Q Padrões de design

Q.1 Introdução

A maioria dos exemplos fornecidos neste livro é relativamente pequena. Estes exemplos não requerem um extenso processo de design, pois utilizam poucas classes e ilustram os conceitos introdutórios de programação. Entretanto, alguns programas são mais complexos — podem requerer milhares de linhas de código ou mais, eles contêm muitas interações entre objetos e envolvem várias interações do usuário. Sistemas maiores, como os de controle de tráfego aéreo ou sistemas que controlam milhares de caixas eletrônicos de um banco importante, poderiam conter milhões de linhas de código. Um design eficaz é crucial à construção adequada desses sistemas complexos.

Nas últimas décadas, ocorreu na indústria de engenharia de software um enorme progresso no campo dos **padrões de design** — arquiteturas testadas para construir softwares orientados a objetos flexíveis e sustentáveis. Utilizar padrões de design reduz substancialmente a complexidade do processo de design. Projetar um sistema de controle de tráfego aéreo será uma tarefa menos complexa se desenvolvedores utilizarem padrões de design. Os padrões de design beneficiam os desenvolvedores de um sistema:

- ajudando a construir um software confiável com arquitetura testada e perícia acumulada pela indústria;
- promovendo a reutilização de designs em futuros sistemas;
- ajudando a identificar equívocos comuns e armadilhas que ocorrem ao se construir sistemas;
- ajudando a projetar sistemas independentemente da linguagem em que eles, em última instância, serão implementados;
- estabelecendo um vocabulário comum de design entre os desenvolvedores;
- encurtando a fase de design no processo de desenvolvimento de um software.

A noção da utilização de padrões de design para construir sistemas de softwares originou-se no campo da arquitetura. Os arquitetos utilizam uma série de elementos de design arquitetônico estabelecidos, como arcos e colunas, ao projetar edifícios. Projetar com arcos e colunas é uma estratégia testada para construir edifícios perfeitos — esses elementos podem ser vistos como padrões de design arquitetônicos.

Nos softwares, os padrões de design não são classes nem objetos. Em vez disso, os projetistas utilizam padrões de design para construir conjuntos de classes e objetos. Para utilizar padrões de design eficientemente, os projetistas devem conhecer os padrões mais famosos e eficientes utilizados na indústria de engenharia de software. Neste apêndice, discutimos padrões e arquiteturas fundamentais de design orientado a objetos e sua importância na construção de softwares bem-elaborados.

Este apêndice apresenta vários padrões de design em Java, mas estes podem ser implementados em qualquer linguagem orientada a objetos, como C++ ou Visual Basic. Descrevemos vários padrões de design utilizados pela Sun Microsystems na Java API. Utilizamos os padrões de design em muitos programas deste livro, identificados por toda a nossa discussão. Esses programas fornecem vários exemplos do uso de padrões de design para construir softwares orientados a objetos robustos e confiáveis.

História dos padrões de design orientado a objetos

Entre 1991 e 1994, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides — coletivamente conhecidos como "*Gang of Four*" — utilizaram suas perícias para escrever o livro *Design Patterns: Elements of Reusable Object-Oriented Software*. Essa obra descreve 23 padrões

de design, cada um fornecendo uma solução a um problema comum de design de software na indústria. O livro agrupa os padrões de design em três categorias — **padrões de design criacionais**, **padrões de design estruturais** e **padrões de design comportamentais**. Padrões de design criacionais descrevem as técnicas para instanciar objetos (ou grupos de objetos). Padrões de design estruturais permitem que os projetistas organizem classes e objetos em estruturas maiores. Padrões de design comportamentais atribuem responsabilidades a classes e objetos.

"*Gang of Four*" mostrou que os padrões de design evoluíram naturalmente ao longo dos anos de experiência da indústria. No seu artigo *Seven Habits of Successful Pattern Writers*,[1] John Vlissides afirma que "a atividade mais importante no processo de escrever padrões é a reflexão". Essa afirmação implica que, para criar padrões, os desenvolvedores devem refletir sobre e documentar seus sucessos (e equívocos). Os desenvolvedores utilizam os padrões de design para capturar e empregar essa experiência coletiva da indústria que, em última instância, ajuda-os a evitar a repetição dos mesmos equívocos. Novos padrões de design são criados o tempo todo e apresentados rapidamente aos projetistas do mundo todo via Internet.

Padrões de design são um tópico mais avançado que talvez não apareça nas sequências introdutórias da maioria dos cursos. À medida que você avança nos seus estudos sobre o Java, é certo que os padrões de design terão um valor maior. Se você é um aluno e seu instrutor não planeja incluir esse material no seu curso, encorajamos a leitura desse material por conta própria.

Q.2 Introduzindo padrões de design criacionais, estruturais e comportamentais

Na Seção Q.1, mencionamos que o "*Gang of Four*" descreveu 23 padrões de design utilizando três categorias — criacional, estrutural e comportamental. Nesta, e nas outras seções deste apêndice, discutimos os padrões de design em cada categoria e sua importância e como cada padrão se relaciona ao material sobre Java neste livro. Por exemplo, vários componentes Java Swing que apresentamos no Capítulo 14 e no Capítulo 25 utilizam o padrão de design Composite. A Figura Q.1 identifica os 18 padrões de design do "*Gang of Four*" discutidos neste apêndice.

Seção	Padrões de design criacionais	Padrões de design estruturais	Padrões de design comportamentais
Seção Q.2	Singleton	Proxy	Memento, State
Seção Q.3	Factory Method	Adapter, Bridge, Composite	Chain of Responsibility, Command, Observer, Strategy, Template Method
Seção Q.5	Abstract Factory	Decorator, Facade	
Seção Q.6	Prototype		Iterator

Figura Q.1 | 18 padrões de design do "Gang of Four" discutidos neste apêndice.

Muitos padrões populares foram documentados a partir do livro do "*Gang of Four*" — estes incluem os **padrões de design de concorrência**, que são especialmente úteis no design de sistemas com múltiplas threads. A Seção Q.4 discute alguns desses padrões utilizados na indústria. Padrões arquitetônicos, como discutido na Seção Q.5, especificam como subsistemas interagem um com o outro. A Figura Q.2 lista os padrões de concorrência e os padrões arquitetônicos que discutimos neste apêndice.

Seção	Padrões de design de concorrência	Padrões arquitetônicos
Seção Q.4	Single-Threaded Execution, Guarded Suspension, Balking, Read/Write Lock, Terminação de duas fases	
Seção Q.5		Model-View-Controller, Layers

Figura Q.2 | Padrões de design de concorrência e padrões arquitetônicos discutidos neste apêndice.

Q.2.1 Padrões de design criacionais

Padrões de design criacionais abordam questões relacionadas à criação de objetos, por exemplo, impedir que um sistema crie mais de um objeto de uma classe (o padrão de design criacional Singleton) ou postergar, até o tempo de execução, a decisão sobre quais tipos de objetos serão criados (o propósito dos outros padrões de design criacionais aqui discutidos). Por exemplo, suponha que estamos projetando um programa de desenho em 3-D, em que o usuário pode criar vários objetos geométricos em 3-D como cilindros, esferas, cubos, tetraedros etc. Suponha

[1] Vlissides, J. *Pattern Hatching: Design Patterns Applied*. Reading, MA: Addison-Wesley, 1998.

ainda que cada forma no programa de desenho seja representada por um objeto. Em tempo de compilação, o programa não sabe quais formas o usuário irá escolher para desenhar. Com base na entrada do usuário, esse programa deve ser capaz de determinar em que classe instanciar um objeto apropriado para a forma que o usuário selecionou. Se o usuário criar um cilindro na GUI, nosso programa deve "saber" como instanciar um objeto da classe `Cylinder`. Quando o usuário decide qual objeto geométrico desenhar, o programa deve determinar em que subclasse específica instanciar esse objeto.

