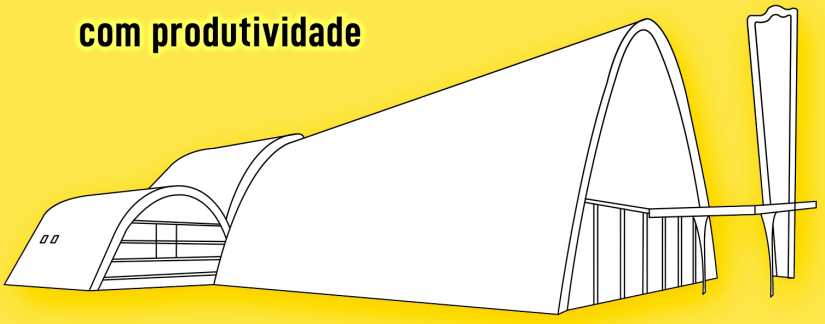
ENGENHARIA *DE* SOFTWARE MODERNA

Princípios e práticas para desenvolvimento de software com produtividade



MARCO TULIO VALENTE

Engenharia de Software Moderna

Princípios e Práticas para Desenvolvimento de Software com Produtividade

Marco Tulio Valente

Versão 2020.1.0 - LeanPub

Para Cynthia, Daniel e Mariana.

Conteúdo

Capítulo 1

Padrões de Projeto

> *A design that doesn't take change into account risks major redesign in the future.* – Gang of Four

Este capítulo inicia com uma introdução ao conceito e aos objetivos de padrões de projeto (Seção 6.1). Em seguida, discutimos com detalhes dez padrões de projetos: Fábrica, Singleton, Proxy, Adaptador, Fachada, Decorador, Strategy, Observador, Template Method e Visitor. Cada um desses padrões é discutido em uma seção separada (Seções 6.1 a 6.11). A apresentação de cada padrão é organizada em três partes: (1) um contexto, isto é, um sistema onde o padrão poderia ser útil; (2) um problema no projeto desse sistema; (3) uma solução para esse problema por meio de padrões. Na Seção 6.12, discutimos brevemente mais alguns padrões. Terminamos o capítulo alertando que padrões de projeto não são uma bala-de-prata, ou seja, discutimos situações onde o uso de padrões não é recomendado (Seção 6.13).

1.1 Introdução

Padrões de projeto são inspirados em uma ideia proposta por Cristopher Alexander, um arquiteto — de construções civis e não de software — e professor da Universidade de Berkeley. Em 1977, Alexander lançou um livro chamado *A Patterns Language*, no qual ele documenta diversos padrões para construção de cidades e prédios. Segundo Alexander:

> Cada padrão descreve um problema que sempre ocorre em nosso contexto e uma solução para ele, de forma que possamos usá-la um milhão de vezes.

Em 1995, Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides lançaram um livro adaptando as ideias de Alexander para o mundo de desenvolvimento de software (link). Em vez de propor um catálogo de soluções para projeto de cidades e prédios, eles propuseram um catálogo com soluções para resolver problemas recorrentes em projeto de software. Eles deram o nome de **Padrões de Projeto** às soluções propostas no livro. Eles definem padrões de projeto da seguinte forma:

> Padrões de projeto descrevem objetos e classes que se relacionam para resolver um problema de projeto genérico em um contexto particular.

Assim, para entender os padrões propostos pela *Gang of Four* — nome pelo qual ficaram conhecidos os autores e também o livro de padrões de projeto — precisamos entender: (1) o problema que o padrão pretende resolver; (2) o contexto em que esse problema ocorre; (3) a solução proposta. Neste livro, vamos descrever alguns padrões de projeto, sempre focando nesses elementos: contexto, problema e solução. Iremos também mostrar vários exemplos de código.

Além de oferecer soluções prontas para problemas de projeto, padrões de projeto transformaram-se em um vocabulário largamente adotado por desenvolvedores de software. Assim, é comum ouvir desenvolvedores dizendo que usaram uma fábrica para resolver um certo problema, enquanto que um segundo problema foi resolvido por meio de decoradores. Ou seja, eles apenas mencionam o nome do padrão e subentendem que a solução adotada já está clara. De forma semelhante, o vocabulário de padrões de projeto é muito usado na documentação de sistemas. Por exemplo, a figura da próxima página mostra a documentação de uma das classes da biblioteca padrão de Java. Podemos ver que o nome da classe termina em `Factory` — que é um dos padrões de projeto que vamos estudar daqui a pouco. Na descrição da classe, volta-se a mencionar que ela é uma fábrica. Portanto, desenvolvedores que conhecem esse padrão de projeto terão mais facilidade para entender e usar a classe em questão.

Um desenvolvedor pode se beneficiar do domínio de padrões de projeto em dois cenários principais:

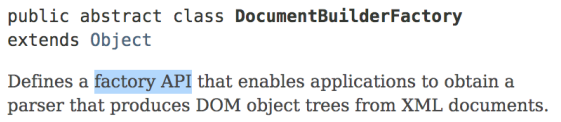


Figura 1.1: Documentação de uma classe `Factory` da API de Java

- Quando ele estiver implementando o seu próprio sistema. Nesse caso, conhecer padrões de projeto pode ajudá-lo a adotar, no seu sistema, uma solução de projeto já testada e validada.
- Quando ele estiver usando um sistema de terceiros, como o pacote de Java que implementa a classe DocumentBuilderFactor da figura. Nesse caso, conhecimento de padrões de projeto pode ajudá-lo a entender o comportamento e a estrutura da classe que ele precisa usar.

É importante entender que padrões de projeto visam a criação de projetos de software flexíveis e extensíveis. Neste livro, antes de explicar cada um dos padrões, vamos apresentar um contexto e um trecho de código que funciona e produz um resultado. Porém, ele não dá origem a um projeto flexível. Para deixar essa inflexibilidade clara, apresentaremos um cenário de extensão do código mostrado, envolvendo a implementação de novos requisitos. Então vamos argumentar que essa extensão exigirá algum esforço, que poderá ser minimizado se usarmos um padrão de projeto.

Os quatro autores do livro de padrões de projeto defendem que devemos projetar um sistema pensando em mudanças que inevitavelmente vão ocorrer — eles chamam essa preocupação de *design for change*. Conforme afirmado por eles na frase que abre este capítulo, se design for change não for uma preocupação, os desenvolvedores correm o risco de em breve ter que planejar um profundo reprojeto de seus sistemas.

No livro sobre padrões de projeto, são propostos 23 padrões, divididos nas seguintes três categorias (os padrões que estudaremos neste capítulo estão em negrito, seguido do número da seção onde eles são apresentados):

- **Criacionais**: padrões que propõem soluções flexíveis para criação de objetos. São eles: **Abstract Factory (6.2)**, Factory Method, **Singleton (6.3)**, **Builder (6.12)** e Prototype.

- **Estruturais**: padrões que propõem soluções flexíveis para composição de classes e objetos. São eles: **Proxy (6.4)**, **Adapter (6.5)**, **Facade (6.6)**, **Decorator (6.7)**, Bridge, Composite e Flyweight.
- **Comportamentais**: padrões que propõem soluções flexíveis para interação e divisão de responsabilidades entre classes e objetos. São eles: **Strategy (6.8)**, **Observer (6.9)**, **Template Method (6.10)**, **Visitor (6.11)**, Chain of Responsibility, Command, Interpreter, **Iterator (6.12)**, Mediator, Memento e State.

Tradução: Pelo fato de a tradução ser direta, vamos traduzir os nomes dos seguintes padrões: Fábrica Abstrata, Método Fábrica, Adaptador, Decorador, Observador e Iterador. Os demais serão referenciados usando o nome original.

