

# Engenharia de Software Moderna

## Cap. 6 - Padrões de Projeto

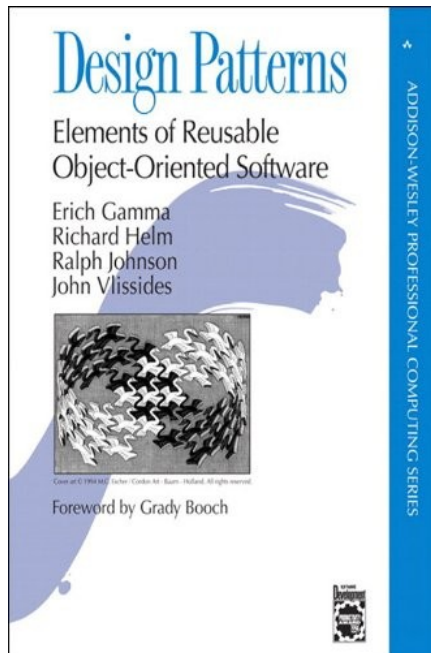
Prof. Marco Tulio Valente

<https://engsoftmoderna.info>, @engsoftmoderna

Licença [CC-BY](#); permite copiar, distribuir, adaptar etc; porém, **créditos devem ser dados ao autor dos slides**

# 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
<b>Abstract Factory</b> Factory Method <b>Singleton</b> Builder Prototype	<b>Proxy</b> <b>Adapter</b> <b>Facade</b> <b>Decorator</b> Bridge Composite Flyweight	<b>Strategy</b> <b>Observer</b> <b>Template Method</b> Visitor Chain of Responsibility Command Interpreter Iterator Mediator Memento State

# 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

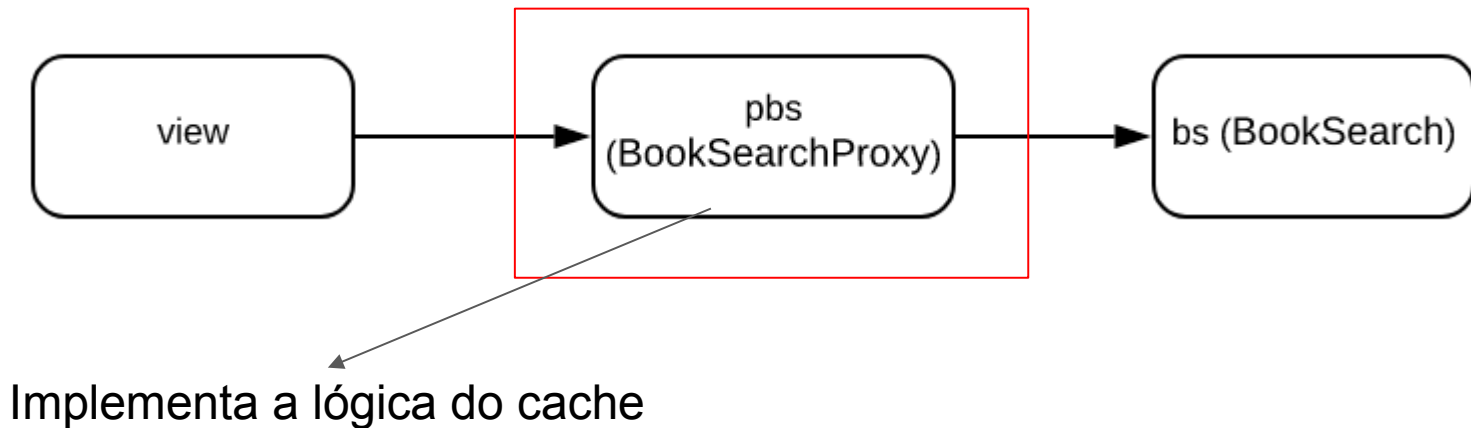
```
class BookSearch {  
    ...  
    Book getBook(String ISBN) { ... }  
    ...  
}
```

