APÊNDICE

2 Conceitos Orientados a Objeto

Conceitos-Chave

O que é um ponto de vista orientado a objeto? Por que um método é considerado orientado a objeto? O que é um objeto? À medida que os conceitos orientados a objeto ganharam ampla aceitação durante as décadas de 1980 e 1990, surgiram muitas opiniões diferentes sobre as respostas corretas a essas perguntas, mas atualmente há uma visão coerente dos conceitos orientados a objeto. Este apêndice lhe proporcionará uma visão rápida do tópico e apresentará os conceitos básicos e a terminologia.

Para entender o ponto de vista orientado a objeto, considere um exemplo prático – o objeto sobre o qual você está sentado agora – uma cadeira (*chair*). **Chair** é uma subclasse de uma classe muito maior que chamamos de PeçaDeMobília (**PieceOfFurniture**). Cadeiras individuais são membros (usualmente chamados de instâncias) da classe **Chair**. Um conjunto de atributos genéricos pode ser associado a cada objeto da classe **PieceOfFurniture**. Por exemplo, todos os móveis têm um custo, dimensões, peso, localização e cor, entre muitos outros atributos possíveis. Isso se aplica independentemente de estarmos falando de uma mesa (table) ou uma cadeira (chair), um sofá (sofa) ou um armário (armoire). Como **Chair** é um membro de **PieceOfFurniture**, **Chair** herda todos os atributos definidos para a classe.

Tentamos dar uma definição bem-humorada de uma classe descrevendo seus atributos, mas algo está faltando. Todo objeto na classe **PieceOfFurniture** pode ser manipulado de várias maneiras. Ele pode ser comprado e vendido, modificado fisicamente (por exemplo, você pode cortar uma perna da cadeira ou pintar o objeto de púrpura) ou movido de um lugar para outro. Cada uma dessas *operações* (outros termos são *serviços ou métodos*) modificará um ou mais atributos do objeto. Por exemplo, se o atributo localização for um item de dado composto definido como

localização = edifício + piso + sala

uma operação denominada *move()* modificaria um ou mais dos itens de dados (edifício, piso ou sala) que formam o atributo localização. Para isso, *move()* deve ter "conhecimento" desses itens de dados. A operação *move()* poderia ser usada para uma cadeira ou mesa, já que ambas são instâncias da classe **PieceOfFurniture**. Operações válidas para a classe **PieceOfFurniture** – *buy(), sell(), weigh()* – são especificadas como parte da definição de classe e herdadas por todas as instâncias da classe.

A classe **Chair** (e todos os objetos em geral) encapsula dados (os valores de atributo que definem a cadeira), operações (as ações aplicadas para mudar os atributos da cadeira), outros objetos, constantes (configuração de valores) e outras informações relacionadas. *Encapsulamento* significa que todas essas informações são empacotadas sob um nome e podem ser reutilizadas como uma especificação ou componente de programa.

Agora que introduzimos alguns conceitos básicos, terá mais sentido uma definição formal de *orientado a objeto*. Coad e Yourdon [Coa91] definem o termo da seguinte maneira:

Orientado a objeto = objetos + classificação + herança + comunicação

Três desses conceitos já foram apresentados. Comunicação será discutida mais adiante neste apêndice.

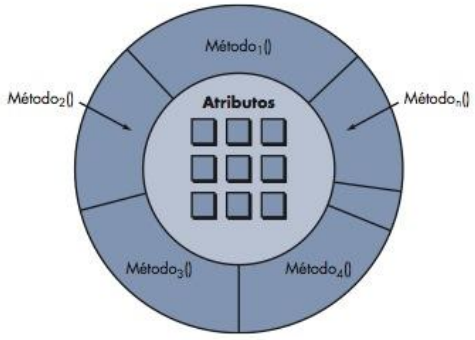
Classes e Objetos

Classe é um conceito orientado a objeto que encapsula dados e abstrações procedurais necessárias para descrever o conteúdo e comportamento de alguma entidade do mundo real. Abstrações de dados que descrevem a classe são envolvidas por uma "parede" de abstrações procedurais [Tay90] (representada na Figura A2.1) capazes de manipular de certa maneira os dados. Em uma classe bem projetada, a única maneira de atingir os atributos (e operar sobre eles) é passar através de um dos métodos que formam a "parede" ilustrada na figura. Portanto, a classe encapsula dados (dentro da parede) e o processamento que manipula os dados (os métodos que formam a parede). Isso possibilita o encapsulamento de informações (Capítulo 8) e reduz o impacto de efeitos colaterais associados a mudança. Como os métodos tendem a manipular um número limitado de atributos, sua coesão é melhorada, e porque a comunicação ocorre somente por meio dos métodos que formam a "parede", a classe tende a ser menos fortemente acoplada em relação a outros elementos de um sistema.[1]

