M

Padrões de projeto

M.1 Introdução

A maioria dos exemplos fornecidos neste livro é relativamente pequena. Não requerem um extenso processo de projeto, 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 sistemas de controle de tráfego aéreo ou sistemas que controlam milhares de caixas automáticos de um banco importante, poderiam conter milhões de linhas de código. Um projeto 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 projeto** — arquiteturas testadas para construir softwares orientados a objetos flexíveis e sustentáveis. Utilizar padrões de projeto 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 projeto. Os padrões de projeto beneficiam os desenvolvedores de um sistema

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

O conceito de utilização de padrões de projeto para construir sistemas de softwares originados no campo da arquitetura. Arquitetos utilizam uma série de elementos de projetos arquitetônicos estabelecidos, como arcos e colunas, ao projetarem edifícios. Projetar com arcos e colunas é uma estratégia testada para construir edifícios perfeitos — esses elementos podem ser vistos como padrões de projeto arquitetônicos.

Nos softwares, os padrões de projeto não são classes nem objetos. Em vez disso, os projetistas utilizam padrões de projeto para construir conjuntos de classes e objetos. Para utilizar padrões de projeto 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 projeto orientados a objetos e sua importância na construção de softwares bem elaborados.

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

O histórico dos padrões de projeto orientados 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*. Esse livro descreve 23 padrões de projeto, cada um fornecendo uma solução a um problema comum de projeto de software na indústria. O livro agrupa os padrões de projeto em três categorias — **padrões de projeto criacionais**, **padrões de projeto estruturais** e **padrões de projeto comportamentais**. Padrões de projeto criacionais descrevem as técnicas para instanciar objetos (ou grupos de objetos). Padrões de projeto estruturais permitem que os projetistas organizem classes e objetos em estruturas maiores. Padrões de projeto comportamentais atribuem responsabilidades a classes e objetos.

O livro da 'Gang of Four' mostrou que os padrões de projeto evoluíram naturalmente ao longo dos anos da experiência na 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 projeto 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 projeto são criados o tempo todo e apresentados rapidamente aos projetistas do mundo todo via Internet.

Os padrões de projeto são um tópico mais avançado que talvez não apareça nas seqüências introdutórias da maioria dos cursos. À medida que você avança nos seus estudos sobre o Java, é certo que os padrões de projeto terão um valor maior. Se você é um aluno e seu instrutor não planeja incluir esse material em seu curso, encorajamos a leitura desse material por conta própria.

A Seção M.8 apresenta uma lista dos recursos da Web que dizem respeito aos padrões de projeto e sua relevância para a programação Java. À medida que avança por este apêndice, é recomendável consultar os URLs fornecidos para aprender mais sobre um padrão de projeto particular introduzido no texto ou para ler sobre novos desenvolvimentos da comunidade dos padrões de projeto.

M.2 Introduzindo padrões de projeto criacionais, estruturais e comportamentais

Na Seção M.1, mencionamos que a 'Gang of Four' descreveu 23 padrões de projeto utilizando três categorias — criacional, estrutural e comportamental. Nesta, e nas outras seções deste apêndice, discutimos os padrões de projeto 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 nos capítulos 11 e 22 utilizam o padrão de projeto Composite. A Figura M.1 identifica os 18 padrões de projeto da Gang of Four discutidos neste apêndice.

Muitos padrões populares foram documentados com base no livro da Gang of Four — estes incluem os **padrões de projeto de concorrência**, especialmente úteis no projeto de sistemas de múltiplas threads. A Seção M.4 discute algum desses padrões utilizados na indústria. Padrões arquitetônicos, como discutido na Seção M.5, especificam como subsistemas interagem um com o outro. A Figura M.2 lista os padrões de concorrência e os padrões arquitetônicos que abordamos neste apêndice.

M.2.1 Padrões de projeto criacionais

Padrões de projeto criacionais examinam questões relacionadas à criação de objetos, por exemplo, impedir que um sistema crie mais de um objeto de uma classe (o padrão de projeto criacional Singleton) ou postergar, até o tempo de execução, a decisão sobre quais tipos de objetos serão criados (o propósito dos outros padrões de projeto criacionais discutidos aqui). Por exemplo, suponha que estejamos 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 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 de 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.