# Problema: Inserir um cache (para melhorar desempenho)

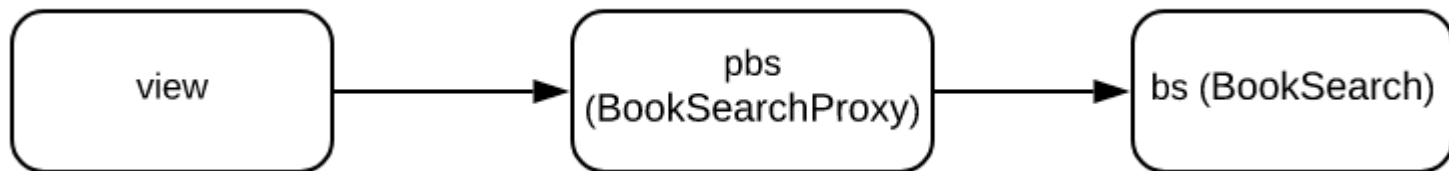
- Se "livro no cache"
  - retorna livro imediatamente
  - caso contrário, continua com a pesquisa
- Porém, não queremos modificar o código de BookSearch
  - Classe já está funcionando
  - Já tem um desenvolvedor acostumado a mantê-la, etc

# Solução: **Padrão Proxy**

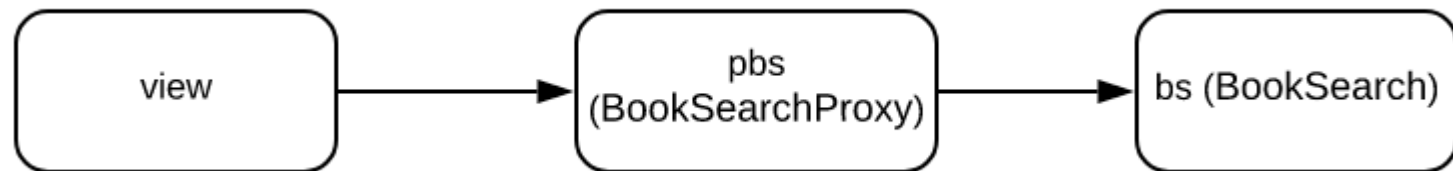
- Proxy: objeto intermediário entre cliente e um objeto base
- Clientes não "falam" mais com o objeto base (diretamente)
- Eles têm que passar antes pelo proxy



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

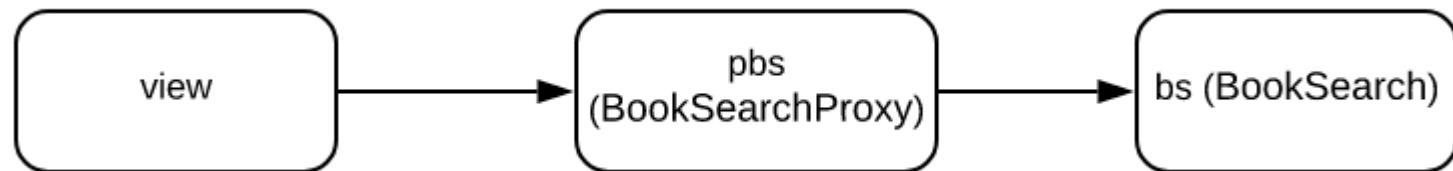


```
void main() {  
    BookSearch bs = new BookSearch();  
    BookSearchProxy pbs;  
    pbs = new BookSearchProxy(bs);  
    ...  
    View view = new View(pbs);  
    ...  
}
```





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



O código de BookSearchProxy está no livro

## **(4) Adaptador**

# Contexto: Sistema para Controlar Projetores Multimídia

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

Classes fornecidas pelos fabricantes dos projetores

# Problema

- No sistema de controle de multimídias, eu gostaria de manipular uma única interface Projektor

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

Sempre usaria objetos dessa interface, sem me preocupar com a classe concreta que implementa a interface

## Problema (cont.)

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

- São classes dos fabricantes dos projetores
- Eu não tenho acesso a elas. Para, por exemplo, fazer com que elas implementem a interface `Projetor`

# Solução: **Padrão Adaptador**



# Solução: Implementar uma Classe Adaptadora

```
class AdaptadorProjetoSamsung implements Projeto {  
  
    private ProjetoSamsung projetor;  
  
    AdaptadorProjetoSamsung (ProjetoSamung projetor) {  
        this.projetor = projetor;  
    }  
  
    public void liga() {  
        projetor.turnOn();  
    }  
  
}
```



# Solução: Implementar uma Classe Adaptadora

```
class AdaptadorProjetoSamsung implements Projeto {  
    private ProjetoSamsung projetor;  
  
    AdaptadorProjetoSamsung (ProjetoSamung projetor) {  
        this.projetor = projetor;  
    }  
  
    public void liga() {  
        projetor.turnOn();  
    }  
}
```

Padrão de  
Entrada

Projeto



Padrão de  
Saída

Projeto

AdaptadorProjeto

## **(5) Fachada**

# Contexto / Problema / Solução

- Contexto: módulo M usado por diversos outros módulos
- Problema: interface de M é complexa; clientes reclamam que é difícil usar a interface pública de M
- Solução: criar uma interface mais simples para M, chamada de **Fachada**.