1.2 Fábrica

Contexto: Suponha um sistema distribuído baseado em TCP/IP. Nesse sistema, três funções `f`, `g` e `h` criam objetos do tipo `TCPChannel` para comunicação remota, como mostra o próximo código.

```
void f() {
  TCPChannel c = new TCPChannel();
  ...
}

void g() {
  TCPChannel c = new TCPChannel();
  ...
}

void h() {
  TCPChannel c = new TCPChannel();
  ...
}
```

Problema: Suponha que — em determinadas configurações do sistema — precisaremos usar UDP para comunicação. Portanto, se considerarmos esse requisito, o sistema não atende ao Princípio Aberto/Fechado, isto é, ele não está fechado para modificações e aberto para extensões nos protocolos de comunicação usados. Sendo mais claro, gostaríamos de "parametrizar" o código acima para criar objetos dos tipos `TCPChannel` ou `UDPChannel`, dependendo dos clientes. O problema é que o operador `new` deve ser seguido do nome literal de uma classe. Esse operador — pelo menos em linguagens como Java, C++ e C# — não permite que a classe dos

objetos que se pretende criar seja passada como um parâmetro. Resumindo, o problema consiste em parametrizar a instanciação dos canais de comunicação no código acima, de forma que ele consiga trabalhar com protocolos diferentes.

Solução: A solução que vamos descrever baseia-se no padrão de projeto **Fábrica**. Esse padrão possui algumas variações, mas no nosso problema vamos adotar um método estático que: (1) apenas cria e retorna objetos de uma determinada classe; (2) e também oculta o tipo desses objetos por trás de uma interface. Um exemplo é mostrado a seguir:

```
class ChannelFactory {
   public static Channel create() { // método fábrica estático
     return new TCPChannel();
   }
}

void f() {
  Channel c = ChannelFactory.create();
  ...
}

void g() {
  Channel c = ChannelFactory.create();
  ...
}

void h() {
  Channel c = ChannelFactory.create();
  ...
}
```

Nessa nova versão, as funções `f`, `g` e `h` não tem consciência do tipo de `Channel` que vão criar e usar. Elas chamam um **Método Fábrica Estático**, que instancia e retorna um objeto de uma classe concreta — para ser claro, essa variante do padrão Fábrica não foi proposta no livro da Gangue dos Quatro, mas sim alguns anos depois por Joshua Bloch (link). É importante também destacar que as três funções usam sempre uma interface `Channel` para manipular os objetos criados pelo método fábrica estático. Ou seja, aplicamos o princípio "Prefira Interfaces a Classes" (ou Inversão de Dependências).

No novo código, o sistema continua funcionando com canais do tipo `TCPChannel`. Porém, se quisermos mudar o tipo de canal, temos agora que modificar um único elemento do código: o método create da classe `ChannelFactory`. Dizendo de outra forma, um método fábrica estático funciona como um "aspirador" de métodos `new`:

todas as chamadas antigas de `new` migram para uma única chamada, no método fábrica estático.

Existem ainda algumas variações do padrão Fábrica. Em uma delas, uma classe abstrata é usada para concentrar vários métodos fábrica. Essa classe recebe então o nome de **Fábrica Abstrata**. Um exemplo é mostrado a seguir:

```
abstract class ProtocolFactory { // Fábrica Abstrata
  abstract Channel createChannel();
  abstract Port createPort();
  ...
}

void f(ProtocolFactory pf) {
  Channel c = pf.createChannel();
  Port p = pf.createPort();
  ...
}
```

No exemplo acima, omitimos as classes que estendem a classe abstrata `ProtocolFactory` e que vão implementar, de fato, os métodos concretos para criação de canais e portas de comunicação. Podemos, por exemplo, ter duas subclasses: `TCPProtocolFactory` e `UDPProtocolFactory`.

1.3 Singleton

Contexto: Suponha uma classe `Logger`, usada para registrar as operações realizadas em um sistema. Um uso dessa classe é mostrado a seguir:

```
void f() {
  Logger log = new Logger();
  log.println("Executando f");
  ...
}
void g() {
  Logger log = new Logger();
  log.println("Executando g");
  ...
}
void h() {
  Logger log = new Logger();
  log.println("Executando h");
  ...
}
```

Problema: No código anterior, cada método que precisa registrar eventos cria sua própria instância de `Logger`. No entanto, gostaríamos que todos os usos de `Logger` tivessem como alvo a mesma instância da classe. Em outras palavras, não queremos uma proliferação de objetos `Logger`. Em vez disso, gostaríamos que existisse, no máximo, uma única instância dessa classe e que ela fosse usada em todas as partes do sistema que precisam registrar algum evento. Isso é importante, por exemplo, caso o registro de eventos seja feito em arquivos. Se for possível a criação de vários objetos `Logger`, todo novo objeto instanciado vai apagar o arquivo anterior, criado por outros objetos `Logger`.

Solução: A solução para esse problema consiste em transformar a classe `Logger` em um **Singleton**. Esse padrão de projeto define como implementar classes que terão, como o próprio nome indica, no máximo uma instância. Mostramos a seguir a versão de `Logger` que funciona como um Singleton:

```
class Logger {

  private Logger() {} // proíbe clientes de chamar new Logger()

  private static Logger instance; // instância única da classe

  public static Logger getInstance() {
    if (instance == null) // 1a vez que chama-se getInstance
      instance = new Logger();
    return instance;
  }

  public void println(String msg) {
    // registra msg na console, mas poderia ser em arquivo
    System.out.println(msg);
  }
}
```

Primeiro, essa classe tem um construtor *default* privado. Com isso, um erro de compilação ocorrerá quando qualquer código fora da classe tentar chamar `new Logger()`. Além disso, um atributo estático armazena a instância única da classe. Quando precisarmos dessa instância, devemos chamar o método público e estático `getInstance()`. Um exemplo é mostrado a seguir:

```
void f() {
  Logger log = Logger.getInstance();
  log.println("Executando f");
  ...
```

```
}

void g() {
  Logger log = Logger.getInstance();
  log.println("Executando g");
  ...
}

void h() {
  Logger log = Logger.getInstance();
  log.println("Executando h");
  ...
}
```

Nesse novo código, temos certeza de que as três chamadas de `getInstance` retornam a mesma instância de `Logger`. Todas as mensagens serão então registradas usando-se essa instância.

Dentre os padrões de projeto propostos no livro da "Gangue dos Quatro", Singleton é o mais polêmico e criticado. O motivo é que ele pode ser usado para camuflar a criação de variáveis e estruturas de dados globais. No nosso caso, a instância única de `Logger` é, na prática, uma variável global que pode ser lida e alterada em qualquer parte do programa. Para isso, basta chamar `Logger.getInstance()`. Como vimos no Capítulo 5, variáveis globais representam uma forma de acoplamento ruim (ou forte) entre classes, isto é, uma forma de acoplamento que não é mediada por meio de interfaces estáveis. Porém, no caso de `Logger`, o uso de Singleton não gera preocupações, pois ele é exatamente aquele recomendado pelo padrão: temos um recurso que é único — um arquivo de log, no caso — e queremos refletir essa característica no projeto, garantindo que ele seja manipulado por meio de uma classe que, por construção, possuirá no máximo uma instância.

Em resumo: Singleton deve ser usado para modelar recursos que, conceitualmente, devem possuir no máximo uma instância durante a execução de um programa. Por outro lado, um uso abusivo do padrão ocorre quando ele é adotado como um artifício para criação de variáveis globais.

Por fim, existe mais uma crítica ao uso de Singletons: eles tornam o teste automático de métodos mais complicado. O motivo é que o resultado da execução de um método pode agora depender de um "estado global" armazenado em um Singleton. Por exemplo, suponha um método `m` que retorna o valor de `x + y`, onde `x` é um parâmetro de entrada e `y` é uma variável global, que é parte de um Singleton. Logo, para testar esse método precisamos fornecer o valor `x`; o que é bastante fácil, pois ele é um parâmetro do método. Mas também precisamos garantir que `y` terá

um valor conhecido; o que pode ser mais difícil, pois ele é um atributo de uma outra classe.

Código Fonte: O código do exemplo de Singleton usado nesta seção está disponível neste link.