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

Figura M.1 18 padrões de projeto da Gang of Four discutidos neste apêndice.

Seção	Padrões de projeto de concorrência	Padrões arquitetônicos
Seção M.4	Single-Threaded Execution, Guarded Suspension, Balking, Read/Write Lock, Two-Phase Termination	
Seção M.5		Model-View-Controller, Layers

Figura M.2 Padrões de projeto de concorrência e arquitetônicos discutidos neste apêndice.

O livro da Gang of Four descreve cinco padrões criacionais (discutiremos quatro neste apêndice):

- Abstract Factory (Seção M.5)
- Builder (não discutido)

1 Vlissides, J. *Pattern hatching:* design patterns applied. Reading, MA: Addison-Wesley, 1998.

- Factory Method (Seção M.3)
- Prototype (Seção M.6)
- Singleton (Seção M.2)

Singleton

Ocasionalmente, um sistema deve conter um único objeto de uma classe — depois que o programa instancia esse objeto, ele 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 projeto Singleton** garante que um sistema instancie no máximo um objeto de uma classe.

A Figura M.3 demonstra o código Java utilizando o padrão de projeto 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` com esse construtor. A linha 7 declara uma referência estática a um objeto `Singleton` e invoca o construtor privado. Isso cria a instância da classe Singleton que será fornecida aos clientes. Quando invocado, o método estático `getSingletonInstance` (linhas 16—19) simplesmente retorna uma cópia dessa referência.

```
1  // Singleton.Java
2  // Demonstra o padrão de projeto 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 M.3 A classe `Singleton` assegura que somente um objeto de sua classe seja criado.

As linhas 9 e 10 da classe `SingletonTest` (Figura M.4) declaram duas referências a objetos `Singleton` — `firstSingleton` e `secondSingleton`. As linhas 13 e 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 M.4 mostra que `firstSingleton` e `secondSingleton` são de fato referências ao mesmo objeto `Singleton`, porque toda vez que o método `getSingletonInstance` é chamado, ele retorna uma referência ao mesmo objeto `Singleton`.

M.2.2 Padrões de projeto estruturais

Padrões de projeto estruturais descrevem maneiras comuns de organizar classes e objetos em um sistema. O livro da Gang of Four descreve sete padrões de projeto estruturais (discutiremos seis neste apêndice) :

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

```
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 M.4 A classe `SingletonTest` cria o objeto Singleton mais de uma vez.

Proxy

Um applet sempre deve exibir algo enquanto imagens são carregadas a fim de fornecer um feedback positivo aos usuários para que 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 projeto 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 projeto Proxy permite que o sistema utilize um objeto — chamado **objeto proxy** — no lugar de um 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.

M.2.3 Padrões de projeto comportamentais

Os **padrões de projeto 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 projeto 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 projeto 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 o recurso 'desfaz' que permite aos usuários reverterem para versões prévias dos seus trabalhos.

O livro da Gang of Four descreve 11 padrões de projeto comportamentais (discutiremos oito neste apêndice):

- Chain of Responsibility (Seção M.3)
- Command (Seção M.3)
- Interpreter (não discutido)
- Iterator (Seção M.2)
- Mediator (não discutido)
- Memento (Seção M.2)

- Observer (Seção M.3)
- State (Seção M.2)
- Strategy (Seção M.3)
- Template Method (Seção M.3)
- Visitor (não discutido)

Memento
Considere um programa de pintura, que permita a um usuário criar imagens gráficas. Ocasionalmente, o usuário talvez posicione uma imagem gráfica de maneira inapropriada na área de desenho. Programas de pintura oferecem o recurso '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 projeto 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 projeto Memento exige três tipos de objeto. 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 (o memento salva o estado da área de desenho). O memento é armazenado como o primeiro item na lista de 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 um outro memento para armazenar essas informações. Esse memento torna-se 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 deseje 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.