Em outras palavras, podemos dizer que classe é uma descrição generalizada (por exemplo, um *template* ou *blueprint*) que descreve uma coleção de objetos similares. Por definição, objetos são instâncias de uma classe específica e herdam seus atributos e propriedades disponíveis para manipular os atributos. Uma *superclasse* (muitas vezes chamada de *classe base*) é a generalização de um conjunto de classes relacionadas a ela. Uma *subclasse* é uma especialização da superclasse. Por exemplo, a superclasse **MotorVehicle** é uma generalização da classe **Truck**, **SUV**, **Automobile** e **Van**. A subclasse **Automobile** herda todos os atributos de **MotorVehicle**, mas, além disso, incorpora atributos adicionais específicos apenas aos automóveis.

Essas definições implicam a existência de uma hierarquia de classes na qual atributos e operações da superclasse são herdados por subclasses que podem, cada uma delas, acrescentar atributos e métodos "privados" adicionais. Por exemplo, as operações *sitOn()* e *turn()* podem ser privadas à subclasse **Chair**.

Figura A2.1
Representação esquemática de uma classe



1 Deve-se notar, no entanto, que o acoplamento pode se tornar um sério problema em sistemas orientados a objeto. Ele surge quando classes de várias partes do sistema são usadas como tipos de dados de atributos e argumentos para métodos. Apesar do acesso aos objetos poder ser apenas por meio de chamadas de procedimento, isso não significa que o acoplamento seja necessariamente baixo, apenas menor do que seria se fosse permitido o acesso direto às partes internas do objeto.

Atributos

Você já aprendeu que os atributos são anexados às classes e que as descrevem de certa maneira. O atributo pode assumir um valor definido por um *domínio* enumerado. Em muitos casos, domínio é apenas um conjunto de valores específicos. Suponha que uma classe **Automobile** tenha o atributo cor. O domínio dos valores de cor é {**white, black, silver, gray, blue, red, yellow, green**}. Em situações mais complexas, o domínio pode ser uma classe. Continuando o exemplo, a classe **Automobile** também tem um atributo conjunto de tração (**powerTrain**) que por si só é uma classe. A classe **PowerTrain** teria os atributos que descrevem o motor e a transmissão específicos para determinado carro.

As *características* (valores do domínio) podem ser ampliadas atribuindo-se um valor-padrão a um atributo. Por exemplo, o atributo **cor** por padrão é **white**. Ele pode também ser útil para associar uma probabilidade com determinada característica atribuindo pares {valor, probabilidade}. Considere o atributo **cor** para automóvel. Em algumas aplicações (planejamento de manufatura) pode ser necessário atribuir uma probabilidade a cada uma das cores (branco e preto são altamente prováveis como cores de automóveis).

Operações, Métodos e Serviços

Um objeto encapsula dados (representados como uma coleção de atributos) e algoritmos que processam os dados. Esses algoritmos são chamados de *operações*, *métodos* ou *serviços*[2] e podem ser vistos como componentes de processamento.

Cada uma das operações encapsulada por um objeto proporciona uma representação de um dos comportamentos do objeto. A operação *GetColor()* para o objeto **Automobile** extrairá a cor armazenada no atributo **cor**. A implicação da existência dessa operação é que a classe **Automobile** foi projetada para receber um estímulo (chamamos o estímulo de *mensagem*) que solicita a cor da instância particular de uma classe. Sempre que um objeto recebe um estímulo, ele inicia algum comportamento. Isso pode ser algo simples como obter a cor do automóvel ou complexo como o início de uma cadeia de estímulos passados entre uma variedade de objetos diferentes. Neste último caso, considere um exemplo no qual o estímulo inicial recebido por **Object 1** resulta na geração de dois outros estímulos enviados para **Object 2** e **Object 3**. Operações encapsuladas pelo segundo e terceiro objetos agem sobre o estímulo, retornando informações necessárias para o primeiro objeto. **Object 1** usa as informações retornadas para satisfazer o comportamento demandado pelo estímulo inicial.