1.4 Proxy

Contexto: Suponha uma classe `BookSearch`, cujo principal método pesquisa por um livro, dado o seu ISBN:

```
class BookSearch {
  ...
  Book getBook(String ISBN) { ... }
  ...
}
```

Problema: Suponha que nosso serviço de pesquisa de livros esteja ficando popular e ganhando usuários. Para melhorar seu desempenho, pensamos em introduzir um sistema de cache: antes de pesquisar por um livro, iremos verificar se ele está no cache. Se sim, o livro será imediatamente retornado. Caso contrário, a pesquisa prosseguirá segundo a lógica normal do método `getBook()`. Porém, não gostaríamos que esse novo requisito — pesquisa em cache — fosse implementado na classe `BookSearch`. O motivo é que queremos manter a classe coesa e aderente ao Princípio da Responsabilidade Única. Na verdade, o cache será implementado por um desenvolvedor diferente daquele que é responsável por manter `BookSearch`. Além disso, vamos usar uma biblioteca de cache de terceiros, com diversos recursos e customizações. Por isso, achamos importante separar, em classes distintas, o interesse "pesquisar livros por ISBN" (que é um requisito funcional) do interesse "usar um cache nas pesquisas por livros" (que é um requisito não-funcional).

Solução: O padrão de projeto **Proxy** defende a inserção de um objeto intermediário, chamado proxy, entre um objeto base e seus clientes. Assim, os clientes não terão mais uma referência direta para o objeto base, mas sim para o proxy. Por sua vez, o proxy possui uma referência para o objeto base. Além disso, o proxy deve implementar as mesmas interfaces do objeto base.

O objetivo de um proxy é mediar o acesso a um objeto base, agregando-lhe funcionalidades, sem que ele tome conhecimento disso. No nosso caso, o objeto base é do tipo `BookSearch`; a funcionalidade que pretendemos agregar é um cache; e o proxy é um objeto da seguinte classe:

```
class BookSearchProxy implements BookSearchInterface {

  private BookSearchInterface base;

  BookSearchProxy (BookSearchInterface base) {
    this.base = base;
  }

  Book getBook(String ISBN) {
    if("livro com ISBN no cache")
      return "livro do cache"
    else {
      Book book = base.getBook(ISBN);
      if(book != null)
        "adicione book no cache"
      return book;
    }
  }
  ...
}
```

Deve ser criada também uma interface `BookSearchInterface`, não mostrada no código. Tanto a classe base como a classe do proxy devem implementar essa interface. Isso permitirá que os clientes não tomem conhecimento da existência de um proxy entre eles e o objeto base. Mais uma vez, estamos lançando mão do Princípio "Prefira Interfaces a Classes".

O próximo código ilustra a instanciação do proxy. Primeiro, mostramos o código antes do proxy. Nesse código (a seguir), um objeto `BookSearch` é criado no programa principal e depois passado como parâmetro de qualquer classe ou função que precise do serviço de pesquisa de livros, como a classe `View`.

```
void main() {
  BookSearch bs = new BookSearch();
  ...
  View view = new View(bs);
  ...
}
```

Com a decisão de usar um proxy, vamos ter que modificar esse código para instanciar o proxy (código a seguir). Além disso, `View` passou a receber como parâmetro de sua construtora uma referência para o proxy, em vez de uma referência para o objeto base.

```
void main() {
  BookSearch bs = new BookSearch();
  BookSearchProxy pbs;
  pbs = new BookSearchProxy(bs);
  ...
  View view = new View(pbs);
  ...
}
```

A próxima figura ilustra os objetos e as referências entre eles, considerando a solução que usa um proxy:

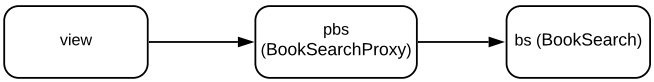


Figura 1.2: Padrão de projeto Proxy

Além de ajudar na implementação de caches, proxies podem ser usados para implementar outros requisitos não-funcionais. Alguns exemplos incluem:

- Comunicação com um cliente remoto, isto é, pode-se usar um proxy para encapsular protocolos e detalhes de comunicação. Esses proxies são chamados de **stubs**.
- Alocação de memória por demanda para objetos que consomem muita memória. Por exemplo, uma classe pode manipular uma imagem em alta resolução. Então, podemos usar um proxy para evitar que a imagem fique carregada o tempo todo na memória principal. Ela somente será carregada, possivelmente do disco, antes da execução de alguns métodos.
- Controlar o acesso de diversos clientes a um objeto base. Por exemplo, os clientes devem estar autenticados e ter permissão para executar certas operações do objeto base. Com isso, a classe do objeto base pode se concentrar na implementação de requisitos funcionais.

1.5 Adaptador

Contexto: Suponha um sistema que tenha que controlar projetores multimídia. Para isso ele deve instanciar objetos de classes fornecidas pelos fabricantes de cada projetor, como ilustrado a seguir:

```
class ProjetorLG {
  public void turnOn() { ... }
  ...
}

class ProjetorSamsung {
  public void enable(int timer) { ... }
  ...
}
```

Para simplificar, estamos mostrando apenas duas classes. Porém, um cenário real pode envolver classes de outros fabricantes de projetores. Também estamos mostrando apenas um método de cada classe, mas elas podem conter outros métodos. Particularmente, o método mostrado é responsável por ligar o projetor. No caso dos projetores da LG, esse método não possui parâmetros. No caso dos projetores da Samsung podemos passar um intervalo em minutos para ligação do projetor. Se esse parâmetro for igual a zero, o projetor é ligado imediatamente. Veja ainda que o nome dos métodos é diferente nas duas classes.

Problema: No sistema de controle de projetores multimídia, queremos usar uma interface única para ligar os projetores, independentemente de marca. O próximo código mostra essa interface e uma classe cliente do sistema:

```
interface Projetor {

  void liga() { ... }

}
...
class SistemaControleProjetores {

   void init(Projetor projetor) {
     projetor.liga();  // liga qualquer projetor
   }

}
```

Porém, as classes de cada projetor — mostradas anteriormente — foram implementadas pelos seus fabricantes e estão prontas para uso. Ou seja, não temos acesso ao código dessas classes para fazer com que elas implementem a interface `Projetor`.

Solução: O padrão de projeto **Adaptador** — também conhecido como **Wrapper** — é uma solução para o nosso problema. Recomenda-se usar esse padrão quando temos que converter a interface de uma classe para outra interface, esperada pelos seus clientes. No nosso exemplo, ele pode ser usado para converter a interface `Projetor` — usada no sistema de controle de projetores — para as interfaces (métodos públicos) das classes implementadas pelos fabricantes dos projetores.

Um exemplo de classe adaptadora, de `ProjetorSamsung` para `Projetor`, é o seguinte:

```
class AdaptadorProjetorSamsung implements Projetor {

   private ProjetorSamung projetor;

   AdaptadorProjetorSamsung (ProjetorSamung projetor) {
     this.projetor = projetor;
   }

   public void liga() {
     projetor.turnOn();
   }

}
```

A classe `AdaptadorProjetorSamsung` implementa a interface `Projetor`. Logo, objetos dessa classe podem ser passados como parâmetro do método `init()` do sistema para controle de projetores. A classe `AdaptadorProjetorSamsung` também possui um atributo privado do tipo `ProjetorSamsung`. A sequência de chamadas é então a seguinte (acompanhe também pelo diagrama de sequência UML, mostrado na próxima página): primeiro, o cliente — no nosso caso, representado pelo método `init` — chama `liga()` da classe adaptadora; em seguida, a execução desse método chama o método equivalente — no caso, `turnOn()` — do objeto que está sendo "adaptado"; no caso, um objeto que acessa projetores Samsung.

Se quisermos manipular projetores LG, vamos ter que implementar uma segunda classe adaptadora. No entanto, seu código será parecido com `AdaptadorProjetorSamsung`.

