# Engenharia de Software

## Capítulo 6 - Padrões de Projeto

# Livro-texto

Slides baseados no conteúdo do livro **Engenharia de Software Moderna** de Marco Tulio Valente

ISBN: 978-65-00-01950-6

Site: https://engsoftmoderna.info

# Introdução

- **Padrões de Projeto** descrevem objetos e classes que se relacionam para resolver um problema genéricos em um contexto particular
- Visam a criação de projetos de softwarte flexíveis e extensíveis
- Para aplicar os padrões propostos, precisamos entender:
  1. o problema que o padrão pretende resolver
  2. o contexto em que o problema ocorre
  3. a solução proposta

# Introdução

- Benefícios do domínio de padrões de projeto por DEVs
  - Quando ele estiver implementando o seu próprio sistema, pois pode ajudá-lo a adotar uma solução de projeto já testada e validada
  - Quando ele estiver usando um sistema de terceiros, pois pode ajudá-lo a entender o comportamento e a estrutura da *classe* que ele precisa usar

# Introdução

- **Criacionais**: propõem soluções flexíveis para criação de objetos
  - Abstract Factory, Factory Method, Singleton, Builder e Prototype
- **Estruturais**: propõem soluções flexíveis para composição de classes e objetos
  - Proxy, Adapter, Facade, Decorator, Bridge, Composite e Flyweight

# Introdução

- **Comportamentais**: propõem soluções flexíveis para a interação e divisão de responsabilidades entre classes e objetos
  - Strategy, Observer, Template Method, Visitor, Chain of Responsability, Command, Interpreter, Iterator, Mediator, Memento e State

# Factory

- **Contexto**: um sistema distribuído baseado em TCP/IP
- **Problema**: "parametrizar" o código para criar objetos dos tipos `TCPChannel` ou `UDPChannel`
- **Solução**: utilizar o padrão **Factory (Fábrica)**. A 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

# Factory (Problema)

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

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

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

# Factory (Solução)

```
class ChannelFactory {
    public static Channel create() {
        return new TCPChannel();
    }
}
void f() {
    Channel c = ChannelFactory.create();
    ...
}
void g() {
    Channel c = ChannelFactory.create();
    ...
}
...
```

# Factory (Fábrica) - Fábrica Abstrata

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

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

# Singleton

- **Contexto**: uma classe `Logger`, usada para registrar as operações realizadas em um sistema
- **Problema**: cada método que precisa registrar eventos cria a sua própria instância de `Logger`, mas gostaríamos que existisse uma única instância dessa classe e que ela fosse usada em todo o sistema, pois se `Logger` for gravar em arquivos, a cada nova instanciação, apagará o arquivo anterior
- **Solução**: transformar a classe `Logger` em **Singleton**

# Singleton (Problema)

```
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");
    ...
}
```

# Singleton (Solução)

```
class Logger {
    private static Logger instance; // Instância única
    private Logger() {} // Proíbe clientes de chamar new Logger()
    public static Logger getInstance() {
        if (instance == null) {
            instance = new Logger();
        }
        return instance;
    }
    public void println(String msg) {
        // Registra msg no console, mas poderia ser em um arquivo
        System.out.println(msg);
    }
}
```

# Singleton (Solução)

```
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");
    ...
}
```

# Singleton (Solução)

- `Logger` é, na prática, uma variável global, que pode ser lida e alterada em qualquer parte do programa
- Mas variáveis globais são um problema, pois representam um acoplamento forte entre classes
- Porém, no caso se `Logger`, o uso de Singleton não gera preocupações, pois temos um recurso que é único e essa característica está sendo refletida no projeto

# Proxy

- **Contexto**: uma classe `BookSearch`, cujo principal método pesquisa por um livro, dado o seu ISBN
- **Problema**: introduzir um sistema de cache, porém, não gostaríamos que fosse implementado na classe `BookSearch`, pois queremos manter a classe coesa e aderente ao Princípio da Responsabilidade Única
- **Solução**: utilizar o padrão de projeto **Proxy**, que defende a inserção de um objeto intermediário, chamado proxy, entre um objeto base e seus clientes

# Proxy (Problema)

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

# Proxy (Solução)

```
interface BookSearchInterface {
    Book getBook(String ISBN);
}
```

# Proxy (Solução)

```
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;
        }
    }
}
```

# Proxy (Solução)

- Além de ajudar na implementação de caches, proxies podem ser usados para implementar outros requisitos não-funcionais, como:
  - Comunicação com um cliente remoto, para encapsular protocolos e detalhes de comunicação (conhecidos como **stubs**)
  - Alocação de memória por demanda de objetos que consomem muita memória
  - Controlar o acesso de diversos clientes a um objeto base, como por exemplo, os clinetes devem estar autenticados e ter permissão para executar certas operações do objeto base

