (realizado principalmente pela camada de domínio). Para esclarecer a importância dessa separação, considere o caso dos chamados *sistemas de informações organizacionais* (*enterprise information systems*, EIS), que manipulam grandes quantidades de dados e fornecem serviços de alta qualidade para integrar diversos processos de negócio em uma grande organização. Um EIS normalmente possui muitos grupos de usuários, cada um deles com sua necessidade em relação à forma de interagir com o sistema. Nesses sistemas, a mesma informação deve ser apresentada em formatos e em diferentes perspectivas. Além disso, as mudanças realizadas em uma perspectiva devem ser refletidas imediatamente em outras. Além disso, há o requisito de que a funcionalidade do núcleo do sistema deve ser independente das diferentes perspectivas fornecidas. Sendo assim, é fundamental estruturar os componentes dessa aplicação de tal forma que ela seja facilmente adaptável a diferentes alternativas de apresentação de suas informações e interação com seu ambiente. Nesses sistemas, é fundamental que a construção de uma nova forma de apresentação não resulte em mudanças nas camadas inferiores (na camada de domínio, por exemplo).

É também importante notar que um sistema de informação pode disponibilizar diversas variantes de uma camada. Por exemplo, em uma aplicação que forneça certa funcionalidade que deve ser acessível tanto por um usuário final (ser humano) quanto por outra aplicação (por meio de um serviço WEB, por exemplo), pode haver duas camadas de apresentação, ambas com acesso aos serviços fornecidos pela mesma camada da aplicação.

É também possível que, em função de sua complexidade, uma camada seja subdividida verticalmente no que costumamos chamar de *partições*. Como exemplo de partições, considere novamente o exemplo dado anteriormente neste capítulo de um sistema de vendas pela WEB, cuja camada de domínio foi decomposta nos subsistemas (partições): Clientes, Pedidos e Entregas. Como outro exemplo, dessa vez no contexto do SCA, a camada do domínio desse sistema poderia ser subdividida em partições, de acordo com os pacotes apresentados na [Seção 4.7.3](#). A propósito, na linguagem Java, o mecanismo utilizado para definir tanto camadas quanto partições é o de *pacote*, que corresponde a um contêiner de classes e de interfaces da aplicação. Nessa linguagem, pode haver a composição recursiva de pacotes.

Outra característica digna de nota acerca das camadas de apresentação e de infraestrutura é que

ambas fornecem serviços de interação com o ambiente do sistema. Por um lado, a camada da apresentação está tipicamente associada à interação com usuários. Já a camada de infraestrutura provê serviços para que a aplicação interaja com sistemas externos (como mecanismos de armazenamento de dados persistentes, servidores, etc).

No início deste capítulo, declaramos que o projeto da arquitetura de um sistema influencia diretamente na qualidade do mesmo com relação ao atendimento de seus *requisitos não funcionais* (ver [Seção 2.1.1](#)). De fato, a separação de um SSOO nas camadas descritas acima permite que o mesmo atenda de forma satisfatória diversos de seus requisitos não funcionais. A *manutenibilidade* e a *flexibilidade* do sistema, em particular, aumentam com a divisão em camadas.

Durante a definição da arquitetura lógica de um SSOO, o uso de *padrões de projeto* (consulte a [Seção 6.6](#)) é bastante comum. Esses padrões são utilizados para definir o que chamamos de microarquitetura do sistema, ou seja, decisões de projeto tomadas durante a construção de uma camada em alguma partição. Seguem alguns exemplos dessa situação:

- Para comunicação entre subsistemas, normalmente o padrão Façade ([Seção 8.6.6](#)) pode ser utilizado.
- Para diminuir o acoplamento entre camadas (ou entre partições dentro de uma camada), os padrões Factory Method ([Seção 8.6.4](#)) e Domain Service ([Seção 5.4.3.2](#)) pode ser utilizado.
- O padrão Observer ([Seção 8.6.2](#)) pode ser utilizado quando uma camada em certo nível precisa se comunicar com uma camada de um nível superior. Nesse caso, o componente da camada inferior representa o *sujeito* (i.e., o observável), enquanto o componente da camada superior representa o *observador*. O observável se comunica com a camada superior por meio de uma interface genérica, e portanto não precisa ter conhecimento da classe específica do observador que reside na camada superior. Na prática, isso evita que um objeto de uma camada guarde uma referência direta para algum objeto de uma camada superior. O padrão Observer alcança esse objetivo, porque permite a comunicação entre objetos por meio do acoplamento abstrato (ver [Seção 8.5.5](#)).

Finalmente, outra questão importante diz respeito à distinção entre as responsabilidades das camadas da aplicação e do domínio. Não raro, essa distinção não é tão nítida, e várias vezes pode surgir a dúvida acerca de onde alocar certa responsabilidade. Neste ponto, deve-se atentar ao seguinte: se a responsabilidade diz respeito ao domínio da aplicação, é sempre mais adequado alocá-la a alguma classe da camada do domínio. A camada da aplicação deve conter apenas classes com *operações de sistemas* (veja a [Seção 7.5.4](#)) que em conjunto correspondem à interface para interação com o SSOO. É importante notar também que nem sempre a camada de aplicação é utilizada: quando o sistema em questão não for complexo, a camada da apresentação pode enviar requisições diretamente à camada do domínio.

11.1.3 O padrão MVC e sua relação com a arquitetura lógica

Na [Seção 11.1.2](#), descrevemos as camadas tipicamente encontradas em sistemas de informação. Em particular, mencionamos que, idealmente, a camada da apresentação não deve conter inteligência, mas sim implementar apenas lógica da apresentação (preenchimento de controles co dados provenientes da camada da aplicação, habilitação de controles, definição de cores etc.). Um padrão

arquitetural normalmente utilizado para alcançar essa separação de responsabilidades entre a *lógica da apresentação* e a *lógica da aplicação* é o Model-View-Controller (MVC). Nessa seção, descrevemos a variante desse padrão mais popular atualmente e a relacionamos com as camadas apresentadas na [Seção 11.1.2](#).

O Model-View-Controller é um padrão de software que descreve a interação entre objetos da interface com o usuário e os demais objetos de uma aplicação. Em particular, esse padrão propõe: (1) uma forma de organizar a interface com o usuário; e (2) descreve de que maneira atualizar o estado dessa interface. A proposta inicial desse padrão foi apresentada na linguagem de programação Smalltalk-80. Ao longo do tempo, diversas variações da proposta original foram proposta. De forma geral, todas as variantes apresentam três componentes, Model, View e Controller, conforme descrito a seguir.

- **Model.** Esse componente é a parte da aplicação que contém os dados e suas validações. Em outras palavras, o componente Model corresponde ao estado, à estrutura e ao comportamento dos dados sendo visualizados e manipulados pelo usuário da aplicação por meio da interface gráfica. Esse componente disponibiliza operações em sua interface para que o restante da aplicação possa manipulá-lo (p. ex., consultá-lo ou editá-lo). O Model não contém dependências (veja a [Seção 8.4.1](#)) para os outros dois componentes.
- **View.** Uma aplicação pode apresentar aos seus usuários diversas perspectivas (visões) de uma mesma informação (ou de um mesmo modelo). Por exemplo, pode haver uma interface gráfica para editar as notas de um aluno em uma turma, assim como outra interface gráfica para permitir a visualização dessas notas para impressão apenas. O componente View do MVC representa cada uma das possíveis formas de apresentar uma informação proveniente do componente Model.
- **Controller.** Cada visão possui a ela associada um objeto controlador, que auxilia na implementação da interface gráfica associada à visão correspondente.

O MVC original foi proposto no contexto de cada controle de uma interface gráfica (p. ex., um botão, uma lista suspensa etc). Nessa proposta original, o padrão Observer (veja a [Seção 8.6.2](#)) é usado para permitir que o View atualize a si próprio ao se inscrever como um observador do Model: no momento em que o Controller atualiza o Model, essa atualização é notificada ao View que por sua vez atualiza o seu estado. Agora, vamos descrever uma das variantes do MVC original que adapta esse padrão para aplicações WEB. Nessa variante, esquematizada na [Figura 11-5](#), o componente Model possui a mesma atribuição do MVC clássico, ou seja, ele representa as informações manipuladas pelo sistema por meio de sua interface com o usuário.

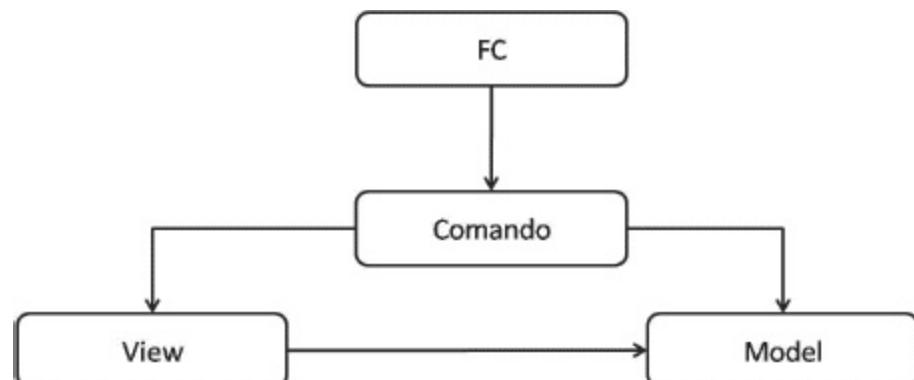


Figura 11-5: Variante do padrão MVC para aplicações Web.

Conforme a [Figura 11-5](#), no MVC para Web, o componente Controller é representado por dois tipos de objetos. O primeiro tipo corresponde ao próprio FC. O primeiro é o **Front Controller** (FC), cujo propósito é servir de ponto único de entrada (do lado do servidor) para as funcionalidades de uma aplicação WEB. O FC manipula requisições HTTP provenientes de um cliente (p. ex., um navegador WEB ou um aplicativo para dispositivos móveis). O segundo tipo corresponde aos denominados **comandos**. Em uma aplicação Web, normalmente existe uma única classe FC (e um único objeto dessa classe a tempo de execução), mas existem diversas classes de comando, uma classe para cada requisição possível da aplicação Web. Essa estratégia torna mais fácil controlar alguns aspectos da aplicação, como, por exemplo, a autenticação dos usuários.

Na [Figura 11-5](#), as setas indicam o sentido do envio de mensagens entre os componentes. Ao receber uma requisição HTTP, o FC realiza operações gerais de recepção, como autorização e autenticação. A seguir, esse objeto, identifica qual o comando adequado para processar a requisição, possivelmente mediante consulta a algum serviço de mapeamento de URLs. Uma vez que identifica o comando adequado para processar a requisição, o FC a despacha para ele. Ao receber o controle da execução, o comando interage com o Model, por vezes alterando, por vezes apenas consultando o estado dele. As mensagens que o comando envia ao Model dependem da requisição específica enviada pelo cliente. Após essa interação com o Model, o comando repassa o controle realiza um controle de navegação ao selecionar qual o componente View adequado para produzir a resposta ao usuário. Na variante do MVC para Web, o View é responsável por gerar o conteúdo (p. ex., a página HTML) a ser enviado ao cliente que originou a requisição HTTP. Ao passar o controle da execução para o View, é possível que o Controller passe também uma parte das informações que resgatou do Model. Isso acontece, por exemplo, quando o View precisa montar uma página HTML que contenha essa informação. Essa página HTML é então repassada ao cliente Web que gerou a requisição HTTP original, o que fecha o ciclo de atendimento de uma requisição no MVC para Web.

É comum a situação em que determinado formulário deve ser apresentado ao usuário com alguns de seus controles (como listas ou menus suspensos) já preenchidos com informações provenientes dos objetos de domínio. Para exemplificar esse aspecto, considere o formulário para realização de inscrições em turmas do SCA. Quando este formulário for apresentado ao usuário (nesse caso, um aluno), devem ser apresentadas as turmas disponíveis para inscrição. Nesse caso, uma vez obtida a lista de turmas a partir de uma camada subjacente, uma classe da camada da apresentação pode assumir a responsabilidade de organizar essa lista em um formato específico. Essas classes são denominadas **View Helpers** (VH). Uma classe VH é, portanto, uma classe integrante do componente View do MVC para Web que auxilia na formatação apropriada de informações provenientes do Model.

No contexto da linguagem Java, as tecnologias Java ServerPages e Servlets são usadas para implementar o MVC para Web. Existem também diversos **frameworks MVC**, que dão suporte à organização da aplicação de acordo com esse padrão. Três exemplos de frameworks MVC são o Java ServerFaces, o *Spring MVC* e o *Apache Struts*. Na plataforma Microsoft, existe o framework **ASP.NET MVC**.

Até aqui, descrevemos resumidamente o funcionamento do MVC original e de sua variante para aplicações Web. Outro aspecto importante é a relação que existe entre o MVC para Web e as camadas da arquitetura lógica descritas na [Seção 11.1.2](#). Nesse contexto, é importante entender que os componentes do MVC não são camadas, embora haja uma correspondência entre esses componentes e as camadas. Em particular, os componentes View e Controller do MVC para Web correspondem a objetos que residem na camada da apresentação. Já o componente Model dessa

variante corresponde às informações a validações realizadas pelas camadas subjacentes à camada da apresentação. Sendo assim, um objeto comando (que reside na camada da apresentação) interage com o **controlador do caso de uso** (que reside na camada da aplicação). Esse controlador de caso de uso, por sua vez, coordena a produção da resposta esperada ao interagir com a camada do domínio.

Portanto, repare que o FC e os comandos residem na camada da apresentação, enquanto os controladores de casos de uso são objetos da camada da aplicação. De fato, para evitar uma possível confusão de nomes, alguns autores Martin Fowler e Eric Evans denominam controladores de casos de uso de classes de serviço.

Por fim, repare que nem sempre haverá uma camada da aplicação. Na situação em que a equipe responsável pela arquitetura decidir não criar uma camada da aplicação, as responsabilidades antes dessa camada são movidas para os comandos. Esta não é a situação ideal do ponto de vista de separação de responsabilidades, mas é uma prática frequentemente utilizada, principalmente para aplicações não tão complexas.

11.2 Arquitetura física

A arquitetura física (ou *arquitetura de implantação*) diz respeito à disposição dos subsistemas de um SSOO pelos nós de processamentos disponíveis. Para sistemas simples, que executam em um único nó de processamento, a definição da arquitetura de implantação não faz sentido. No entanto, para sistemas mais complexos, é fundamental conhecer quais são os componentes físicos do sistema, quais são as interdependências entre eles e de que forma as camadas lógicas do sistema são dispostas por esses componentes.

Nesta seção, discutimos aspectos relativos à implantação física de um SSOO. Outro aspecto que descrevemos aqui é o suporte da UML para o projeto da implantação física de um SSOO. O diagrama de implementação da UML é utilizado para representar a arquitetura física de um sistema. O modelo construído a partir desse diagrama é denominado modelo de implementação. Esse modelo também é denominado modelo da arquitetura física. Há dois tipos de diagramas de implementação que são descritos nesta seção: o diagrama de implantação e o diagrama de componentes.