Conceitos de Análise e Projeto Orientado a Objeto

A modelagem de requisitos (também chamada de modelagem de análise) concentra-se primeiro sobre classes extraídas diretamente do enunciado do problema. Essas *classes de entidade* tipicamente representam itens que devem ser armazenados em uma base de dados e persistem durante toda a aplicação (a menos que sejam excluídas especificamente).

O projeto refina e amplia o conjunto de classes de entidade. Classes de fronteira e controladoras são desenvolvidas e/ou refinadas durante o projeto. Classes de fronteira criam a interface (por exemplo, telas interativas e relatórios impressos) que o usuário vê e com os quais interage à medida que o software é usado. Classes de fronteira são projetadas com a responsabilidade de controlar a maneira como os objetos entidade são representados para os usuários.

Classes controladoras são projetadas para controlar (1) a criação ou atualização de objetos entidade, (2) a instanciação de objetos de fronteira, já que obtêm informações dos objetos entidade, (3) comunicação complexa entre conjuntos de objetos, e (4) validação de dados transferidos entre objetos ou entre o usuário e a aplicação.

---

2 No contexto dessa discussão, é usado o termo *operações*, mas os termos *métodos* e *serviços* são igualmente populares.

Os conceitos discutidos nos próximos parágrafos podem ser úteis no trabalho de análise e projeto.

Herança. É um dos diferenciadores-chave entre sistemas convencionais e orientados a objeto. Uma subclasse **Y** herda todos os atributos e operações associadas a sua superclasse **X**. Isso significa que todas as estruturas de dados e algoritmos originalmente desenhados e implementados para **X** ficam imediatamente disponíveis para **Y** – nenhum trabalho adicional precisa ser feito. A reutilização foi conseguida diretamente.

Qualquer alteração nos atributos ou operações contidas dentro de uma superclasse é imediatamente herdada por todas as subclasses. Portanto, a hierarquia de classes torna-se um mecanismo pelo qual alterações (em altos níveis) podem ser imediatamente propagadas através de um sistema.

É importante notar que em cada nível de hierarquia de classe, novos atributos e operações podem ser acrescentados àqueles que foram herdados de níveis mais altos na hierarquia. De fato, sempre que uma nova classe deve ser criada, você tem um conjunto de opções:

- A classe pode ser projetada e construída do início. Isto é, não é usada a herança.
- A hierarquia de classe pode ser pesquisada para determinar se uma classe mais alta na hierarquia contém grande parte dos atributos e operações necessários. A nova classe herda da classe mais alta e podem ser feitas adições quando necessário.
- A hierarquia de classe pode ser reestruturada para que os atributos e operações necessários possam ser herdados pela nova classe.
- As características de uma classe existente podem ser anuladas, e diferentes versões de atributos ou operações são implementadas para a nova classe.

Como todos os conceitos fundamentais de projeto, a herança pode proporcionar benefício significativo ao projeto, mas se ela for usada de forma não apropriada,[3] pode complicar um projeto desnecessariamente e levar a um software passível de erros difícil de manter.

Mensagens. As classes devem interagir umas com as outras para atingir os objetivos do projeto. Uma mensagem estimula a ocorrência de algum comportamento no objeto receptor. O comportamento ocorre quando uma operação é executada.

A interação entre objetos é apresentada na Figura A2.2. Uma operação dentro de **SenderObject** gera uma mensagem da forma *mensagem* (<parâmetros>) em que os parâmetros identificam **ReceiverObject** como o objeto a ser estimulado pela mensagem, a operação dentro de **ReceiverObject** que deve receber a mensagem e os itens de dados que fornecem as informações necessárias para que a operação seja bem-sucedida. A colaboração definida entre classes como parte do modelo de requisitos fornece diretrizes úteis na criação das mensagens.

