1
Visão geral

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

No decorrer da história, diversos tipos de bens serviram de 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 a suas competidoras.

Há um ditado segundo o qual "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, dessa necessidade, surgiram 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 tal forma a compensar 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 compreende 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 cresce o tamanho do sistema. 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 construir essa casa, 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 construção de sistemas habitacionais, também há uma gradação de complexidade. Para a construção de sistemas de software mais complexos, também é 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 dessas razões.

1. **Gerenciamento da complexidade:** um dos principais motivos de utilizar modelos é que há limitações no ser humano em 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 modelo para representar sua parte aerodinâmica etc. Através 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:** certamente, 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 estes servem como um ponto de referência comum.
3. **Redução dos custos no desenvolvimento:** no desenvolvimento de sistemas, seres humanos estão invariavelmente sujeitos a cometerem 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 (por exemplo, é muito mais fácil corrigir uma maquete do que pôr abaixo uma 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 sobre como são os modelos de sistemas de software. Em construções civis, frequentemente há profissionais para analisar as plantas da construção. A partir dessas, 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 figura 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, juntamente com a informação textual associada, 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 uma outra, mais consisa 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". Nessa analogia, ele imaginou como seria um sistema de software que funcionasse como um ser vivo. Nesse sistema, cada "célula" interagiria com outras células através do envio de mensagens para 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, então, estabeleceu 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. Informa ao atendente (digamos, o José) seu nome, as características da pizza desejada e o seu endereço. José, que só tem a função de atendente, comunica à 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 obje-

[1] Talvez tal história não seja verídica, mas ilustra bem o conceito que quero passar.

tos 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, também mamífero, também um 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, um outro paradigma era utilizado na modelagem de sistemas:[2] 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 estas realizem serviços. Por que não aplicar essa mesma forma de pensar à modelagem de sistemas?

O paradigma da orientação a objetos visualiza um sistema de software como uma coleção de agentes interconectados chamados *objetos*. Cada objeto é responsável por realizar tarefas específicas. 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 papel importante 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 30 anos. As seções a seguir continuam descrevendo os conceitos principais da orientação a objetos.

---

[2] Na verdade, tanto o paradigma estruturado quanto o paradigma orientado a objetos surgiram nas linguagens de programação, para depois serem aplicados à modelagem de sistemas. De fato, as ideias de Alan Kay foram aplicadas na construção de uma das primeiras linguagens de programação orientadas a objetos: o SmallTalk.

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, os seres humanos realizam esse processo mental de agrupamento para tentar gerenciar a complexidade de entender as coisas do mundo real. Realmente, 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 infeliz acaso, 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 possam 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 Mensagens

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 um objeto for visto como uma entidade ativa que representa uma abstração de algo do mundo real, então faz sentido dizer que tal objeto pode responder a estímulos a ele enviados (assim como faz sentido dizer que seres vivos reagem a 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.

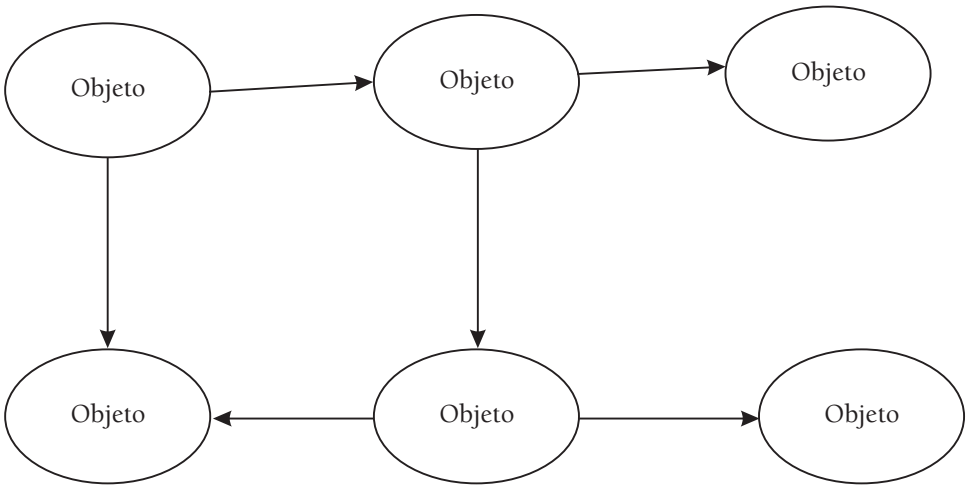

Figura 1-1: Objetos interagem através do envio de mensagens.

