

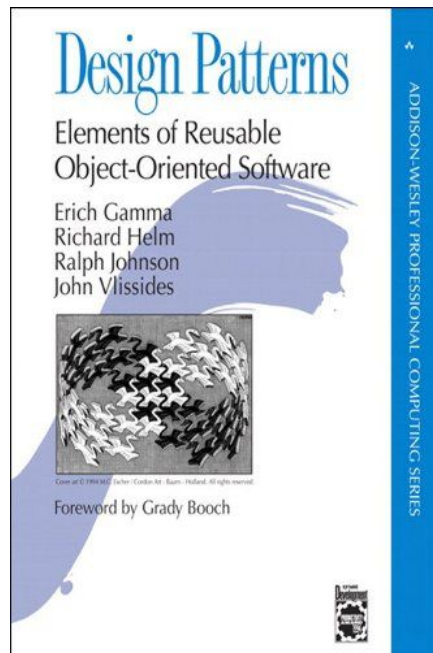
Padrões de Projeto

Profa. Fabíola S. F. Pereira

fabiola.pereira@ufu.br

Padrões de Projeto

- Soluções recorrentes para problemas de projeto enfrentados por engenheiros de software



1994, conhecido como livro da
"Gangue dos Quatro" ou GoF

Utilidade #1: Reúso de projeto

- Suponha que eu tenho um problema de projeto
- Pode existir um padrão que resolve esse problema
- Vantagem: ganho tempo e não preciso reinventar a roda

Utilidade #2: Vocabulário para comunicação

- Vocabulário para discussões, documentação, etc

```
public abstract class DocumentBuilderFactory  
extends Object
```

Defines a **factory API** that enables applications to obtain a parser that produces DOM object trees from XML documents.

Criacionais	Estruturais	Comportamentais
Abstract Factory Factory Method Singleton Builder Prototype	Proxy Adapter Facade Decorator Bridge Composite Flyweight	Strategy Observer Template Method Visitor Chain of Responsibility Command Interpreter Iterator Mediator Memento State

23 padrões

Estrutura da Apresentação dos Padrões

- Contexto (sistema ou parte de um sistema)
- Problema (de projeto)
- Solução (por meio de um padrão de projeto)

Importante

- Padrões de projeto ajudam em **design for change**
- Facilitam mudanças futuras no código
- Se o código dificilmente vai precisar de mudar, então uso de padrões não é interessante

(1) Fábrica

Relembrando a estrutura da explicação/slides

- Contexto
- Problema
- Solução

Contexto: Sistema que usa canais de comunicação

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

Problema

- Alguns clientes vão precisar de usar UDP, em vez de TCP
- Como parametrizar as chamadas de **new**?
 - Espalhadas em vários pontos do código
- Como criar um projeto que facilite essa mudança?

Solução: Padrão Fábrica

- Fábrica: método que centraliza a criação de certos objetos

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

Único ponto de mudança se quisermos usar UDP

Sem Fábrica

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

Sem Fábrica

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



Com Método Fábrica Estático

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

(2) Singleton

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

Contexto:
Classe Logger


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

Contexto:
Classe Logger

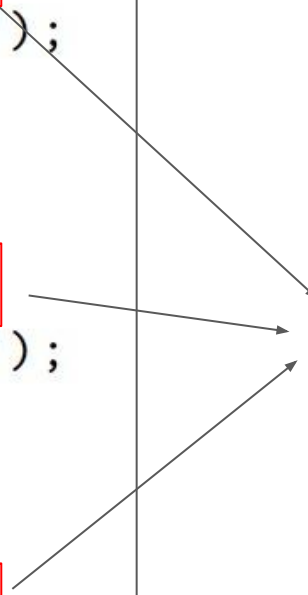
```
void f() {  
    Logger log = new Logger();  
    log.println("Executando f");  
    ...  
}  
void g() {  
    Logger log = new Logger();  
    log.println("Executando g");  
    ...  
}  
void h() {  
    Logger log = new Logger();  
    log.println("Executando h");  
    ...  
}
```

Contexto:
Classe Logger

```
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:

Cada método cliente
usa sua própria
instância de Logger



Problema

- Como fazer com que todos os clientes usem a mesma instância de Logger?
- Na verdade, deve existir uma única instância de Logger
 - Toda operação, por exemplo, seria registrada no mesmo arquivo

Solução: **Padrão Singleton**

- Transformar a classe Logger em um Singleton
- Singleton: classe que possui no máximo uma instância

```
class Logger {  
    private Logger() {} // proíbe clientes 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);  
    }  
}
```

```
class Logger {  
  
    private Logger() {} // proíbe clientes 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);  
    }  
}
```

```
class Logger {  
  
    private Logger() {} // proíbe clientes chamar new Logger()  
  
    private static Logger instance; // instância única da classe  
  
    public static Logger getInstance() {  
        1 if (instance == null) // 1a vez que chama-se getInstance  
            instance = new Logger();  
        2 return instance;  
    }  
  
    public void println(String msg) {  
        // registra msg na console, mas poderia ser em arquivo  
        System.out.println(msg);  
    }  
}
```