# Adapter (Adaptador)

- **Contexto**: um sistema que tenha que controlar projetores multimídia
- **Problema**: queremos utilizar uma interface única para ligar os projetores, independente de marca. Porém, não temos acesso ao código dessas classes para fazer com que elas implementem a interface `Projetor`
- **Solução**: o padrão de projeto **Adapter** (também conhecido como **Wrapper**). Recomenda-se utilizar esse padrão quando temos que converter a interface de uma classe para outra interface, esperada pelos seus clientes

# Adapter (Problema)

```
interface Projetor {
    void liga();
}
...
class SistemaControleProjetores() {
    void init(Projetor projetor) {
        projetor.liga(); // Liga qualquer projetor
    }
}
```

# Adapter (Solução)

```
class AdaptadorProjetorSamsung implements Projetor {
    private ProjetorSamsung projetor;

    AdaptadorProjetoSamsung(ProjetorSamsung projetor) {
        this.projetor = projetor;
    }

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

# Facade (Fachada)

- **Contexto**: implementar um interpretador para uma linguagem X
- **Problema**: os usuários pedem uma interface mais simples para chamar o interpretador da linguagem X
- **Solução**: o padrão de projeto **Facade**. Um Facade é uma classe que oferece uma interface mais simples para um sistema. O objetivo é evitar que usuarios tenham que conhecer classes internas desse sistema

# Facade (Problema)

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

# Facade (Solução)

```
class InterpretadorX {
    private String arq;

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

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

# Facade (Solução)

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

# Decorator (Decorador)

- **Contexto**: as classes `TCPChannel` e `UDPChannel` implementam uma interface `Channel`
- **Problema**: os clientes dessas classes precisam adicionar funcionalidades extras em canais, tais como buffers, compactação das mensagens, log das mensagens trafegadas, etc.
- **Solução**: o padrão de projeto **Decorator**, que ao contrário da herança que nesse contexto causaria uma explosão combinatória do número de classes relacionadas com canais de comunicação, apresenta uma alternativa a herança quando se precisa adicionar novas funcionalidades em uma classe base

# Decorator (Contexto)

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

class TCPChannel implements Channel {
    ...
}

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

# Decorator (Problema)

```
TCPZipChannel extends TCPChannel
TCPBufferedChannel extends TCPChannel
TCPBufferedZipChannel extends TCPZipChannel
TCPLogChannel extends TCPChannel
TCPLogBufferedZipChannel extends TCPBufferedZipChannel

UDPZipChannel extends UDPChannel
UDPBufferedChannel extends UDPChannel
UDPBufferedZipChannel extends UDPZipChannel
UDPLogChannel extends UDPChannel
UDPLogBufferedZipChannel extends UDPBufferedZipChannel
```

# Decorator (Solução)

```
class ChannelDecorator implements Channel {
    private Channel channel;

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

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

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

# Decorator (Solução)

```
class ZipDecorator extends ChannelDecorator {
    public ZipChannel(Channel channel) {
        super(channel);
    }
    public void send(String msg) {
        "compacta mensagem msg"
        super.send(msg);
    }
    public String receive() {
        String msg = super.receive();
        "descompacta mensagem msg"
        return msg;
    }
}
```

# Decorator (Solução)

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

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

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

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

# Strategy

- **Contexto**:
- **Problema**: os clientes querem ter a opção de alterar e definir, por conta própria, o algoritmo de ordenação
- **Solução**: o padrão de projeto **Strategy**. Ele vai permitir "abrir" a classe `Mylist` para novos algoritmos de ordenação, mas sem alterar o seu código fonte, prescrevendo como encapsular uma família de algoritmos e como torná-los intercambiáveis.

# Strategy (Problema)

```
class MyList {
    // Dados de uma lista
    // Métodos de uma lista: add, delete, search

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

# Strategy (Solução)

```
class MyList {
    // Dados de uma lista
    // Métodos de uma lista: add, delete, search
    private SortStrategy strategy;
    public MyList() {
        this.strategy = new QuickSortStrategy();
    }
    public void setSortStrategy(SortStrategy strategy) {
        this.strategy = strategy;
    }
    public void sort() {
        this.strategy.sort(this);
    }
}
```