O livro do "*Gang of Four*" descreve cinco padrões criacionais (quatro deles discutidos neste apêndice):

- Abstract Factory (Seção Q.5)
- Builder (não discutido)
- Factory Method (Seção Q.3)
- Prototype (Seção Q.6)
- Singleton (Seção Q.2)

Singleton

Ocasionalmente, um sistema deve conter exatamente um único objeto de uma classe — isto é, depois que o programa instancia esse objeto, o programa não deve ter permissão de criar objetos adicionais dessa classe. Por exemplo, alguns sistemas são conectados a um banco de dados utilizando apenas um objeto que gerencia as conexões ao banco de dados, isso assegura que outros objetos não inicializem conexões desnecessárias que tornariam o sistema lento. O **padrão de design Singleton** garante que um sistema instancia no máximo um objeto de uma classe.

A Figura Q.3 demonstra o código Java utilizando o padrão de design Singleton. A linha 4 declara a classe `Singleton` como `final` de modo que não possam ser criadas subclasses que forneceriam múltiplas instanciações. As linhas 10–13 declaram um construtor `private` — somente a classe `Singleton` pode instanciar um objeto `Singleton` utilizando esse construtor. A linha 7 declara uma referência estática a um objeto `Singleton` e invoca o construtor privado. Isso cria uma instância da classe Singleton que será fornecida aos clientes. Quando invocado, o método estático `getSingletonInstance` (linhas 16–19) simplesmente retorna uma cópia dessa referência.

As linhas 9–10 da classe `SingletonTest` (Figura. Q.4) declaram duas referências a objetos `Singleton` — `firstSingleton` e `secondSingleton`. As linhas 13–14 chamam o método `getSingletonInstance` e atribuem referências `Singleton` a `firstSingleton` e `secondSingleton`, respectivamente. A linha 17 testa se essas referências se referem ao mesmo objeto `Singleton`. A Figura Q.4 mostra que `firstSingleton` e `secondSingleton` são de fato referências ao mesmo objeto `Singleton`, porque toda vez que o método `getSingleton-Instance` é chamado, ele retorna uma referência ao mesmo objeto `Singleton`.

```
1  // Singleton.java
2  // Demonstra o padrão de design Singleton
3
4  public final class Singleton
5  {
6     // Objeto Singleton a ser retornado por getSingletonInstance
7     private static final Singleton singleton = new Singleton();
8
9     // construtor privado impede a instanciação pelos clientes
10    private Singleton()
11    {
12       System.err.println( "Singleton object created." );
13    } // fim do construtor Singleton
14
15    // retorna o objeto Singleton estático
16    public static Singleton getInstance()
17    {
18       return singleton;
19    } // fim do método getInstance
20 } // fim da classe Singleton
```

Figura Q.3 | A classe `Singleton` assegura que somente um objeto de sua classe seja criado.

```
1  // SingletonTest.java
2  // Tenta criar dois objetos Singleton
3
4  public class SingletonTest
5  {
```

```
6       // executa SingletonExample
7       public static void main( String[] args )
8       {
9          Singleton firstSingleton;
10         Singleton secondSingleton;
11
12         // cria objetos Singleton
13         firstSingleton = Singleton.getInstance();
14         secondSingleton = Singleton.getInstance();
15
16         // o dois "Singletons" devem se referir ao mesmo Singleton
17         if (firstSingleton == secondSingleton )
18            System.err.println( "firstSingleton and secondSingleton " +
19               "refer to the same Singleton object" );
20      } // fim de main
21   } // fim da classe SingletonTest
```

```
Singleton object created.
firstSingleton and secondSingleton refer to the same Singleton object
```

Figura Q.4 | A classe `SingletonTest` cria um objeto `Singleton` mais de uma vez.

Q.2.2 Padrões de design estruturais

Padrões de design estruturais descrevem maneiras comuns de organizar classes e objetos em um sistema. O livro do "*Gang of Four*" descreve sete padrões de design estruturais (seis deles discutidos neste apêndice) :

- Adapter (Seção Q.3)
- Bridge (Seção Q.3)
- Composite (Seção Q.3)
- Decorator (Seção Q.5)
- Facade (Seção Q.5)
- Flyweight (não discutido)
- Proxy (Seção Q.2)

Proxy

Um applet sempre deve exibir algo enquanto imagens são carregadas a fim de fornecer um feedback positivo aos usuários para que eles saibam que o applet está funcionando. Se esse "algo" for uma imagem menor ou uma string de texto informando o usuário de que as imagens estão sendo carregadas, o **padrão de design Proxy** poderá ser aplicado para alcançar esse efeito. Considere o carregamento de várias imagens grandes (vários megabytes) em um applet Java. Idealmente, gostaríamos de ver essas imagens instantaneamente — entretanto, o processo para carregar imagens grandes na memória pode demorar (especialmente por uma rede). O padrão de design Proxy permite que o sistema utilize um objeto — chamado **objeto proxy** — no lugar de outro. No nosso exemplo, o objeto proxy poderia ser uma medida que mostra ao usuário a porcentagem carregada de uma grande imagem. Quando essa imagem termina de carregar, o objeto proxy não é mais necessário — o applet pode então exibir uma imagem em vez do objeto proxy. A classe `javax.swing.JProgressBar` pode ser utilizada para criar esses objetos proxy.

Q.2.3 Padrões de design comportamentais

Os **padrões de design comportamentais** fornecem estratégias testadas para modelar a maneira como os objetos colaboram entre si em um sistema e oferecem comportamentos especiais apropriados para uma ampla variedade de aplicativos. Vamos considerar o padrão de design comportamental Observer — um exemplo clássico que ilustra colaborações entre objetos. Por exemplo, componentes GUI colaboram com seus ouvintes para responder a interações do usuário. Os componentes GUI utilizam esse padrão para processar eventos da interface com o usuário. Um ouvinte observa alterações de estado em um componente GUI particular registrando-se para tratar os eventos nessa GUI. Quando o usuário interage com esse componente GUI, o componente notifica seus ouvintes (também conhecido como observadores) de que seu estado mudou (por exemplo, um botão foi pressionado).

Também consideramos o padrão de design comportamental Memento — um exemplo para oferecer um comportamento especial a muitos aplicativos. O padrão Memento permite que um sistema salve o estado de um objeto, de modo que esse estado possa ser restaurado posteriormente. Por exemplo, muitos aplicativos fornecem uma capacidade de "desfazer" que permite aos usuários reverterem para versões prévias dos seus trabalhos.

O livro do "*Gang of Four*" descreve 11 padrões de design comportamentais (discutimos oito neste apêndice):

- Chain of Responsibility (Seção Q.3)
- Command (Seção Q.3)
- Interpreter (não discutido)
- Iterator (Seção Q.2)
- Mediator (não discutido)
- Memento (Seção Q.2)
- Observer (Seção Q.3)
- State (Seção Q.2)
- Strategy (Seção Q.3)
- Template Method (Seção Q.3)
- Visitor (não discutido)

Memento

Considere um programa de pintura, que permite a um usuário criar imagens gráficas. Ocasionalmente, o usuário talvez posicione uma imagem gráfica impropriamente na área de desenho. Programas de pintura oferecem um recurso de "desfazer" ("undo") que permite ao usuário reverter esse erro. Especificamente, o programa restaura a área de desenho ao seu estado antes de o usuário ter posicionado a imagem gráfica. Programas de pintura mais sofisticados oferecem um histórico, que armazena vários estados em uma lista, permitindo que o usuário restaure o programa de acordo com qualquer estado no histórico.O **padrão de design Memento** permite a um objeto salvar seu estado, de modo que — se necessário — o objeto possa ser restaurado ao seu estado inicial.

O padrão de design Memento exige três tipos de objetos. O **objeto originador** ocupa algum estado — o conjunto de valores dos atributos em um momento específico na execução do programa. No nosso exemplo do programa de pintura, a área de desenho atua como o originador, pois contém informações sobre o atributo descrevendo seu estado — quando o programa é executado pela primeira vez, a área não conterá nenhum elemento. O **objeto memento** armazena uma cópia dos atributos necessários associados com o estado do originador (isto é, o memento salva o estado da área de desenho). O memento é armazenado como o primeiro item na lista do histórico, que atua como o **objeto "zelador"** — o objeto que contém as referências a todos os objetos memento associadas ao originador. Agora, suponha que o usuário desenhe um círculo na área de desenho. A área contém diferentes informações que descrevem seu estado — um objeto círculo centralizado nas coordenadas *x*-*y* especificadas. A área de desenho então utiliza outro memento para armazenar essas informações. Esse memento se torna o segundo item na lista de histórico. A lista de histórico exibe todos os mementos na tela, assim o usuário pode selecionar qual estado restaurar. Suponha que o usuário deseja remover o círculo — se o usuário selecionar o primeiro memento, a área de desenho irá utilizá-lo para restaurar a área de desenho em branco.