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 que ignoramos os aspectos menos importantes. Esse processo mental nos permite gerenciar a complexidade de um objeto, ao mesmo tempo que concentramos 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 se-

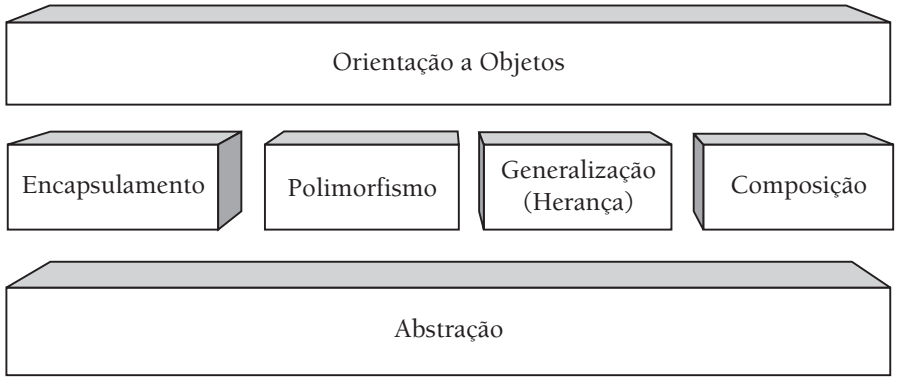

Figura 1-2: Princípios da orientação a objetos podem ser vistos como aplicações de um princípio mais básico, o da abstração.

ções, descrevemos alguns conceitos fundamentais da orientação a objetos e estabelecemos sua correlação com o conceito de abstração.

1.2.3.1 Encapsulamento

Objetos possuem *comportamento*. O termo comportamento diz respeito a operações realizadas por um objeto, conforme este objeto receba 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 ele sabe fazer, sem, no entanto, descrever *como* ele 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.

Através do encapsulamento, a única coisa que um objeto precisa saber para pedir a colaboração de outro objeto é conhecer a sua interface. Nada mais. Isso contribui para a autonomia dos objetos, pois cada objeto envia mensagens a outros objetos 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 software. 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 software moderno, no qual 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 software orientado 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 software, 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.

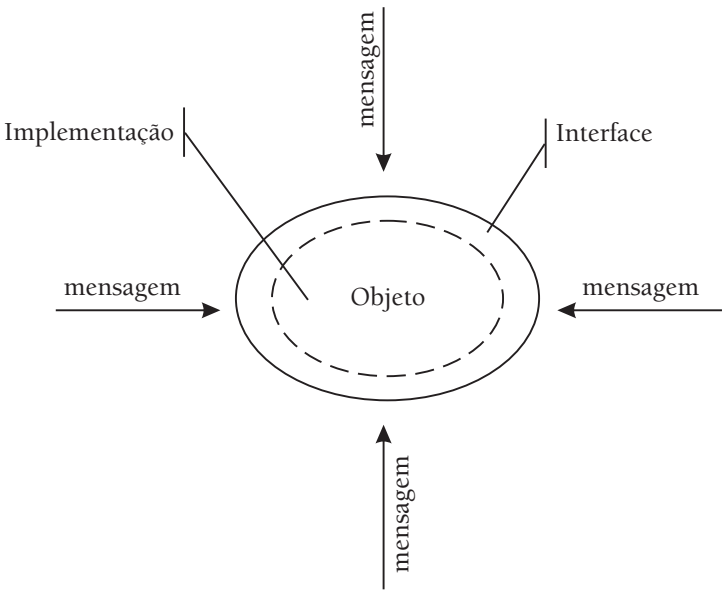

Figura 1-3: Princípio do encapsulamento: visto externamente, o objeto é a sua interface.