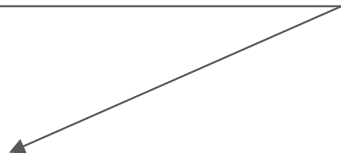
Exemplo: Interpretador

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

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



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



Interface mais simples,  
que a de cima

## **(6) Decorador**



# Contexto: Sistema que usa canais de comunicação

- Já foi usado para explicar o padrão Fábrica

```
interface Channel {  
    void send(String msg);  
    String receive();  
}  
  
class TCPChannel implements Channel {  
    ...  
}  
  
class UDPChannel implements Channel {  
    ...  
}
```

# Problema: Precisamos adicionar funcionalidades extras em canais

- Em muitos usos, os canais "default" (TCP ou UDP) não são suficientes
- Por exemplo, precisamos também de canais com:
  - Compactação e descompactação de dados
  - Buffers (ou caches) de dados
  - Logging dos dados enviados e recebidos
  - etc

# Possível solução: via herança

- Subclasses são criadas para essas novas funcionalidades
- `TCPZipChannel extends TCPChannel`
- `TCPBufferedChannel extends TCPChannel`
- `TCPBufferedZipChannel extends TCPZipChannel extends TCPChannel`
- `TCPLogChannel extends TCPChannel`
- `TCPLogBufferedZipChannel extends TCPBufferedZipChannel extends TCPZipChannel extends TCPChannel`

# Possível solução: via herança

- Subclasses são criadas para essas novas funcionalidades
- TCPZipChannel `extends` TCPChannel
- TCPBufferedChannel `extends` TCPChannel
- TCPBufferedZipChannel `extends` TCPZipChannel `extends` TCPChannel
- TCPLogChannel `extends` TCPChannel
- TCPLogBufferedZipChannel `extends` TCPBufferedZipChannel `extends`  
TCPZipChannel `extends` TCPChannel

# Possível solução: via herança

- Subclasses são criadas para essas novas funcionalidades
- `TCPZipChannel extends TCPChannel`
- `TCPBufferedChannel extends TCPChannel`
- `TCPBufferedZipChannel extends TCPZipChannel extends TCPChannel`
- `TCPLogChannel extends TCPChannel`
- `TCPLogBufferedZipChannel extends TCPBufferedZipChannel extends TCPZipChannel extends TCPChannel`

Problema: explosão no número de subclasses

# E precisamos de classes similares para UDP

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

## Solução: **Padrão Decorador**

- Resolve o nosso problema -- que é adicionar novas funcionalidades em uma classe -- por meio de composição e não mais por meio de herança



# Exemplo

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

# Exemplo

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

## Mais um exemplo

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

## Mais um exemplo

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

## Mais um exemplo

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

# Comparação

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



Dentro de uma caixa, tem outra caixa, que tem outra caixa ... até chegar no presente. Isto é, até chegar em TCPChannel ou UDPChannel

# Implementação do Padrão Decorador

# Implementação

- Uma classe base, chamada ChannelDecorator
- Uma subclasse para cada funcionalidade
  - Exemplos: ZipChannel, BufferedChannel, LogChannel



# Implementação

- Não são mais necessárias:
  - Subclasses para combinações de funcionalidades.  
Exemplo: TCPBufferedZipChannel
- Logo, não existe explosão no número de subclasses