State
Em alguns projetos, devemos comunicar as informações sobre o estado de um objeto ou representar os vários estados que um objeto pode ocupar. O **padrão de projeto 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 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.

M.2.4 Conclusão

Nesta seção, listamos os três tipos de padrões de projeto introduzidos no livro da Gang of Four, identificamos 18 desses padrões de projeto abordados neste apêndice e discutimos padrões de projeto específicos, incluindo o Singleton, Proxy, Memento e State. Na próxima seção, introduziremos alguns padrões de projeto associados com o AWT e componentes Swing GUI. Depois de ler a próxima seção, você entenderá melhor como os componentes GUI Java tiram proveito dos padrões de projeto.

M.3 Padrões de projeto utilizados nos pacotes `java.awt` e `javax.swing`

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

M.3.1 Padrões de projeto criacionais

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

Factory Method
Suponha que estejamos projetando um sistema que abra uma imagem 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 por meio 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 projeto 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 permita 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/j2se/5.0/docs/api/java/awt/Component.html
```

M.3.2 Padrões de projeto estruturais

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

Adapter

O **padrão de projeto Adapter** fornece a um objeto uma nova interface que se *adapta* à interface de um outro objeto, permitindo que os dois objetos colaborem entre si. Poderíamos equiparar o padrão adpater 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 um soquete europeu e vice-versa.

O Java fornece várias classes que utilizam o padrão de projeto 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 11.14, adapta um objeto que gera `MouseEvents` para um objeto que trata `MouseEvents`.

Bridge

Suponha que estejamos 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. Projetamos as 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` (um `Button` com uma `Image` sobreposta) que estende a classe `Button`, precisaremos criar subclasses `Win32ImageButton` e `MacImageButton` adicionais. De fato, precisaremos criar subclasses `Button` para cada sistema operacional que desejarmos suportar, o que aumenta o tempo de desenvolvimento. Segundo, quando um novo sistema operacional aparecer no mercado, precisaremos criar subclasses `Button` adicionais específica para esse novo sistema.

O **padrão de projeto 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 projeto 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 projeto 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 projeto permite que projetistas criem novos componentes GUI para diversas plataformas utilizando uma 'ponte' para ocultar detalhes específicos à plataforma.

Dica de portabilidade M.1

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

Composite

Projetistas freqüentemente 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 conter, é chamado de **ramificação** (por exemplo, um diretório contendo arquivos); caso contrário, é chamado de **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 de **cliente** — que quer percorrer a estrutura para determinar a classe em particular para cada nó. Fazer essa determinação pode ser demorado, e a estrutura pode tornar-se difícil de manter.