Estado

Em alguns designs, devemos comunicar as informações sobre o estado de um objeto ou representar os vários estados que um objeto pode ocupar. O **padrão de design State** utiliza uma superclasse abstrata — chamada **classe State** — que contém os métodos que descrevem os comportamentos dos estados que um objeto (chamado **objeto de contexto**) pode ocupar. Uma **subclasse State**, que estende a classe State, representa um estado individual que o contexto pode ocupar. Cada subclasse State contém os métodos que implementam os métodos abstratos da classe State. O contexto contém exatamente uma única referência a um objeto da classe State — esse objeto é chamado **objeto state**. Quando o contexto altera o estado, o objeto state faz referência ao objeto da subclasse State associado a esse novo estado.

Q.2.4 Conclusão

Nesta seção, listamos os três tipos de padrões de design introduzidos no livro do "*Gang of Four*", identificamos 18 desses padrões de design discutidos neste apêndice e discutimos padrões de design específicos: Singleton, Proxy, Memento e State. Na próxima seção, introduziremos alguns padrões de design associados com o AWT e componentes Swing GUI.

Q.3 Padrões de design nos pacotes `java.awt` e `javax.swing`

Esta seção introduz aqueles padrões de design associados com componentes GUI Java. Ela ajuda a entender melhor como esses componentes tiram proveito dos padrões de design e como os desenvolvedores integram padrões de design a aplicativos da GUI Java.

Q.3.1 Padrões de design criacionais

Agora, continuaremos nosso tratamento dos padrões de design criacionais que fornecem maneiras de instanciar objetos em um sistema.

Factory Method

Suponha que estamos projetando um sistema que abre uma imagem a partir de um arquivo especificado. Há vários diferentes formatos de imagens, como GIF e JPEG. Podemos utilizar o método `createImage` da classe `java.awt.Component` para criar um objeto `Image`. Por exemplo, para criar uma imagem JPEG e GIF em um objeto de uma subclasse `Component` — como um objeto `JPanel` — passamos o nome do arquivo da imagem para o método `createImage`, que retorna um objeto `Image` o qual armazena os dados da imagem. Podemos criar dois objetos `Image`, cada um contendo dados para duas imagens com estruturas completamente diferentes. Por exemplo, uma imagem JPEG pode conter até 16,7 milhões de cores, uma imagem GIF apenas 256. Além disso, uma imagem GIF pode conter pixels transparentes que não são renderizados na tela, enquanto uma imagem JPEG não pode fazer isso.

A classe `Image` é uma classe abstrata que representa uma imagem que podemos exibir na tela. Utilizando o parâmetro passado pelo programador, o método `createImage` determina a subclasse `Image` específica a partir da qual é possível instanciar o objeto `Image`. Podemos projetar sistemas para permitir que o usuário especifique a imagem a ser criada e o método `createImage` determinará em que subclasse instanciar a `Image`. Se o parâmetro passado para o método `createImage` fizer referência a um arquivo JPEG, o método `createImage` irá instanciar e retornar um objeto de uma subclasse `Image` adequada para imagens JPEG. Se o parâmetro fizer referência a um arquivo GIF, `createImage` irá instanciar e retornar um objeto de uma subclasse `Image` adequada para imagens GIF.

O método `createImage` é um exemplo do **padrão de design Factory Method**. O propósito exclusivo desse **método factory** é criar objetos permitindo que o sistema determine qual classe instanciar em tempo de execução. Podemos projetar um sistema que permite a um usuário especificar qual tipo de imagem criar em tempo de execução. A classe `Component` talvez não seja capaz de determinar qual subclasse `Image` instanciar até que o usuário especifique a imagem a ser carregada. Para informações adicionais sobre o método `createImage`, visite

```
java.sun.com/javase/6/docs/api/java/awt/Component.html
```

Q.3.2 Padrões de design estruturais

Agora, discutiremos mais três padrões de design estruturais. O padrão de design Adapter ajuda os objetos com interfaces incompatíveis a colaborarem entre si. O padrão de design Bridge ajuda os projetistas a aprimorarem a independência de plataformas nos seus sistemas. O padrão de design Composite fornece uma maneira para que os projetistas organizem e manipulem objetos.

Adapter

O **padrão de design Adapter** fornece a um objeto uma nova interface que *se adapta* à interface de outro objeto, permitindo que os dois objetos colaborem entre si. Poderíamos equiparar o adaptador nesse padrão a um adaptador de tomada em um dispositivo elétrico — soquetes elétricos na Europa têm uma forma diferente daqueles nos Estados Unidos, portanto, é necessário um adaptador para conectar um dispositivo norte-americano a uma soquete europeu e vice-versa.

O Java fornece várias classes que utilizam o padrão de design Adapter. Os objetos das subclasses concretas dessas classes atuam como adaptadores entre objetos que geram certos eventos e objetos que tratam os eventos. Por exemplo, um `MouseAdapter`, que explicamos na Seção 14.15, adapta um objeto que gera `MouseEvents` para um objeto que trata `Mouse-EventS`.

Bridge

Suponha que estamos projetando a classe `Button` tanto para sistemas operacionais Windows como Macintosh. A classe `Button` contém informações específicas sobre o botão como um `ActionListener` e um rótulo. Criamos classes `Win32Button` e `MacButton` para estender a classe `Button`. A classe `Win32Button` contém informações sobre a "aparência e comportamento" de como exibir um `Button` no sistema operacional Windows e a classe `MacButton` contém informações sobre a "aparência e comportamento" de como exibir um `Button` no sistema operacional Macintosh.

Aqui, surgem dois problemas. Primeiro, se criarmos novas subclasses `Button`, precisaremos criar subclasses `Win32Button` e `MacButton` correspondentes. Por exemplo, se criarmos a classe `ImageButton` (uma `Button` com uma sobreposição `Image`) que estende a classe `Button`, precisaremos criar subclasses adicionais `Win32ImageButton` e `MacImageButton`. De fato, precisaremos criar subclasses `Button` para cada sistema operacional que desejamos suportar, o que aumenta o tempo de desenvolvimento. Segundo, quando um novo sistema operacional aparece no mercado, precisaremos criar subclasses `Button` adicionais específicas para esse novo sistema.

O **padrão de design Bridge** evita esses problemas dividindo uma abstração (por exemplo, um `Button`) e suas implementações (por exemplo, `Win32Button`, `MacButton` etc.) em hierarquias de classes separadas. Por exemplo, as classes Java AWT utilizam o padrão de design Bridge para permitir que os projetistas criem subclasses `Button` AWT sem a necessidade de criar subclasses adicionais específicas ao sistema operacional. Cada `Button` AWT mantém uma referência a um `ButtonPeer`, que é a superclasse para implementações específicas de plataforma, como `Win32ButtonPeer`, `MacButtonPeer` etc. Quando um programador cria um objeto `Button`, a classe `Button` chama o método factory `createButton` da classe `Toolkit` para criar o objeto `ButtonPeer` específico à plataforma. O objeto `Button` armazena uma referência ao seu `ButtonPeer` — essa referência é a "ponte" no padrão de design Bridge. Quando o programador invoca os métodos no objeto `Button`, o objeto `Button` delega o trabalho ao método de nível mais baixo apropriado no seu `ButtonPeer` para atender à solicitação. Um projetista que cria uma subclasse `Button` chamada, por exemplo, `ImageButton`, não precisa criar uma `Win32ImageButton` ou `MacImageButton` correspondente com capacidades de desenho de imagens específicas à plataforma. Um `ImageButton` *é um* `Button`. Portanto,

quando um `ImageButton` precisa exibir sua imagem, o `ImageButton` utiliza o objeto `Graphics` do seu `ButtonPeer` para renderizar essa imagem em cada plataforma. Esse padrão de design permite que projetistas criem novos componentes GUI para diversas plataformas utilizando uma "ponte" para ocultar detalhes específicos à plataforma.