11.2.1 Alocação de camadas lógicas aos nós de processamento

Na implantação física de um sistema construído segundo a arquitetura a cliente-servidor, é comum utilizar as definições das camadas lógicas como um guia para a alocação física dos subsistemas pelos nós de processamento existentes. Sendo assim, a cada nó de processamento são alocadas uma ou mais camadas lógicas. Note que, nesta seção, o termo *camada* é utilizado com dois sentidos diferentes: para indicar uma camada lógica (conforme a Seção 11.1.1) ou uma camada física, esta última normalmente associada a um nó de processamento. Em inglês, esses significados normalmente correspondem aos termos chamados de *layers* e de *tiers*, respectivamente (embora também haja confusão de uso dos mesmos). Nesta seção, o contexto da utilização do termo *camada* deve ajudar a eliminar qualquer eventual ambiguidade com relação ao significado.

A alocação das camadas lógicas a diferentes nós de processamento possui diversas vantagens, conforme descrito a seguir.

- Em primeiro lugar, a divisão dos objetos permite um maior grau de manutenção e reutilização desses objetos, porque sistemas de software construídos em camadas podem ser mais

facilmente estendidos. Por exemplo, se surgir a necessidade de construir uma versão de sistema preexistente para funcionar no ambiente da Internet, é necessário apenas adicionar uma nova camada de apresentação. Em tese, as demais camadas não precisariam de modificação, e a nova versão do sistema poderia utilizar as mesmas camadas mais internas que as utilizadas pela versão anterior.

- Os sistemas em camadas também são mais adaptáveis a uma quantidade maior de usuários (comparativamente à arquitetura cliente-servidor em duas camadas). De fato, na arquitetura em camadas, servidores novos ou mais potentes podem ser acrescentados para compensar um eventual crescimento no número de usuários do sistema.

No entanto, a divisão do sistema em camadas apresenta a desvantagem de *potencialmente* diminuir seu desempenho: a cada camada, as representações dos objetos sofrem modificações, e essas modificações levam tempo para serem realizadas.

Na implantação física de um sistema construído segundo a arquitetura cliente-servidor, a camada de apresentação é alocada na máquina do usuário e é normalmente responsável pela interface gráfica com o usuário. O servidor (em que estão alocadas as camadas lógicas subjacentes) é normalmente executado em outra máquina, que possui uma capacidade de processamento maior e pode servir a diversos clientes. No entanto, nada impede que possa haver outras configurações de alocação entre cliente e servidor. Com efeito, dependendo da carga de processamento destinada a ele, um cliente pode ser *magro* ou *gordo*.

Um *cliente magro* (tradução para *thin client*) possui apenas a camada de apresentação, que representa o papel de cliente na comunicação com os demais subsistemas. O processamento da lógica da aplicação (cálculos, regras de negócio, validações de campos etc.) está no servidor (demais camadas). O cliente fica com a responsabilidade de prover a interface gráfica com o usuário.

Um *cliente gordo* (tradução para *fat client*), também conhecido como *cliente rico* (tradução para *rich client*), se caracteriza, por outro lado, por conter a interface gráfica com o usuário e a maior parte da (ou toda) lógica da aplicação; nesse caso, o servidor funciona como um repositório de dados.

Independentemente de o cliente ser gordo ou magro, um SSOO que divide a interação com o usuário e o acesso aos dados em dois subsistemas é denominado *sistema cliente-servidor em duas camadas*. Sistemas cliente-servidor em duas camadas foram dominantes durante aproximadamente toda a década de 1990. A construção de sistemas em duas camadas é vantajosa quando o número de clientes não é tão grande (p. ex., uma centena de clientes interagindo com o servidor por uma rede local). No entanto, um acontecimento importante fez com que sistemas cliente-servidor em duas camadas se tornassem obsoletos: o surgimento da Internet. Esse acontecimento gerou uma demanda pela construção de sistemas de software que pudessem ser utilizados pela Internet. Isso causou problemas em relação à estratégia cliente-servidor em duas camadas, principalmente quanto à construção de clientes gordos. Isso porque a ideia básica da Internet é permitir o acesso a variados recursos por meio de um *programa navegador* (*browser*), que não fornece grande suporte à construção daquele tipo de cliente.

A solução encontrada para o problema da arquitetura em duas camadas foi simplesmente dividir o sistema em mais camadas de software. Sistemas construídos segundo essa estratégia foram denominados *sistemas cliente-servidor em três camadas* ou *sistemas cliente-servidor em quatro camadas*. Entretanto, a ideia básica original permanece: dividir o processamento do sistema em

diversos nós. Em particular, uma disposição bastante popular, principalmente em aplicações para a WEB, é a *arquitetura cliente-servidor em três camadas* (ver [Figura 11-6](#)). Nessa arquitetura, a camada lógica de apresentação fica em um nó de processamento (conhecido como *presentation tier*). As camadas lógicas da aplicação e do domínio ficam juntas em outro nó (correspondente à camada física denominada *middle tier*). Essa camada do meio é normalmente associada ao *servidor da aplicação*. A camada física de apresentação se comunica (requisita serviços) com o servidor da aplicação. Aliás, é possível a situação em que há mais de um servidor de aplicação que os clientes podem ter acesso por intermédio da camada física de apresentação, com o objetivo de aumentar a disponibilidade e o desempenho da aplicação. Finalmente, a camada do meio faz acesso a um terceiro nó de processamento, no qual está executando um mecanismo de armazenamento persistente, normalmente representado por um sistema de gerenciamento de bancos de dados (SGBD). Esta última camada física é normalmente chamada de *camada de dados* (tradução para *data tier*).



Figura 11-6: Esquema de uma arquitetura cliente-servidor em três camadas.

Uma vez que definimos as alocações das camadas lógicas aos nós de processamento, como podemos representar isso de forma gráfica? A UML dá suporte a essa representação por meio do seu diagrama de implantação (*deployment diagram*). Esse diagrama representa a topologia física do sistema e, opcionalmente, os componentes que são executados nessa topologia. Pode-se dizer que esse diagrama apresenta um mapeamento entre os componentes de software e o hardware utilizados pelo sistema.

Os elementos de um diagrama de implantação são os *nós* e as *conexões*. Um nó é uma unidade física que representa um recurso computacional e normalmente possui uma memória e alguma capacidade de processamento. Quando um sistema está em execução, seus componentes residem em nós. Há diversos tipos de nós: processadores, dispositivos, sensores, roteadores ou qualquer objeto físico de importância para o sistema de software. Graficamente, um nó é representado por um cubo. O nome e o tipo do nó são definidos no interior desse cubo.

A sintaxe para o nome e o tipo do nó é análoga à utilizada para os diagramas de objetos (ver

[Seção 5.3](#)): o nome e o tipo são sublinhados e separados um do outro por um sinal de dois pontos. Tanto o nome quanto o tipo são opcionais. Os nós são ligados uns aos outros por meio de conexões. As conexões mostram mecanismos de comunicação entre os nós: meios físicos (cabo coaxial, fibra ótica etc.) ou protocolos de comunicação (TCP/IP, HTTP etc.). Uma conexão é representada graficamente por uma linha ligando dois nós. Para denotar o tipo de comunicação ao qual ela corresponde, uma conexão pode ser estereotipada.

A [Figura 11-7](#) fornece um exemplo que ilustra a notação da UML para os elementos de um diagrama de implantação. Esse diagrama informa que, no sistema em questão, há computadores pessoais se comunicando por intermédio do protocolo HTTP ao servidor de aplicação, que, por sua vez, se comunica com o sistema de gerência de banco de dados via ODBC.

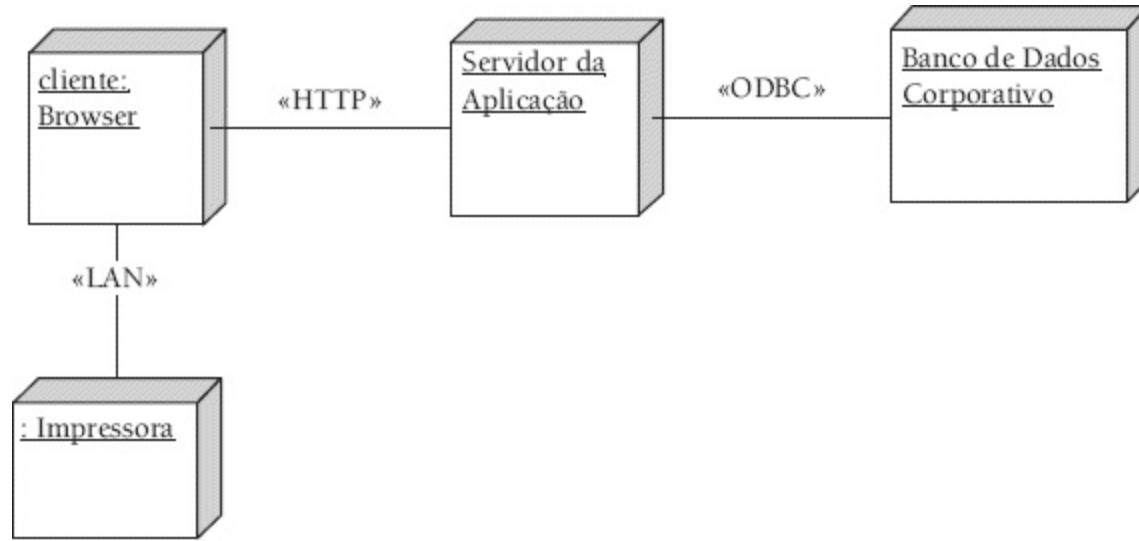


Figura 11-7: Exemplo de diagrama de implantação.

11.2.2 Alocação de componentes aos nós de processamento

Uma tarefa relacionada à alocação de camadas lógicas em nós de processamento é aquela cujo propósito é definir quais são os *componentes* ([Seção 8.1](#)) alocados em cada nó de processamento e quais são as dependências entre eles. Nesta seção, descrevemos a notação da UML que dá suporte a essa representação, assim como apresentamos diversos fatores que influenciam na alocação de componentes a nós de processamento.

A UML define o diagrama de componentes, cuja finalidade é apresentar os vários componentes de software e suas dependências. Os elementos gráficos que podem ser usados nesse diagrama são ilustrados na [Figura 11-8](#). Mais à esquerda, temos o símbolo de componente. O nome do componente pode ser posicionado dentro ou abaixo da caixa. Os dois símbolos mais à direita já são conhecidos: o de interface e o de dependência. Um componente pode realizar *interfaces* (ver [Seção 8.5.4](#)) e utilizar os serviços de outros componentes por meio das interfaces deles. Um relacionamento de dependência é utilizado para ligar um componente a outro componente, quando um é consumidor e o outro é produtor de algum serviço.

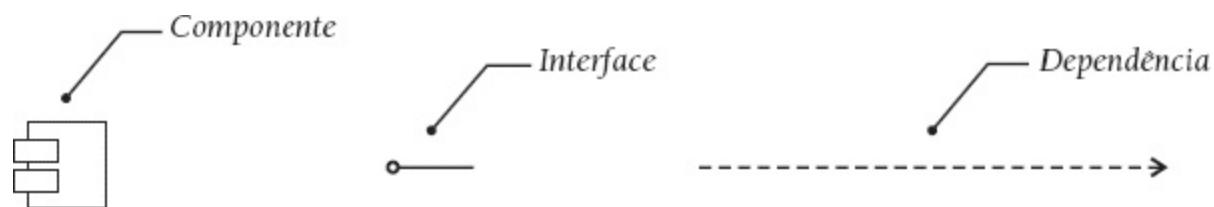


Figura 11-8: Elementos da notação utilizada para diagramas de componentes.

A [Figura 11-9](#) apresenta um exemplo de diagrama de componente. Para entender esse exemplo, considere que foi definido um novo caso de uso no SCA, denominado Montar Grade de Horários. Como o próprio nome diz, a finalidade desse caso de uso é permitir que o coordenador do curso (i.e., o ator desse caso de uso) monte a grade de horários para um determinado período letivo. Uma grade de horários é composta de turmas, cada uma com horários e locais de aulas definidos, assim como de um professor alocado para cada turma. O processo de definição da grade de horários é bastante complexo e por conta disso foi alocado a um componente, denominado PlanejadorGradeHorários. Para usar o serviço desse componente, seu cliente (o controlador do caso de uso, cuja classe é denominada MontarGradeHorariosControlador) deve fornecer as disponibilidades e restrições que precisam ser aplicadas durante a montagem. Ele pode fazer isso por meio da instanciação de um objeto cuja classe implemente a interface Recursos, que é utilizada (requerida) pelo componente. Além disso, o componente implementa a interface PlanoGradeHorários, que retorna uma possível grade de horários consistentes com os recursos e restrições fornecidas. Quando a operação getAlocacoes é invocada, o componente invoca as operações getDisponibilidades e getRestricoes para obter os dados para montagem da grade de horários. Após montar a grade, o componente retorna o resultado do processamento por meio de um objeto da classe Alocacoes.

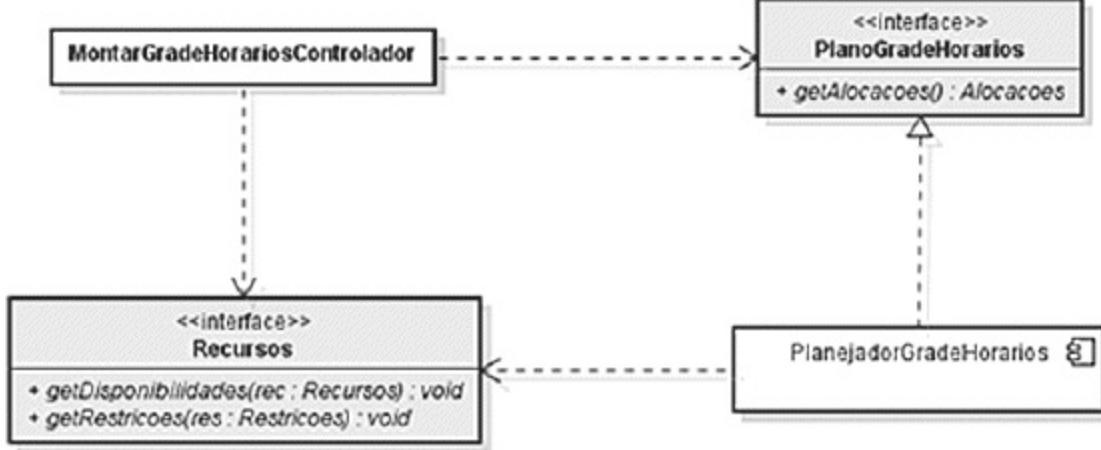


Figura 11-9: Exemplo de diagrama de componentes.