# Observer (Observador)

- **Contexto**: implementar um sistema para controlar uma estação meteorológica
- **Problema**: não queremos acoplar `Temperatura` (classe de modelo) a `Termometro` (classe de interface), pois classes de interface mudam com frequência
- **Solução**: o padrão de projeto **Observer**. Este padrão define como implementar uma relação do tipo um-para-muitos entre sujeito e observadores. Quando o estado de um sujeito muda, seus observadores devem ser notificados

# Observer (Problema)

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

# Observer (Solução)

```
class Temperatura extends Subject {
    private double temp;
    public double getTemp() {
        return this.temp;
    }
    public void setTemp() {
        this.temp = temp;
        notifyObservers();
    }
}
class TermometroCelsius implements Observer {
    public void update(Subject s) {
        double temp = ((Temperatura) s).getTemp();
        System.out.println("Temperatura Celsius: " + temp);
    }
}
```

# Vantagens do Padrão Observer

- Não acopla o sujeito a seus observadores
- Disponibiliza um mecanismo de notificação que pode ser reusado por diferentes pares de sujeito-observador

# Template Method

- **Contexto**: desenvolvimento de uma folha de pagamento em que temos uma classe `Funcionario`, com duas subclasses: `FuncionarioPublico` e `FuncionarioCLT`
- **Problema**: padronizar um modelo (ou template) para cálculo dos salários na base `Funcionario`, que possa ser herdado pelas subclasses
- **Solução**: o padrão de projeto **Template Method**. Ele especifica como implementar o "esqueleto" de um algoritmo de uma classe abstrata X, mas deixando pendente alguns passos (ou métodos abstratos)

# Template Method (Solução)

```
abstract class Funcionario {
    double salario;
    ...
    abstract double calcDescontosPrevidencia();
    abstract double calcDescontosPlanoSaude();
    abstract double calcOutrosDescontos();

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

# Template Method (Solução)

```
class FuncionarioPublico extends Funcionario {
    ...
    double calcDescontosPrevidencia() {
        // Calcula os descontos da Previdência para o funcionário público
    }
    double calcDescontosPlanoSaude() {
        // Calcula os descontos do Plano de Saúde para o funcionário público
    }
    double calcOutrosDescontos() {
        // Calcula outros descontos para o funcionário público
    }
    // calcSalarioLiquido() é herdado de Funcionario, assim não precisa ser
    // redeclarado
}
```

# Visitor

- **Contexto**: um sistema de estacionamento de veículos, em que existe a classe `Veiculo`, com subclasses `Carro`, `Onibus` e `Motocicleta` e que esses veículos estão armazenados em uma lista
- **Problema**: realizar uma operação em todos os veículos estacionados, como imprimir informações dos veículos, persistir os dados ou enviar uma mensagem aos donos dos veículos
- **Solução**: o padrão de projeto **Visitor**. Esse padrão define como "adicionar" uma operação em uma família de objetos, sem que seja preciso modificar a classe dos mesmos

# Visitor (Problema)

```
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 carro"
    }
    public void visit(Onibus o) {
        "imprime dados de onibus"
    }
    public void visit(Motocicleta m) {
        "imprime dados de motocicleta"
    }
}
```

# Visitor (Problema)

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

# Visitor (Solução)

```
abstract class Veiculo {
    abstract public void accept(Visitor v);
}
class Carro extends Veiculo {
    ...
    public void accept(Visitor v) {
        v.visit(this);
    }
    ...
}
// Mesmo código para Onibus e Motocicleta
```

# Visitor (Solução)

```
PrintVisitor visitor = new PrintVisitor();
for (Veiculo veiculo : listaDeVeiculosEstacionados) {
    veiculo.accept(visitor); // Erro de compilação
}
```

# Iterator (Iterador)

- Padroniza a interface para caminhar sobre uma estrutura de dados
- Permite percorrer uma estrutura de dados sem conhecer o seu tipo concreto
- A interface inclui métodos como `hasNext` e `next`

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

# Builder

- Facilita a instanciação de objetos que têm muitos atributos, sendo alguns deles opcionais
- Em vez de criar diversos construtores, um método para cada combinação possível de parâmetros, podemos delegar o processo de inicialização dos campos de um objeto para uma classe `Builder`
- A chamada dos construtores poderia gerar confusão, pois o DEV teria que conhecer exatamente a ordem dos diversos parâmetros

# Builder

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

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

# Quando Não Usar Padrões de Projeto

- O uso de padrões têm um custo. Vale a pena esse custo extra?
  - Requer a criação de classes/interfaces extras
  - Requer a criação de código (no corpo dos métodos) extras
- *O maior risco de padrões de projetos é a sua super-aplicacão (over-application)*