No **padrão de projeto 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 projeto 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 M.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` pode ser adicionado a qualquer objeto de uma subclasse `Component` e qualquer objeto de uma subclasse `Component` pode ser adicionado 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.

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 adicionados 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`.

M.3.3 Padrões de projeto comportamentais

Esta seção continua a discussão sobre os padrões de projeto comportamentais. Discutiremos os padrões de projeto 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. Freqüentemente, um sistema precisa determinar em tempo de execução o objeto que tratará uma mensagem particular. Por exemplo, considere o projeto 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 tratará a chamada e se a segunda estiver ocupada, a terceira tratará a chamada. Se todas as linhas no sistema estiverem ocupadas, uma secretária eletrônica irá instruir o chamador a esperar a próxima linha disponível. Quando uma linha estiver disponível, essa tratará a chamada.

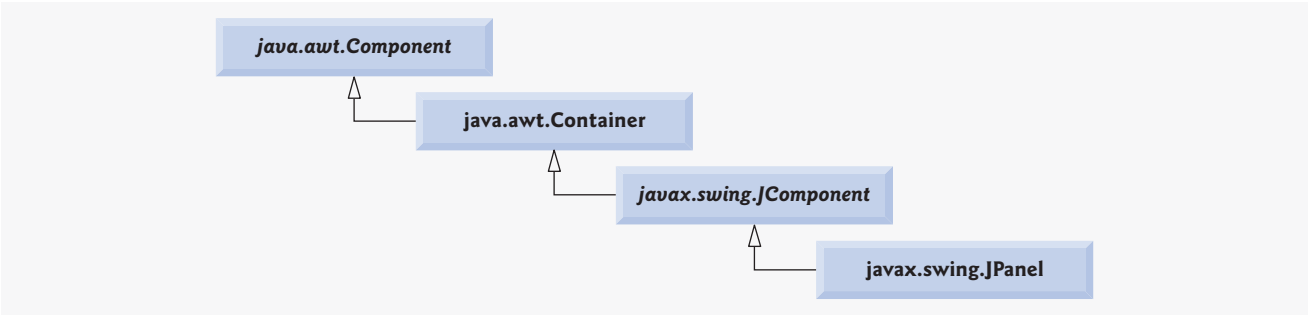

Figura M.5 Hierarquia de herança da classe `JPanel`.

O **padrão de projeto 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 projeto 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` (que 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).

Command

Freqüentemente, 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 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 projeto 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 projeto é chamado de 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 M.5.3.] A Figura M.6 mostra o projeto 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.

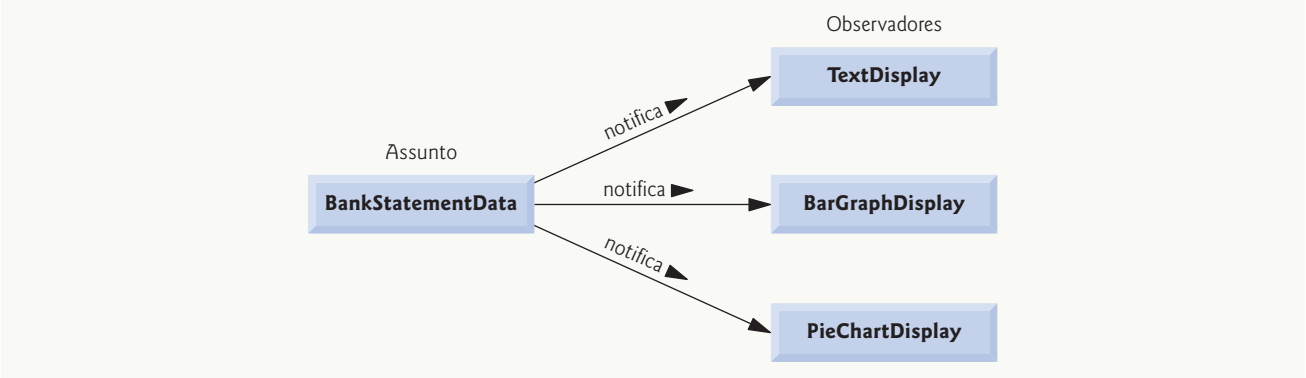

Figura M.6 A base do padrão de projeto `Observer`.

Observação de engenharia de software M.1

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

O **padrão de projeto Observer** é apropriado para sistemas como o da Figura M.6. Ele promove o acoplamento fraco entre um **objeto-assunto** e **objetos observadores** — um objeto-assunto notifica os objetos observadores quando o assunto altera o 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 projeto 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.

Os componentes Swing GUI utilizam o padrão de projeto Observer. Os componentes GUI colaboram com seus ouvintes para responder as 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 (que o `JButton` foi pressionado).

Strategy

O **padrão de projeto Strategy** é semelhante ao padrão de projeto State (discutido na Seção M.2.3). Mencionamos que o padrão de projeto State contém um objeto estado, que encapsula o estado de um objeto de contexto. O padrão de projeto Strategy contém um **objeto strategy**, que é análogo ao objeto state do padrão de projeto 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 projeto Strategy utilizando `LayoutManagers` (discutidos na Seção 11.17) 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 seqüência 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 (a referência ao objeto `LayoutManager`) pode conter referências aos objetos das classes que implementam essa interface (os objetos `FlowLayout`, `BorderLayout` ou `GridLayout`) de modo que o objeto `LayoutManager` possa referir-se a um `FlowLayout`, `BorderLayout` ou `GridLayout` a qualquer momento. A classe `Container` pode alterar essa referência por meio do método `setLayout` para selecionar diferentes layouts em tempo de execução.

A classe `FlowLayoutFrame` (Figura 11.39) 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`. Nesse exemplo, o `FlowLayout` fornece a estratégia para organizar os componentes.