Dica de portabilidade Q.1

Os projetistas costumam utilizar o padrão de design Bridge a fim de aprimorar a independência de plataforma dos seus sistemas. Esse padrão de design permite que os projetistas criem novos componentes para diversas plataformas utilizando uma "ponte" para ocultar detalhes específicos à plataforma.

Composite

Os projetistas frequentemente organizam componentes em estruturas hierárquicas (por exemplo, uma hierarquia de diretórios e arquivos em um sistema de arquivos) — cada nó na estrutura representa um componente (por exemplo, um arquivo ou diretório). Cada nó pode conter referências a um ou mais outros nós e, se contiver, será chamado **ramificação** (por exemplo, um diretório contendo arquivos); caso contrário, é chamado **folha** (por exemplo, um arquivo). Algumas vezes, uma estrutura contém objetos de várias classes diferentes (por exemplo, um diretório pode conter arquivos e diretórios). Um objeto — chamado **cliente** — que quer percorrer a estrutura precisa determinar a classe específica de cada nó. Fazer essa determinação pode ser demorado e a estrutura pode tornar-se difícil de manter.

No **padrão de design Composite**, cada componente em uma estrutura hierárquica implementa a mesma interface ou estende uma superclasse comum. Esse polimorfismo (introduzido no Capítulo 10) assegura que os clientes possam percorrer todos os elementos — a ramificação ou folha — uniformemente na estrutura sem precisar determinar cada tipo de componente, porque todos os componentes implementam a mesma interface ou estendem a mesma superclasse.

Componentes GUI Java utilizam o padrão de design Composite. Considere a classe de componente Swing `JPanel`, que estende a classe `JComponent`. A classe `JComponent` estende a classe `java.awt.Container`, que estende a classe `java.awt.Component` (Figura. Q.5). A classe `Container` fornece o método `add`, que acrescenta um objeto `Component` (ou objeto da subclasse `Component`) a esse objeto `Container`. Portanto, um objeto `JPanel` poderia ser adicionado a qualquer objeto de uma subclasse `Component`, e qualquer objeto de uma subclasse `Component` poderia ser adicionada a esse objeto `JPanel`. Um objeto `JPanel` pode conter qualquer componente GUI sem precisar conhecer seu tipo específico. Quase todas as classes GUI são contêineres e componentes, permitindo aninhamento e estruturação arbitrariamente complexa de GUIs.

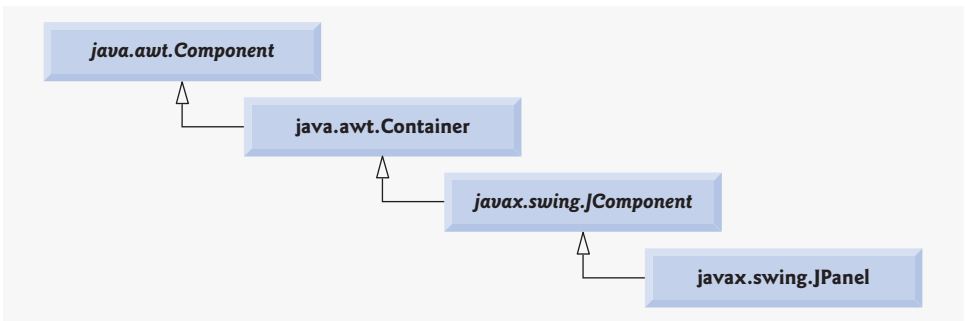

Figura Q.5 | Hierarquia de herança da classe `JPanel`.

Um cliente, como um objeto `JPanel`, pode percorrer todos os componentes uniformemente na hierarquia. Por exemplo, se o objeto `JPanel` chamar o método `repaint` da superclasse `Container`, o método `repaint` exibirá o objeto `JPanel` e todos os componentes adicionado ao objeto `JPanel`. O método `repaint` não tem de determinar cada tipo do componente, uma vez que todos os componentes herdam da superclasse `Container`, que contém o método `repaint`.

Q.3.3 Padrões de design comportamentais

Esta seção continua nossa discussão sobre os padrões de design comportamentais. Discutimos os padrões de design Chain of Responsibility, Command, Observer, Strategy e Template Method.

Chain of Responsibility

Em sistemas orientados a objetos, os objetos interagem por meio do envio de mensagens. Frequentemente, um sistema precisa determinar em tempo de execução o objeto que tratará uma mensagem particular. Por exemplo, considere o design de um sistema de telefonia com três linhas para um escritório. Quando alguém chama o escritório, a primeira linha trata a chamada — se a primeira linha estiver ocupada, a segunda linha tratará a chamada e se a segunda linha estiver ocupada, a terceira linha tratará a chamada. Se todas as linhas no sistema estiverem ocupadas, um sistema de voz automatizado instrui o usuário a esperar a próxima linha disponível. Quando uma linha estiver disponível, essa linha tratará a chamada.

O **padrão de design Chain of Responsibility** permite que um sistema determine em tempo de execução o objeto que tratará uma mensagem. Esse padrão permite que um objeto envie uma mensagem para vários objetos em uma **cadeia**. Cada objeto na cadeia pode tratar a mensagem ou passá-la para o próximo objeto. Por exemplo, a primeira linha no sistema de telefonia é o primeiro objeto na cadeia de responsabilidades, a segunda linha é o segundo objeto, a terceira linha é o terceiro objeto e a secretária eletrônica é o quarto objeto. O objeto final na cadeia é a próxima linha disponível que trata a mensagem. A cadeia é criada dinamicamente em resposta à presença ou ausência de operadores específicos de mensagens.

Vários componentes GUI Java AWT utilizam o padrão de design Chain of Responsibility para tratar certos eventos. Por exemplo, a classe `java.awt.Button` sobrescreve o método `processEvent` da classe `java.awt.Component` para processar objetos `AWTEvent`. O método `processEvent` tenta tratar o `AWTEvent` no recebimento dele como um argumento. Se o método `processEvent` determinar que o `AWTEvent` é um `ActionEvent` (isto é, o `Button` foi pressionado), ele tratará o evento invocando o método `processActionEvent`, que informa a qualquer `ActionListener` registrado no `Button` de que o `Button` foi pressionado. Se o método `processEvent` determinar que o `AWTEvent` não é um `ActionEvent`, o método não será capaz de tratá-lo e irá passá-lo para o método `processEvent` da superclasse `Component` (o próximo ouvinte na cadeia).

Comando

Frequentemente, os aplicativos fornecem aos usuários várias maneiras de realizar uma dada tarefa. Por exemplo, em um processador de texto pode haver um menu **Edit** com itens no menu para cortar, copiar e colar texto. Uma barra de ferramentas ou um menu pop-up também poderia oferecer os mesmos itens. A funcionalidade que o aplicativo fornece é a mesma em cada caso — os diferentes componentes de interface para invocar a funcionalidade são oferecidos como uma conveniência ao usuário. Entretanto, a mesma instância do componente GUI (por exemplo, `JButton`) não pode ser utilizada para menus, barras de ferramentas e menus pop-up, portanto o desenvolvedor deve codificar a mesma funcionalidade três vezes. Se houver muitos desses itens na interface, repetir essa funcionalidade seria entediante e propenso a erros.

O **padrão de design Command** soluciona esse problema permitindo que os desenvolvedores encapsulem, uma vez, a funcionalidade desejada (por exemplo, copiar texto) em um objeto reutilizável; essa funcionalidade pode então ser adicionada a um menu, barra de ferramentas, menu pop-up ou outro mecanismo. Esse padrão de design é chamado Command porque ele define um comando, ou instrução, a ser executado. Ele permite que um projetista encapsule um comando de modo que ele possa ser utilizado entre vários objetos.