Um dos principais objetivos da alocação de componentes aos nós físicos é distribuir a carga de processamento do sistema. No entanto, nem sempre isso aumenta o desempenho, pois a sobrecarga de comunicação entre os nós de processamento pode anular qualquer ganho obtido com a distribuição do processamento. O envio de uma mensagem entre objetos dentro de um processo executando em um nó de processamento é bastante rápido. No entanto, uma mensagem enviada entre processos no mesmo nó pode ser executada em um intervalo de tempo até duas vezes maior. Mais ainda, se a mensagem ultrapassar as fronteiras de um nó para ser executada em outra máquina, a lentidão provavelmente aumentará mais uma ou duas ordens de grandeza, dependendo das características da rede de comunicação entre os nós (FOWLER, 2002). Portanto, durante a alocação de componentes, o modelador deve considerar diversos fatores de desempenho, sendo que muitos deles se sobrepõem ou são incompatíveis. Alguns desses fatores são listados a seguir.

- Utilização de dispositivos: considera a distribuição física dos dispositivos de entrada e saída pertencentes ao sistema e de que forma os componentes os utilizam.

- Carga computacional: esse fator considera a necessidade de processamento simultâneo de dois ou mais componentes.
- Capacidade de processamento dos nós: provavelmente alguns nós são computacionalmente mais potentes que outros. Os componentes com exigências mais urgentes de processamento devem ser alocados aos nós mais rápidos.
- Realização de tarefas: considera qual a comunicação dos componentes em relação a um ou mais processos intimamente relacionados em um nó. Quanto mais relacionados dois componentes (processos) estão, maior é a probabilidade de eles serem alocados ao mesmo nó.
- Tempo de resposta: leva em conta o tempo de resposta do sistema. Envolve alocar componentes de tal forma que o desempenho seja o máximo possível e que as dependências entre eles não cruzem as fronteiras de um nó. A ideia é minimizar o tráfego pelos canais de comunicação entre nós.
- Há também fatores não relacionados ao desempenho, mas que também influenciam na alocação dos componentes. Alguns desses fatores são listados a seguir.
- Outros requisitos não funcionais do sistema: pode haver algum requisito não funcional que restrinja a localização de certos componentes. Por exemplo, talvez o sistema deva se comunicar com um sistema legado¹ que se localiza em uma máquina específica e não pode ser realocado.
- Segurança: envolve alocar componentes de acordo com critérios de segurança.
- Diferenças de plataformas (de hardware ou de sistema operacional) dos componentes do sistema. Por exemplo, pode ser que um componente somente seja executado em plataforma *Windows* ou somente em *Linux*.
- Características dos usuários do sistema: leva em conta a localização física dos usuários, que máquinas eles utilizam, como estão conectados etc.
- Necessidade ou benefícios da distribuição das camadas lógicas do sistema. Por exemplo, foi decidido que o sistema será em duas camadas (cliente-servidor), ou em três camadas que devem ser alocadas a nós diferentes.
- Redundância: o mesmo componente pode ter de ser alocado a mais de um nó. Isso normalmente ocorre em aplicações que devem ter alta tolerância a falhas. Dessa forma, se um dos nós não está disponível, o sistema pode obter acesso aos serviços do componente em outros nós.

11.2.3 Padrões e tecnologias para distribuição de objetos

Quando falamos em uma aplicação não distribuída, uma requisição de um objeto a outro não ultrapassa os limites do nó de processamento no qual o sistema está hospedado. Entretanto, atualmente são comuns as aplicações distribuídas nas quais as partes (subsistemas) componentes estão fisicamente separadas em diferentes nós de processamento. Nesse último caso, surge o seguinte complicador: como dois objetos residentes em partes fisicamente separadas da aplicação podem trocar mensagens? Por exemplo, considere uma situação em que os objetos componentes da interface gráfica com usuários estão localizados em um nó de processamento, e os demais objetos da aplicação estão em outro nó.

Uma alternativa para resolver esse problema é com o uso do padrão de projeto denominado **Proxy** (GAMMA et al., 2004). Esse padrão recomenda a criação de **classes procuradoras** (tradução

para proxy classes). Classes procuradoras são aquelas cuja responsabilidade é fazer com que mensagens entre objetos cruzem as fronteiras entre os nós de processamento por meio de uma rede de comunicação. O que acontece é que um objeto residente em um nó de processamento pode enviar uma mensagem para um procurador, que, por sua vez, cuida de repassar essa mensagem para um objeto residente em outro nó de processamento, de forma transparente para o remetente. Ou seja, para o objeto remetente, tudo se passa como se o receptor estivesse residente no nó de processamento local.

Outro padrão de projeto encontrado no contexto de distribuição de objetos é o **Data Transfer Object** (DTO). Esse padrão é utilizado quando as informações que estão espalhadas por diversos objetos relacionados devem ser enviadas a outro nó de processamento. Por exemplo, considere a classe Turma no SCA. Considere ainda que, para cada turma, informações de código da turma, nome da disciplina e nome e matrícula do professor responsável devam ser enviadas para outro nó de processamento. Nesse contexto, poderia ser criada uma classe denominada TurmaInfoDTO que contivesse apenas os atributos correspondentes às informações necessárias. Uma lista de objetos desse tipo seria, então, criada no nó de processamento de origem e enviada ao nó de processamento de destino. Essa abordagem tem a vantagem de evitar problemas de desempenho, pois apenas as informações necessárias dos objetos de domínio envolvidos são trafegadas de um nó a outro.

Duas tecnologias que podem ser utilizadas para implementar classes procuradoras são RMI (*Remote Method Invocation*) e CORBA (*Common Object Request Broker Architecture*). Uma estratégia de distribuição alternativa para o uso de classes procuradoras é copiar objetos de um nó de processamento para outro. Dessa forma, cópias do mesmo objeto são distribuídas e, assim, cada cópia pode receber mensagens sem que haja a necessidade de transporte da requisição pela rede de comunicação (como acontece com o uso de objetos procuradores). Entretanto, a cópia de objetos tem a desvantagem de poder ocasionar inconsistência entre as cópias criadas.

11.3 Projeto da arquitetura no processo de desenvolvimento

A construção dos diagramas de componentes é iniciada na fase de elaboração e refinada na fase de construção de um processo de desenvolvimento iterativo. Note, contudo, que nem sempre a construção dos diagramas aqui descritos é necessária. O fator determinante nessa decisão é a *complexidade* do SSOO em questão. A construção de diagramas de componentes se justifica para componentes de execução. A sua utilização nesse caso permite visualizar as dependências entre os componentes e a utilização de interfaces a tempo de execução do sistema. Se o sistema for executado em modo distribuído (em vários nós de uma rede), o melhor a fazer é representar os seus componentes utilizando os diagramas de implantação. Por outro lado, não é adequado construir diagramas de componentes para representar dependências de compilação entre os elementos do código-fonte do sistema. Isso porque a maioria dos ambientes de desenvolvimento tem capacidade de manter as dependências entre códigos-fonte, códigos-objeto, códigos executáveis e páginas de script.

Em relação ao diagrama de implantação, sua construção tem início na fase de elaboração. Na fase de construção, os componentes são adicionados aos diversos nós. Cada versão do sistema corresponde a uma versão do diagrama de implantação, que exibe os componentes utilizados na construção daquela versão. O diagrama de implantação deve fazer parte dos manuais para instalação e operacionalização do sistema. Note que nem todo sistema necessita de um modelo de implementação completo. Por exemplo, sistemas implantados em um único computador não necessitam de diagramas de implantação (embora possam precisar de diagramas de componentes).

Se ele for bastante simples, o modelo de implementação é completamente desnecessário. Ou seja, a atividade de alocação de componentes aos nós físicos só tem sentido para sistemas distribuídos. Para aqueles que utilizam um único processador, não há necessidade dessa atividade.

O arquiteto de software também deve examinar a tecnologia que tem em mãos durante a definição do projeto da arquitetura do sistema. De fato, pode haver algumas restrições relativas à tecnologia que já foram declaradas como requisitos não funcionais (ver [Seção 2.1.1](#)).

A alocação de mão de obra especializada em um processo de desenvolvimento iterativo pode ser feita com base na arquitetura lógica definida para o sistema. De fato, a alocação de desenvolvedores pode ser feita de acordo com a especialidade de cada um: projetistas da interface gráfica ficam na camada da apresentação, outro grupo pode ser alocado ao desenvolvimento da camada de persistência, e assim por diante. Essa estratégia ajuda a especializar o trabalho e a aumentar a produtividade do desenvolvimento.

► EXERCÍCIOS

11-1: Frequentemente, diagramas de implementação são desenhados com estereótipos gráficos que lembram os elementos do sistema. Por exemplo, pode-se encontrar um diagrama de implantação que apresenta ícones para computadores pessoais, servidores de bancos de dados, subsistemas de monitoração de tráfego (*firewalls*) etc. Discuta as vantagens ou desvantagens dessa abordagem.

11-2: Realize uma pesquisa sobre o padrão de projeto denominado *Active Record*. Qual o propósito dele? Em que situações pode ser aplicado?

11-3: Na [Seção 11.1.2](#), apresentamos as camadas típicas de um sistema de informação. Outra proposta de organização arquitetural é a denominada *Onion Architecture* (Arquitetura Cebola). Realize uma pesquisa sobre essa proposta. Quais são seus diferenciais em relação à organização que apresentamos na [Seção 11.1.2](#)? De que forma ela pode ser aplicada em sistemas de informação?

11-4: Realize uma pesquisa sobre o termo Smart UI no contexto do DDD (*Domain Driven Design*). Qual a relação desse termo com o conceito de arquitetura lógica apresentado neste Capítulo? E com o padrão MVC?

¹. Esses são sistemas preeexistentes, normalmente não orientados a objetos, com os quais a aplicação em desenvolvimento deve se comunicar. Sistemas legados normalmente são encapsulados em um conjunto de *classes de interface* para que possam ser utilizados.

Mapeamento de objetos para o modelo relacional

Na época, Nixon estava normalizando as relações com a China. Eu pensei que, se ele podia normalizar relações, eu também podia.

— E.F. CODD

Aquestão do mapeamento de objetos para o modelo relacional é um problema relevante por dois motivos. Em primeiro lugar, porque a tecnologia de orientação a objetos se consolidou como a forma usual de desenvolver sistemas de software. Em segundo, porque a tecnologia de bancos de dados relacionais é uma das tecnologias que tiveram maior êxito na área de computação¹ e sem dúvida os sistemas de gerência de bancos de dados relacionais (SGBDR) dominam o mercado comercial.

No entanto, essas duas tecnologias nasceram de princípios teóricos bastante diferentes. A tecnologia de orientação a objetos se baseia no princípio do Encapsulamento (ver Seção 1.2.3.1), que diz que objetos são abstrações de um comportamento, não importando como são representados. Além disso, sistemas de software orientados a objetos são construídos utilizando-se os objetos, que são constituídos de dados e de funções. Por outro lado, a tecnologia relacional lida com o armazenamento de dados tabulares. Essas diferenças fundamentais entre as tecnologias OO e relacional se refletem em termos práticos quando da construção de sistemas de software orientados a objetos que utilizam um SGBDR. O **descasamento de informações** (*impedance mismatch*) é um termo utilizado para denotar o problema das diferenças entre as representações do modelo de objetos e do modelo relacional. Uma proporção significativa do esforço de desenvolvimento recai sobre a solução que o programador deve dar a este problema.

Os princípios básicos do paradigma da orientação a objetos e do modelo relacional são bastante diferentes. No modelo de objetos, os elementos (objetos) correspondem a abstrações de comportamento. No modelo relacional, os elementos correspondem a dados no formato tabular.

Um aspecto importante a considerar sobre classes de um SSOO é identificar quais delas geram objetos que devem ser persistentes. De fato, objetos de um sistema de software podem ser classificados em dois tipos, *objetos transientes* e *persistentes*. Um objeto transiente existe somente durante uma sessão de uso do sistema. Objetos de controle, objetos DAO, serviços do domínio e objetos de fronteira (ver Seção 5.4.2) são alguns exemplos de objetos transientes. Por outro lado, objetos persistentes têm uma existência que perdura durante várias sessões de uso do sistema. Objetos de classes do domínio são normalmente persistentes. Objetos persistentes precisam ser armazenados quando uma execução do sistema termina e restaurados quando outra é iniciada. Durante este capítulo, o termo *objeto* diz respeito a objetos persistentes.

Neste capítulo, apresentamos detalhes a respeito dos diversos aspectos relativos ao mapeamento

de objetos para um mecanismo de armazenamento persistente. Nossa descrição focaliza principalmente o contexto particular em que esse mecanismo de armazenamento é representado por um SGBDR. Na [Seção 12.1](#), descrevemos as regras para realizar o projeto de banco de dados a partir de um modelo de classes do domínio. Na [Seção 12.2](#), apresentamos detalhes da construção da camada de persistência de um SSOO, cujo propósito é permitir a movimentação (em duas vias) de objetos entre o espaço de memória do SSOO para o mecanismo de armazenamento.

12.1 Projeto de banco de dados

Uma das principais atividades do projeto detalhado (ver [Seção 2.1.3](#)) é o desenvolvimento do banco de dados a ser utilizado, se este não existir. Essa atividade normalmente é chamada de *projeto de banco de dados*. Tal projeto envolve diversas atividades, algumas delas enumeradas a seguir:

- Construção do esquema do banco de dados.
- Criação de índices² para agilizar o acesso aos dados armazenados.
- Definição das estruturas de dados a serem utilizadas no armazenamento físico dos dados.
- Definição de visões sobre as dados armazenados.
- Atribuição de direitos de acesso (que usuários podem acessar que recursos).
- Definição de políticas de backup dos dados.

Pode-se notar pela lista anterior que o projeto de banco de dados, por si só, é uma atividade bastante extensa. A descrição de todos esses assuntos está fora do escopo deste livro. Neste capítulo são considerados apenas os aspectos de mapeamento de informações entre as tecnologias de orientação a objetos e relacional. Mais especificamente, é descrito como realizar o mapeamento do modelo de classes para o modelo relacional, uma atividade que resulta em um *esquema de banco de dados*, ou seja, na criação de um conjunto de representações que podem ser diretamente definidas no SGBD para a criação do banco de dados.

Na verdade, existem ferramentas CASE (ver [Seção 2.6](#)) que fornecem a funcionalidade de mapeamento *automático* para a criação de um esquema relacional. A partir do modelo de classes e da definição do SGBDR a ser utilizado, a própria ferramenta CASE gera o esquema do banco de dados correspondente. Há, inclusive, ferramentas que, dado um banco de dados armazenado em um SGBDR, conseguem gerar diagramas representando esse banco. Entretanto, a discussão neste capítulo ainda é relevante, pois nem sempre uma ferramenta CASE está disponível para a equipe de desenvolvimento. Além disso, mesmo na existência de uma ferramenta, é importante que seu usuário tenha um conhecimento básico dos procedimentos existentes para a realização do mapeamento.

Antes da descrição das regras de mapeamento, a [Seção 12.1.1](#) faz uma descrição sucinta do modelo relacional, somente para estabelecer um contexto. Uma descrição mais detalhada deste modelo está fora do escopo deste livro e pode ser encontrada em ELMASRI e NAVATHE (2011).