Template Method

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

Por exemplo, considere o projeto da Figura M.6, que apresentamos na discussão sobre o padrão de projeto 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 por meio do objeto `BankStatementData`, analisam sintaticamente e exibem as instruções. O padrão de projeto Template Method permite criar uma superclasse abstrata chamada de `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 projeto Template Method permite estender o algoritmo para outras subclasses `BankStatementDisplay` — por exemplo, poderíamos criar classes, como `LineGraphDisplay` ou a classe `3DimensionalDisplay`, que utilizassem o mesmo algoritmo herdado da classe `BankStatementDisplay` e fornecessem implementações diferentes dos métodos abstratos que o algoritmo chama.

M.3.4 Conclusão

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

M.4 Padrões de projeto de concorrência

Muitos padrões de projeto adicionais foram descobertos desde a publicação do livro da 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, abordaremos os padrões de concorrência para complementar nossa discussão sobre a programação de múltiplas threads do Capítulo 23.

Padrões de projeto de concorrência

Linguagens de programação de múltiplas threads como o Java permitem que projetistas especifiquem atividades concorrentes — aquelas que operam em paralelo umas com as outras. Projetar sistemas concorrentes de maneira inapropriada 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 esperarem que um ou outro termine as tarefas e se nenhum puder completar sua tarefa, eles potencialmente poderão esperar eternamente — uma situação denominada de **impasse**. Utilizando o Java, Doug Lea[2] e Mark Grand[3] documentaram os **padrões de concorrência** para arquiteturas de projeto com múltiplas threads a fim de evitar vários problemas associados com o multithreading. Fornecemos a seguir uma lista parcial desses padrões de projeto:

- O **padrão de projeto para execução de uma única thread** (Grand, 2002) impede que várias threads executem o mesmo método de outro objeto concorrentemente. O Capítulo 23 discute várias técnicas que podem ser utilizadas para aplicar este padrão.
- O **Padrão de projeto Guarded Suspension** (Lea, 2000) suspende a atividade de uma thread e retoma a atividade dessa thread quando alguma condição for satisfeita. As linhas 87—90 e linhas 41—44 da classe `RunnableObject` (Figura 23.17) utilizam

2 Lea, D. *Concurrent programming in Java*. 2. ed. *Design principles and patterns*. Boston: Addison-Wesley, 2000.

3 Grand, M. *Patterns in Java*; a catalog of reusable design patterns illustrated with UML. 2. ed. v. I. Nova York: John Wiley and Sons, 2002.

esse padrão de projeto — os métodos `await` e `signal` suspendem e retomam as threads de programa e a linha 72 da classe `RandomCharacters` (Figura 23.18) alterna a variável de guarda que a condição avalia.

- O **padrão de projeto 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 deste 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 ao acessar uma estrutura de dados que não existe.
- O **padrão de projeto 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 projeto Two-Phase Termination** (Grand, 98) utiliza um processo de término de duas fases para uma thread a fim de assegurar que ela 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 encerrado por alguma alteração de estado — no término, o método `run` pode invocar um método `private` responsável por parar quaisquer outras threads geradas (primeira fase). A thread então termina depois de o método `run` ser encerrado (segunda fase).