# ChannelDecorator

```
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();  
    }  
  
}
```

Todos os decoradores são subclasses dessa classe única

# ChannelDecorator

```
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();  
    }  
}
```

# ChannelDecorator

```
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();  
    }  
  
}
```

# ChannelDecorator

```
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();  
    }  
}
```

Implementação: ZipChannel

# ZipChannel

```
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;  
    }  
  
}
```

# ZipChannel

```
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;  
    }  
  
}
```

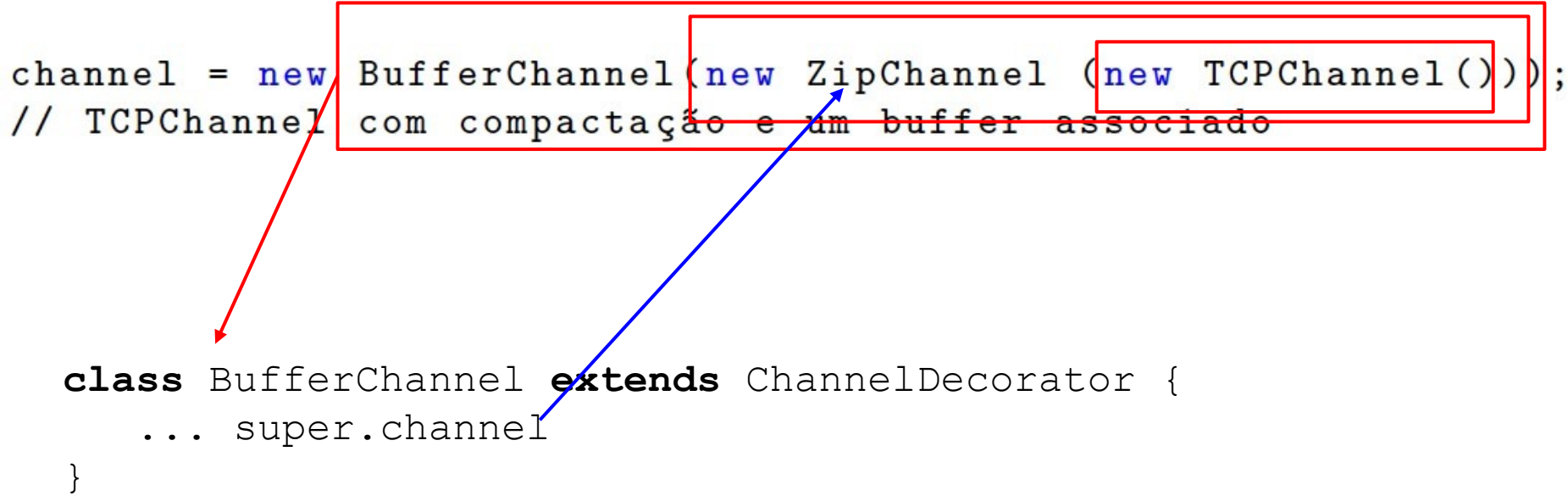


# ZipChannel

```
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;  
    }  
  
}
```

# Exemplo

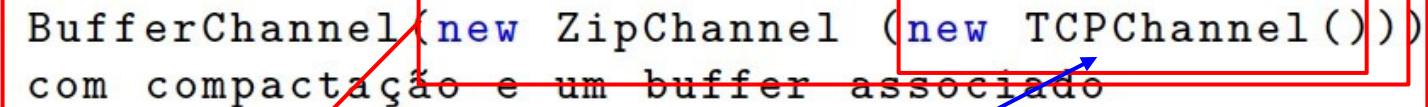
```
channel = new BufferChannel(new ZipChannel (new TCPChannel()));  
// TCPChannel com compactação e um buffer associado
```



```
class BufferChannel extends ChannelDecorator {  
    ... super.channel  
}
```

# Exemplo

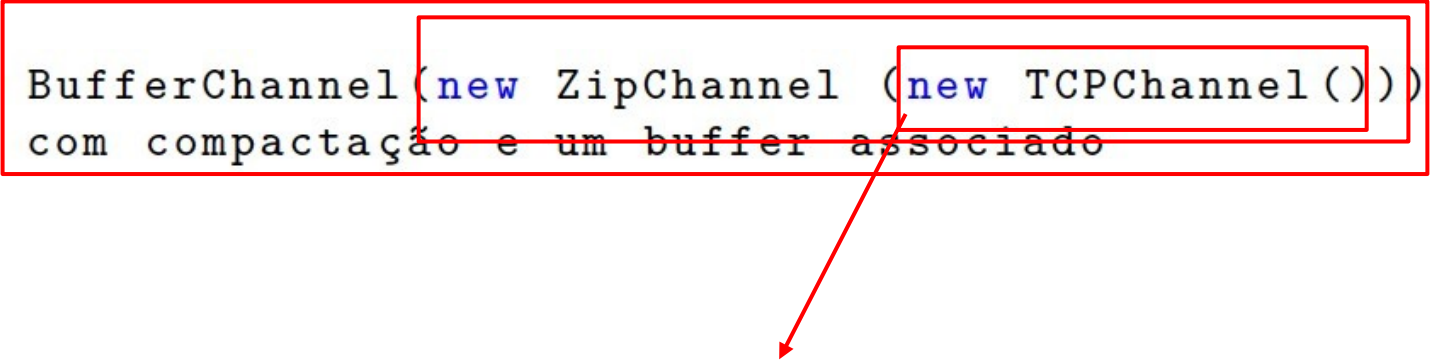
```
channel = new BufferChannel(new ZipChannel (new TCPChannel()));  
// TCPChannel com compactação e um buffer associado
```



```
class ZipChannel extends ChannelDecorator {  
    ... super.channel  
}
```