12.1.1 Conceitos do modelo de dados relacional

O modelo relacional se fundamenta no conceito de *relação*. De forma bastante simplista, pode-se pensar em uma relação como uma tabela, composta de linhas e de colunas. Cada relação possui um nome. Cada coluna de uma relação possui um nome e um domínio. A [Tabela 12-1](#) ilustra o uso de relações para armazenar informações sobre departamentos, empregados, projetos e alocações de projetos de uma empresa.

Tabela 12-1: Exemplos de relações. Chaves primárias estão sublinhadas e chaves estrangeiras estão tracejadas

Departamento			
id	sigla	nome	<u>idGerente</u>
13	RH	Recursos Humanos	5
14	INF	Informática	2
15	RF	Recursos Financeiros	6

Empregado						
id	matrícula	CPF	nome	endereço	CEP	<u>idDepartamento</u>
1	10223	038488847-89	Carlos	Rua 24 de Maio, 40	22740-002	13
2	10490	024488847-67	Marcelo	Rua do Bispo, 1.000	22733-000	13
3	10377	NULL	Adelci	Av. Rio Branco, 9	NULL	NULL
4	11057	0345868378-20	Roberto	Av. Apiacás, 50	NULL	14
5	10922	NULL	Aline	R. Uruguaiana, 50	NULL	14
6	11345	0254647888-67	Marcelo	NULL	NULL	15

Projeto		
id	nome	verba
1	PNADO	R\$ 7.000
2	BMMO	R\$ 3.000
3	SGILM	R\$ 6.000
4	ACME	R\$ 8.000

Alocação		
id	<u>idProjeto</u>	<u>idEmpregado</u>
100	1	1
101	1	2
102	2	1
103	3	5
104	4	2

No modelo relacional, cada coluna pode conter apenas *valores atômicos*. Ou seja, uma coluna de uma relação não pode armazenar uma informação estruturada (exceção feita aos tipos de dados para datas, que existem na maioria dos SGBDR).

Um conceito importante no modelo relacional é o de *chave primária*. Uma chave primária é uma coluna ou conjunto de colunas cujos valores podem ser utilizados para identificar unicamente cada linha de uma relação. Note que todas as relações apresentadas na Tabela 12-1 possuem uma coluna chamada *id*. Essa coluna é a chave primária de cada relação na qual aparece.

Outro conceito importante do modelo relacional é o de *chave estrangeira*. Linhas de uma relação podem estar associadas às de outras relações. Estas associações são representadas pela única

maneira possível, considerando-se os recursos de notação do modelo relacional: deve existir, em uma das duas relações, uma coluna cujos valores fazem referência aos de uma coluna da outra relação. Na terminologia do modelo relacional, esta coluna de referência é denominada *chave estrangeira*.

Considere novamente a [Tabela 12-1](#). Note que Empregado tem uma coluna `idDepartamento`. Esta coluna é um exemplo de chave estrangeira. Note que os valores nessa coluna estão contidos no conjunto de valores da coluna `id` de Departamento. Em cada relação apresentada na [Tabela 12-1](#), as chaves estrangeiras estão tracejadas.

Além de conter valores que podem ser encontrados em uma chave primária, uma chave estrangeira também pode conter *valores nulos*. Um valor nulo é representado pela constante `NULL` em um SGBDR. Esse valor normalmente é usado para indicar que um valor não se aplica, ou é desconhecido, ou não existe. Por exemplo, na relação Empregado, o empregado Aldeci contém o valor `NULL` para a chave estrangeira `idDepartamento`. Isso significa que este empregado não está associado a departamento algum.

12.1.2 Mapeamento de objetos para o modelo relacional

Quando um SGBDR deve ser utilizado como *mecanismo de armazenamento persistente* de informações para um sistema de software orientado a objetos, há a necessidade de se realizar o mapeamento dos valores de atributos de objetos persistentes do sistema para tabelas.

É a partir do modelo de classes que o mapeamento de objetos para o modelo relacional é realizado. Esse procedimento de mapeamento é bastante semelhante ao de mapeamento do Modelo de Entidades e Relacionamentos (MER) para o modelo relacional. De fato, as regras de mapeamento a partir do MER são bastante semelhantes às utilizadas no mapeamento a partir do modelo de classes. No entanto, em virtude de o modelo de classes possuir mais recursos de representação que o MER, regras adicionais são definidas.

Entretanto, é importante enfatizar que o MER e o modelo de classes *não* são equivalentes. Principalmente para desenvolvedores iniciantes, esses modelos são frequentemente confundidos. Isso talvez se deva à semelhança sintática dos dois diagramas.³ No entanto, essa semelhança existe somente no nível sintático. O MER é um modelo de dados, enquanto o modelo de classes modela objetos (dados e comportamento). Para maiores detalhes, ver [Seção 5.4.6](#).

As próximas seções discutem o procedimento de mapeamento de diversos elementos do modelo de classes para o modelo relacional. Durante essa discussão, os exemplos de diagramas de classes não apresentam o comportamento das operações de cada retângulo de classe. Isso porque as operações não têm relevância para o mapeamento aqui descrito.

O leitor deve atentar para a notação utilizada neste capítulo para representar relações. Cada relação é representada pelo seu nome e pelos nomes de suas colunas entre parênteses. Além disso, as chaves primárias são sublinhadas, e as estrangeiras são tracejadas. Uma definição mais completa do modelo relacional conteria o tipo de dados de cada coluna e as restrições de integridade sobre as mesmas.

Além disso, nos exemplos de mapeamento apresentados a seguir, utiliza-se sempre uma *coluna de implementação* como chave primária de cada relação. Uma coluna de implementação é um identificador sem significado no domínio de negócios. Essa abordagem é utilizada para manter uma padronização nos exemplos e por ser uma das melhores maneiras de associar identificadores a objetos mapeados para tabelas. Por convenção, a coluna de implementação usada em nossos

exemplos é denominada *id*.

12.1.3 Classes e seus atributos

Classes são mapeadas para relações. O caso mais simples é o de mapear cada classe como uma relação e cada um de seus atributos como uma coluna da relação correspondente. No entanto, muito frequentemente não há uma correspondência unívoca entre classes e relações. Pode ser que várias classes sejam mapeadas para uma única relação ou que uma classe seja mapeada para várias relações.

Para atributos o que vale de forma geral é que *um atributo será mapeado para uma ou mais colunas*. Lembre-se, também, de que nem todos os atributos são persistentes. Por exemplo, pode ser que uma classe Pedido tenha um atributo derivado, total, utilizado para guardar o valor total a ser pago por um pedido, mas que este atributo não seja armazenado no banco de dados. Em geral, atributos derivados não são mapeados para o banco de dados. No entanto, por questões de desempenho, seu projetista pode optar por mapear esses atributos.

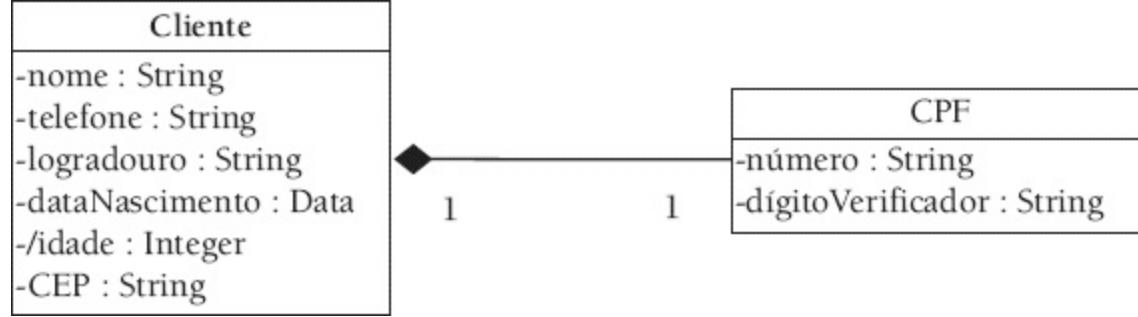


Figura 12-1: Classes Cliente e CPF.

Considere a classe Cliente ilustrada na [Figura 12-1](#). A [Tabela 12-2](#) exibe duas alternativas de mapeamento possíveis para essa classe. Note que, em ambos os casos, o atributo derivado *idade* não é mapeado. Além disso, nas duas alternativas, as classes Cliente e CPF são mapeadas para uma única relação. No entanto, na primeira alternativa, o CEP de um cliente é mapeado para uma relação separada e o atributo CEP da classe é dividido em duas partes (*número* e *dígitoVerificador*).

Tabela 12-2: Mapeamentos possíveis para a classe Cliente

1 ^a	Cliente(<u>id</u> , CPF, nome, telefone, logradouro, dataNascimento, <u>idCEP</u>) CEP(<u>id</u> , número, sufixo)
2 ^a	Cliente(<u>id</u> , nome, telefone, logradouro, dataNascimento, CPF, CEP)

O exemplo anterior também mostra que vários atributos podem ser mapeados para uma única coluna. Por exemplo, os atributos da classe CPF são ambos armazenados em uma coluna da relação Cliente.⁴

12.1.4 Associações

O procedimento utilizado para mapear associações utiliza o conceito de *chave estrangeira* (ver [Seção 12.1.1](#)), que são utilizadas para relacionar linhas de relações diferentes ou linhas de uma mesma relação.

Há três casos para mapeamento de associações, cada um correspondente a um tipo de *conectividade* (ver [Seção 5.2.2.1](#)).

Na discussão a seguir, considere, sem perda de generalidade, que há uma associação entre objetos de duas classes, C_a e C_b , e que as duas classes foram mapeadas para duas relações separadas, T_a e T_b . Considere, ainda, o diagrama de classes da [Figura 12-2](#) a ser utilizado nos exemplos desta seção.

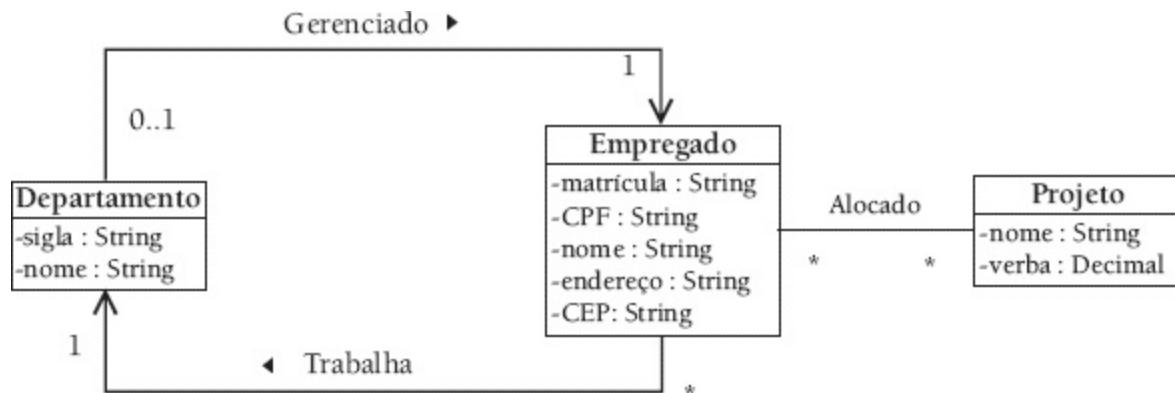


Figura 12-2: Diagrama de classes ilustrando os três tipos de conectividade em associações.

12.1.4.1 Associações de conectividade um-para-um

Quando há uma associação um-para-um entre C_a e C_b , deve-se adicionar uma chave estrangeira em uma das duas relações para referenciar a chave primária da outra relação.

Com respeito à escolha da relação na qual a chave estrangeira deve ser adicionada, deve-se observar se a *participação* (ver [Seção 5.2.2.2](#)) na associação é opcional ou obrigatória em ambos os extremos da associação. Há três possibilidades, enumeradas a seguir.

1. A associação é obrigatória em ambos os extremos.
2. A associação é opcional em ambos os extremos.
3. A associação é obrigatória em um extremo e opcional no outro extremo.

Nos casos (1) e (2), a escolha da relação na qual se deve adicionar a chave estrangeira é aleatória. No entanto, no caso (3), deve-se optar pela relação que corresponde à classe de participação obrigatória. Se isso for feito, a coluna de chave estrangeira sempre terá valores não nulos, ao contrário do que aconteceria se a outra relação fosse escolhida para adicionar a coluna de chave estrangeira.

A [Figura 12-2](#) apresenta um exemplo de associação um para um. O mapeamento dessa associação é ilustrado a seguir. Note que, no extremo de participação total, foi criada uma coluna de chave estrangeira, `idEmpregadoGerente`.

Departamento(`id`, sigla, nome, `idEmpregadoGerente`)
Empregado(`id`, matrícula, CPF, nome, endereço, CEP)

Como o objetivo de aumentar o desempenho de processamento, as classes que participam de uma associação um para um também podem ser mapeadas para uma única relação. Nesse caso, não há necessidade do uso de uma chave estrangeira. Os mapeamentos apresentados na [Figura 12-2](#) são exemplos dessa situação: há uma associação (agregação) de conectividade um para um, mas ambas

as classes são mapeadas para uma mesma relação.

12.1.4.2 Associações de conectividade um para muitos

Em uma associação um-para-muitos entre objetos de C_a e de C_b , seja C_a a classe na qual cada objeto se associa com muitos objetos da classe C_b . Neste caso, deve-se adicionar uma chave estrangeira em T_a para referenciar a chave primária de T_b .

Como exemplo, considere novamente a [Figura 12-2](#), na qual se apresenta a associação um para muitos Trabalha. A seguir é apresentada uma extensão do mapeamento anterior considerando essa associação.

Departamento(id, sigla, nome, idEmpregadoGerente)
Empregado(id, matrícula, CPF, nome, endereço, CEP, idDepartamento)

12.1.4.3 Associações de conectividade muitos para muitos

Quando há uma associação de conectividade muitos para muitos entre objetos de C_a e de C_b , uma *relação de associação* deve ser criada. Uma relação de associação tem o objetivo de representar a associação muitos para muitos entre duas ou mais relações.⁵

Diferentemente do mapeamento dos outros tipos de conectividade, a conectividade muitos para muitos exige a criação de uma nova relação. Nas outras conectividades, ao contrário, apenas adiciona-se uma nova coluna de chave estrangeira a uma das relações correspondentes às classes participantes da associação.

Seja T_{assoc} o nome da relação de associação. O mapeamento de uma associação muitos para muitos é feito aplicando-se a regra para o mapeamento um para muitos duas vezes, considerando-se separadamente os pares de relações (T_a, T_{assoc}) e (T_b, T_{assoc}).

Há duas alternativas para se definir a chave primária de T_{assoc} . Primeiramente, pode-se definir uma chave primária composta para T_{assoc} . Uma chave primária composta é formada por mais de uma coluna.

Uma segunda alternativa é criar uma coluna de implementação que sirva como chave primária simples da relação de associação. Estas duas alternativas são ilustradas a seguir, no mapeamento da associação Alocado (ver [Figura 12-2](#)).