Na próxima seção, retornaremos aos padrões de projeto da Gang of Four. Utilizando o material apresentado nos capítuloc 14 e 24, identificamos as classes no pacote `java.io` e `java.net` que usam padrões de projeto.

M.5 Padrões de projeto utilizados nos pacotes `java.io` e `java.net`

Esta seção introduz os padrões de projeto associados a pacotes de arquivos, fluxos e redes do Java.

M.5.1 Padrões de projeto criacionais

Agora, prosseguiremos nossa discussão sobre padrões de projeto criacionais.

Abstract Factory

Como ocorre com o padrão de projeto de Factory Method, o **padrão de projeto Abstract Factory** permite que um sistema determine em que subclasse instanciar um objeto em tempo de execução. Com freqüência, essa subclasse não é conhecida durante o desenvolvimento. O **Abstract Factory,** porém, utiliza um objeto conhecido como uma **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 sockets Java no pacote `java.net` utiliza o padrão de projeto Abstract Factory. Um socket descreve uma conexão, ou um fluxo de dados, entre dois processos. A classe `Socket` faz referência a um objeto de uma subclasse `SocketImpl` (Seção 24.5). 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 a 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` por meio da qual instancia até o tempo de execução, pois ele não conhece o tipo de implementação de `Socket` requerido (por exemplo, um socket 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.

M.5.2 Padrões de projeto estruturais

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

Decorator

Vamos reexaminar a classe `CreateSequentialFile` (Figura 14.18). As linhas 20 e 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` aparece para '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 de `ObjectFileOutputStream`, que implementaria os comportamentos de ambas as classes.

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

Vamos considerar um exemplo mais complexo para descobrir como o padrão de projeto 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 projeto 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 (se não atendessem a 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 projeto Facade**, que permite a um objeto — chamado de **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 atrás da fachada. O cliente continua a não saber como os objetos por atrás 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 projeto 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. Esses desenvolvedores de aplicações de escudos de padrão de complexidades de subsistema. Os desenvolvedores só precisam conhecer as operações do objeto fachada, em vez de das operações mais detalhadas do subsistema inteiro. A implementação por atrás da fachada pode ser alterada sem modificações para os clientes.

No pacote java.net, o 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.

M.5.3 Padrões arquitetônicos

Os padrões de projeto permitem que os desenvolvedores projetem partes específicas dos sistemas, como abstrair instanciações de objetos ou agregar classes a estruturas maiores. Os padrões de projeto 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 a seguir os populares padrões arquitetônicos Model-View-Controller e Camadas (Layers).

MVC

Pense no projeto 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-as 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 M.7 mostra os relacionamentos entre componentes no MVC.

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 poderia exibir uma estrutura de tópicos do documento e a outra, 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 notificar sobre todas as visualizações das alterações.