1.6 Fachada

Contexto: Suponha que implementamos um interpretador para uma linguagem X. Esse interpretador permite executar programas X a partir de uma linguagem hospedeira, no caso Java. Se quiser tornar o exemplo mais real, imagine que

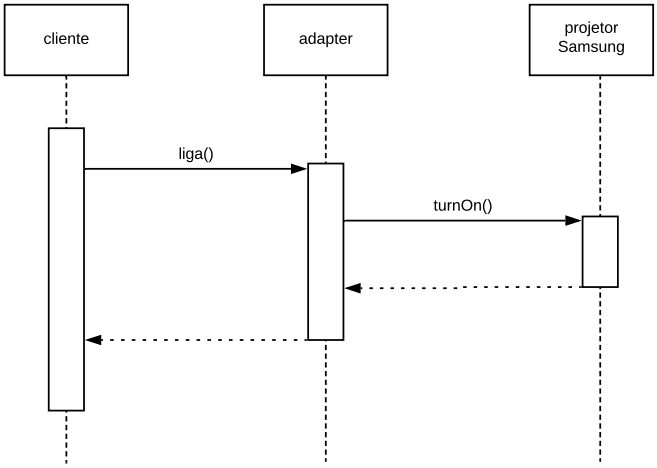


Figura 1.3: Padrão de projeto Adaptador

X é uma linguagem para consulta a dados, semelhante a SQL. Para executar programas X, a partir de Java, os seguintes passos são necessários:

```
Scanner s = new Scanner("prog1.x");
Parser p = new Parser(s);
AST ast = p.parse();
CodeGenerator code = new CodeGenerator(ast);
code.eval();
```

Problema: Como a linguagem X está ficando popular, os desenvolvedores estão reclamando da complexidade do código acima, pois ele requer conhecimento de classes internas do interpretador de X. Logo, os usuários frequentemente pedem uma interface mais simples para usar o interpretador de X.

Solução: O padrão de projeto **Fachada** é uma solução para o nosso problema. Uma Fachada é uma classe que oferece uma interface mais simples para um sistema. O objetivo é evitar que os usuários tenham que conhecer classes internas desse sistema; em vez disso, eles precisam interagir apenas com a classe de Fachada. As classes internas ficam encapsuladas por trás da Fachada.

No nosso problema, a Fachada poderia ser:

```
class InterpretadorX {

  private String arq;

  InterpretadorX(arq) {
    this.arq = arq;
  }

  void eval() {
    Scanner s = new Scanner(arq);
    Parser p = new Parser(s);
    AST ast = p.parse();
    CodeGenerator code = new CodeGenerator(ast);
    code.eval();
  }
}
```

Assim, os desenvolvedores que precisam executar programas X, a partir de Java, poderão fazê-lo por meio de uma única linha de código:

```
new InterpretadorX("prog1.x").eval();
```

Antes de implementar a fachada, os clientes precisavam criar três objetos de tipos internos do interpretador e chamar dois métodos. Agora, basta criar um único objeto e chamar `eval`.

1.7 Decorador

Contexto: Vamos voltar ao sistema de comunicação remota usado para explicar o Padrão Fábrica. Suponha que as classes `TCPChannel` e `UDPChannel` implementam uma interface `Channel`:

```
interface Channel {
   void send(String msg);
   String receive();
}

class TCPChannel implements Channel {
   ...
}
class UDPChannel implements Channel {
   ...
}
```

Problema: Os clientes dessas classes precisam adicionar funcionalidades extras em canais, tais como buffers, compactação das mensagens, log das mensagens trafegadas, etc. Mas essas funcionalidades são opcionais: dependendo do cliente precisamos de apenas algumas funcionalidades ou, talvez, nenhuma delas. Uma primeira solução consiste no uso de herança para criar subclasses com cada possível seleção de funcionalidades. No quadro abaixo, mostramos algumas das subclasses que teríamos que criar (`extends` significa relação de herança):

- `TCPZipChannel extends TCPChannel`
- `TCPBufferedChannel extends TCPChannel`
- `TCPBufferedZipChannel extends TCPZipChannel extends TCPChannel`
- `TCPLogChannel extends TCPChannel`
- `TCPLogBufferedZipChannel extends TCPBufferedZipChannel extends TCPZipChannel extends TCPChannel`
- `UDPZipChannel extends UDPChannel`
- `UDPBufferedChannel extends UDPChannel`
- `UDPBufferedZipChannel extends UDPZipChannel extends UDPChannel`
- `UDPLogChannel extends UDPChannel`
- `UDPLogBufferedZipChannel extends UDPBufferedZipChannel extends UDPZipChannel extends UDPChannel`

Nessa solução, usamos herança para implementar subclasses para cada conjunto de funcionalidades. Suponha que o usuário precise de um canal UDP com buffer e compactação. Para isso, tivemos que implementar `UDPBufferedZipChannel` como subclasse de `UDPZipChannel`, que por sua vez foi implementada como subclasse de `UDPChannel`. Como o leitor deve ter percebido, uma solução via herança é quase que inviável, pois ela gera uma explosão combinatória do número de classes relacionadas com canais de comunicação.

Solução: O Padrão Decorador representa uma alternativa a herança quando se precisa adicionar novas funcionalidades em uma classe base. Em vez de usar herança, usa-se composição para adicionar tais funcionalidades dinamicamente nas classes base. Portanto, Decorador é um exemplo de aplicação do princípio de projeto "Prefira Composição a Herança", que estudamos no capítulo anterior.

No nosso problema, ao optarmos por decoradores, o cliente poderá configurar um `Channel` da seguinte forma:

```
channel = new ZipChannel(new TCPChannel());
// TCPChannel que compacte/descompacte dados enviados/recebidos

channel = new BufferChannel(new TCPChannel());
// TCPChannel com um buffer associado

channel = new BufferChannel(new UDPChannel());
// UDPChannel com um buffer associado

channel = new BufferChannel(new ZipChannel (new TCPChannel()));
// TCPChannel com compactação e um buffer associado
```

Portanto, em uma solução com decoradores, a configuração de um `Channel` é feita no momento da sua instanciação, por meio de uma sequência aninhada de operadores `new`. O `new` mais interno sempre cria uma classe base, no nosso exemplo `TCPChannel` ou `UDPChannel`. Feito isso, os operadores mais externos são usados para "decorar" o objeto criado com novas funcionalidades.

Falta então explicar as classes que são os decoradores propriamente ditos, como `ZipChannel` e `BufferChannel`. Primeiro, elas são subclasses da seguinte classe que não aparece no exemplo, mas que é fundamental para o funcionamento do padrão Decorador:

```
class ChannelDecorator implements Channel {

  protected Channel channel;

  public ChannelDecorator(Channel channel) {
    this.channel = channel;
  }

  public void send(String msg) {
    channel.send(msg);
  }

  public String receive() {
    return channel.receive();
  }

}
```

Essa classe tem duas características importantes:

- Ela é uma `Channel`, isto é, ela implementa essa interface e, portanto, os seus dois métodos. Assim, sempre que for esperado um objeto do tipo `Channel` podemos passar um objeto do tipo `ChannelDecorator` no lugar.
- Ela possui internamente um objeto do tipo `Channel` para o qual delega as chamadas aos métodos `send` e ‘receive. Em outras palavras, um decorador, no nosso caso, vai sempre referenciar um outro decorador. Após implementar a funcionalidade que lhe cabe — um buffer, compactação, etc — ele repassa a chamada para esse decorador.

Por fim, chegamos aos decoradores reais. Eles são subclasses de `ChannelDecorator`, como no código a seguir, que implementa um decorador que compacta e descompacta as mensagens trafegadas pelo canal:

```
class ZipChannel extends ChannelDecorator {

   public ZipChannel(Channel c) {
    super(c);
   }

   public void send(String msg) {
    "compacta mensagem msg"
    super.channel.send(msg);
   }

   public String receive() {
    String msg = super.channel.receive();
    "descompacta mensagem msg"
    return msg;
   }

}
```

Para entender o funcionamento de `ZipChannel`, suponha o seguinte código cliente:

```
Channel c = new ZipChannel(new TCPChannel());
c.send("Hello, world")
```

A chamada de `send` na última linha do exemplo dispara as seguintes execuções de métodos:

- Primeiro, executa-se `ZipChannel.send`, que vai compactar a mensagem.