1 ^a alternativa	Departamento(<u>id</u> , sigla, nome, <u>idEmpregadoGerente</u>) Empregado(<u>id</u> , matrícula, CPF, nome, endereço, CEP, <u>idDepartamento</u>) Alocação(<u>idProjeto</u> , <u>idEmpregado</u> , nome, verba) Projeto(<u>id</u> , nome, verba)
2 ^a alternativa	Departamento(<u>id</u> , sigla, nome, <u>idEmpregadoGerente</u>) Empregado(<u>id</u> , matrícula, CPF, nome, endereço, CEP, <u>idDepartamento</u>) Alocação(<u>id</u> , <u>idProjeto</u> , <u>idEmpregado</u> , nome, verba) Projeto(<u>id</u> , nome, verba)

Resumindo, o leitor pode notar que, independentemente do tipo de conectividade envolvida (um para um, um para muitos ou muitos para muitos), o conceito de chave estrangeira é sempre utilizado no mapeamento de associações. Além disso, embora os exemplos ilustrados nesta seção tenham envolvido apenas associações, as mesmas regras de mapeamento se aplicam a agregações e a composições.

12.1.5 Agregações e composições

Uma agregação (ou composição) é uma forma especial de associação. Portanto, o *mesmo* procedimento para realizar o mapeamento de associações pode ser utilizado para agregações e composições. No entanto, a diferença semântica entre uma associação e uma agregação (composição) influí na forma como o SGBDR deve agir quando um registro da relação correspondente ao todo precisa ser excluído ou atualizado.

Ou seja, pode ser necessário que, quando um objeto inteiro for removido, os objetos parte sejam também removidos. De fato, não faz sentido continuar armazenando os itens de pedido, de um pedido que já foi removido, por exemplo. Isso pode ser implementado em situações como quando se utilizam recursos de um SGBDR, como *gatilhos (triggers)* e *procedimentos armazenados (stored procedures)*. Para estudo adicional sobre estes assuntos, recomenda-se ler ELMASRI e NAVATHE (2011).

O padrão de acesso em agregações (composições) também é diferente do encontrado nas associações. Usualmente, quando um objeto todo deve ser restaurado, é natural restaurar também os objetos parte. Em associações, isso nem sempre é o caso. Nessa situação, a definição de *índices* adequados no SGBDR é importante para que o acesso aos objetos parte seja feito da forma mais eficiente possível.

12.1.6 Associações reflexivas

Uma associação reflexiva (ver [Seção 5.2.2.6](#)) é um tipo especial de associação. Portanto, o mapeamento aplicado a associações visto na [Seção 12.1.4](#) se aplica igualmente aqui.

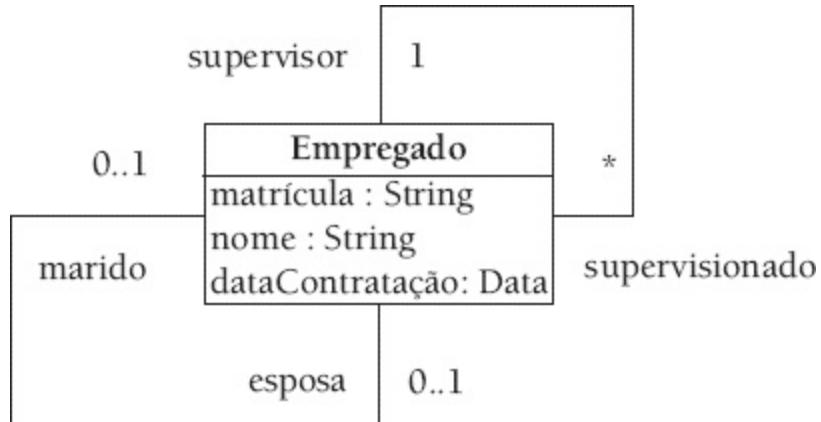


Figura 12-3: Associações reflexivas entre objetos da classe Empregado.

Para um exemplo, ver [Figura 12-3](#), que apresenta um fragmento de diagrama em que cada empregado tem um supervisor, ele próprio um outro empregado. Além disso, este diagrama representa relacionamentos de marido/esposa entre os empregados. Um possível mapeamento desse diagrama é apresentado a seguir. Note que as chaves estrangeiras idConjunge e idSupervisor foram definidas na relação **Empregado** como era esperado, visto que ambas as associações são reflexivas.

Empregado(id, matrícula, nome, dataContratação, idConjunge, idSupervisor)

No caso de uma associação reflexiva de conectividade muitos para muitos, uma relação de associação deve ser criada para implementar o mapeamento, no mesmo molde das associações

muitos para muitos não reflexivas (ver [Seção 12.1.4.3](#)).

12.1.7 Associações ternárias

Associações ternárias (ver [Seção 5.2.2.5](#)) e, de modo geral, associações n -árias, podem ser mapeadas por meio de um procedimento semelhante ao utilizado para associações binárias de conectividade muitos para muitos. Uma relação para representar a associação é criada e são adicionadas nelas chaves estrangeiras para as n classes participantes da associação. Se a associação ternária possuir uma classe associativa, os atributos dela são mapeados como colunas da relação de associação.

Como exemplo, considere a [Figura 12-4](#), na qual uma associação ternária é ilustrada. O mapeamento desse diagrama de classe é apresentado a seguir:

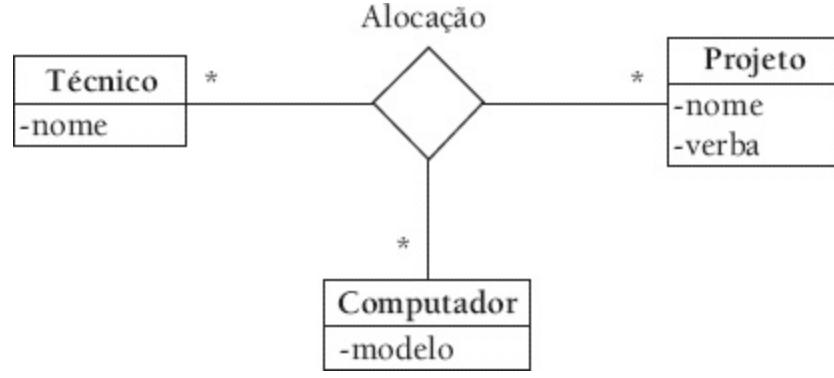


Figura 12-4: Associação ternária Projeto – Computador – Técnico.

12.1.8 Classes associativas

Uma classe associativa (ver [Seção 5.2.2.4](#)) é mais comumente encontrada em associações de conectividade muitos para muitos. No entanto, nada impede que uma classe associativa seja utilizada em associações de conectividade um para muitos ou um para um. Consequentemente, para cada um dos três casos de mapeamento de associações entre objetos (ver [Seção 12.1.4](#)), há uma variante em que uma classe associativa é utilizada.

De uma forma geral, o mapeamento de uma classe associativa é feito pela criação de uma relação para representá-la. Os atributos da classe associativa são mapeados para colunas dessa relação. Além disso, essa relação deve conter chaves estrangeiras que referenciem as relações correspondentes às classes que participam da associação.

Considere estender o diagrama de classes exibido na [Figura 12-2](#) para exemplificar o mapeamento de classes associativas. Essa extensão é ilustrada no diagrama da [Figura 12-5](#). Agora, para cada utilização de uma ferramenta por um projeto, a data dessa utilização é representada pela classe associativa Utilização. Além disso, são adicionados atributos para representar tanto a quantidade de horas semanais em que um empregado está alocado a um projeto, como sua remuneração em tal projeto. Esses atributos são cargaHorária e remuneração, respectivamente, e são representados em uma classe associativa Trabalho. No mapeamento, note que uma relação é criada para cada classe associativa.

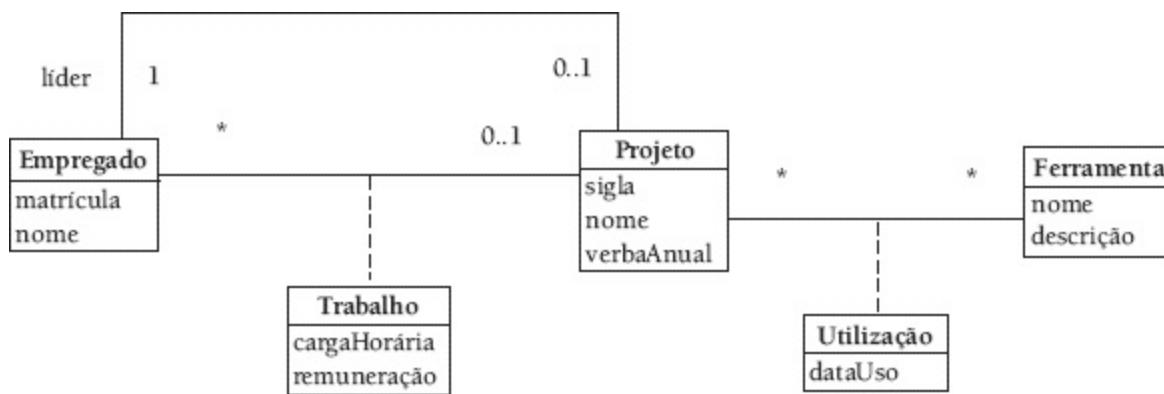


Figura 12-5: Diagrama de classe envolvendo o uso de classes associativas.

Há outra forma de realizar o mapeamento da classe associativa Trabalho: não criar uma relação, mas sim mapear os atributos dessa classe na relação correspondente à classe Empregado. Essa alternativa tem a desvantagem de apresentar um potencial desperdício de espaço, pois as colunas cargaHorária e remuneração não seriam utilizadas para empregados que não estivessem trabalhando em projeto algum.

12.1.9 Generalização

Há basicamente três alternativas de se mapear relacionamentos de generalização (AMBLER, 1997). Para a descrição dessa alternativas, considere a [Figura 12-6](#) que ilustra esquematicamente uma hierarquia de generalização, onde a superclasse, C_0 , possui diversas subclasses C_i , $1 \leq i \leq n$.

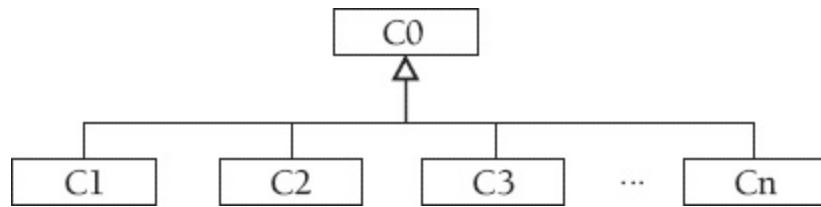


Figura 12-6: Hierarquia de generalização: a classe C_0 é superclasse de n outras classes.

12.1.9.1 Uma relação para cada classe da hierarquia

Nesta alternativa, uma relação é criada para cada uma das classes ($C_0, C_1, C_2, \dots, C_n$). A seguir, considera-se que há uma correspondência unívoca entre objetos de C_0 e objetos de C_i .⁶ Uma vez feita esta consideração de correspondência, pode-se aplicar a mesma regra de mapeamento aplicada para associações de conectividade “um para um” (ver [Seção 12.1.4.1](#)), sendo que a chave estrangeira deve ser adicionada na relação que implementa a subclasse.

12.1.9.2 Uma relação para toda a hierarquia

Nesta alternativa, cria-se uma única relação com colunas para representar os atributos da superclasse e também os atributos de todas as subclasses. Além disso, a relação deve conter outra coluna cujos valores servem para identificar, dado um objeto (linha) desta relação, a qual classe da hierarquia esse objeto pertence.

12.1.9.3 Uma relação para cada classe concreta da hierarquia

Nesta alternativa, para cada subclasse é criada uma relação. Como é de se esperar, os atributos específicos de cada subclasse correspondem a colunas na relação correspondente. Além disso, cada relação que implementa uma subclasse C_i também possui colunas para cada atributo herdado da superclasse C_o . Dessa forma, os atributos da superclasse são replicados pelas n relações que implementam as subclasses.

12.1.9.4 Exemplo de mapeamento de generalização

Vamos dar um exemplo para esclarecer melhor as alternativas de mapeamento de generalização. Considere o diagrama de classes exibido na [Figura 12-7](#), no qual as classes PessoaFísica e PessoaJurídica são subclasses da classe Contribuinte.

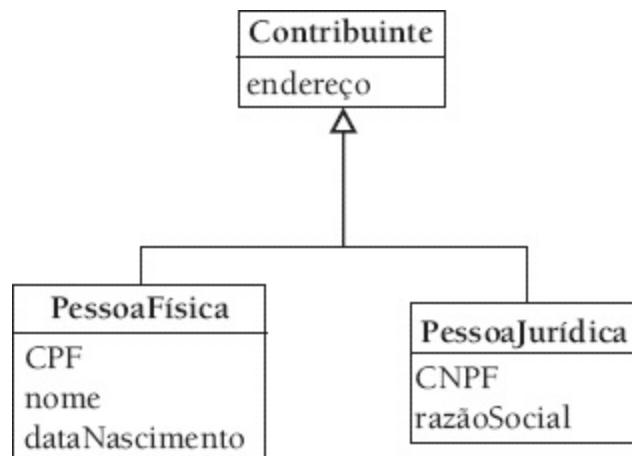


Figura 12-7: Hierarquia de generalização para exemplificar o mapeamento.

A [Tabela 12-3](#) exibe as soluções de acordo com cada uma das alternativas de mapeamento descrita há pouco. Na primeira alternativa (uma relação para cada classe da hierarquia), há uma chave estrangeira (idContribuinte) em cada relação de subclasse por conta do artifício utilizado de se considerar a existência de uma associação um-para-um.

Na segunda alternativa (uma relação para toda a hierarquia), note a existência do atributo tipo, utilizado para fazer a diferenciação entre os tipos de pessoa.

Finalmente, a terceira alternativa (uma relação para cada classe concreta da hierarquia) apresenta uma relação que implementa a hierarquia e nas quais os atributos da superclasse estão replicados.

Tabela 12-3: Mapeamento de uma hierarquia de generalização

1 ^a alternativa	Contribuinte(<u>id</u> , endereço) PessoaFísica(<u>id</u> , nome, dataNascimento, CPF, <u>idContribuinte</u>) PessoaJurídica(<u>id</u> , CNPJ, razãoSocial, <u>idContribuinte</u>)
2 ^a alternativa	Pessoa(<u>id</u> , nome, endereço, dataNascimento, CPF, CNPJ, razãoSocial, tipo)
3 ^a alternativa	PessoaFísica(<u>id</u> , dataNascimento, nome, endereço, CPF) PessoaJurídica(<u>id</u> , CNPJ, endereço, razãoSocial)

12.1.9.5 Comparação entre as estratégias de mapeamento

A primeira alternativa (uma relação para cada classe da hierarquia) é a que melhor reflete o modelo

orientado a objetos. Isto porque cada classe é mapeada para uma relação, e as colunas desta relação são correspondentes aos atributos específicos da classe. Entretanto, essa alternativa apresenta desvantagens com relação ao desempenho da manipulação das relações.