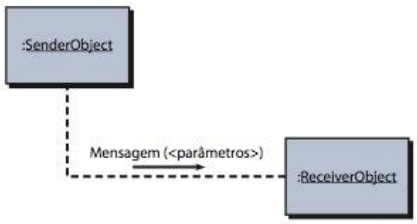
Cox [Cox86] descreve o intercâmbio entre classes da seguinte maneira:

> É solicitado a um objeto [classe] que execute uma de suas operações enviando-lhe uma mensagem com as informações do que fazer. O receptor [objeto] responde a mensagem primeiro escolhendo a operação que implementa o nome da mensagem, executando essa operação e, então, retornando o controle para o chamador. As mensagens unem um sistema orientado a objeto. As mensagens proporcionam informações sobre o comportamento dos objetos individuais e do sistema orientado a objeto como um todo.

Polimorfismo. Característica que reduz bastante o esforço necessário para ampliar o projeto de um sistema orientado a objeto. Para entender o polimorfismo, considere uma aplicação convencional que deve traçar quatro tipos diferentes de gráficos: gráficos de colunas, gráficos de pizza, histogramas e diagramas Kiviat. Idealmente, uma vez coletados os dados para um tipo particular de gráfico, o gráfico será traçado. Para conseguir isso em uma aplicação convencional

3 Projetar uma subclasse que herda atributos e operações de mais de uma superclasse (às vezes chamada de "herança múltipla") não é visto com simpatia por muitos projetistas.

FIGURA A2.2
Mensagens entre objetos



(e manter a coesão do módulo), seria necessário desenvolver módulos de desenho para cada tipo de gráfico. Então, no projeto, estaria embutida lógica de controle similar a esta a seguir:

```
case of graphtype:
   if graphtype = linegraph then DrawLineGraph (data);
   if graphtype = piechart then DrawPieChart (data);
   if graphtype = histogram then DrawHisto (data);
   if graphtype = kiviat then DrawKiviat (data);
end case;
```

Embora esse projeto seja razoavelmente simples, seria complicado adicionar novos tipos de gráficos. Um novo módulo de desenho teria de ser criado para cada tipo de gráfico e a lógica de controle teria de ser atualizada para refletir o novo tipo de gráfico.

Para resolver esse problema em um sistema orientado a objeto, todos os gráficos se tornam subclasses de uma classe geral denominada **Graph**. Usando um conceito chamado *sobrecarga* [Tay90], cada subclasse define uma operação *draw*. O objeto pode enviar uma mensagem *draw* a qualquer um dos objetos instanciados a partir de qualquer uma das subclasses. O objeto que está recebendo a mensagem chamará sua própria operação *draw* para criar o gráfico apropriado. Portanto, o projeto é reduzido a

```
draw <graphtype>
```

Quando um novo tipo de gráfico é acrescentado ao sistema, cria-se uma subclasse com sua própria operação *draw*. Mas não são necessárias alterações em qualquer objeto que queira que um gráfico seja desenhado, pois a mensagem **draw <graphtype> permanece inalterada**. Resumindo, o polimorfismo permite que várias operações diferentes tenham o mesmo nome. Isso, por sua vez, desacopla os objetos uns dos outros, tornando-os mais independentes.

Classes de Projeto. O modelo de requisitos define um conjunto completo de classes de análise. Cada uma descreve algum elemento do domínio do problema, focalizando os aspectos visíveis ao usuário ou ao cliente. O nível de abstração de uma classe de análise é relativamente alto.

Conforme evolui o modelo de projeto, a equipe de software deve definir um conjunto de *classes de projeto* que (1) refina as *classes de análise*, fornecendo detalhe de projeto que permitirá que sejam implementadas e (2) crie um novo conjunto de classes de projeto que implemente uma infraestrutura de software que suporte a solução de negócio. São sugeridos cinco tipos diferentes de classes de projeto, cada um representando uma camada diferente da arquitetura de projeto [Amb01]:

- *Classes de interface de usuário* definem todas as abstrações necessárias para a interação homem-computador (*human-computer interaction* – HCI).
- *Classes do domínio de negócio* são muitas vezes refinamentos das classes de análise definidas anteriormente. As classes identificam os atributos e operações (métodos) necessários para implementar algum elemento do domínio de negócio.

- *Classes de processo* implementam abstrações de negócio de baixo nível necessárias para gerenciar completamente as classes do domínio de negócios.
- *Classes persistentes* representam armazenamento de dados (por exemplo, uma base de dados) que persistirá além da execução do software.
- *Classes de sistema* implementam funções de gerenciamento e controle de software que permitem ao sistema operar e comunicar-se em seu ambiente de computação e com o mundo exterior.