```
class Logger {  
  
    private Logger() {} // proíbe clientes 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);  
    }  
}
```

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

Mesma
instância

(3) Proxy

Contexto: Função que pesquisa livros

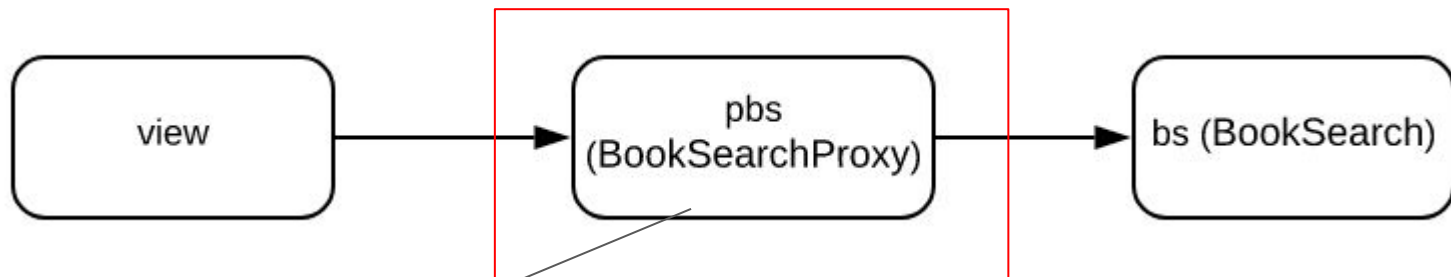
```
class BookSearch {  
    ...  
    Book getBook(String ISBN) { ... }  
    ...  
}
```

Problema: Inserir um cache (para melhorar desempenho)

- Se "livro no cache"
 - retorna livro imediatamente
 - caso contrário, continua com a pesquisa
- Porém, não queremos modificar o código de BookSearch
 - Classe já está funcionando
 - Já tem um desenvolvedor acostumado a mantê-la, etc

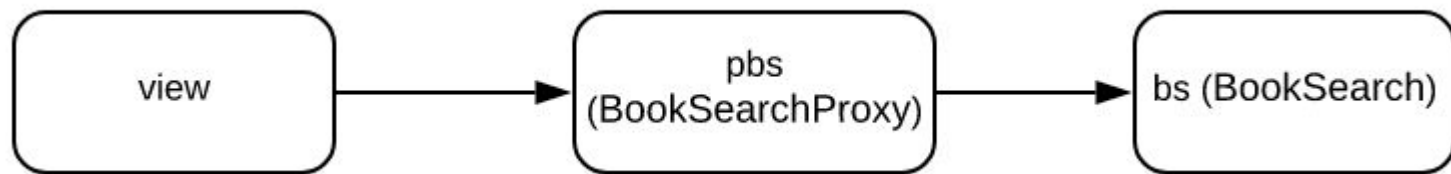
Solução: Padrão Proxy

- Proxy: objeto intermediário entre cliente e um objeto base
- Clientes não "falam" mais com o objeto base (diretamente)
- Eles têm que passar antes pelo proxy

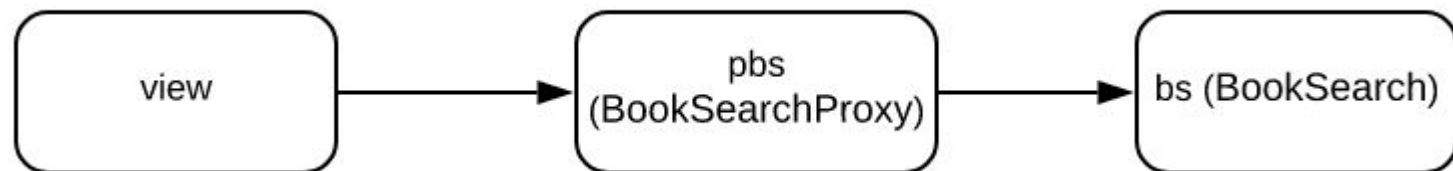


Implementa a lógica do cache

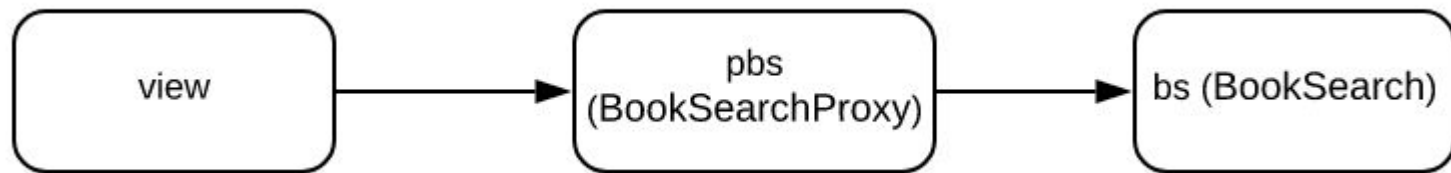
```
void main() {  
    BookSearch bs = new BookSearch();  
    BookSearchProxy pbs;  
    pbs = new BookSearchProxy(bs);  
    ...  
    View view = new View(pbs);  
    ...  
}
```



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




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



Exercícios de Fixação

Assinale V ou F

- () Singleton é um padrão que garante que uma classe possui, no máximo, uma instância e oferece um ponto único de acesso a ela.
- () Fábrica Abstrata é um padrão que oferece uma interface ou classe abstrata para criação de uma família de objetos relacionados
- () Proxy é um padrão que funciona como um intermediário que controla o acesso a um objeto base
- () Um proxy tem que encaminhar todas as chamadas realizadas em métodos de sua interface para métodos equivalentes do objeto base, podendo, no entanto, realizar algum processamento antes ou depois desse encaminhamento.

Fim

Assinale V ou F

(V) Singleton é um padrão que garante que uma classe possui, no máximo, uma instância e oferece um ponto único de acesso a ela.

(V) Fábrica Abstrata é um padrão que oferece uma interface ou classe abstrata para criação de uma família de objetos relacionados

(V) Proxy é um padrão que funciona como um intermediário que controla o acesso a um objeto base

(F) Um proxy tem que encaminhar todas as chamadas realizadas em métodos de sua interface para métodos equivalentes do objeto base, podendo, no entanto, realizar algum processamento antes ou depois desse encaminhamento.