- Após a compactação, `ZipChannel.send` chama `super.channel.send`, que vai executar `ChannelDecorator.send`, pois `ChannelDecorator` é a superclasse de `ZipChannel`.
- `ChannelDecorator.send` apenas repassa a chamada para o `Channel` por ele referenciado, que no caso é um `TCPChannel`.
- Finalmente, chegamos a `TCPChannel.send`, que vai transmitir a mensagem via TCP.

Código Fonte: O código do exemplo de Decorador usado nesta seção está disponível neste link.

1.8 Strategy

Contexto: Suponha que estamos implementando um pacote de estruturas de dados, com a seguinte classe lista:

```
class MyList {

  ... // dados de uma lista
  ... // métodos de uma lista: add, delete, search

  public void sort() {
    ... // ordena a lista usando Quicksort
  }

}
```

Problema: os nossos clientes estão solicitando que novos algoritmos de ordenação possam ser usados para ordenar os elementos da lista. Explicando melhor, eles querem ter a opção de alterar e definir, por conta própria, o algoritmo de ordenação. No entanto, a versão atual da classe sempre ordena a lista usando o algoritmo Quicksort. Se lembrarmos dos princípios de projeto que estudamos no capítulo anterior, podemos dizer que a classe `MyList` não segue o princípio Aberto/Fechado, considerando o algoritmo de ordenação.

Solução: o Padrão **Strategy** é a solução para o nosso problema de "abrir" a classe `MyList` para novos algoritmos de ordenação, mas sem alterar o seu código fonte. O objetivo do padrão é parametrizar os algoritmos usados por uma classe. Ele prescreve como encapsular uma família de algoritmos e como torná-los intercambiáveis. Assim, seu uso é recomendado quando uma classe é usuária de um certo algoritmo (de ordenação, no nosso exemplo). Porém, como existem diversos

algoritmos com esse propósito, não se quer antecipar uma decisão e implementar apenas um deles no corpo da classe, como ocorre na primeira versão de `MyList`.

Mostra-se a seguir o novo código de `MyList`, usando o Padrão Strategy para configuração do algoritmo de ordenação:

```
class MyList {

  ... // dados de uma lista
  ... // métodos de uma lista: add, delete, search

  private SortStrategy strategy;

  public MyList() {
    strategy = new QuickSortStrategy();
  }

  public void setSortStrategy(SortStrategy strategy) {
    this.strategy = strategy;
  }

  public void sort() {
    strategy.sort(this);
  }

}
```

Nessa nova versão, o algoritmo de ordenação transformou-se em um atributo da classe `MyList` e um método `set` foi criado para configurar esse algoritmo. O método `sort` repassa a tarefa de ordenação para um método de mesmo nome do objeto com a estratégia de ordenação. Nessa chamada, passa-se `this` como parâmetro, pois o algoritmo a ser executado deve ter acesso à lista para ordenar seus elementos.

Para encerrar a apresentação do padrão, mostramos o código das classes que implementam as estratégias — isto é, os algoritmos — de ordenação:

```
abstract class SortStrategy {
  abstract void sort(MyList list);
}
class QuickSortStrategy extends SortStrategy {
  void sort(MyList list) { ... }
}
class ShellSortStrategy extends SortStrategy {
  void sort(MyList list) { ... }
}
```

1.9 Observador

Contexto: Suponha que estamos implementando um sistema para controlar uma estação meteorológica. Nesse sistema, temos que manipular objetos de duas classes: `Temperatura`, que são objetos de "modelo" que armazenam as temperaturas monitoradas na estação meteorológica; e `Termometro`, que é uma classe usada para criar objetos visuais que exibem as temperaturas sob monitoramento. Termômetros devem exibir a temperatura atual que foi monitorada. Se a temperatura mudar, os termômetros devem ser atualizados.

Problema: Não queremos acoplar `Temperatura` (classe de modelo) a `Termometro` (classe de interface). O motivo é simples: classes de interface mudam com frequência. Na versão atual, o sistema possui uma interface textual, que exibe temperaturas em Celsius na console do sistema operacional. Mas, em breve, pretendemos ter interfaces Web, para celulares e para outros sistemas. Pretendemos também oferecer outras interfaces de termômetros, tais como digital, analógico, etc. Por fim, temos mais classes semelhantes a `Temperatura` e `Termometro` em nosso sistema, tais como: `PressaoAtmosferica` e `Barometro`, `UmidadeDoAr` e `Higrometro`, `VelocidadeDoVento` e `Anemometro`, etc. Logo, na medida do possível, gostaríamos de reusar o mecanismo de notificação também nessas classes.

Solução: O padrão **Observador** é a solução recomendada para o nosso contexto e problema. Esse padrão define como implementar uma relação do tipo um-para-muitos entre objetos sujeito e observadores. Quando o estado de um sujeito muda, seus observadores devem ser notificados.

Primeiro, vamos mostrar um possível programa principal para o nosso problema:

```
void main() {
  Temperatura t = new Temperatura();
  t.addObserver(new TermometroCelsius());
  t.addObserver(new TermometroFahrenheit());
  t.setTemp(100.0);
}
```

Esse programa cria um objeto do tipo `Temperatura` (um sujeito) e então adiciona dois observadores nele: um `TermometroCelsius` e um `TermometroFahrenheit`. Por fim, define-se o valor da temperatura para 100 graus Celsius. A suposição é que temperaturas são, por *default*, monitoradas na escala Celsius.

As classes `Temperatura` e `TermometroCelsius` são mostradas a seguir:

```
class Temperatura extends Subject {

  private double temp;

  public double getTemp() {
    return temp;
  }

  public void setTemp(double temp) {
    this.temp = temp;
    notifyObservers();
  }

}

class TermometroCelsius implements Observer{

  public void update(Subject s){
    double temp = ((Temperatura) s).getTemp();
    System.out.println("Temperatura Celsius: " + temp);
  }

}
```

Veja que `Temperatura` herda de uma classe chamada `Subject`. Na solução proposta, todos os sujeitos devem estender essa classe. Ao fazer isso, eles herdam dois métodos:

- `addObserver`. No exemplo, esse método é usado no programa principal para adicionar dois termômetros como observadores de uma instância de `Temperatura`.
- `notifyObservers`. No exemplo, esse método é chamado por `Temperatura` para notificar seus observadores de que o seu valor foi alterado no método `setTemp`.

A implementação de `notifyObservers` — que é omitida no exemplo — chama o método `update` dos objetos que se registraram como observadores de uma determinada instância de `Temperatura`. O método `update` faz parte da interface `Observer`, que deve ser implementada por todo observador, como é o caso de `TermometroCelsius`.

A figura da próxima página mostra um diagrama de sequência UML que ilustra a comunicação entre uma temperatura (sujeito) e três possíveis termômetros (observadores). Assume-se que os três termômetros estão registrados como observadores da temperatura. A sequência de chamadas começa com temperatura recebendo uma chamada para executar `setTemp()`.