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

4 R. Hartman. Building on patterns. *Application Development Trends,* p. 19—26, maio 2001.

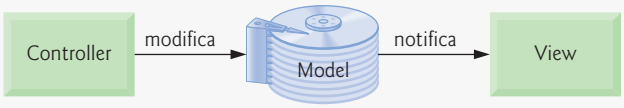

Figura M.7 Arquitetura Model-View-Controller.

Camadas (Layers)

Pense no projeto na Figura M.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 de 'camada inferior', mantém os dados do aplicativo, em geral 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 à 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.

A **camada de cliente** (*client tier*), também chamada de '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.

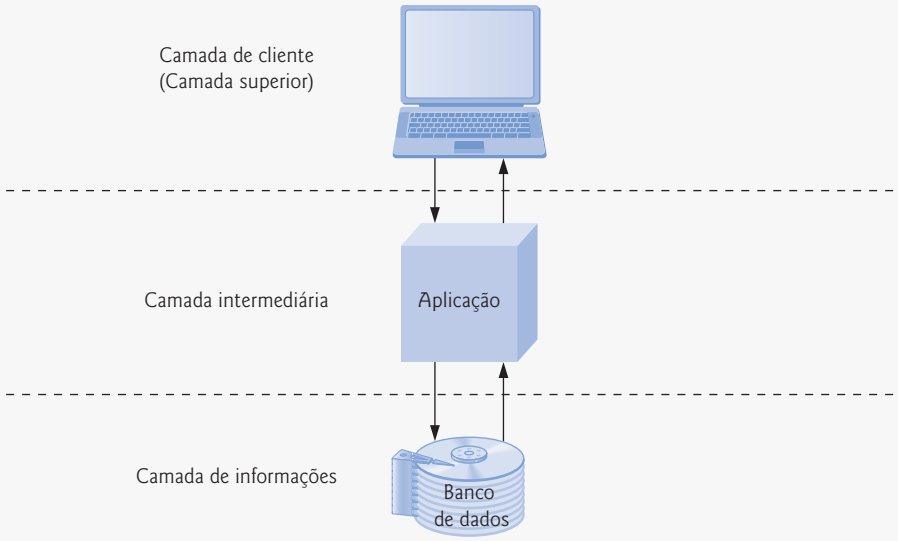

Figura M.8 Modelo de aplicativo de três camadas (*three-tier*).

A Figura M.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 M.8, cada camada corresponde a uma camada. Esse padrão arquitetônico é útil, porque os projetistas podem modificar uma camada sem alterar as outras. Por exemplo, um projetista poderia modificar a camada de informações na Figura M.8 a fim de armazenar um produto particular no banco de dados sem alterar a camada do cliente ou a camada intermediária.

M.5.4 Conclusão

Nesta seção, discutimos como os pacotes `java.io` e `java.net` tiram proveito dos padrões de projeto específicos e como os desenvolvedores podem integrar padrões de projeto com aplicativos de processamento de redes e arquivos em Java. Também apresentamos os padrões

arquitetônicos Model-View-Controller e Camadas (Layers) que atribuem funcionalidades de sistema a subsistemas separados. Esses padrões tornam o projeto de um sistema mais fácil aos desenvolvedores. Na próxima seção, concluiremos nossa apresentação dos padrões de projeto discutindo os padrões utilizados no pacote `java.util`.

M.6 Padrões de projeto utilizados no pacote `java.util`

Nesta seção, utilizamos o material sobre estruturas de dados e coleções discutido nos capítulos 17 a 19 para identificar classes no pacote `java.util` que usam os padrões de projeto.

M.6.1 Padrões de projeto criacionais

Concluiremos nossa discussão sobre os padrões de projeto criacionais apresentando o padrão de projeto 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 projeto do programa de desenho do Exercício 10.1 no estudo de caso opcional de GUIs e imagens gráficas — as classes `MyLine`, `MyOval` e `MyRect` representam as classes 'forma' que estendem a superclasse abstrata `MyShape`. Podemos modificar esse exercício para permitir que o usuário crie, copie e cole novas instâncias da classe `MyLine` ao programa. O **padrão de projeto Prototype** permite que um objeto — chamado de **protótipo** — retorne uma cópia desse protótipo a um objeto solicitante — chamado de **cliente**. Cada protótipo deve pertencer a uma classe que implementa uma interface comum que permite ao protótipo clonar a si próprio. Por exemplo, a Java API fornece o método `clone` da classe `java.lang.Object` e a interface `java.lang.Cloneable` — qualquer objeto de uma classe que implementa `Cloneable` pode utilizar o método `clone` para copiar a si mesmo. 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 criarmos uma linha no nosso desenho, clonamos o protótipo de `MyLine`. Para copiarmos uma linha preexistente, clonamos esse objeto. O método `clone` também é útil para métodos que retornam uma referência a um objeto, mas em situações que o desenvolvedor não queira que o objeto seja alterado por essa referência — o método `clone` retorna 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/j2se/5.0/docs/api/java/lang/Cloneable.html
```