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 contexto da orientação a objetos, qual é a importância e quais são as consequências do polimorfismo? Nesse contexto, o polimorfismo diz respeito 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 adicionado. 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, uma classe pode ser criada a partir do reuso da definição de classes 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 ao qual é comum entre elas podem ser organizadas de forma mais clara.

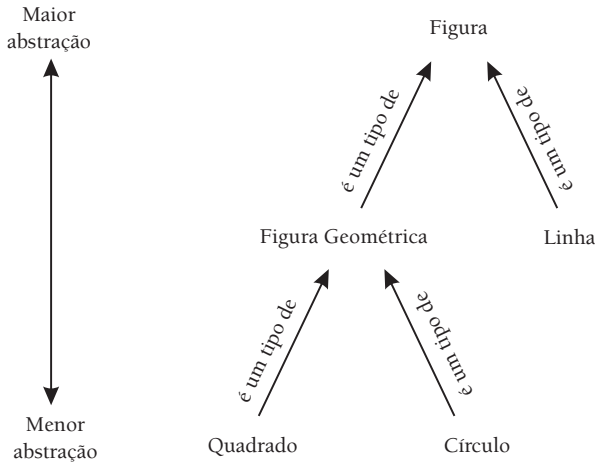

Figura 1-4: Princípio da generalização: classes podem ser organizadas em hierarquias.

1.2.3.4 Composição

É natural para nós seres humanos pensarmos 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 eles 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.

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 densi-

dade 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, pode-se 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.[3]

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 aplique 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 de tal 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, notavelmente no que tange 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.[4]

- **Décadas de 1950/60:** os sistemas de software eram bastante simples. O desenvolvimento desses sistemas era feito de forma *ad hoc*.[5] Os sistemas eram significativamente mais simples e, consequentemente, as técnicas de modelagem também eram mais simples: 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 começavam a surgir. Por conseguinte, modelos mais robustos foram propostos. Neste período, surgem a *programação estruturada* , baseada nos trabalhos de David Parnas. No final dessa década, surge 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.

[3] 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.

[4] 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.

[5] 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.

- **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 metada 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).

Um estudo mais detalhado da primeira metade da década de 1990 pode mostrar que surgiram 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

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 *et al.* (OMT – Object Modeling Technique)
1996	Wirfs-Brock (Responsibility Driven Design)
1996	(Fusion)

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 nesse sentido de 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.

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 procuraram aproveitar o melhor das características das notações preexistentes, principalmente das técnicas propostas anteriormente pelos três amigos (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 elementos adicionados com o objetivo de torná-la mais expressiva.

Finalmente, em 1997, a UML foi aprovada como padrão pelo OMG.[6] 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.[7] Desde o seu surgimento, várias atualizações foram feitas no sentido de torná-la mais clara e útil. Atualmente, a especificação do padrão UML está na versão 2.0 (OMG, 2003).

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. Através 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

---

[6] Sigla para Object Management Group. O OMG é um consórcio internacional de empresa que define e ratifica padrões na área da orientação a objetos.

[7] Algumas das empresas que participam da definição da UML são: Digital, HP, IBM, Oracle, Microsoft, Unisys, IntelliCorp, i-Logix e Rational.

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 a UML pode ser utilizada para a modelagem de sistemas, não importa 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 a desenvolvedores de 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 diversas perspectivas. 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 diferentes 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 inicialmente 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 (paralelismo), 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.