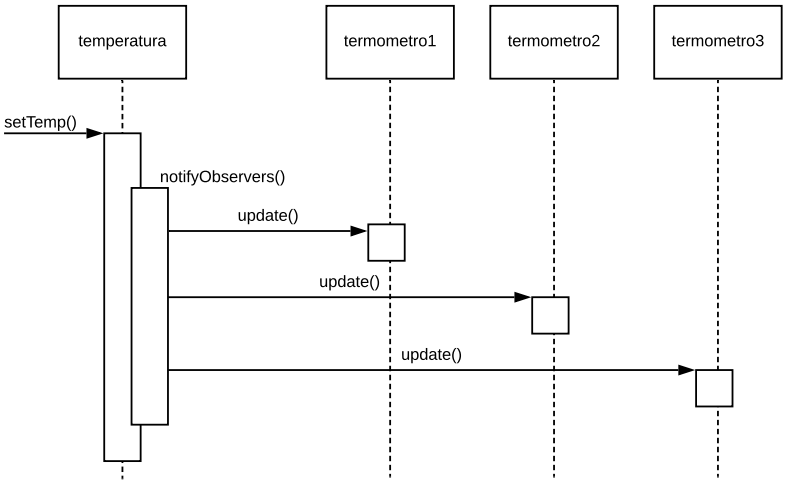


Figura 1.4: Padrão de projeto Observador

O padrão Observador possui as seguintes vantagens principais:

- Ele não acopla os sujeitos a seus observadores. Na verdade, os sujeitos — como `Temperatura`, no exemplo — não conhecem os seus observadores. De forma genérica, os sujeitos publicam um evento anunciando a mudança de seu estado — chamando `notifyObservers` — e os observadores interessados são notificados. Esse comportamento facilita o reúso dos sujeitos em diversos cenários e, também, a implementação de diversos tipos de observadores para o mesmo tipo de sujeito.
- Uma vez implementado, o padrão Observador disponibiliza um mecanismo de notificação que pode ser reusado por diferentes pares de sujeito-observador. Por exemplo, podemos reusar a classe `Subject` e a interface `Observer` para notificações envolvendo pressão atmosférica e barômetros, umidade do ar e higrômetros, velocidade do vento e anemômetros, etc.

Código Fonte: Se quiser conferir o código completo do nosso exemplo de Observador, incluindo o código das classes `Subject` e da interface `Observer`, acesse o seguinte link.

1.10 Template Method

Contexto: Suponha que estamos desenvolvendo uma folha de pagamento. Nela, temos uma classe `Funcionario`, com duas subclasses: `FuncionarioPublico` e `FuncionarioCLT`.

Problema: Pretendemos padronizar um modelo (ou template) para cálculo dos salários na classe base `Funcionario`, que possa depois ser herdado pelas suas subclasses. Assim, as subclasses terão apenas que adaptar a rotina de cálculo de salários às suas particularidades. Mais especificamente, as subclasses saberão exatamente os métodos que precisam implementar para calcular o salário de um funcionário.

Solução: O padrão de projeto **Template Method** resolve o problema que enunciamos. Ele especifica como implementar o "esqueleto" de um algoritmo em uma classe abstrata X, mas deixando pendente alguns passos — ou métodos abstratos. Esses métodos serão implementados nas subclasses de X. Em resumo, um Template Method permite que subclasses customizem um algoritmo, mas sem mudar a sua estrutura geral implementada na classe base.

Um exemplo de Template Method para o nosso contexto e problema é mostrado a seguir:

```
abstract class Funcionario {

   double salario;
   ...
   private abstract double calcDescontosPrevidencia();
   private abstract double calcDescontosPlanoSaude();
   private abstract double calcOutrosDescontos();

   public double calcSalarioLiquido { // template method
     double prev = calcDescontosPrevidencia();
     double saude = calcDescontosPlanoSaude();
     double outros = calcOutrosDescontos();
     return salario - prev - saude - outros;
   }

}
```

Nesse exemplo, `calcSalarioLiquido` é um método template para cálculo do salário de funcionários. Ele padroniza que temos que calcular três descontos: para a previdência, para o plano de saúde do funcionário e outros descontos. Feito isso, o salário líquido é o salário do funcionário subtraído desses três descontos. Porém,

em `Funcionario`, não sabemos ainda como calcular os descontos, pois eles variam conforme o tipo de funcionário (público ou CLT). Logo, são criados métodos abstratos para representar cada um desses passos da rotina de cálculo de salários. Como eles são abstratos, a classe `Funcionario` também foi declarada como abstrata. Como o leitor já deve ter percebido, subclasses de `Funcionario` — como `FuncionarioPublico` e `FuncionarioCLT` — vão herdar o método `calcSalarioLiquido`, que não precisará sofrer nenhuma modificação. No entanto, caberá às subclasses implementar os três passos (métodos) abstratos: `calcDescontosPrevidencia`, `calcDescontosPlanoSaude` e `calcOutrosDescontos`.

Métodos template permitem que "código antigo" chame "código novo". No exemplo, a classe `Funcionario` provavelmente foi implementada antes de `FuncionarioPublico` e `FuncionarioCLT`. Logo, dizemos que `Funcionario` é mais antiga do que as suas subclasses. Mesmo assim, `Funcionario` inclui um método que vai chamar "código novo", implementado nas subclasses. Esse recurso de sistemas orientados a objetos é chamado de **inversão de controle**. Ele é fundamental, por exemplo, para implementação de **frameworks**, isto é, aplicações semi-prontas, que antes de serem usadas devem ser customizadas por seus clientes. Apesar de não ser o único instrumento disponível para esse fim, métodos template constituem uma alternativa interessante para que um cliente implemente o código faltante em um framework.

1.11 Visitor

Contexto: Suponha o sistema de estacionamentos que usamos no Capítulo 5. Suponha que nesse sistema existe uma classe `Veiculo`, com subclasses `Carro`, `Onibus` e `Motocicleta`. Essas classes são usadas para armazenar informações sobre os veículos estacionados no estacionamento. Suponha ainda que todos esses veículos estão armazenados em uma lista. Dizemos que essa lista é uma estrutura de dados **polimórfica**, pois ela pode armazenar objetos de classes diferentes, desde que eles sejam subclasses de `Veiculo`.

Problema: Com frequência, no sistema de estacionamentos, temos que realizar uma operação em todos os veículos estacionados. Como exemplo, podemos citar: imprimir informações sobre os veículos estacionados, persistir os dados dos veículos ou enviar uma mensagem para os donos dos veículos.

No entanto, o objetivo é implementar essas operações fora das classes de `Veiculo` por meio de um código como o seguinte:

```
interface Visitor {
   void visit(Carro c);
   void visit(Onibus o);
   void visit(Motocicleta m);
}

class PrintVisitor implements Visitor {
   public void visit(Carro c) { "imprime dados de um carro" }
   public void visit(Onibus o) { "imprime dados de um onibus" }
   public void visit(Motocicleta m) { "imprime dados de moto" }
}
```

Nesse código, a classe `PrintVisitor` inclui métodos que imprimem os dados de um `Carro`, `Onibus` e `Motocicleta`. Uma vez implementada essa classe, gostaríamos de usar o seguinte código para "visitar" todos os veículos do estacionamento::

```
PrintVisitor visitor = new PrintVisitor();
foreach (Veiculo veiculo: listaDeVeiculosEstacionados) {
   visitor.visit(veiculo); // erro de compilação
}
```

No entanto, no código mostrado, o método `visit` a ser chamado depende do tipo dinâmico do objeto alvo da chamada (`visitor`) e do tipo dinâmico de um parâmetro (`veiculo`). Porém, em linguagens como Java, C++ ou C# apenas o tipo do objeto alvo da chamada é considerado na escolha do método a ser chamado. Dizendo de outro modo, em Java e em linguagens similares, o compilador somente conhece o tipo estático de `veiculo`, que é `Veiculo`. Por isso, ele não consegue inferir qual implementação de `visit` deve ser chamada.

Para ficar mais claro, o seguinte erro ocorre ao compilar o código anterior:

```
visitor.visit(veiculo);
        ^
method PrintVisitor.visit(Carro) is not applicable
  (argument mismatch; Veiculo cannot be converted to Carro)
method PrintVisitor.visit(Onibus) is not applicable
  (argument mismatch; Veiculo cannot be converted to Onibus)
```

Na verdade, esse código somente compila em linguagens que oferecem **despacho duplo** de chamadas de métodos (*double dispatch*). Nessas linguagens, os tipos do objeto alvo e de um dos parâmetros de chamada são usados para escolher o

método que será invocado. No entanto, despacho duplo somente está disponível em linguagens mais antigas e menos conhecidas hoje em dia, como Common Lisp.