# Exemplo

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

A diagram consisting of three nested red rectangular boxes. The innermost box encloses the expression 'new TCPChannel()' in the code. The middle box encloses the entire argument '(new ZipChannel (new TCPChannel()))' of the 'new ZipChannel' constructor. The outermost box encloses the entire right-hand side of the assignment 'new BufferChannel(new ZipChannel (new TCPChannel()))'. A red arrow originates from the bottom of the outermost box and points towards the 'TCPChannel' class definition below.

```
class TCPChannel implements Channel {  
    // canal "final"  
}
```

## **(7) Strategy**

# Contexto: Biblioteca de Estruturas de Dados

```
class MyList {  
  
    ... // dados de uma lista  
    ... // métodos de uma lista: add, delete, search  
  
    public void sort() {  
        ... // ordena a lista usando Quicksort  
    }  
  
}
```

Problema: lembram do Princípio Aberto/Fechado?

# Problema

- Classe MyList **não** está aberta a extensões
- Possível extensão: mudar o algoritmo de ordenação
- Usar ShellSort, HeapSort, etc



## Solução: **Padrão Strategy**

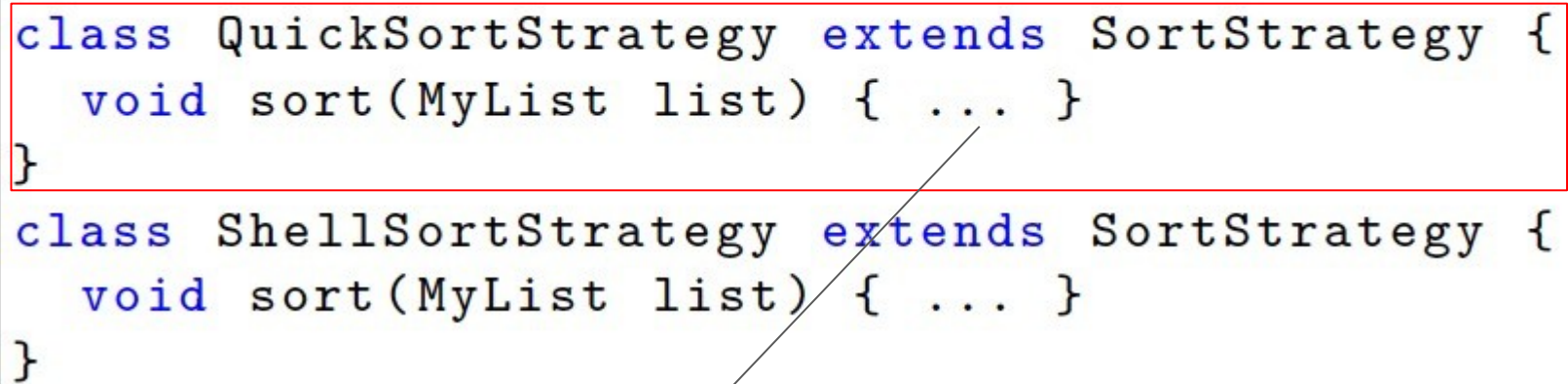
- Objetivo: parametrizar os algoritmos usados por uma classe; tornar uma classe "aberta" a novos algoritmos
- No nosso exemplo, a novos algoritmos de ordenação

Passo #1: Criar uma hierarquia de "estratégias"  
(estratégia = algoritmo)

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

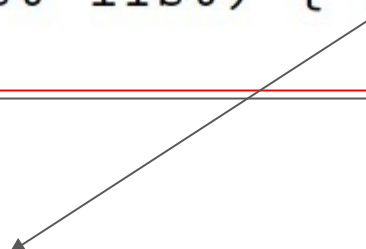
```
abstract class SortStrategy {  
    abstract void sort(MyList list);  
}  
  
class QuickSortStrategy extends SortStrategy {  
    void sort(MyList list) { ... }  
}  
  
class ShellSortStrategy extends SortStrategy {  
    void sort(MyList list) { ... }  
}
```