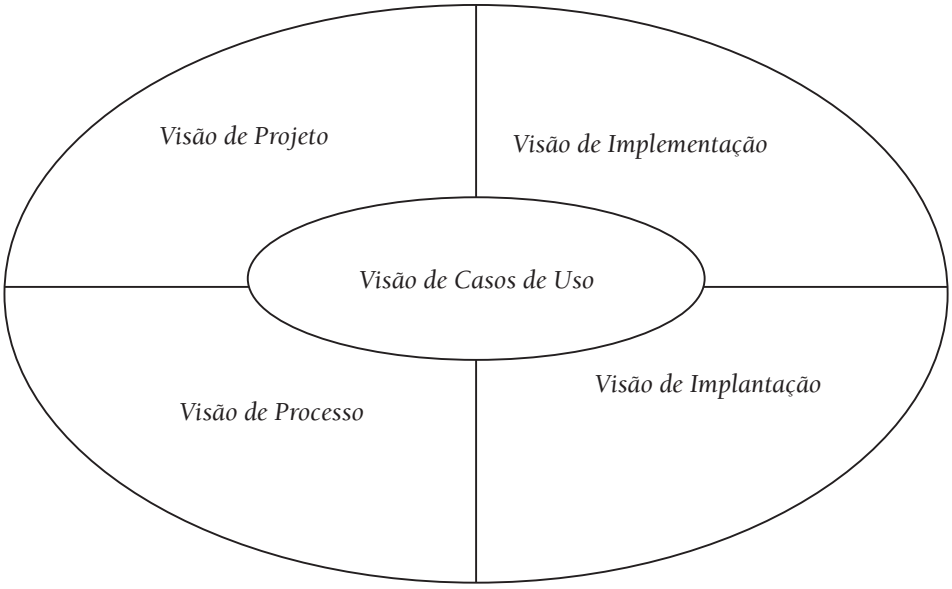

Figura 1-5: Visões (perspectivas) de um sistema de software.

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, esses documentos 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. Na figura, 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 exis-

tência de vários diagramas, vamos utilizar novamente uma analogia. Carrinhos em miniatura são cópias fiéis de carros de verdade. Cada um dos carrinhos é um modelo físico tridimensional que pode ser analisado de diversas perspectivas (por cima, por baixo, por dentro etc.). O mesmo não ocorre com os diagramas, que são desenhos bidimensionais.

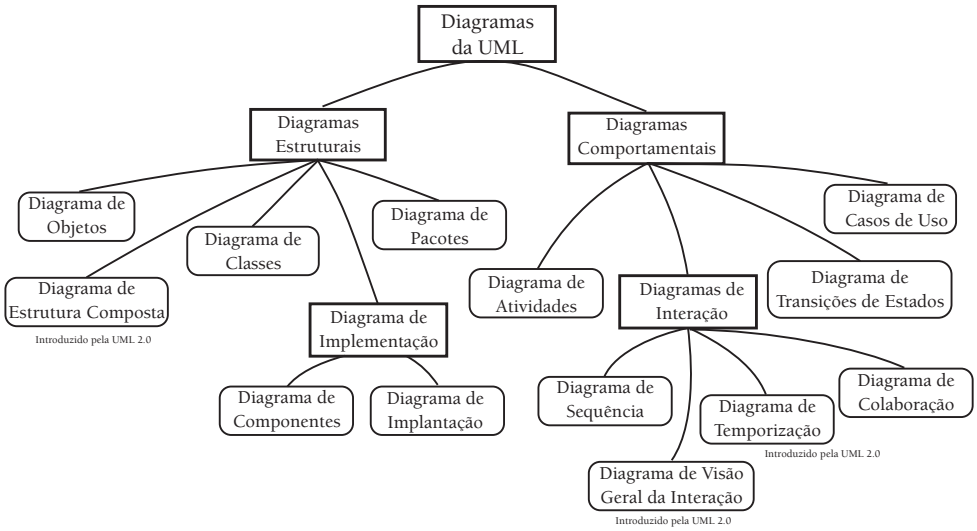

Figura 1-6: Diagramas definidos pela UML.

Para compensar essa dimensão a menos, utilizam-se diversos diagramas para construir modelos de várias perspectivas do sistema. Cada um dos diagramas 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.

A UML é uma linguagem de modelagem visual, ou seja, é um conjunto de notações e 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 de fora.
c. A células podem se reagrupar para resolver problemas ou para 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 algumas 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 relacione os termos objeto, classe, generalização e mensagem. Dê exemplos de cada um desses conceitos.