Observer

Suponha que queiramos projetar um programa para visualizar as informações sobre uma conta bancária. Esse sistema inclui a classe `BankStatementData` para armazenar dados relacionados às instruções do banco e as classes `TextDisplay`, `BarGraphDisplay` e `PieChartDisplay` para exibir os dados. [*Nota*: Essa abordagem é a base do padrão arquitetônico Model-View-Controller, discutido na Seção Q.5.3] A Figura Q.6 mostra o design do nosso sistema. Os dados são exibidos pela classe `TextDisplay` no formato de texto, pela classe `BarGraphDisplay` no formato de gráfico de barras e pela classe `PieChartDisplay` como um gráfico de pizza. Queremos projetar o sistema de modo que o objeto `BankStatementData` notifique os objetos que exibem os dados sobre uma alteração nos dados. Também queremos projetar o sistema com um **acoplamento** fraco — o grau de dependência entre classes em um sistema.

Observação de engenharia de software Q.1

Classes com acoplamento fraco são mais fáceis de reutilizar e modificar do que classes com um acoplamento forte, as quais dependem maciçamente uma da outra. Uma modificação de uma classe em um sistema com um acoplamento fraco normalmente resulta na modificação das outras classes nesse sistema. Uma modificação para uma classe em um grupo de classes com acoplamento fraco exigiria pouca ou nenhuma modificação nas outras classes.

O **padrão de design Observer** é apropriado para sistemas como o da Figura Q.6. Esse padrão promove o acoplamento fraco entre um **objeto-assunto** e **objetos observadores** — um objeto-assunto notifica os objetos observadores quando o assunto muda de estado. Quando notificado pelo assunto, os observadores mudam em resposta. No nosso exemplo, o objeto `BankStatementData` é o assunto e os objetos que exibem os dados são os observadores. Um assunto pode notificar vários observadores; portanto, o assunto tem um relacionamento de um para muitos com os observadores.

A Java API contém as classes que utilizam o padrão de design Observer. A classe **`java.util.Observable`** representa um assunto. A classe `Observable` fornece o método `addObserver`, que recebe um argumento **`java.util.Observer`**. A interface `Observer` permite que o objeto `Observable` notifique o `Observer` quando o objeto `Observable` altera o estado. O `Observer` pode ser uma instância de qualquer classe que implementa a interface `Observer`; visto que o objeto `Observable` invoca os métodos declarados na interface `Observer`, os objetos permanecem fracamente acoplados. Se um desenvolvedor alterar a maneira como um `Observer` particular responde às alterações no objeto `Observable`, o desenvolvedor não precisará alterar esse objeto. O objeto `Observable` só interage com seus `Observers` por meio da interface `Observer`, que permite acoplamento fraco.

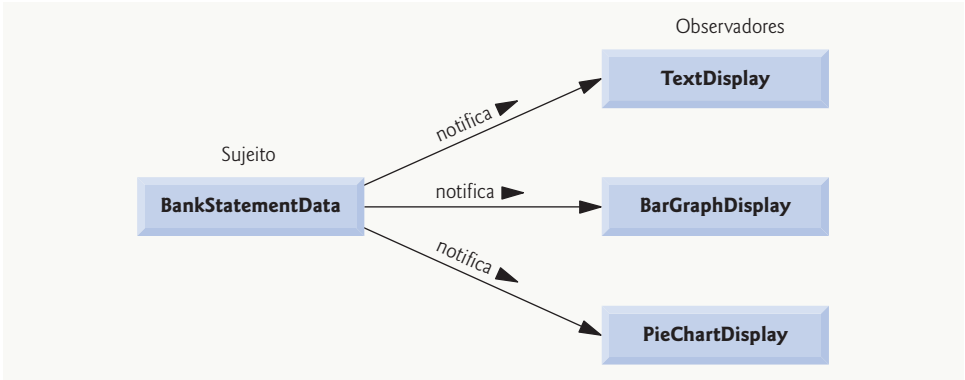

Figura Q.6 | A base do padrão de design `Observer`.

Os componentes Swing GUI utilizam o padrão de design Observer. Os componentes GUI colaboram com seus ouvintes para responder às interações do usuário. Por exemplo, um `ActionListener` observa alterações de estado em um `JButton` (o assunto) registrando-se para tratar eventos desse `JButton`. Quando pressionado pelo usuário, o `JButton` notifica seus objetos `ActionListener` (os observadores) de que o estado do `JButton` mudou (isto é, o `JButton` foi pressionado).

Strategy

O **padrão de design Strategy** é semelhante ao padrão de design State (discutido na Seção Q.2.3). Mencionamos que o padrão de design State contém um objeto estado, que encapsula o estado de um objeto de contexto. O padrão de design Strategy contém um **objeto strategy**, que é análogo ao objeto state do padrão de design State. A principal diferença é que o objeto strategy encapsula um algoritmo em vez de informações sobre o estado.

Por exemplo, os componentes `java.awt.Container` implementam o padrão de design Strategy utilizando `LayoutManagers` (discutidos na Seção 14.18) como objetos strategy. No pacote `java.awt`, as classes `FlowLayout`, `BorderLayout` e `GridLayout` implementam a interface `LayoutManager`. Cada classe utiliza o método `addLayoutComponent` para adicionar componentes GUI a um objeto `Container`. Cada método, porém, utiliza um diferente algoritmo para exibir estes componentes GUI: um `FlowLayout` exibe-os em uma sequência da esquerda para a direita, um `BorderLayout` exibe-os em cinco regiões e um `GridLayout` exibe-os no formato linha/coluna.

A classe `Container` contém uma referência a um objeto `LayoutManager` (o objeto strategy). Uma referência de interface (isto é, a referência ao objeto `LayoutManager`) pode conter referências aos objetos das classes que implementam essa interface (isto é, os objetos `FlowLayout`, `BorderLayout` ou `GridLayout`) de modo que o objeto `LayoutManager` possa referenciar um `FlowLayout`, `BorderLayout` ou `GridLayout` a qualquer momento. A classe `Container` pode alterar essa referência por meio do método `setLayout` para selecionar distintos layouts em tempo de execução.

A classe `FlowLayoutFrame` (Figura 14.39) demonstra o aplicativo do padrão Strategy — a linha 23 declara um novo objeto `FlowLayout` e a linha 25 invoca o método `setLayout` do objeto `Container` para atribuir o objeto `FlowLayout` ao objeto `Container`. Neste exemplo, o `FlowLayout` fornece a estratégia para organizar os componentes.

Template Method

O **padrão de design Template Method** também lida com algoritmos. O padrão de design Strategy permite que vários objetos contenham algoritmos distintos. Entretanto, o padrão de design Template Method requer que todos os objetos compartilhem um único algoritmo definido por uma superclasse.

Por exemplo, considere o design da FiguraQ.6, que apresentamos na discussão sobre o padrão de design Observer anteriormente nesta seção. Os objetos das classes `TextDisplay`, `BarGraphDisplay` e `PieChartDisplay` utilizam o mesmo algoritmo básico para adquirir e exibir os dados — obtêm todas as instruções a partir do objeto `BankStatementData`, analisam sintaticamente e exibem as instruções. O padrão de design Template Method permite criar uma superclasse abstrata chamada `BankStatementDisplay` que fornece o algoritmo comum para exibição dos dados. Nesse exemplo, o algoritmo invoca os métodos abstratos `getData`, `parseData` e `displayData`. As classes `TextDisplay`, `BarGraphDisplay` e `PieChartDisplay` estendem a classe `BankStatementDisplay` para herdar o algoritmo, assim, cada objeto pode utilizar o mesmo algoritmo. Cada subclasse `BankStatementDisplay` então sobrescreve cada método de uma maneira específica a essa subclasse, porque cada classe implementa o algoritmo de maneira diferente. Por exemplo, as classes `TextDisplay`, `BarGraphDisplay` e `PieChartDisplay` poderiam obter e analisar sintaticamente os dados de maneira idêntica, mas cada uma exibe esses dados de maneira diferente.

O padrão de design Template Method permite estender o algoritmo para outras subclasses `BankStatementDisplay` — por exemplo, poderíamos criar classes, como `LineGraphDisplay` ou a classe `3DimensionalDisplay`, que utilizem o mesmo algoritmo herdado da classe `BankStatementDisplay` e fornecer implementações diferentes dos métodos abstratos que o algoritmo chama.