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



Código do QuickSort

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



Código do ShellSort

Passo #2: Modificar MyList para usar a hierarquia de estratégias

```
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);  
    }  
  
}
```



```
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);  
    }  
  
}
```

```
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);  
    }  
  
}
```

```
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);  
    }  
}
```

## **(8) Observador**

# Contexto: Sistema de uma Estação Meteorológica

- Duas classes desse sistema:
  - Temperatura
  - Termômetro
- Existem diversos tipos de Termômetro (digital, analógico, web, celular, console, etc)
- Se a temperatura mudar, os termômetros devem ser atualizados

# Problema

- Não queremos acoplar Temperatura a Termômetros
- Motivos:
  - Tornar classe de dados (modelo) independente de classes de visão (ou de interface com usuários)
  - Tornar flexível a adição de um novo tipo de termômetro no sistema

# Solução: **Padrão Observador**

- Implementa uma relação do tipo um-para-muitos entre objetos chamados de Sujeito e Observadores
  - Sujeito: Temperatura
  - Observadores: Termômetros
- Quando o estado de um Sujeito muda, seus Observadores são notificados.
- Mas Sujeito não conhece o tipo concreto de seus Observadores

# Programa Principal

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



Sujeito



# Programa Principal

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



Dois observadores

# Programa Principal

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



Notifica os termômetros

# Classe Temperatura

```
class Temperatura extends Subject {  
  
    private double temp;  
  
    public double getTemp() {  
        return temp;  
    }  
  
    public void setTemp(double temp) {  
        this.temp = temp;  
        notifyObservers();  
    }  
  
}
```

# Classe Temperatura

```
class Temperatura extends Subject {  
  
    private double temp;  
  
    public double getTemp() {  
        return temp;  
    }  
  
    public void setTemp(double temp) {  
        this.temp = temp;  
        notifyObservers();  
    }  
  
}
```

Classe que  
implementa  
addObservers e  
notifyObservers

# Classe Temperatura

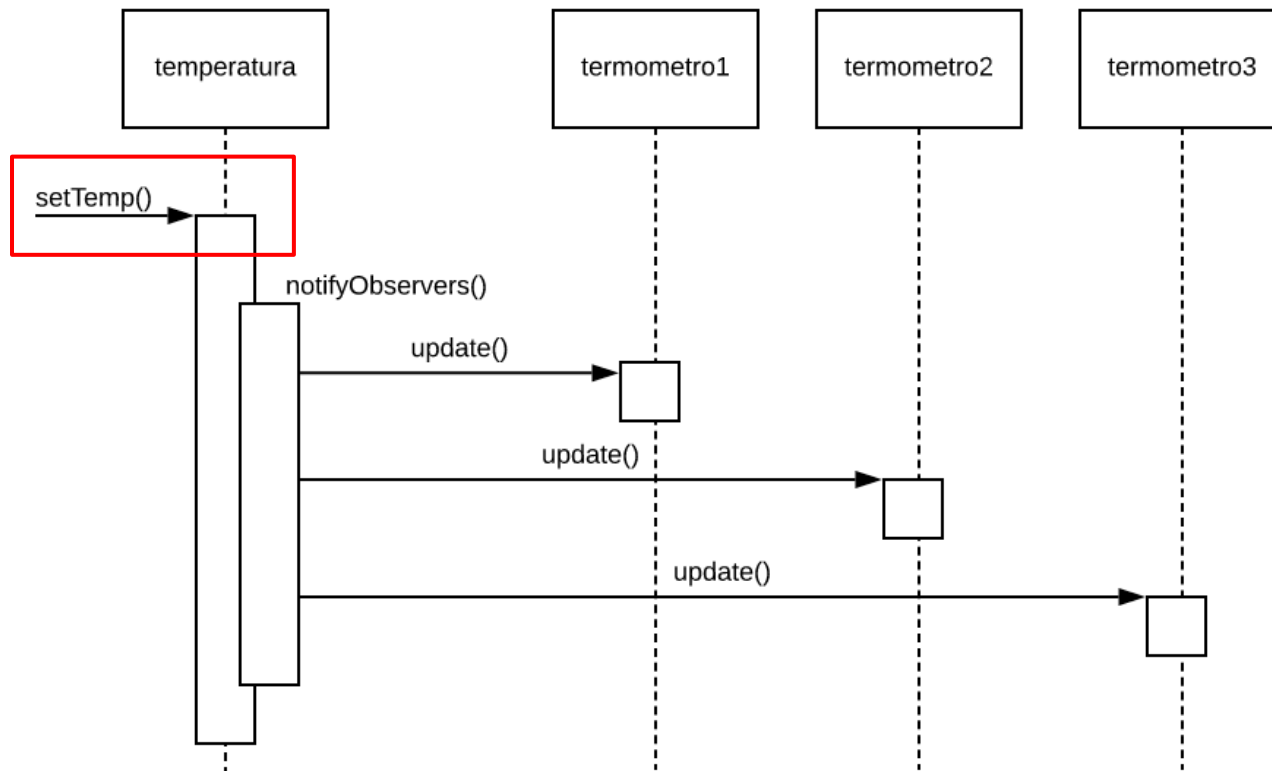
```
class Temperatura extends Subject {  
  
    private double temp;  
  
    public double getTemp() {  
        return temp;  
    }  
  
    public void setTemp(double temp) {  
        this.temp = temp;  
        notifyObservers();  
    }  
  
}
```