2

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 de 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 chegarem 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 (simplesmente *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 a forma de desenvolver software 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 o melhor processo de desenvolvimento, 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 componente deste para satisfazer um contrato, padrão, especificação ou outros documentos formalmente impostos (Maciaszek, 2000). Normalmente os requisitos de um sistema são identificados

[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.

a partir de um *domínio*. Denomina-se domínio a área de conhecimento ou de 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 (se este existir). 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 tempos de resposta esperados para as funcionalidades do sistema.

---

[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.

   c. Portabilidade: restrições sobre as plataformas de hardware e de software nas quais o sistema será implantado e sobre o 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 eles possam ser verificados e comunicados a leitores técnicos e não técnicos. A equipe técnica (leitores técnicos) deve entender o documento de requisitos de tal forma a poder obter 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 sobre o 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 uso 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 e o que não faz parte 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 mudar: as tecnologias utilizadas, as expectativas dos usuários, as regras do negócio etc. E isso para mencionar apenas algumas possíveis mudanças.

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

[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.

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 de desenvolvimento 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, 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 requisito deve ser considerado durante o desenvolvimento. As prioridades atribuídas aos requisitos permitirão ao gerente de projeto tomar decisões acerca 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, damos 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 tais componentes interagem com o objetivo de entender como esse sistema funciona. No contexto dos sistemas de software, esta é a etapa em que 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 inte-

resse 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*, através da 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 valida um modelo, quer dizer que entendeu o modelo construído e que, segundo esse entendimento, o modelo 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 tardiamente, 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 os modelos. 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 & 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 do domínio, por conta de corresponderem a conceitos com o 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 uma 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. Note que análise da aplicação envolve apenas a *identificação* desses objetos. Essa subfase *não* envolve o detalhamento desses objetos. 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 reuso. 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 ferramen-

ta é útil para modelar determinado aspecto de um sistema. É comum utilizarem-se diversas ferramentas para capturar aspectos diferentes de um problema dado. 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, diagrama de estados e diagrama de atividades.

2.1.3 Projeto (desenho)

O foco principal da análise são os aspectos lógicos e independentes de implementação de um sistema (os requisitos). 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 do sistema, padrão de interface gráfica, a linguagem de programação, o gerenciador de banco de dados etc.

Esta fase produz uma descrição *computacional* do que o software deve fazer e deve ser coerente com a descrição feita na análise. Em alguns casos, algumas restrições da tecnologia a ser utilizada já foram amarradas no levantamento de requisitos. Em outros casos, essas restrições devem ser especificadas. Mas, em todos os casos, a fase de projeto do sistema é direcionada pelos modelos construídos na fase de análise e pelo planejamento do sistema.

O 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*).

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 nesta fase do projeto são os diagramas de implementação. 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 as funcionalidades do módulo. 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 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. Principalmente no desenvolvimento de sistemas orientados a objetos, as atividades dessas duas fases frequentemente se misturam (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 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 na fase de projeto. O principal produto dessa fase é o relatório de testes, com informações sobre erros detectados no software. Após a atividade de testes, os diversos módulos do sistema são integrados, resultando finalmente no produto de software.

2.1.6 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, aqui 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 para fornecer o conhecimento técnico necessário ao desenvolvimento do sistema de software e 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 mais um fim 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 sua 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*. Esse profissional deve entender os problemas do domínio do negócio 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, ele deve ter suficiente domínio do vocabulário da área de conhecimento na qual o sistema será implantado para que, ao se comunicar com o especialista de domínio, este 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 entender não só 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, que mapeia 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 ele o faz, e determinar se as práticas atuais da organização realmente fazem sentido. 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 em seus quadros 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. Muitas vezes, esse profissional está 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 esses tipos é que eles 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 são os subsistemas que 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 (por exemplo, decisões que têm influência no desempenho do sistema). Esse profissional também trabalha em conjunto com o gerente de projeto para priorizar e organizar o plano de projeto.