De fato, no exemplo da primeira alternativa, sempre que houver a necessidade de inserção (remoção) de um novo objeto, seja ele da classe PessoaFísica ou da classe PessoaJurídica, deverá haver inserção (remoção) de dois registros, um em cada relação. Isso porque as colunas que representam objetos dessas classes estão divididas entre as duas relações.

A primeira alternativa também é desvantajosa em consultas que precisem processar dados em que as colunas necessárias não estão na mesma relação. Neste caso, uma operação de *junção* (bastante cara do ponto de vista de desempenho) sobre as relações envolvidas deverá ser realizada.

A segunda alternativa de implementação é bastante simples, além de facilitar situações em que objetos mudam de classe (ver [Seção 8.5.7](#)).

Uma desvantagem da segunda alternativa é na situação em que um novo atributo deve ser adicionado ou removido de uma das classes da hierarquia. Nesse caso, não importa qual seja esta classe, a relação que mapeia toda a hierarquia é sempre modificada.

Outra desvantagem é que ela tem o potencial de desperdiçar bastante espaço de armazenamento, principalmente no caso em que a hierarquia tem uma largura grande (várias classes “irmãs”) e os objetos pertencem a uma, e somente uma, classe da hierarquia.

Finalmente, a terceira alternativa (uma relação para cada classe da hierarquia) apresenta a vantagem de agrupar os objetos de uma classe em uma única relação.

Entretanto, uma de suas desvantagens é que, quando uma classe é modificada, cada uma das relações correspondentes às suas subclasses deve ser modificada. Considere, por exemplo, a quantidade de trabalho a ser feita quando um atributo deve ser adicionado à classe Contribuinte do diagrama exibido na [Figura 12-7](#): as relações PessoaFísica e PessoaJurídica devem ser alteradas. De forma geral, se houver n relações correspondentes a subclasses, estas n relações devem ser modificadas quando a definição da superclasse é modificada.

A conclusão a que se pode chegar é que nenhuma das alternativas de mapeamento de generalização é a melhor. Cada uma possui vantagens e desvantagens, e a escolha da alternativa a ser utilizada depende das características do sistema de software sendo desenvolvido. Mais que isso, a equipe de desenvolvimento pode mesmo decidir implementar mais de uma alternativa. Isto poderia ser feito por meio do conceito de visões em um SGBDR.

12.2 Construção da camada de persistência

Na [Seção 12.1](#), são descritas formas de mapeamento da *estrutura* dos objetos para que estes possam ser armazenados de forma persistente em um SGBDR. Por outro lado, quando uma sessão de uso de um SSOO é iniciada, objetos têm que ser resgatados a partir do mecanismo de armazenamento persistente (ou seja, deve ser alocado espaço em memória principal para armazenamento desses objetos). Quando essa sessão de uso termina, todos os objetos são destruídos (ou seja, o espaço alocado para eles na memória principal é retornado ao sistema operacional). Mais que isso, eventuais alterações nos objetos resgatados devem ser refletidas no mecanismo de armazenamento, assim como objetos criados durante a sessão devem ser armazenados nesse mecanismo. Portanto, *objetos persistentes* são aqueles cujo estado é mantido entre sucessivas sessões de uso do sistema. Essa manutenção é feita com o uso de algum *mecanismo de armazenamento persistente*, que

corresponde a um recurso externo ao SSOO usado para armazenar os objetos entre sessões de uso desse sistema.

Pela descrição acima, podemos concluir que devem ser definidos procedimentos relativos ao *armazenamento* de objetos em um SGBDR em tempo de execução do SSOO. Alguns desses procedimentos são enumerados a seguir.

- a. Materialização: corresponde a restaurar um objeto a partir do banco de dados quando necessário.
- b. Atualização: consiste em enviar modificações sobre um objeto para o banco de dados.
- c. Remoção: corresponde a remover um objeto do mecanismo de armazenamento persistente.

Esses procedimentos estão relacionados ao transporte de objetos da memória principal do sistema para um SGBD e vice-versa. Tais funcionalidades permitem que objetos perdurem e sejam modificados em diversas execuções do sistema. Nesta seção, descrevemos diversas estratégias que podem ser utilizadas para implementar o mapeamento objeto-relacional, ou seja, a transformação da representação do modelo relacional para o modelo de objetos, e vice-versa.

Para isolar os objetos do domínio de detalhes de comunicação com o SGBD, uma *camada de persistência* pode ser utilizada. O objetivo de uma camada de persistência é isolar os objetos do sistema de mudanças no mecanismo de armazenamento persistente. Se um SGBD diferente tiver que ser utilizado pelo sistema,⁷ por exemplo, somente a camada de persistência é modificada; os objetos da camada do domínio permanecem intactos. Em uma perspectiva arquitetural, a camada de persistência é parte da camada de infraestrutura (veja a [Seção 11.1.2.4](#)).

Com a diminuição do acoplamento entre os objetos e a estrutura do banco de dados, o sistema se torna mais flexível (podendo ser modificado para se adaptar a novos requisitos) e mais portável (podendo ser transportado para outras plataformas de hardware ou de software).

No entanto, as vantagens de uma camada de persistência não vêm de graça. A intermediação feita por essa camada entre os objetos do domínio e o SGBD traz uma sobrecarga de processamento ao sistema, o que pode diminuir o seu desempenho. Outra desvantagem é que a camada de persistência pode aumentar a complexidade da realização de certas operações que seriam triviais com o uso direto de SQL. Entretanto, as vantagens adquiridas pela utilização de uma camada de software, principalmente em sistemas complexos, geralmente compensam a perda no desempenho e a dificuldade de implementação.

12.2.1 Acesso direto ao banco de dados

Uma estratégia simples utilizada para realizar o mapeamento objeto-relacional é fazer com que cada objeto persistente do sistema possua comportamento que permita a sua restauração, atualização ou remoção do mecanismo persistente conforme necessário. Há um código escrito em SQL (*Structured Query Language*) para realizar a inserção, remoção, atualização e consulta das tabelas onde estão armazenados os objetos. Essa solução é de fácil implementação em Linguagens de Quarta Geração, como o *Visual Basic*, o *PowerBuilder* e o *Delphi*. Esses ambientes possuem os denominados controles *data aware*, objetos para construção da interface gráfica com o usuário que se conectam diretamente a dados armazenados em um mecanismo de armazenamento.

No entanto, a solução descrita anteriormente apresenta algumas desvantagens para sistemas mais complexos. Uma delas é que as classes relativas à lógica do negócio ficam muito acopladas às

classes relativas à interface e ao acesso ao banco de dados. Pode-se tornar muito complexo migrar o sistema de um SGBD para outro. Ainda por cima, a lógica da aplicação fica desprotegida de eventuais modificações na estrutura do banco de dados.

Outro problema é que a *coesão* das classes (ver [Seção 7.5.2](#)) diminui. Isso porque cada classe deve possuir responsabilidades relativas ao armazenamento e à materialização de seus objetos, além de ter responsabilidades inerentes ao negócio. A dificuldade de manutenção e extensão do código-fonte resultante praticamente impede a utilização desta abordagem para sistemas complexos.

A busca de soluções para os problemas suscitados por essa estratégia (acesso direto) gerou diversas outras propostas. Descrevemos diversas dessas propostas nas próximas seções.

12.2.2 Uso de um SGBDOO ou de um SGBDOR

Na metade da década de 1980, começou-se a falar em um novo modelo para SGBDs, o orientado a objetos. Nesse modelo, em vez de tabelas, os conceitos principais eram classes e objetos. A teoria desse modelo se consolidou e, já no início da década de 1990, foram criados alguns produtos comerciais de *sistemas de gerência de bancos de dados orientados a objetos* (SGBDOO). Alguns exemplos de SGBDOO: ORION (MOC), OPENOODB (Texas Instruments), Iris (HP), GEMSTONE (GEMSTONE Systems), ONTOS (Ontos), Objectivity (Objectivity Inc.), ARDENT (ARDENT software), POET (POET Software).

Em 1991, foi formado um grupo para padronizar as funcionalidades de um SGBDOO, o ODMG (*Object Database Management Group*). Entre outras coisas, o ODMG definiu um modelo de objetos e uma linguagem de consulta para SGBDOO. O modelo de objetos, denominado ODL (*Object Definition Language*), permite a definição de estruturas de dados arbitrariamente complexas (classes) no SGBDOO. Nesse modelo, atributos de um objeto podem conter valores de tipos de dados estruturados, diferente do modelo relacional, onde as tabelas só podem armazenar itens atômicos. A ODL também fornece a possibilidade de definir hierarquias de herança entre classes. A linguagem de consulta definida pelo ODMG foi denominada OQL (*Object Query Language*). A OQL permite consultar e manipular objetos armazenados em um banco de dados, além de possuir extensões para identidade de objetos, objetos complexos, expressões de caminho, chamada de operações (métodos) e herança.

Algumas pessoas pensavam que a tecnologia de SGBDOO suplantaria a velha tecnologia relacional do matemático E. F. Codd. Afinal de contas, a primeira estava em perfeita sintonia com o paradigma da orientação a objetos que, no contexto das linguagens de programação, vinha ganhando mais e mais força. No entanto, o que aconteceu foi que os principais vendedores de SGBDR existentes na época começaram a incorporar características de orientação a objetos em seus produtos. Esses SGBDR passaram a adotar o modelo de dados *objeto-relacional*. Esse modelo de dados é uma extensão do modelo relacional, onde são adicionadas características da orientação a objetos, entre outras.

Hoje em dia os principais produtos de SGBD existentes são *sistemas de gerência de bancos de dados objeto-relacionais* (SGBDOR). Um SGBDOR é também conhecido pelo nome de SGBD relacional estendido. Atualmente, os SGBDs puramente orientados a objetos (SGBDOO) são pouco utilizados. Dois dos principais representantes dessa geração de SGBDOR encontrados no mercado são o Oracle 9i™ e o DB2 Universal Server™.

Assim como os SGBDOO, os SGBDOR são ideais para certas aplicações especiais, como CAD/CAM (*Computer Aided Design/Computer Aided Manufacturing*). Além disso, em tese, se uma

aplicação desenvolvida em uma linguagem orientada a objetos utiliza um SGBDOR ou um SGBDOO em vez de um SGBDR, o mapeamento de objetos pode ser feito de forma mais direta. No entanto, os SGBDOO não se desenvolveram comercialmente. Na maioria das vezes, o desenvolvedor de aplicações orientadas a objetos que precisam armazenar dados persistentes acaba tendo a necessidade de interagir com um SGBDR ou com um SGBDOR. Além disso, é um fato que existe uma quantidade imensa de organizações que utilizam um SGBDR puro. Mais que isso, existe uma grande resistência das organizações em substituir esses sistemas. Isso leva a crer que o mapeamento de objetos para o modelo puramente relacional ainda irá durar por muitos anos. Por conta disso, nas próximas duas seções, detalhamos estratégias de mapeamento de objetos mais realistas: o *padrão DAO* e *frameworks ORM*.

12.2.3 Padrão DAO

Existem no mercado diversos tipos de armazenamento persistente: arquivos no sistema operacional, sistemas de gerência de bancos de dados, diretórios LDAP, sistemas legados etc. Cada um desses tipos de armazenamento contém suas particularidades de acesso aos dados. Falando especificamente de sistemas de gerência de bancos de dados, cada um deles fornece sua API (*Application Programming Interface*) para acesso aos dados armazenados. Sistemas que fazem acesso direto a determinado mecanismo de armazenamento ficam atrelados a suas particularidades. Além disso, sistemas assim construídos frequentemente misturam aspectos da lógica do negócio com aspectos de acesso aos dados persistentes. Por consequência, esses sistemas se tornam menos portáveis e desnecessariamente atrelados a uma tecnologia específica de armazenamento.

Tecnologias como *JDBC* (Java Database Connectivity) e *ADO.NET* são uma tentativa de “blindar” as aplicações das particularidades de cada SGBDR. Essas tecnologias fornecem APIs que podem ser utilizadas para acesso padronizado a diversos SGBDR. Com o uso de uma dessas tecnologias, uma mesma API é utilizada pelo programador para ter acesso a informações em um SGBDR, não importando qual é o SGBDR específico em uso no momento. Se, por exemplo, os dados da aplicação tiverem que ser movidos para outro SGBDR, essa aplicação não precisa ser alterada, por conta de estar utilizando uma API padronizada.

Entretanto, mesmo nessas APIs padronizadas fornecidas pelo JDBC e pelo ADO.NET, há o risco de a aplicação ficar acoplada a um SGBDR específico. Um exemplo típico dessa situação é o caso em que é necessário obter a chave primária do último registro inserido em uma tabela; cada SGBDR tem sua estratégia para fornecer tal valor. Além disso, pode ser que a aplicação tenha a necessidade de armazenar informações não somente em diferentes SGBDR, mas também em outros tipos de mecanismos de armazenamento persistentes, como arquivos no sistema operacional, sistemas legados, serviços externos etc. Uma solução para esse dilema é o padrão DAO, que descrevemos a seguir.

Padrão DAO: uso de um objeto do acesso dos dados (DAO) para obter acesso e armazenar dados em uma fonte. O objeto DAO controla a conexão com a fonte dos dados para obter e armazenar dados.

O padrão DAO é uma forma de desacoplar as classes do negócio dos aspectos relativos ao acesso ao(s) mecanismo(s) de armazenamento persistente. DAO é uma sigla para *Data Access Object* (Objeto de Acesso a Dados). Graças ao uso desse padrão, uma aplicação obtém acesso a objetos de

negócio (que estão associados a informações persistentes) por meio de uma *interface* (ver [Seção 8.5.4](#)), a chamada *interface DAO*. Classes que implementam essa interface se comprometem a prover os meios de transformar as informações provenientes do mecanismo de armazenamento em objetos de negócio. Também se comprometem a armazenar informações de um objeto de negócios no mecanismo de armazenamento em uso. A aplicação interage com o objeto DAO mediante uma interface. A implementação desse objeto por si só não faz diferença para a aplicação. O objeto DAO isola completamente os seus clientes das particularidades da fonte de dados sendo acessada. Apresentamos a estrutura genérica do padrão DAO na [Figura 12-8](#).