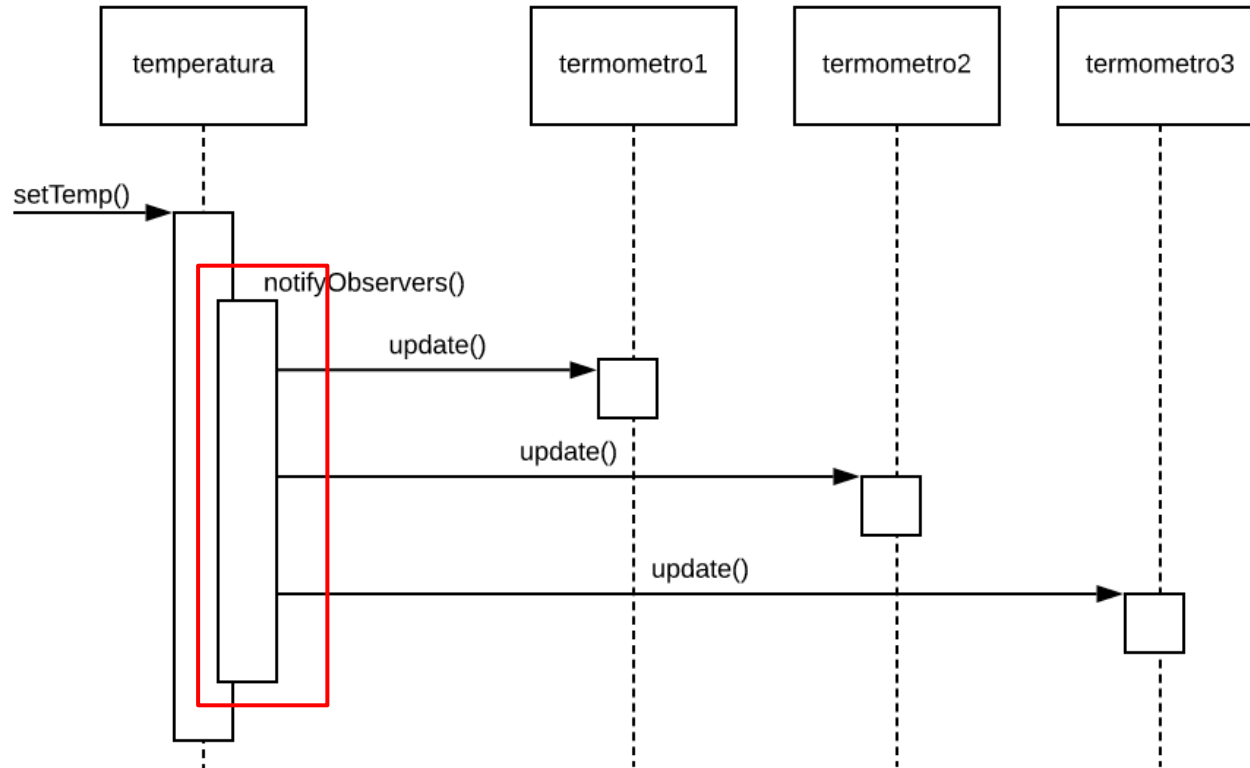
Notifica todos os observadores (isto é, Termômetros) que foram adicionados a uma Temperatura