Q.3.4 Conclusão

Nesta seção, discutimos como os componentes Swing tiram proveito dos padrões de design e como os desenvolvedores podem integrar os padrões de design a aplicativos GUI em Java. Na seção a seguir, discutiremos os padrões de design de concorrência, que são particularmente úteis para desenvolver sistemas com múltiplas threads.

Q.4 Padrões de design de concorrência

Muitos padrões de design adicionais foram descobertos desde a publicação do livro "*Gang of Four*", o qual introduziu padrões que envolvem sistemas orientados a objetos. Alguns desses novos padrões envolvem tipos específicos de sistemas orientados a objetos, como sistemas concorrentes, distribuídos ou paralelos. Nesta seção, discutimos os padrões de concorrência para complementar nossa discussão sobre a programação de múltiplas threads do Capítulo 26.

Padrões de design de concorrência

Linguagens de programação de múltiplas threads como o Java permitem que os projetistas especifiquem atividades concorrentes — isto é, aquelas que operam em paralelo umas com as outras. Projetar sistemas concorrentes impropriamente pode introduzir problemas de concorrência. Por exemplo, dois objetos que tentam, ao mesmo tempo, alterar dados compartilhados poderiam corromper esses dados. Além disso, se dois objetos esperam que um ou outro termine as tarefas e se nenhum puder completar sua tarefa, esses objetos potencialmente esperariam eternamente — uma situação denominada **impasse**. Utilizando o Java, Doug Lea[2] e Mark Grand[3] documentaram os **padrões de concorrência** para arquiteturas de design com múltiplas threads a fim de evitar vários problemas associados com o multithreading. Fornecemos uma lista parcial desses padrões de design:

- O **padrão de design para execução de uma única thread** (Grand, 2002) impede que várias threads executem o mesmo método de outro objeto concorrentemente. O Capítulo 26 discute várias técnicas que podem ser utilizadas para aplicar esse padrão.
- O **Padrão de design Guarded Suspension** (Lea, 2000) suspende a atividade de uma thread e retoma a atividade dessa thread quando alguma condição for satisfeita.
- O **padrão de design Balking** (Lea, 2000) assegura que um método irá **emperrar** — isto é, retornar sem realizar nenhuma ação — se um objeto ocupar um estado que não possa executar esse método. Uma variação desse padrão é que o método lança uma exceção que descreve por que esse método não é capaz de executar — por exemplo, um método que lança uma exceção quando ao acessar uma estrutura de dados que não existe.
- O **padrão de design Read/Write Lock** (Lea, 2000) permite que múltiplas threads obtenham acesso de leitura concorrente em um objeto, mas impede que múltiplas threads obtenham acesso de gravação concorrente nesse objeto. Somente uma thread por vez pode obter acesso de gravação a um objeto — quando essa thread obtém acesso de gravação, o objeto permanece **bloqueado** para todas as outras threads.
- O **padrão de design Two-Phase Termination** (Grand, 2002) utiliza um processo de término de duas fases para uma thread a fim de assegurar que uma thread tenha a oportunidade de liberar recursos — como outras threads geradas — na memória (primeira fase) antes do término (segunda fase). No Java, um objeto `Runnable` pode utilizar esse padrão no método `run`. Por exemplo, o método `run` pode conter um loop infinito que é terminado por alguma alteração de estado — no término, o método `run` pode invocar um método `private` responsável por parar quaisquer outras threads geradas (primeira fase). O thread então termina depois de o método `run` terminar (segunda fase).

Na próxima seção, retornaremos aos padrões de design do "*Gang of Four*". Utilizando o material apresentado nos Capítulos 17 e 27, identificamos as classes do pacote `java.io` e `java.net` que utilizam padrões de design.

Q.5 Padrões de design utilizados nos pacotes `java.io` e `java.net`

Esta seção introduz esses padrões de design associados com pacotes de arquivos, fluxos e redes do Java.

Q.5.1 Padrões de design criacionais

Agora, continuamos a nossa discussão sobre padrões de design criacionais.

Abstract Factory

Como ocorre com o padrão de design Factory Method, o **padrão de design Abstract Factory** permite que um sistema determine em que subclasse instanciar um objeto em tempo de execução. Frequentemente, essa subclasse não é conhecida durante o desenvolvimento.

---

[2] Lea, D. *Concurrent Programming in Java, Second Edition: Design Principles and Patterns*. Boston: Addison-Wesley, 2000.

[3] Grand, M. *Patterns in Java; A Catalog of Reusable Design Patterns Illustrated with UML, Second Edition, Volume I*. Nova York: John Wiley and Sons, 2002.

O **Abstract Factory,** porém, utiliza um objeto conhecido como **fábrica** que usa uma interface para instanciar objetos. Uma fábrica cria um produto que, nesse caso, é um objeto de uma subclasse determinada em tempo de execução.

A biblioteca de soquetes Java no pacote `java.net` utiliza o padrão de design Abstract Factory. Um soquete descreve uma conexão, ou um fluxo de dados, entre dois processos. A classe `Socket` refere-se a um objeto de uma subclasse `SocketImpl`. A classe `Socket` também contém uma referência `static` a um objeto que implementa a interface `SocketImplFactory`. O construtor `Socket` invoca o método `createSocketImpl` da interface `SocketImplFactory` para criar o objeto `SocketImpl`. O objeto que implementa interface `SocketImplFactory` é a fábrica, e um objeto de uma subclasse `SocketImpl` é o produto dessa fábrica. O sistema não pode especificar a subclasse `SocketImpl` a partir da qual instanciar até o tempo de execução, porque o sistema não conhece o tipo de implementação de `Socket` que é requerida (por exemplo, um soquete configurado para os requisitos de segurança da rede local). O método `createSocketImpl` decide a subclasse `SocketImpl` da qual instanciar o objeto em tempo de execução.

Q.5.2 Padrões de design estruturais

Esta seção conclui nossa discussão sobre os padrões de design estruturais.

Decorator

Vamos reexaminar a classe `CreateSequentialFile` (Figura 17.17). As linhas 20–21 dessa classe permitem a um objeto `FileOutputStream`, que grava bytes em um arquivo, ganhar a funcionalidade de um `ObjectOutputStream`, que fornece métodos para gravar objetos inteiros em um `OutputStream`. A classe `CreateSequentialFile` parece "empacotar" um objeto `ObjectOutputStream` em torno de um objeto `FileOutputStream`. O fato de ser possível adicionar dinamicamente o comportamento de um `ObjectOutputStream` a um `FileOutputStream` evita a necessidade de uma classe separada chamada `ObjectFileOutputStream`, que implementaria os comportamentos de ambas as classes.

As linhas 20–21 da classe `CreateSequentialFile` mostram um exemplo do **padrão de design Decorator**, que permite a um objeto ganhar funcionalidade adicional dinamicamente. Utilizando esse padrão, projetistas não têm de criar classes desnecessárias separadas para adicionar responsabilidades a objetos de uma dada classe.

Vamos considerar um exemplo mais complexo para descobrir como o padrão de design Decorator pode simplificar a estrutura de um sistema. Suponha que queiramos aprimorar o desempenho de E/S do exemplo anterior utilizando um `BufferedOutputStream`. Com o padrão de design Decorator, escreveríamos

```
output = new ObjectOutputStream(
   new BufferedOutputStream(
      new FileOutputStream( fileName ) ) );
```

Podemos combinar objetos dessa maneira, porque `ObjectOutputStream`, `BufferedOutputStream` e `FileOutputStream` estendem a superclasse abstrata `OutputStream` e cada construtor de subclasse recebe um objeto `OutputStream` como um parâmetro. Se os objetos de fluxo no pacote `java.io` não utilizassem o padrão Decorator (isto é, não atendessem esses dois requisitos), o pacote `java.io` teria de fornecer as classes `BufferedFileOutputStream`, `ObjectBufferedOutputStream`, `ObjectBufferedFileOutputStream` e `ObjectFileOutputStream`. Pense no número de classes que teríamos de criar se combinássemos mais objetos de fluxo sem aplicar o padrão Decorator.