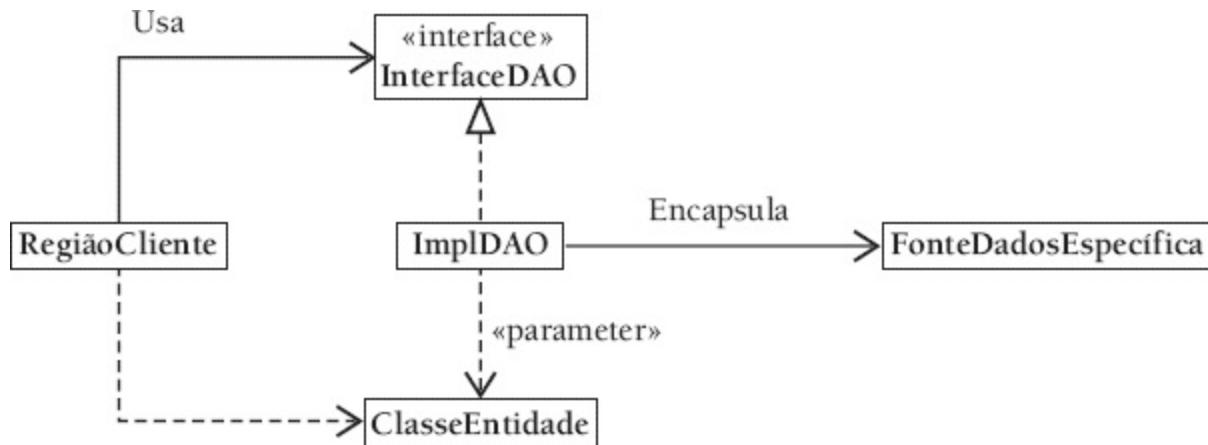


Figura 12-8: Estrutura do padrão DAO.

Na [Figura 12-8](#), temos diversos componentes. Em primeiro lugar, o componente Cliente é a região da aplicação que necessita ter acesso a objetos persistentes. Outro componente dessa estrutura é a InterfaceDAO, que define as operações que devem ser implementadas pela classe ImplDAO. A classe ImplDAO representa uma implementação para acesso a uma fonte de dados específica. Para cada fonte de dados específica, a aplicação deve ter uma classe ImplDAO. Por fim, ClasseEntidade representa a classe cujos objetos são criados pelo objeto ImplDAO. Esses objetos encapsulam as informações provenientes da fonte de dados. Uma vez que esses objetos estão criados, a região cliente manipula as informações persistentes por intermédio dos mesmos.

Outra questão importante a respeito da [Figura 12-8](#) é que embora a Região-Cliente use as implementações fornecidas por ImplDAO, ela o faz por meio da InterfaceDAO. Isso significa que RegiãoCliente não tem e não precisa ter conhecimento do objeto específico que está implementando a interface. Esse aspecto é importante porque permite que a RegiãoCliente obtenha acesso à fonte de dados sem estar acoplada a uma forma de acesso específica.

Para dar uma visão mais clara do padrão DAO, o [Quadro 12-1](#) fornece um exemplo em linguagem Java de uma InterfaceDAO, chamada AlunoDAO. Note que essa interface define operações para manipulação de objetos da classe Aluno, que corresponde ao ClasseEntidade na estrutura genérica da [Figura 12-8](#).

Quadro 12-1: Exemplo em linguagem Java de uma InterfaceDAO

```

public interface AlunoDAO {
    public void inserir(Aluno aluno)
        throws AlunoDAOException;
  
```

```
public void atualizar(Aluno aluno)
    throws AlunoDAOException;

public void remover(Aluno aluno)
    throws AlunoDAOException;

public Set <Aluno>encontrarTodos( )
    throws AlunoDAOException;

public Aluno encontrarPorMatrícula(String matrícula)
    throws AlunoDAOException;

public Set <Aluno>encontrarPorTurma(int idTurma)
    throws AlunoDAOException;
}
```

Um comentário relevante a respeito de objetos de acesso a dados é que eles podem ser aplicados não somente ao caso do mapeamento de objetos para um SGBD relacional, mas também para mecanismos de armazenamento de outras naturezas (p. ex., quando os objetos devem ser serializados para armazenamento em arquivos em formato XML).

Como comentário final sobre objetos DAO, é importante entender a diferença entre eles e os repositórios, descritos na [Seção 5.4.3.2](#). Enquanto que um DAO torna o restante da aplicação independente da natureza do mecanismo de armazenamento (SGBD, arquivos etc), um repositório torna o restante da aplicação independente da própria existência de um mecanismo de armazenamento persistente. Por outro lado, é também possível construir uma solução em que um repositório é implementado por meio da delegação de tarefas a um objeto DAO subjacente. Repare também que repositórios são objetos da camada do domínio, enquanto que objetos DAO residem na camada de infraestrutura.

12.2.4 Frameworks ORM

Outra estratégia de implementação da camada de persistência que está se tornando bastante popular é por meio de um *framework ORM*. Antes de definir o que significa esse termo, podemos primeiramente definir o que significa *framework*.

Sucintamente falando, um framework é um conjunto de classes (abstratas e concretas) que podem ser usadas para prover uma estrutura ou implementação inicial para uma aplicação. Graças ao uso de um ou mais frameworks, uma aplicação pode ser construída bem mais rapidamente do que se fosse construída a partir de nada. O uso de um framework, que também chamamos de *instanciação* do mesmo, consiste em derivar subclasses a partir das classes que ele fornece, ou de criar classes que são composições das classes desse framework. Existem frameworks para a construção de diferentes aspectos de uma aplicação: interface gráfica com o usuário, criação de testes, persistência de objetos em um SGBDR etc. São frameworks desse último tipo que nos interessam nesta seção.

Um framework ORM é um conjunto de classes que realiza o mapeamento transparente entre objetos da aplicação e tabelas em um SGBDR. O termo ORM é uma sigla para *Object-Relational Mapping* (mapeamento objeto-relacional). Os frameworks ORM tentam resolver o problema do *descasamento de informações* fornecendo classes que o realizam de forma transparente para o programador da aplicação. Normalmente um framework necessita que o desenvolvedor forneça a correspondência entre a estrutura de objetos da aplicação e o esquema relacional do banco de dados. Essa correspondência tende a ser fornecida por um arquivo de configuração, denominado *arquivo de*

mapeamento. De posse dessa correspondência, o framework está apto a mapear qualquer requisição por uma informação armazenada no SGBDR.

Por exemplo, suponha que um objeto `Aluno` em uma aplicação receba uma mensagem para informar todas as disciplinas que ele já cursou. Se as informações da aplicação estão armazenadas em um SGBDR, essa mensagem ocasiona uma junção entre a tabela que guarda registros de alunos e a tabela que guarda registros de disciplinas. Com o uso de um framework ORM, tudo que o programador tem a fazer é implementar um método `obterDisciplinasCursadas` na classe `Aluno`; quando uma mensagem é recebida para ativar esse método, o framework ORM é responsável por mapear essa chamada em uma expressão SQL para ser enviada ao SGBDR, receber o resultado e criar uma lista de objetos `Disciplina` para ser retornada pelo método. Note que tudo isso é feito de forma transparente pelo framework ORM, uma vez que a correspondência entre a estrutura de objetos e o esquema relacional é conhecida por ele.

Pelo exposto no parágrafo anterior, deve haver classes no framework que implementam este mapeamento automático de objetos para linhas de uma ou mais tabelas do banco de dados. Essas classes também têm a responsabilidade de mapear relacionamentos entre os objetos. Note que esse mapeamento automático proporciona um acoplamento mínimo entre as classes da lógica do negócio e o banco de dados. Mais especificamente, modificações em alguma classe da lógica do negócio têm menor potencial de resultar na modificação do esquema do banco de dados e vice-versa. Em algumas situações, somente o arquivo de mapeamento deve ser alterado.⁸

12.2.4.1 Implementação de um framework ORM

O framework ORM pode ser implementado pela própria equipe de desenvolvimento, ou pode ser adquirido. Essa escolha depende de vários fatores como conhecimento técnico da equipe, disponibilidade de recursos financeiros etc. Se a decisão for a de implementar o framework, é preciso levar em conta que isso não é algo trivial. Os desenvolvedores podem levar bastante tempo nessa implementação.

A descrição completa da implementação de um framework está fora do escopo deste livro. No entanto, o restante desta seção descreve alguns aspectos de implementação de um framework. Um estudo mais detalhado do assunto pode ser encontrado em AMBLER (2000).

Gerenciamento de transações

Frequentemente um conjunto de operações deve ser submetido de uma única vez ao SGBD. A ideia é que todos os itens do conjunto sejam processados; se algum item não puder ser processado, todo o conjunto de operações é cancelado. Na terminologia de banco de dados, esta forma de processamento é denominada *transação*. O framework ORM deve também fornecer funcionalidades para que certa transação sobre um ou mais objetos seja realizada de modo atômico. Isto significa que deve haver uma ou mais classes nesta camada para que transações sejam submetidas ao mecanismo de persistência. Essas classes devem também permitir que uma transação seja refeita ou mesmo desfeita.

Linguagem de consulta a objetos

A maioria dos frameworks para ORM fornece uma *linguagem de consulta sobre objetos*.

Diferentemente da SQL, não fazemos referências em uma linguagem de consulta sobre objetos a nomes de tabelas ou colunas. Em vez disso, fazemos consultas com referências a nomes de classes e a nomes de atributos definidos em alguma classe de nosso modelo de objetos. É tarefa do framework ORM traduzir expressões definidas sobre essa linguagem em comandos válidos em SQL.

Manipulação de coleções de objetos

Outra funcionalidade de um framework é permitir a manipulação (inserção, remoção, atualização e recuperação) de uma coleção de objetos de uma única vez. Esta funcionalidade é implementada por meio de várias classes que permitem a montagem de comandos SQL (INSERT, UPDATE, DELETE e SELECT) sem que a lógica da aplicação tenha conhecimento do esquema do banco de dados utilizado. Essas classes permitem que um critério de manipulação seja definido, para que somente um subconjunto dos registros de uma tabela seja considerado na operação.

- Inserção: funcionalidade para permitir a inserção de vários objetos de uma única vez.
- Remoção: funcionalidade para permitir a remoção de vários objetos de uma única vez. As classes que implementam esta funcionalidade podem ter uma característica de marcar os objetos como removidos, ao invés de propriamente removê-los. Isso permite que os desenvolvedores do sistema implementem funcionalidades para permitir desfazer operações na camada de apresentação.
- Atualização: os objetos são materializados a partir do banco de dados segundo algum critério de seleção e, em seguida, os valores de seus atributos podem ser alterados.
- Recuperação: a coleção de objetos é recuperada e pode ser navegada por meio de classes que implementam funcionalidades semelhantes aos cursores de um banco de dados.

Uma característica das classes que implementam a funcionalidade de manipulação de coleções de objetos é que elas manipulam o estado de objetos, em vez de colunas de tabelas do banco de dados.

12.2.4.2 Especificações e tecnologias relacionadas a ORM

Nas grandes plataformas de desenvolvimento, é comum encontrar especificações e tecnologias relacionadas ao mapeamento objeto-relacional. Em um SSOO que possua um modelo do domínio complexo, o uso desses recursos é fundamental para atingir o isolamento da camada de domínio mencionado na [Seção 11.1.2.3](#). A seguir, apresentamos alguns desses recursos.

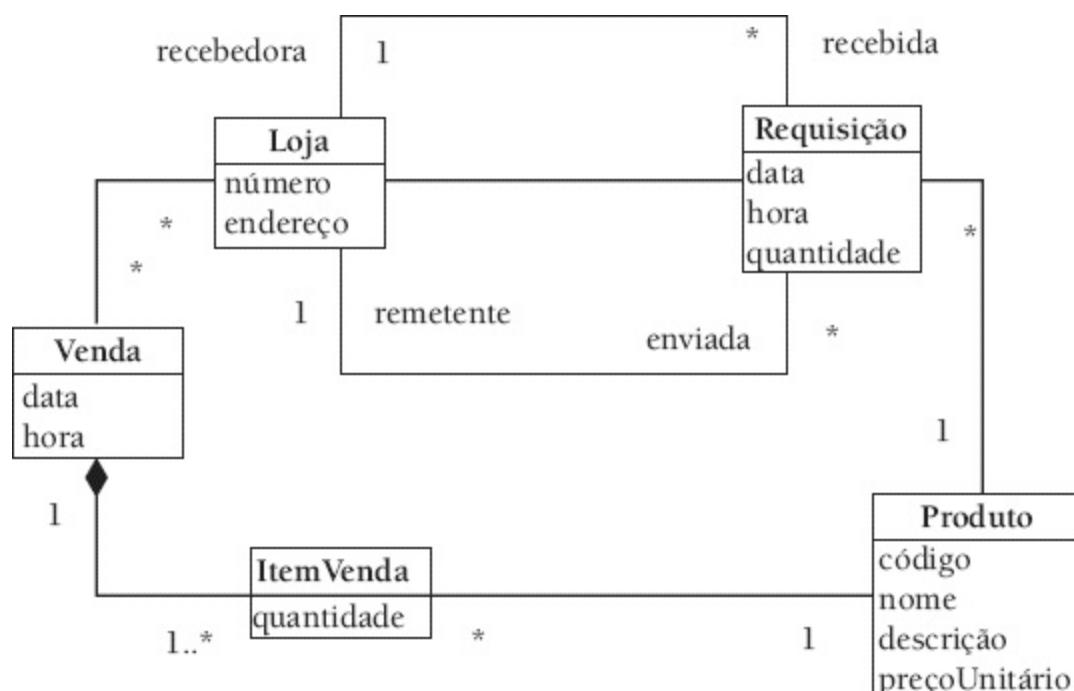
Em Java, existe a especificação JPA (*Java Persistence API*), um conjunto interfaces para realizar a persistência de objetos. Essa API fornece operações para realizar operações CRUD sobre objetos persistentes e gerenciar transações. A configuração de associações entre objetos pode ser realizada por meio de arquivos em formato XML ou de anotações (*annotations*). Também faz parte da especificação JPA a JPQL (*Java Persistence Query Language*), que permite definir expressões (cuja sintaxe é semelhante ao SQL) para consultar o grafo de objetos persistentes de um SSOO. Uma funcionalidade interessante da JPA com relação ao desempenho é a capacidade de permitir a **carga tardia de objetos**. Por meio dessa funcionalidade, é possível resgatar um objeto do mecanismo de armazenamento persistente sem resgatar objetos associados a ele. Os objetos dependentes são resgatados apenas quando referenciados explicitamente. Na literatura essa característica é conhecida por meio de um padrão de projeto denominado *Lazy Loading*. O Hibernate (<http://hibernate.org>) é uma das implementações da JPA para Java.