2.2.5 Programadores

Esse profissional é o responsável 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, diferente-

[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.

mente 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.

Muitas vezes, bons programadores são "promovidos" a analistas de sistemas. Essa é uma prática comum nas empresas de desenvolvimento de software e é 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 para serem passadas a um programador.

2.2.6 Especialistas do domínio

Um outro componente da equipe de desenvolvimento é o *especialista do domínio*, também conhecido como *especialista do negócio*. Esse componente é o indivíduo, ou grupo de indivíduos, 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. O cliente usuário 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 esse usuário 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 de software 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 os clientes reais.

Em qualquer caso, a participação do cliente usuário no processo de desenvolvimento é de suma importância. O distanciamento de usuários do desenvolvimento do sistema se manifesta, sobretudo, em projetos que excedem o seu orçamento, estão atrasados ou que não correspondam 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 qual seja 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 diversas 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 é também 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 fase 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:

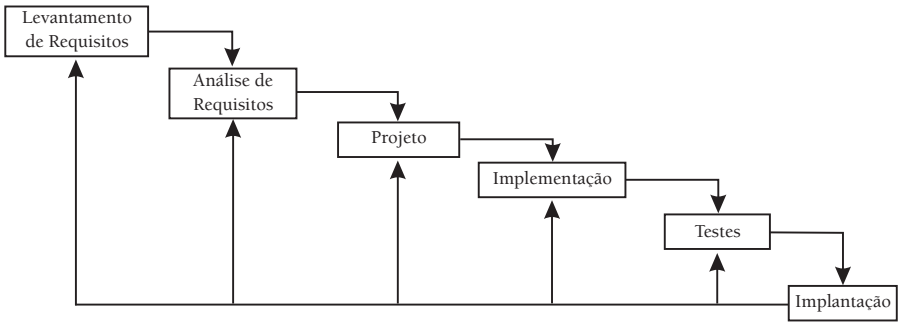

Figura 2-1: A abordagem de desenvolvimento de software em cascata.

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 antes 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, só sendo detectada 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 grandes em complexidade 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. Atualmente, os modelos de ciclo

[6] Uma versão de produção (em contraposição ao termo versão de desenvolvimento) de um software é uma versão que pode ser utilizada pelo usuário.

de vida mais utilizados para o desenvolvimento de sistemas complexos 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 segundo essa abordagem divide o desenvolvimento de um produto de software em ciclos. Em cada ciclo de desenvolvimento, 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, um 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 e cada incremento é desenvolvido 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 no par-

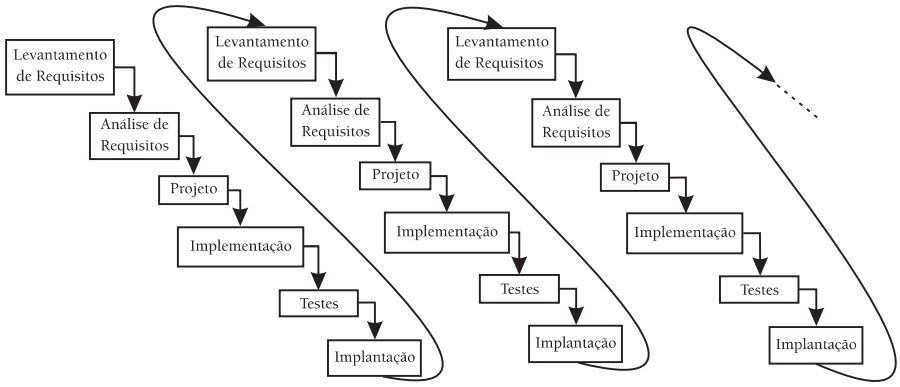

Figura 2-2: No processo incremental e iterativo, cada iteração é uma "minicascata".

ticionamento 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, através dos *casos de uso* do sistema.