M.6.2 Padrões de projeto comportamentais

Concluiremos nossa discussão sobre os padrões de projeto comportamentais abordando o padrão de projeto Iterator.

Iterator

Projetistas utilizam estruturas de dados, como arrays, listas vinculadas e tabelas de hash para organizar os dados em um programa. O **padrão de projeto 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 de **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 iterador que pertence a uma classe que implementa uma interface comum. O Java fornece a interface `Iterator` no pacote `java.util`, discutida na Seção 19.3 — a classe `CollectionTest` (Figura 19.3) que utiliza um objeto `Iterator`.

M.7 Conclusão

Neste apêndice, apresentamos a importância, utilidade e predomínio dos padrões de projeto. Em seu livro *Design patterns, elements of reusable object-oriented software*, a Gang of Four descreveu 23 padrões de projeto que fornecem estratégias testadas para construir sistemas. Cada padrão pertence a uma entre três categorias — padrões criacionais abordam questões relacionada 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 objetos colaboram uns com os outros em um sistema.

Dos 23 padrões de projeto, 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` — usam esses padrões de projeto. Também discutimos os padrões não descritos pela Gang of Four, como os de concorrência, que são úteis em sistemas de múltiplas threads, e os arquitetônicos, que ajudam os projetistas a atribuir funcionalidades a vários subsistemas em um sistema. Motivamos o uso de cada padrão — explicamos por que é importante e também como pode ser utilizado. Quando apropriado, fornecemos vários exemplos de analogias práticas (por exemplo, o adaptador no padrão de projeto Adapter é semelhante a um adaptador de tomada em um dispositivo elétrico). Você também aprendeu como os pacotes da Java API tiram proveito dos padrões de projeto (por exemplo, componentes Swing GUI usam o padrão de projeto 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 projeto.

Esperamos que examine este apêndice como o início de um estudo mais aprofundado dos padrões de projeto. Esses 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 projetos. Recomendamos que você visite os vários URLs que fornecemos na Seção M.8 e que então leia o livro da Gang of Four. Essas informações o ajudarão a construir sistemas melhores utilizando a sabedoria coletiva da tecnologia de objetos da indústria.

Esperamos que você continue seus estudos dos padrões de projeto. Envie seus comentários, críticas e sugestões para aperfeiçoar ainda mais o *Java Como Programar* a `deitel@deitel.com`. Boa sorte!

M.8 Recursos da Web

Os URLs a seguir fornecem informações adicionais sobre a natureza, importância e aplicações dos padrões de projeto.

Padrões de projeto

`www.hillside.net/patterns`

Exibe links às informações sobre padrões de projetos e linguagens.

`www.hillside.net/patterns/books/`

Lista livros sobre padrões de projeto.

`www.netobjectives.com/design.htm`

Introduz a importância dos padrões de projeto.

`umbc7.umbc.edu/~tarr/dp/dp.html`

Conecta a sites da Web de tutoriais e artigos dobre padrões de projetos.

`www.c2.com/ppr/`

Discute os avanços recentes nos padrões de projeto e idéias para projetos futuros.

`www.dofactory.com/patterns/Patterns.aspx`

Fornece diagramas de classes da UML que ilustram cada um dos 23 padrões de projeto da Gang of Four.

Padrões de projeto no Java

`java.sun.com/blueprints/patterns/index.html`

Página de recursos da Sun Microsystems descrevendo os padrões de projeto para aplicativos Java 2 Platform, Enterprise Edition (J2EE).

`www.javaworld.com/channel_content/jw-patterns-index.shtml`

Contém artigos que discutem quando utilizar e como implementar padrões de projeto populares em Java, demonstrando-os com diagramas de classes da UML.

`www.fluffycat.com/java/patterns.html`

Fornece código Java de exemplo e diagramas de classes da UML para ilustrar cada um dos 23 padrões de projeto da Gang of Four.

`www.cmcrossroads.com/bradapp/javapats.html`

Discute os padrões de projeto Java e padrões de projeto presentes na computação distribuída.

`www.javacamp.org/designPattern/`

Fornece definições e código de exemplo para vários padrões de projeto, descrevendo onde cada padrão deve ser utilizado e seus benefícios.

Padrões arquitetônicos

`www.javaworld.com/javaworld/jw-04-1998/jw-04-howto.html`

Contém um artigo sobre como os componentes Swing utilizam a arquitetura Model-View-Controller.

`www.ootips.org/mvc-pattern.html`

Fornece informações e dicas sobre a utilização do MVC.

`www.tml.hut.fi/Opinnot/Tik-109.450/1998/niska/sld001.htm`

Fornece informações sobre o padrão de projeto arquitetônico e idiomas (padrões que têm por alvo um idioma específico).