Portanto, o nosso problema é o seguinte: como simular *double dispatch* em uma linguagem como Java? Se conseguirmos fazer isso, poderemos contornar o erro de compilação que ocorre no código que mostramos.

Solução: A solução para o nosso problema consiste em usar o padrão de projeto **Visitor**. Esse padrão define como "adicionar" uma operação em uma família de objetos, sem que seja preciso modificar as classes dos mesmos. Além disso, o padrão Visitor deve funcionar mesmo em linguagens com *single dispatching* de métodos, como Java.

Como primeiro passo, temos que implementar um método `accept` em cada classe da hierarquia. Na classe raiz, ele é abstrato. Nas subclasses, ele recebe como parâmetro um objeto do tipo `Visitor`. E a sua implementação apenas chama o método `visit` desse `Visitor`, passando `this` como parâmetro. Porém, como a chamada ocorre no corpo de uma classe, o compilador conhece o tipo de `this`. Por exemplo, na classe `Carro`, o compilador sabe que o tipo de `this` é `Carro`. Logo, ele sabe que deve chamar a implementação de `visit` que tem `Carro` como parâmetro. Para ser preciso, o método exato a ser chamado depende do tipo dinâmico do objeto alvo da chamada (`v`). Porém, isso não é um problema, pois significa que temos um caso de *single dispatch*, que é permitido em linguagens como Java.

```
abstract class Veiculo {
  abstract public void accept(Visitor v);
}
class Carro extends Veiculo {
  ...
  public void accept(Visitor v) {
   v.visit(this);
  }
  ...
}
class Onibus extends Veiculo {
  ...
  public void accept(Visitor v) {
    v.visit(this);
  }
  ...
}

// Idem para Motocicleta
```

Por último, temos que modificar o laço que percorre a lista de veículos estacionados. Agora, chamaremos os métodos `accept` de cada veículo, passando o visitor como parâmetro.

```
PrintVisitor visitor = new PrintVisitor();
foreach (Veiculo veiculo: listaDeVeiculosEstacionados) {
  veiculo.accept(visitor);
}
```

Resumindo, visitors facilitam a adição de um método em uma hierarquia de classes. Um visitor congrega operações relacionadas — no exemplo, impressão de dados de `Veiculo` e de suas subclasses. Mas poderia também existir um segundo visitor, com outras operações — por exemplo, persistir os objetos em disco. Por outro lado, a adição de uma nova classe na hierarquia, por exemplo, `Caminhao`, requer a atualização de todos os visitors com um novo método: `visit(Caminhao)`.

Antes de concluir, é importante mencionar que visitors possuem uma desvantagem importante: eles podem forçar uma quebra no encapsulamento das classes que serão visitadas. Por exemplo, `Veiculo` pode ter que implementar métodos públicos expondo seu estado interno para que os visitors tenham acesso a eles.

Código Fonte: O código do exemplo de Visitor usado nesta seção está disponível neste link.

1.12 Outros Padrões de Projeto

Iterador é um padrão de projeto que padroniza uma interface para caminhar sobre uma estrutura de dados. Normalmente, essa interface inclui métodos como `hasNext()` e `next()`, como mostrado no seguinte exemplo:

```
List<String> list = Arrays.asList("a","b","c");
Iterator it = list.iterator();
while(it.hasNext()) {
  String s = (String) it.next();
  System.out.println(s);
}
```

Um iterador permite percorrer uma estrutura de dados sem conhecer o seu tipo concreto. Em vez disso, basta conhecer os métodos da interface Iterator. Iteradores também permitem que múltiplos caminhamentos sejam realizadas de forma simultânea em cima da mesma estrutura de dados.

Builder é um padrão de projeto que facilita a instanciação de objetos que têm muitos atributos, sendo alguns deles opcionais. Se o valor desses atributos opcionais não for informado, eles devem ser inicializados com um valor *default*. Em vez de criar diversos métodos construtores, um método para cada combinação possível de parâmetros, podemos delegar o processo de inicialização dos campos de um objeto para uma classe `Builder`. Um exemplo é mostrado a seguir, para uma classe `Livro`.

```
Livro esm = new Livro.Builder().
                    setNome("Engenharia Soft Moderna").
                    setEditora("UFMG").setAno(2020).build();

Livro gof = new Livro.Builder().setName("Design Patterns").
                    setAutores("GoF").setAno(1995).build();
```

Uma primeira alternativa ao uso de um Builder seria implementar a instanciação por meio de construtores. Porém, teríamos que criar diversos construtores, pois `Livro` possui diversos atributos, nem todos obrigatórios. Além disso, a chamada desses construtores poderia gerar confusão, pois o desenvolvedor teria que conhecer exatamente a ordem dos diversos parâmetros. Com o padrão Builder, os métodos `set` deixam claro qual atributo de `Livro` está sendo inicializado. Uma segunda alternativa seria implementar os métodos `set` diretamente na classe `Livro`. Porém, isso quebraria o princípio de ocultamento da informação, pois tornaria possível alterar, a qualquer momento, qualquer atributo da classe. Por outro lado, com um Builder, os atributos somente podem ser definidos em tempo de instanciação da classe.

Em tempo, a versão de Builder que apresentamos não corresponde à descrição original do padrão contida no livro da Gangue dos Quatro. Em vez disso, apresentamos uma versão proposta por Joshua Bloch (link). Acreditamos que essa versão, hoje em dia, corresponde ao uso mais comum de Builders. Ela é usada, por exemplo, em classes da API de Java, como `Calendar.Builder` (link).

Código Fonte: O código do exemplo de Builder — incluindo as classes `Livro` e `Livro.Builder` — está disponível neste link. Ao estudá-lo, você perceberá que `Livro.Builder` é uma classe interna, pública e estática de `Livro`. Por isso, é que podemos chamar `new Livro.Builder()` diretamente, sem precisar de instanciar antes um objeto do tipo `Livro`.

1.13 Quando Não Usar Padrões de Projeto

Padrões de projeto têm como objetivo tornar o projeto de um sistema mais flexível. Por exemplo, fábricas facilitam trocar o tipo dos objetos manipulados

por um programa. Um decorador permite personalizar uma classe com novas funcionalidades, tornando-a flexível a outros cenários de uso. O padrão Strategy permite configurar os algoritmos usados por uma classe, apenas para citar alguns exemplos.

Porém, como quase tudo em Computação, o uso de padrões implica em um custo. Por exemplo, uma fábrica requer a implementação de pelo menos mais uma classe no sistema. Para citar um segundo exemplo, Strategy requer a criação de uma classe abstrata e mais uma classe para cada algoritmo. Por isso, a adoção de padrões de projeto exige uma análise cuidadosa. Para ilustrar esse tipo de análise, vamos continuar a usar os exemplos de Fábrica e Strategy:

- Antes de usar uma fábrica, devemos fazer (e responder) a seguinte pergunta: vamos mesmo precisar criar objetos de tipos diferentes no nosso sistema? Existem boas chances de que tais objetos sejam, de fato, necessários? Se sim, então vale a pena usar uma Fábrica para encapsular a criação de tais objetos. Caso contrário, é melhor criar os objetos usando o operador `new`, que é a solução nativa para criação de objetos em linguagens como Java.
- De forma semelhante, antes de incluir o padrão Strategy em uma certa classe devemos nos perguntar: vamos mesmo precisar de parametrizar os algoritmos usados na implementação dessa classe? Existem, de fato, usuários que vão precisar de algoritmos alternativos? Se sim, vale a pena usar o padrão Strategy. Caso contrário, é preferível implementar o algoritmo diretamente no corpo da classe.

Apesar de usarmos apenas dois padrões como exemplo, perguntas semelhantes podem ser feita para outros padrões.

No entanto, em muitos sistemas observa-se um uso exagerado de padrões de projeto, em situações onde os ganhos de flexibilidade e extensibilidade são questionáveis. Existe até um termo para se referir a essa situação: **paternite**, isto é, uma "inflamação" associada ao uso precipitado de padrões de projeto.