No modelo de ciclo de vida incremental e iterativo, um sistema de software é desenvolvido em vários passos similares (*iterativo*). Em cada passo, o sistema é estendido com mais funcionalidades (*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 tal versão já corresponde ao sistema como um todo. De qualquer forma, o fato de essa abordagem incentivar a participação do usuário no processo de desenvolvimento de longe compensa qualquer falsa expectativa que este possa ter 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 considerados primeiramente. 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 tantas consequências graves para o projeto.

Os requisitos a serem considerados primeiramente devem ser selecionados com base nos riscos que eles fornecem. Os requisitos mais arriscados devem ser considerados tão logo 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 estes irão ativamente atacar você" (Gilb, 1988). Ou seja, quanto mais cedo 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 iteração, é produzido um *incremento*, ou seja, uma parte do sistema final. Um incremento pode ser liberado para os usuários, ou pode ser somente um incremento interno.

A dimensão de atividades compreende as 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 pro-

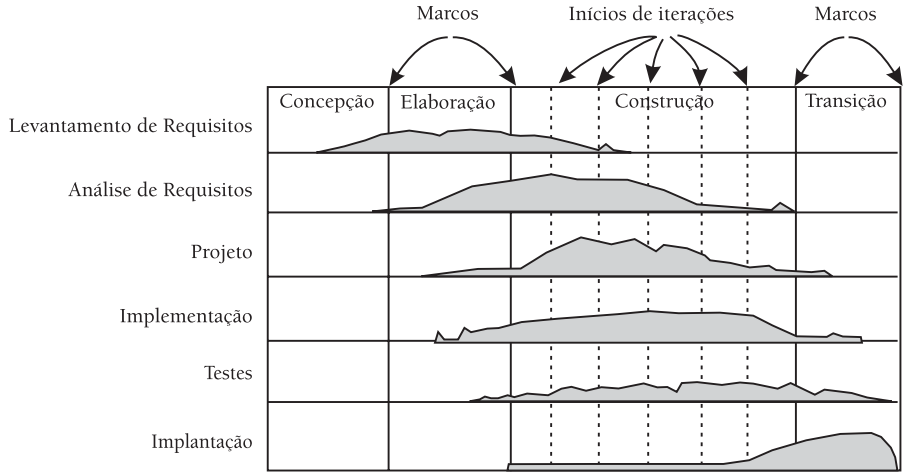

Figura 2-3: Estrutura geral de um processo de desenvolvimento incremental e iterativo.

jeto 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.

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, a Figura 2-3 permite perceber 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 que é 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 através 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 software, 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 necessariamente precisam ser construídos para aspectos da interface gráfica com o usuário. Em vez disso, qualquer aspecto que precise ser mais bem entendido é alvo 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 acerca de uma ou outra característica. O protótipo é então corrigido ou refinado de acordo com tais críticas. 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 através do 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 um substituto 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, pois são eles que guiam as demais fases do projeto de desenvolvimento de software. O ideal é os erros detectados na validação do protótipo serem 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. Tal atividade cooperativa pode ser facilitada pelo uso de ferramentas que auxi-

liam 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 que têm esse objetivo são as *ferramentas CASE* e os ambientes de desenvolvimento. Uma discussão detalhada sobre esses sistemas de apoio ao desenvolvimento está fora do escopo deste livro. Descrevemos sucintamente cada um desses dois tipos de software a seguir.

O termo CASE é uma sigla em inglês para *Engenharia de Software Auxiliada por Computador* (*Computer Aided Software Engineering*). 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, a seguir, algumas características são sucintamente descritas.

- Criação de diagramas e manutenção da consistência entre os mesmos. É comum a quase todas 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 um 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: *engenharia direta* e *engenharia 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, geração de diagramas a partir de código-fonte preexistente.
- Rastreamento de requisitos: uma facilidade 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 fornecerem 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 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, sim, 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: Baseado 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 (por exemplo, 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.

3
Mecanismos gerais

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

A UML consiste em três grandes componentes: blocos de construção básicos, regras que restringem como os blocos de construção 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 estão 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 a sua forma: *estereótipos gráficos* (ou ícones) e *estereótipos 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.