À medida que evolui o projeto arquitetural, a equipe de software deverá desenvolver um conjunto completo de atributos e operações para cada classe de projeto. O nível de abstração é reduzido conforme cada classe de análise é transformada em uma representação de projeto. Isto é, classes de análise representam objetos (e métodos associados aplicados a eles) usando o jargão do domínio de negócio. Classes de projeto apresentam um detalhe significativamente mais técnico como um guia para a implementação.

Arlow e Neustadt [Arl02] sugerem que cada classe de projeto seja revista para garantir sua "boa formação". Eles definem quatro características de uma classe de projeto bem formada:

Completa e suficiente. Uma classe de projeto deverá ser o encapsulamento completo de todos os atributos e métodos requeridos a uma classe (com base em uma interpretação reconhecível do nome da classe). Por exemplo, a classe **Scene** definida para software de edição de vídeo é completa apenas se contiver todos os atributos e métodos que podem ser razoavelmente associados à criação de uma cena de vídeo. A suficiência garante que a classe de projeto contenha somente os métodos necessários para atingir a finalidade da classe, nem mais nem menos.

Primitivismo. Os métodos associados a uma classe de projeto devem se concentrar na execução de uma função específica para a classe. Uma vez que a função tenha sido implementada com um método, a classe não deve proporcionar nenhuma outra maneira de fazê-lo. Por exemplo, a classe **VideoClip** do software de edição de vídeo pode ter atributos **start-point** e **end-point** para indicar os pontos de início e fim do videoclipe (note que o vídeo "bruto" carregado no sistema pode ser maior do que o videoclipe usado). Os métodos, *setStartPoint()* e *setEndPoint()* proporcionam o único meio para estabelecer pontos de início e fim para o videoclipe.

Alta coesão. Uma classe de projeto coesa é limitada. Ela tem um conjunto de responsabilidades pequeno e concentrado e aplica de forma simples atributos e métodos para implementar aquelas responsabilidades. Por exemplo, a classe **VideoClip** do software de edição de vídeo pode conter um conjunto de métodos para editar o videoclipe. Contanto que cada método se concentre somente em atributos associados ao videoclipe, a coesão é mantida.

Baixo acoplamento. No modelo de projeto, é necessário que as classes de projeto colaborem umas com as outras. No entanto, a colaboração deverá ser mantida em um nível mínimo aceitável. Se um modelo de projeto é altamente acoplado (todas as classes de projeto colaboram com todas as outras classes de projeto), o sistema é difícil de implementar, testar e manter com o decorrer do tempo. Em geral, classes de projeto em um subsistema deverão ter apenas um limitado conhecimento das outras classes. Essa restrição, chamada de *Lei de Demeter* [Lie03], sugere que um método deverá mandar mensagens apenas para métodos em classes vizinhas.[4]

Leituras e Fontes de Informação Complementares

Durante as últimas três décadas, centenas de livros foram escritos sobre programação orientada a objeto, análise e projeto. Weisfeld (*The Object-Oriented Thought Process*, 2d ed., Sams

4 Uma maneira menos formal de enunciar a Lei de Demeter é: "Cada unidade deve falar somente com seus amigos; não deve falar com estranhos."

Publishing, 2003) apresenta um excelente tratamento dos conceitos e princípios gerais orientados a objeto. McLaughlin e seus colegas (*Head First Object-Oriented Analysis and Design: A Brain Friendly Guide to OOA&D*, O'Reilly Media, Inc., 2006) proporcionam um tratamento acessível e leve sobre abordagens de análise e projeto orientado a objeto. Um tratamento mais aprofundado sobre análise e projeto orientado a objeto é apresentado por Booch e seus colegas (*Object-Oriented Analysis and Design with Applications*, 3d ed., Addison-Wesley, 2007). Wu (*An Introduction to Object-Oriented Programming with Java*, McGraw-Hill, 2005) escreveu um livro esclarecedor sobre programação orientada a objeto, típico de dezenas de outras publicações escritas para muitas linguagens diferentes.

Uma grande variedade de fontes de informação sobre tecnologias orientadas a objeto está disponível na Internet. Uma lista atualizada das referências da Web pode ser encontrada em "análise" e "design" no site **www.mhhe.com/engcs/compsci/pressman/professional/olc/ser.htm**.