Na plataforma .NET, encontramos a LINQ (*Language-Integrated Query*) que, além do

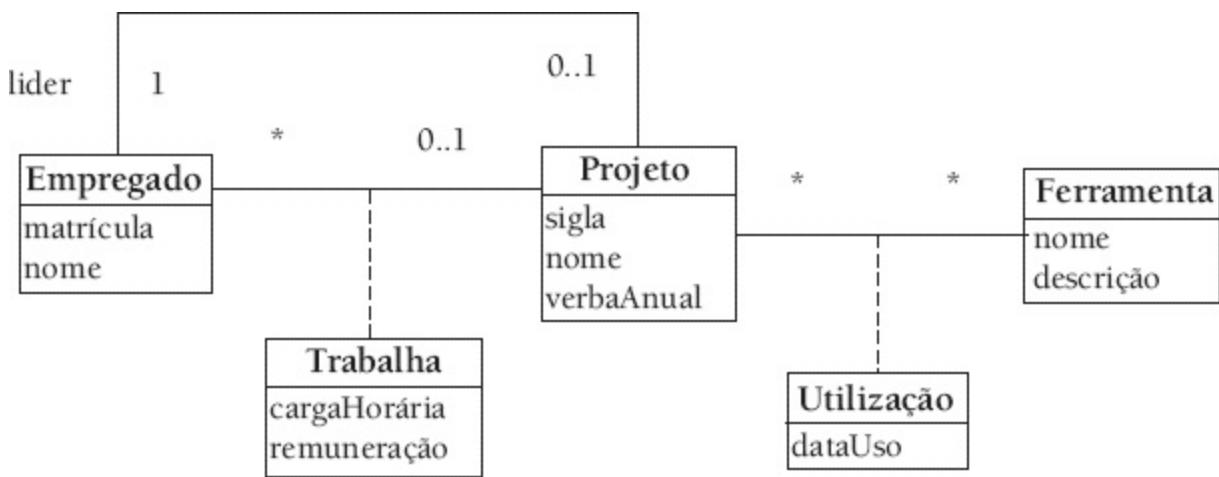
mapeamento objeto-relacional, permite o mapeamento de dados para diversas outras representações. A LINQ é integrada às linguagens C# e VB.NET. Outro framework ORM para a plataforma .NET é o Entity Framework, para qual a Microsoft recentemente divulgou a abertura de seu código.

► EXERCÍCIOS

12-1: Veja o diagrama de classes apresentado a seguir. Realize o mapeamento deste diagrama para o modelo relacional.

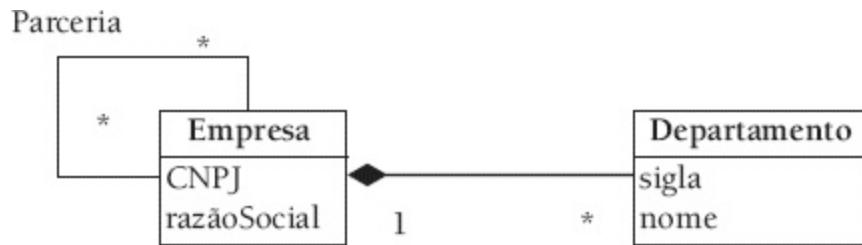


12-2: Veja o diagrama de classe a seguir. Este diagrama informa que empregados podem liderar um projeto por vez, enquanto cada projeto tem, necessariamente, um único líder. O diagrama também informa que pode haver diversos empregados trabalhando em um projeto, embora não possa haver um empregado trabalhando em mais de um projeto. Finalmente, projetos utilizam determinadas ferramentas, e uma ferramenta pode ser utilizada por vários projetos. Realize o mapeamento deste diagrama para o modelo relacional.



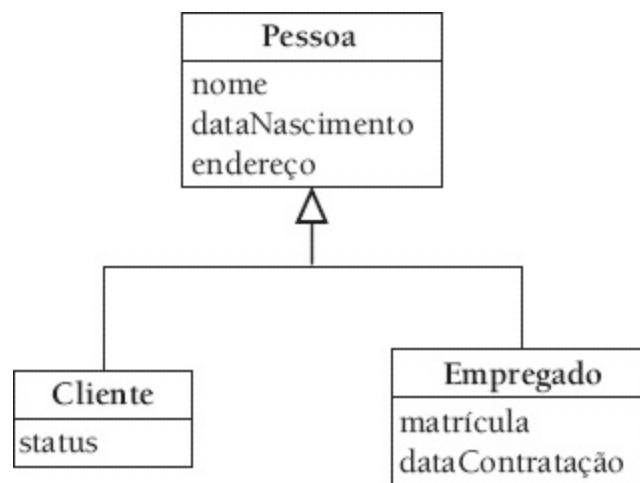
12-3: Considere a hierarquia de generalização a seguir, onde Empregado e Cliente são subclasses de Pessoa. Discuta o que acontece em termos dos objetos armazenados em relações em cada uma das alternativas de mapeamento de generalização nas situações a seguir.

- a) Um empregado se torna uma cliente da loja.
- b) Um cliente se torna um empregado da loja.
- c) Uma pessoa é tanto um empregado quanto um cliente da loja.



12-4: Considere a mesma hierarquia de generalização da questão anterior. Suponha que uma nova classe, Gerente, é adicionada à hierarquia como subclasse de Empregado. Suponha, ainda, que esta nova classe tem um atributo que informa a comissão para gerentes (considere que só empregados gerentes têm comissão). Reflita quanto às modificações que devem ser feitas ao esquema de banco de dados em cada uma das alternativas de mapeamento de generalização.

12-5: Realize o mapeamento para o modelo relacional do seguinte diagrama de classe.



1. A tecnologia relacional se baseia na Teoria de Conjuntos e foi proposta pelo matemático E. F. Codd, no início da década de 1970 (CODD, 1970). Por sua contribuição na área de banco de dados, em 1981, Codd recebeu o equivalente ao Prêmio Nobel da computação, o *ACM Turing Award*.
2. Um índice de banco de dados é uma estrutura de dados persistente que permite que o acesso a um determinado item de informação de um banco de dados seja agilizado. De uma forma bastante simples, pode-se dizer que um índice de banco de dados é semelhante ao índice de uma obra literária.
3. Historicamente, esta semelhança sintática é uma contribuição da OMT, uma das técnicas que influenciaram para a notação utilizada na UML. A OMT foi inicialmente proposta como uma técnica para modelagem relacional estendida.
4. Pode-se argumentar que não há necessidade de criação de uma classe para representar o CPF e que ele pode ser definido como um atributo. No entanto, essa solução pode se justificar para casos em que a classe CPF implementa o comportamento de validação de números de CPF.
5. Na verdade, uma relação de associação pode também representar ligações entre linhas de uma mesma relação, no caso de associações reflexivas.
6. Na verdade, essa correspondência entre objetos no relacionamento de herança não existe. O relacionamento de herança *não* é um relacionamento entre objetos, mas, sim, entre classes de objetos. A consideração que se faz nesta alternativa de mapeamento é somente um artifício.
7. Isso acontece muitas vezes no desenvolvimento de um sistema de software: inicialmente (provavelmente na prototipagem) é utilizado um SGBD menos potente (e mais barato) e, posteriormente, este é substituído pelo SGBD definitivo quando o sistema tiver que entrar em produção.
8. Note, entretanto, que em alguns casos todas as camadas precisam de modificações. Por exemplo, no caso em que um novo campo de dado precisa ser exibido na camada de apresentação, e o seu valor precisa ser armazenado no banco de dados.

Referências

- (Abbott, 1983) Russell J. Abbott. *Program Design By Informal English Descriptions Program Design By Informal English Descriptions*. Communications of the ACM, Volume 26, Issue 11, 1983.
- (Alur et al, 2003) Deepak Alur, Dan Malks e John Crupi. *Core J2EE Patterns: Core J2EE Patterns: As melhores práticas e estratégias de design*. 2^a ed. Rio de Janeiro: Elsevier, 2003.
- (Ambler, 1997) Scott W. Ambler. “Mapping Object to Relational Databases”. 1997. White Paper. Documento disponível em <http://www.amblysoft.com/> mappingObjects.pdf.
- (Ambler, 2000) Ambler, Scott W. “The Design of a Robust Persistence Layer for Relational Databases”. 2000. Disponível em <http://www.amblysoft.com/> persistenceLayer.pdf.
- (Beck e Cunningham, 1989) Kent Beck e Ward Cunningham. “A Laboratory For Teaching Object-Oriented Thinking”. Proc. of the 1989 OOPSLA Conference, 1-6 Outubro, New Orleans, Louisiana. Disponível em <http://c2.com/doc/oopsla89/paper.html>. 1989.
- (Bjork, 1998) R. C. Bjork. “An Example of Object-Oriented Design: An ATM Simulation”. (Notas de aula do Dr. Bjork no Gordon College). Estas notas estão disponíveis em http://www.mathcs.gordon.edu/local/courses/cs320/ATM_Example.
- (Blaha e Rumbaugh, 2006) James Rumbaugh e Michael Blaha. *Modelagem e projetos baseados em objetos com UML*. Rio de Janeiro: Elsevier, 2006.
- (Booch et al., 2006) Grady Booch, James Rumbaugh e Ivar Jacobson. *UML – Guia do usuário*. 2^a ed., Rio de Janeiro: Elsevier, 2006.
- (Buschmann et al, 1996) Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad e Michael Stal. *Pattern-Oriented Software Architecture – A System of Patterns*. Wiley & Sons, 1996.
- (Cantor, 1998) Murray Cantor. *Object Oriented Project Management with UML*. Wiley & Sons, 1998.
- (Chaos, 1994) Standish Group. “The CHAOS Report (1994)”. Documento disponível em http://www.pm2go.com/sample_research/chaos_1994_2.php.
- (Cockburn, 2005) Alistair Cockburn. *Escrevendo casos de uso eficazes: Um guia prático para desenvolvedores de software*. Porto Alegre: Bookman, 2005. 254 pp.
- (Cockburn, 2006) Alistair Cockburn. “An Open Letter to Object Technology Newcomers. Humans and Technology”. 1996. Último acesso em maio de 2006, Disponível em: <http://alistair.cockburn.us/crystal/articles/oltoon/openlettertoonewcomers.html>.
- (Constantine e Lockwood, 1999) Larry Constantine e Lucy A. D. Lockwood. *Software For Use: A Practical Guide To The Models And Methods Of Usage-Centered Design*. Reading, MA: Addison-Wesley, 1999.
- (Elmasri e Navathe, 2011) Ramez Elmasri e Shamkant B. Navathe. *Sistemas de banco de dados: fundamentos e aplicações*. 6^a ed. Reading, MA: Addison-Wesley, 2011.
- (Evans, 2003) Eric Evans. *Domain-Driven Design: Tacking Complexity In the Heart of Software*. MA: Addison-Wesley Longman Publishing Co., Inc., 2003.

- (Filho, 2003) Wilson de Pádua Paula Filho. *Engenharia de Software: fundamentos, métodos e padrões*. 2^a ed. Rio de Janeiro: LTC. 2003.
- (Fowler, 1997) Martin Fowler. *Analysis Patterns: Reusable Object Models*. Reading, MA: Addison-Wesley. 1997.
- (Fowler, 2005) Martin Fowler. *Refatoração: Aperfeiçoando o projeto de código existente*. Porto Alegre: Bookman, 2005.
- (Fowler, 2006) Martin Fowler. *Padrões de arquitetura de aplicações corporativas*. Porto Alegre: Bookman, 2002.
- (Gamma et al., 2000) Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. *Padrões de projeto: soluções reutilizáveis de software orientado a objetos*. 2^a ed. Porto Alegre: Bookman, 2000.
- (Gilb e Finz, 1988) Tom Gilb e Susannah Finz. *Principles of Software Engineering Management*. Addison-Wesley, Wokingham, England. 1988.
- (Gottesdiener, 2001) Ellen Gottesdiener. “Rules Rule: Business Rules as Requirements”. In *Beyond chaos*, Larry L. Constantine (eds.). ACM, New York, NY, USA 219-225. DOI=10.1145/379391.379419 <http://dx.doi.org/10.1145/379391.379419>. 2001.
- (Harel, 1987) David Harel. *Statecharts: A Visual Formalism For Complex Systems*. Sci. Comput. Program., 8, pp. 231-274. 1987.
- (Hay, 1996) David Hay. *Data Model Patterns: Conventions of Thought*. New York, NY: Dorset House, 1996.
- (Jacobson et al., 1992) Ivar Jacobson, Magnus Christerson, Patrick Jonsson e Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach*. MA: Addison-Wesley, 1992.
- (Jones, 1997) Carper Jones. *Applied Software Measurement: Assuring Productivity and Quality*. New York, NY: McGraw-Hill, Inc., 1997.
- (Larman, 2003) Craig Larman. *Utilizando UML e padrões - um guia para a análise e projetos orientados a objetos*. 3^a ed. Porto Alegre: Bookman. 2003.
- (Lee e Tepfenhart, 2001) R. C. Lee, e W. M. Tepfenhart. *UML e C++: Guia prático de desenvolvimento orientado a objeto*. São Paulo: Makron Books, 2001.
- (Liskov e Wing, 1994) Barbara Liskov e J. Wing. “A Behavioral Notion of Subtyping”. ACM TOPLAS, 16, 6 (Nov. 1994), 1811-1841. 1994.
- (Maciaszek, 2000) L. A. Maciaszek. *Requirements Analysis and System Design: Developing Information Systems with UML*. Reading, MA: Addison-Wesley, 2000.
- (Meyer, 1997) Bertrand Meyer. *Object-Oriented Software Construction*. 2nd ed. Prentice Hall, 1997.
- (Muller, 1999) Robert J. Muller. *Database Design for Smarties: Using UML for Data Modeling*. San Francisco, CA: Morgan Kaufmann Publishers Inc., 1999.
- (OMG, 2003) Object Management Group. “UML 2.0 Superstructure Final Adopted specification”. 2003. Disponível em www.omg.org/technology/documents/formal/uml.htm.
- (Parnas, 1972) David L. Parnas. “On The Criteria To Be Used In Decomposing Systems Into Modules”. Communications of the ACM 15, 12, 1053-1058. DOI=10.1145/361598.361623 <http://doi.acm.org/10.1145/361598.361623>. 1972.
- (Rosenberg e Scott, 2001) Doug Rosenberg e Kendall Scott. *Use Case Driven Object Modeling With*

UML: A Practical Approach. Reading, MA: Addison-Wesley, 2001.

(RUP, 2002) Rational Unified Process. “Best Practices for Software Development Teams”. White Paper. Disponível em <http://www.rational.com/products/rup/> index.jtmpl, 2002.

(Szwarcfiter e Markenzon, 2010) Jayme Luiz Szwarcfiter e Lilian Markenzon. *Estruturas de dados e seus algoritmos.* 3^a ed. São Paulo: Saraiva, 2010.

(Wazlawick, 2011) Raul Sidnei Wazlawick. *Análise e projeto de sistemas de informação orientados a objetos.* Rio de Janeiro: Elsevier, 2011. 302p.

(Wazlawick, 2014) Raul Sidnei Wazlawick. *Object-Oriented Analysis and Design for Information Systems: Modeling with UML, OCL and IFML.* New York: Morgan Kaufman, 2014. v. 1. 341p.

(Wirfs-Brock e Wilkerson, 1989) Rebecca Wirfs-Brock e Brian Wilkerson. “Object-Oriented Design: A Responsibility Driven Approach”. In: Norman Meyrowitz. (Ed.), Proc. OOPSLA-89: ACM Conference on Object-Oriented Programming Systems Languages and Applications, p. 71–75. 1989.

(Wirfs-Brock et al., 1990) Rebecca Wirfs-Brock, Brian Wilkerson e Lauren Wiener. *Designing Object-Oriented Software.* Prentice Hall, 1990.

(Yourdon, 1990) Edward Yourdon. *Análise estruturada moderna.* Rio de Janeiro: Elsevier, 1990.