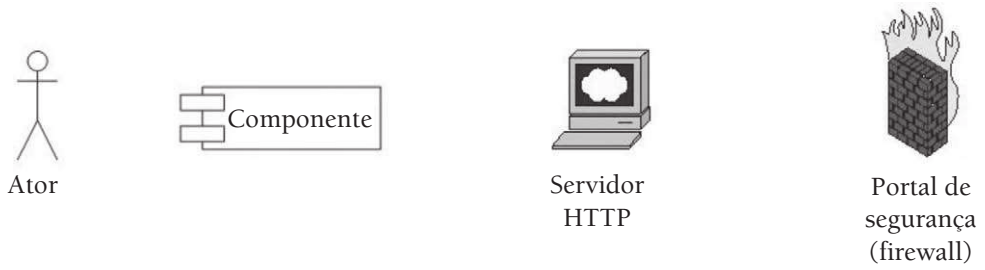

Figura 3-1: Exemplos de estereótipos gráficos.

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>>`, `<<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 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 através 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 elas 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 for realmente necessário.

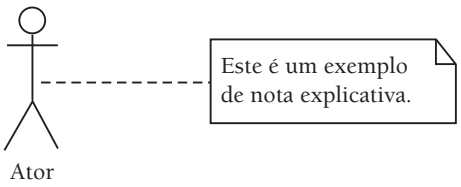

Figura 3-2: Exemplo de nota explicativa.

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. (Este último pode ser predefinido ou definido pelo usuário). Dessa forma, um determinado elemento de um modelo deve primeiramente ser estendido por um estereótipo antes de ser estendido por uma etiqueta valorada.

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

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

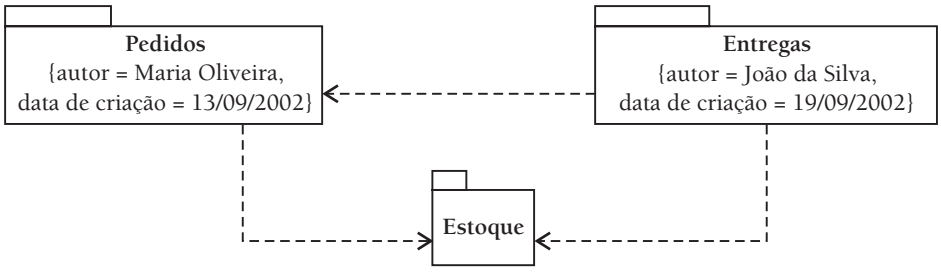

Figura 3-3: Utilização de etiquetas.

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 utilização.

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 através 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, um pacote tem um nome.

As ligações entre pacotes na Figura 3-3 são relacionamentos de dependência entre os pacotes. 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 forma é "pendurar" os elementos agrupados no ícone do pacote. A Figura 3-4 ilustra essas duas formas de representação.

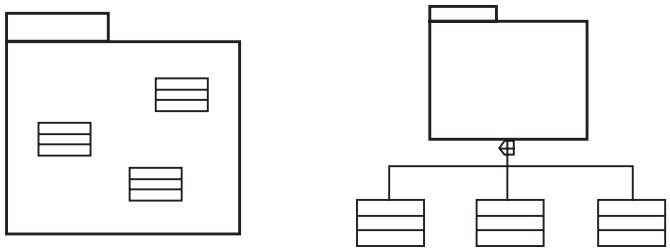

Figura 3-4: Formas de representar o conteúdo de um pacote.

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

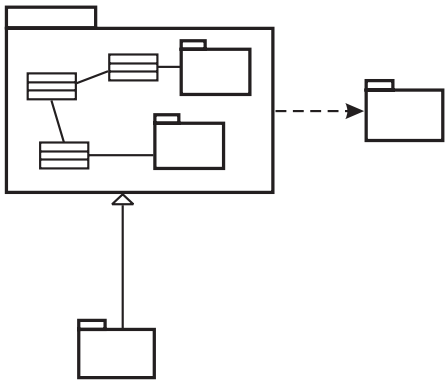

Figura 3-5: Pacotes podem conter outros pacotes.

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 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. 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.