Facade

Ao dirigir, você sabe que pisar no acelerador aumenta a velocidade do seu carro, mas não sabe como isso ocorre exatamente. Esse princípio é a base do **padrão de design Facade**, que permite a um objeto — chamado **objeto fachada** — fornecer uma interface simples para os comportamentos de um **subsistema** (um agregado de objetos que abrange coletivamente uma responsabilidade importante de sistema). O acelerador, por exemplo, é o objeto fachada para o subsistema de aceleração do carro, a direção é o objeto fachada para o subsistema de direção do carro e o freio é o objeto fachada para o subsistema de desaceleração do carro. Um **objeto cliente** utiliza o objeto fachada para acessar os objetos por trás da fachada. O cliente continua a não saber como os objetos por trás da fachada cumprem com as responsabilidades, a complexidade de subsistema permanece assim oculta do cliente. Ao pressionar o acelerador, você atua como um objeto cliente. O padrão de design Facade reduz a complexidade do sistema, porque um cliente interage apenas com um objeto (a fachada) para acessar os comportamentos do subsistema que a fachada representa. Esse desenvolvedores de aplicações de escudos de padrão de complexidades de subsistema. Os desenvolvedores só precisam conhecer as operações do objeto fachada, em vez das operações mais detalhadas do subsistema inteiro. A implementação por trás da fachada pode ser alterada sem modificações para os clientes.

No pacote `java.net`, um objeto da classe `URL` é um objeto de fachada. Esse objeto contém uma referência a um objeto `InetAddress` que especifica o endereço IP do computador host. O objeto fachada da classe `URL` também faz referência a um objeto da classe `URLStreamHandler`, que abre a conexão URL. O objeto cliente que utiliza o objeto fachada da classe `URL` acessa o objeto de `InetAddress` e o objeto de `URLStreamHandler` por meio do objeto fachada. Entretanto, o objeto cliente não sabe como os objetos por trás do objeto fachada da `URL` cumprem suas responsabilidades.

Q.5.3 Padrões arquitetônicos

Os padrões de design permitem que os desenvolvedores projetem partes específicas dos sistemas, como abstrair instanciações de objetos ou agregar classes a estruturas maiores. Os padrões de design também promovem o acoplamento fraco entre objetos. **Padrões arquitetônicos** promovem o acoplamento fraco entre subsistemas. Esses padrões especificam como os subsistemas interagem um com o outro.[4] Introduzimos os populares padrões arquitetônicos Model-View-Controller and Layers.

MVC

Pense no design de um editor de textos simples. Nesse programa, o usuário insere texto pelo teclado e o formata utilizando o mouse. Nosso programa armazena esse texto e informações de formato em uma série de estruturas de dados e então exibe essas informações na tela para que o usuário leia o que foi inserido.

Esse programa obedece ao **padrão arquitetônico Model-View-Controller (MVC)**, que separa os dados do aplicativo (contidos no **modelo**), de um lado, dos componentes gráficos de apresentação (a **visualização**) e lógica de processamento de entrada (o **controlador**), de outro. A Figura Q.7 mostra os relacionamentos entre componentes no MVC.

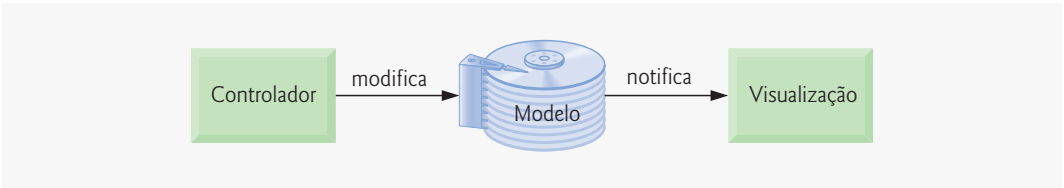

Figura Q.7 | Arquitetura Model-View-Controller.

O controlador implementa a lógica para processar entradas do usuário. O modelo contém dados do aplicativo e a visualização apresenta os dados armazenados no modelo. Quando um usuário fornece alguma entrada, o controlador modifica o modelo com a entrada dada. Com referência ao exemplo do editor de textos, o modelo poderia conter somente os caracteres que compõem o documento. Quando o modelo muda, ele notifica a visualização sobre essa alteração de modo que possa atualizar sua apresentação de acordo com os dados alterados. A visualização em um processador de textos poderia exibir caracteres que utilizam uma fonte particular com um tamanho particular etc.

O MVC não restringe um aplicativo a uma única visualização e a um único controlador. Em um programa mais sofisticado (como um processador de textos), há duas visualizações de um modelo de documentos. Uma visualização poderia exibir uma estrutura de tópicos do documento e a outra poderia exibir o documento completo. O processador de textos também poderia implementar múltiplos controladores — um para tratar entrada pelo teclado e outro para tratar seleções de mouse. Se um dos controladores fizer uma alteração no modelo, tanto a visualização da estrutura de tópicos como a janela de visualização de impressão mostrarão a alteração imediatamente quando o modelo notifica sobre todas as visualizações das alterações.

Outro benefício-chave do padrão arquitetônico MVC é que os desenvolvedores podem modificar cada componente individualmente sem modificar os outros. Por exemplo, poderiam modificar a visualização que exibe a estrutura de tópicos do documento sem modificar o modelo ou outras visualizações ou controladores.

Camadas

Pense no design na Figura Q.8, que apresenta a estrutura básica de um **aplicativo de três camadas (three-tier application)**, no qual cada camada contém um componente único de sistema.

A **camada de informações** (**information tier,** também chamada "camada inferior") mantém os dados do aplicativo, geralmente armazenando esses dados em um banco de dados. A camada de informações para uma loja on-line poderia conter informações sobre produtos, como descrições, preços e quantidades em estoque e informações sobre clientes, como nomes dos usuários, endereços de cobrança e números de cartão de crédito.

A **camada intermediária** (**middle tier**) atua como um intermediário entre a camada de informações e a camada de cliente. A camada intermediária processa as solicitações da camada de cliente e lê e grava os dados no banco de dados.

Ela então processa os dados da camada de informações e apresenta o conteúdo na camada de cliente. Esse processamento é a **lógica do negócio** do aplicativo, a qual trata de tarefas como recuperar dados da camada de informações, assegurando que esses dados são confiáveis antes de atualizar o banco de dados e apresentar os dados na camada de cliente. Por exemplo, a lógica do negócio associada com a camada intermediária para a loja on-line pode verificar o cartão de crédito de um cliente com o emissor do cartão de crédito antes da loja enviar o pedido do cliente. Essa lógica do negócio poderia então armazenar (ou recuperar) as informações sobre o crédito no banco de dados e notificar a camada de cliente de que a verificação foi bem-sucedida.

[4] R. Hartman. "Building on Patterns." *Application Development Trends,* maio de 2001: 19–26.

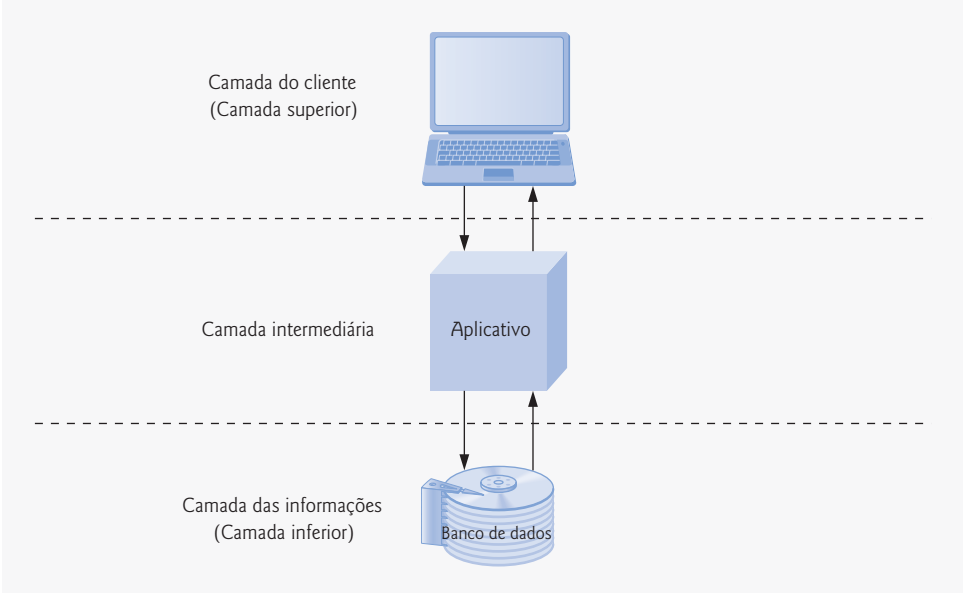

Figura Q.8 | Modelo de aplicativo de três camadas.