John Ousterhout tem um comentário relacionado a essa "doença":

> "O maior risco de padrões de projetos é a sua super-aplicação (*over-application*). Nem todo problema precisa ser resolvido por meio dos padrões de projeto; por isso, não tente forçar um problema a caber em um padrão de projeto quando uma abordagem tradicional funcionar melhor. O uso de padrões de projeto não necessariamente melhora o projeto de um sistema de software; isso só acontece se esse uso for justificado. Assim como ocorre com outros conceitos, a noção de que padrões de projetos são uma boa coisa não significa que quanto mais padrões de projeto usarmos, melhor será nosso sistema."

Ousterhout ilustra seu argumento citando o emprego de decoradores durante a abertura de arquivos em Java, como mostrado no seguinte trecho de código:

```
FileInputStream fs = new FileInputStream(fileName);
BufferedInputStream bs = new BufferedInputStream(fs);
ObjectInputStream os = new ObjectInputStream(bs);
```

Segundo Ousterhout, decoradores adicionam complexidade desnecessária ao processo de criação de arquivos em Java. O principal motivo é que, via de regra, iremos sempre nos beneficiar de um buffer ao abrir qualquer arquivo. Portanto, buffers de entrada/saída deveriam ser oferecidos por *default*, em vez de por meio de uma classe decoradora específica. Assim, as classes `FileInputStream` e `BufferedInputStream` poderiam ser fundidas em uma única classe.

Bibliografia

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

Joshua Bloch. Effective Java. 3rd edition. Prentice Hall, 2017.

Eric Freeman, Elisabeth Robson, Bert Bates, Kathy Sierra. Head First Design Patterns: A Brain-Friendly Guide. O'Reilly, 2004.

Eduardo Guerra. Design Patterns com Java: Projeto Orientado a Objetos guiado por Padrões. Caso do Código, 2014.

Fernando Pereira, Marco Tulio Valente, Roberto Bigonha, Mariza Bigonha. Arcademis: A Framework for Object Oriented Communication Middleware Development. Software: Practice and Experience, 2006.

Fabio Tirelo, Roberto Bigonha, Mariza Bigonha, Marco Tulio Valente. Desenvolvimento de Software Orientado por Aspectos. XXIII Jornada de Atualização em Informática (JAI), 2004.

Exercícios de Fixação

1. (ENADE 2011, adaptado) Em relação a padrões de projeto, assinale V ou F.

( ) Prototype é um tipo de padrão estrutural.

( ) Singleton tem por objetivo garantir que uma classe tenha ao menos uma instância e fornecer um ponto global de acesso para ela.

( ) Template Method tem por objetivo definir o esqueleto de um algoritmo em uma operação, postergando a definição de alguns passos para subclasses.

( ) Iterator fornece uma maneira de acessar sequencialmente os elementos de um objeto agregado sem expor sua representação subjacente.

2. Dê o nome dos seguintes padrões de projeto:

   a) Oferece uma interface unificada e de alto nível que torna mais fácil o uso de um sistema:

   b) Garante que uma classe possui uma única instância e oferece um ponto único de acesso a ela:

   c) Facilita a construção de objetos complexos com vários atributos, sendo alguns deles opcionais:

   d) Converte a interface de uma classe para outra interface esperada pelos clientes. Permite que classes trabalhem juntas, o que não seria possível devido à incompatibilidade de suas interfaces:

   e) Oferece uma interface ou classe abstrata para criação de uma família de objetos relacionados:

   f) Oferece um método para centralizar a criação de um tipo de objeto:

   g) Funciona como um intermediário que controla o acesso a um objeto base:

   h) Permite adicionar dinamicamente novas funcionalidades a uma classe:

   i) Oferece uma interface padronizada para caminhar em estruturas de dados:

   j) Permite parametrizar os algoritmos usados por uma classe:

k) Torna uma estrutura de dados aberta a extensões, isto é, permite adicionar uma função em cada elemento de uma estrutura de dados, mas sem alterar o código de tais elementos:

l) Permite que um objeto avise outros objetos de que seu estado mudou:

m) Define o esqueleto de um algoritmo em uma classe base e delega a implementação de alguns passos para subclasses:

3. Dentre os padrões de projeto que respondeu na questão (2), quais são criacionais?

4. Considerando as respostas da questão (2), liste padrões de projeto que:

a) Ajudam a tornar uma classe aberta a extensões, sem que seja preciso modificar o seu código fonte (isto é, padrões que colocam em prática o princípio Aberto/Fechado).

b) Ajudam a desacoplar dois tipos de classes.

c) Ajudam a incrementar a coesão de uma classe (isto é, fazem com que a classe tenha Responsabilidade Única).

d) Simplificam o uso de um sistema.

5. Qual a semelhança entre Proxy, Decorador e Visitor? E qual a diferença entre esses padrões?

6. No exemplo de Adaptador, mostramos o código de uma única classe adaptadora (AdaptadorProjetorSamsung). Escreva o código de uma classe semelhante, mas que adapte a interface Projetor para a interface ProjetorLG (o código de ambas interfaces é mostrado na Seção 6.5). Chame essa classe de AdaptadorProjetorLG.

7. Suponha uma classe base A. Suponha que queremos adicionar quatro funcionalidades opcionais F1, F2, F3 e F4 em A. Essas funcionalidades podem ser adicionadas em qualquer ordem, isto é, a ordem não é importante. Se usarmos herança, quantas subclasses de A teremos que implementar? Se optarmos por uma solução por meio de decoradores, quantas classes teremos que implementar (sem contar a classe A). Justifique e explique sua resposta.

8. No exemplo de Decorador, mostramos o código de um único decorador (`ZipChannel`). Escreva o código de uma classe semelhante, mas que imprima a mensagem a ser transmitida ou recebida na console. Chame essa classe de `LogChannel`.

9. Dado o código abaixo de uma classe `Subject` (do padrão Observador):

```
interface Observer {
  public void update(Subject s);
}

class Subject {

  private List<Observer> observers = new ArrayList<Observer>();

  public void addObserver(Observer observer) {
    observers.add(observer);
  }

  public void notifyObservers() {
    (A)
  }

}
```

Implemente o código de `notifyObservers`, comentado com um (A) acima.

10. Suponha o exemplo de Visitor que usamos na Seção 6.11. Especificamente, suponha o seguinte código, mostrado no final da seção.

```
PrintVisitor visitor = new PrintVisitor();

foreach(Veiculo veiculo: listaDeVeiculosEstacionados) {
  veiculo.accept(visitor);
}
```

Suponha que `listaDeVeiculosEstacionados` armazene quatro objetos: `umCarro`, `umOnibus`, `umOutroCarro` e `umOutroOnibus`. Desenhe um diagrama de sequência UML que mostre os métodos executados por esse trecho de código (suponha que ele é executado por um objeto `main`).

11. Suponha a API de Java para E/S. Suponha que para evitar o que chamamos de paternite, você fez a união das classes `FileInputStream` e `BufferedInputStream` em uma única classe. Como discutimos na Seção 6.13, o mecanismo de buffer será ativado por *default* na classe que você criou. Porém, como você tornaria possível ativar buffers na nova classe, caso isso fosse necessário?

12. Em uma entrevista dada ao site InformIT, em 2009, por ocasião dos 15 anos do lançamento da primeira edição do GoF, três dos autores do livro mencionaram que — se tivessem que lançar uma segunda edição do trabalho — provavelmente manteriam os padrões originais e incluiriam alguns novos, que se tornaram comuns

desde o lançamento da primeira edição, em 1994. Um dos novos padrões que eles mencionaram na entrevista é chamado de **Null Object**. Estude e explique o funcionamento e os benefícios desse padrão de projeto. Para isso, você vai encontrar diversos artigos na Web. Mas se preferir consultar um livro, uma boa referência é o Capítulo 25 do livro *Agile Principles, Patterns, and Practices in C#*, de Robert C. Martin e Micah Martin. Ou então o refactoring chamado "Introduce Null Object" do livro de Refactoring de Martin Fowler.