A **camada de cliente** (**client tier,** também chamada "camada superior") é a interface com o usuário do aplicativo, como um navegador da Web padrão. Os usuários interagem diretamente com o aplicativo por meio da interface com o usuário. A camada de cliente interage com a camada intermediária para fazer solicitações e recuperar dados da camada de informações. A camada de cliente exibe então os dados recuperados na camada intermediária.

A Figura Q.8 é uma implementação do **padrão arquitetônico Layers**, que divide as funcionalidades em **camadas (layers)** separadas. Cada camada contém um conjunto de responsabilidades de sistema e só depende dos serviços da próxima camada inferior. Na Figura Q.8, cada tier corresponde a um layer. Esse padrão arquitetônico é útil, porque os projetistas podem modificar uma camada sem modificar as outras. Por exemplo, um projetista poderia modificar a camada de informações na Figura Q.8 a fim de armazenar um produto particular no banco de dados sem modificar a camada de cliente ou a camada intermediária.

Q.5.4 Conclusão

Nesta seção, discutimos como os pacotes `java.io` e `java.net` tiram proveito dos padrões de design específicos e como os desenvolvedores podem integrar padrões de design com aplicativos de processamento de redes e arquivos em Java. Também apresentamos os padrões arquitetônicos MVC e Layers, que atribuem funcionalidades de sistema a subsistemas separados. Esses padrões tornam o design de um sistema mais fácil aos desenvolvedores. Na próxima seção, concluiremos nossa apresentação dos padrões de design discutindo os padrões utilizados no pacote `java.util`.

Q.6 Padrões de design utilizados no pacote `java.util`

Nesta seção, utilizamos o material sobre estruturas de dados e coleções discutido nos Capítulos 22–20 para identificar classes do pacote `java.util` que utilizam padrões de design.

Q.6.1 Padrões de design criacionais

Concluímos nossa discussão dos padrões de design criacionais apresentando o padrão de design Prototype.

Prototype

Às vezes, um sistema deve fazer uma cópia de um objeto sem "conhecer" a classe desse objeto até o tempo de execução. Por exemplo, pense no design do programa de desenho do Exercício 10.1 no estudo de caso opcional de GUIs e imagens gráficas — as classes `MyLine`, `MyOval` e `MyRectangle` representam as classes "forma" que estendem a superclasse abstrata `MyShape`. Poderíamos modificar esse exercício para permitir que o usuário crie, copie e cole novas instâncias da classe `MyLine` ao programa. O **padrão de design Prototype** permite que um objeto — chamado **protótipo** — retorne uma cópia desse protótipo a um objeto solicitante — chamado um **cliente**. Cada protótipo deve pertencer a uma classe que implementa uma interface comum que permite ao protótipo clonar a si próprio. Por exemplo, a API do Java fornece o método `clone` a partir da classe `java.lang.Object` e interface `java.lang.Cloneable` — qualquer objeto a partir de uma classe que implementa `Cloneable` pode utilizar o método `clone` para copiar a si próprio. Especificamente, o método `clone` cria uma cópia de um objeto e então retorna uma referência a esse objeto. Se projetarmos a classe `MyLine` como o protótipo para o Exercício 10.1, a classe `MyLine` deve então implementar a interface `Cloneable`. Para criar uma nova linha no nosso desenho, clonamos o protótipo da

`MyLine`. Para copiar uma linha preexistente, clonamos esse objeto. O método `clone` também é útil nos métodos que retornam uma referência a um objeto, mas o desenvolvedor não quer que o objeto seja alterado por essa referência — o método `clone` retornar uma referência à cópia do objeto em vez de retornar a referência a esse objeto. Para informações adicionais sobre a interface `Cloneable`, visite

```
java.sun.com/javase/6/docs/api/java/lang/Cloneable.html
```

Q.6.2 Padrões de design comportamentais

Concluímos nossa discussão sobre os padrões de design comportamentais discutindo o padrão de design Iterator.

Iterator

Os projetistas utilizam estruturas de dados, como arrays, listas vinculadas e tabelas de hash para organizar os dados em um programa. O **padrão de design Iterator** permite que objetos acessem objetos individuais em qualquer estrutura de dados sem "conhecer" o comportamento da estrutura de dados (como percorrer a estrutura ou remover um elemento dessa estrutura) ou como essa estrutura de dados armazena objetos. As instruções para percorrer a estrutura de dados e acessar seus elementos são armazenadas em um objeto separado chamado **iterador**. Cada estrutura de dados pode criar um iterador — cada iterador implementa os métodos de uma interface comum para percorrer a estrutura de dados e acessar seus dados. Um cliente pode percorrer duas estruturas de dados diferentemente estruturadas — como uma lista vinculada e uma tabela de hash — de uma mesma maneira, pois as duas estruturas de dados fornecem um objeto de iterador que pertence a uma classe que implementa uma interface comum. O Java fornece a interface `Iterator` do pacote `java.util`, que utilizamos na Figura 20.2.

Q.7 Conclusão

Neste apêndice, apresentamos a importância, utilidade e predomínio dos padrões de design. Em seu livro *Design Patterns, Elements of Reusable Object-Oriented Software*, o *Gang of Four* descreveu 23 padrões de design que fornecem estratégias testadas para construir sistemas. Cada padrão pertence a uma entre três categorias de um padrão — padrões criacionais abordam questões relacionadas à criação de objetos, padrões estruturais fornecem maneiras de organizar classes e objetos em um sistema e padrões comportamentais oferecem estratégias para modelar a maneira como os objetos colaboram uns com os outros em um sistema.

Dos 23 padrões de design, discutimos 18 dos mais populares utilizados pela comunidade Java. A discussão foi dividida de acordo com a maneira como certos pacotes da Java API — pacotes `java.awt`, `javax.swing`, `java.io`, `java.net` e `java.util` — utilizam esses padrões de design. Também discutimos os padrões não descritos pelo *Gang of Four*, como os padrões de concorrência, que são úteis em sistemas de múltiplas threads, e os padrões arquitetônicos, que ajudam os projetistas a atribuir funcionalidades a vários subsistemas em um sistema. Estimulamos o uso de cada padrão — explicamos por que é importante e explicamos como ele pode ser utilizado. Quando apropriado, fornecemos vários exemplos de analogias práticas (por exemplo, o adaptador no padrão de design Adapter é semelhante a um adaptador de uma tomada em um dispositivo elétrico). Você também aprendeu como os pacotes da Java API tiram proveito dos padrões de design (por exemplo, componentes Swing GUI utilizam o padrão de design Observer para colaborar com seus ouvintes a fim de responder às interações dos usuários). Fornecemos exemplos de como certos programas neste livro utilizaram os padrões de design.

Esperamos que examine este apêndice como um início para um estudo mais aprofundado dos padrões de design. Os padrões de design são utilizados mais predominantemente pela comunidade J2EE (Java 2 Platform, Enterprise Edition), em que os sistemas tendem a ser excessivamente grandes e complexos e nos quais a robustez, portabilidade e desempenho são bastante críticos. Entretanto, mesmo os programadores iniciantes podem se beneficiar da exposição inicial aos padrões de designs. Recomendamos visitar nosso Java Design Patterns Resource Center (`www.deitel.com/JavaDesignPatterns/`) e ler o livro do *Gang of Four*. Essas informações lhe ajudarão a construir sistemas melhores utilizando a sabedoria coletiva da tecnologia de objetos da indústria.