# Uma classe Termômetro

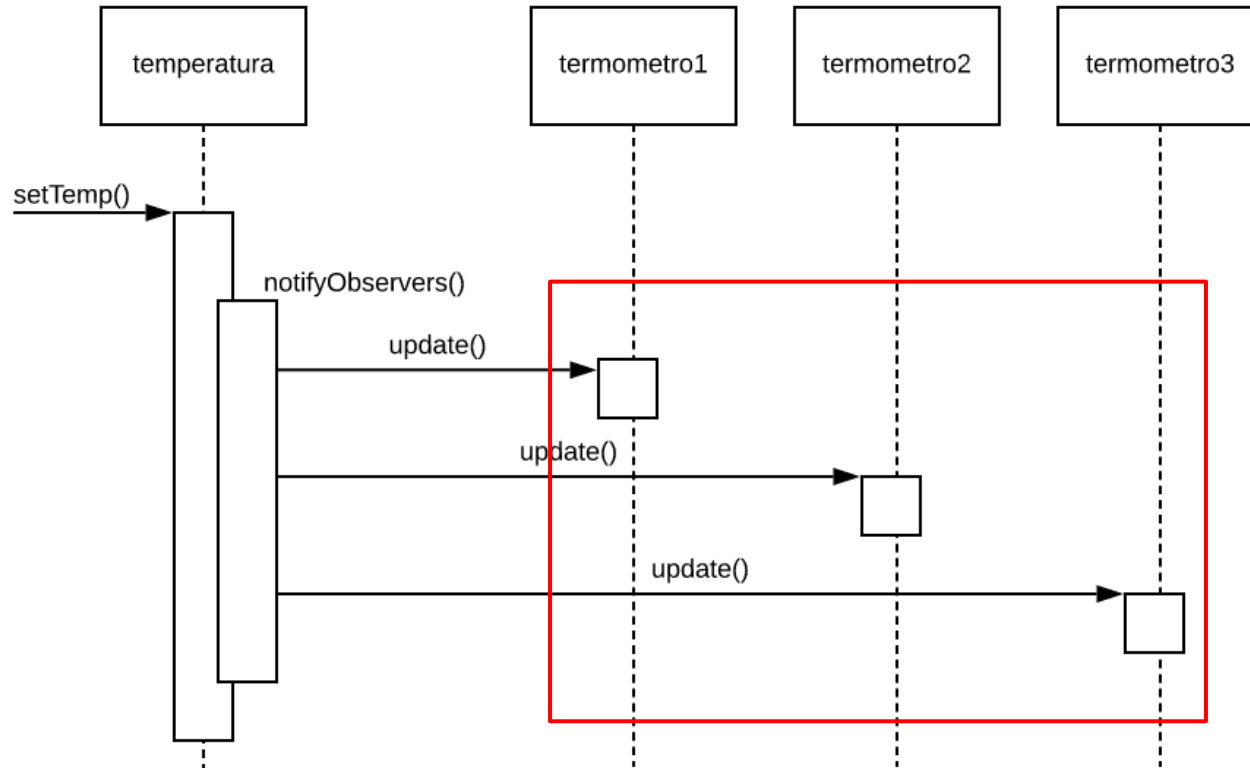
```
class TermometroCelsius implements Observer{  
  
    public void update(Subject s){  
        double temp = ((Temperatura) s).getTemp();  
        System.out.println("Temperatura Celsius: " + temp);  
    }  
  
}
```

Todos Observadores devem implementar esse método.  
Chamada de **notifyObservers** (em Temperatura) resulta na execução de **update** de cada um de seus observadores









## **(9) Template Method**

# Contexto: Folha de Pagamento

- Uma classe base: Funcionario
- Duas subclasses: FuncionarioPublico e FuncionarioCLT

# Problema

- Função que calcula salário de funcionários:
  - Passos são semelhantes para func. públicos e CLT
  - Porém, existem alguns detalhes diferentes
- Na classe pai (Funcionario) queremos definir o "workflow principal" (ou o template) para cálculo de salários
- E deixar aberto para as subclasses  
os refinamentos desses passos

# Solução: Padrão **Template Method**

- Especifica como implementar o esqueleto de um algoritmo em uma classe abstrata X
- Mas deixando pendente alguns passos (ou métodos abstratos) para serem implementados nas subclasses
- Ou seja, esse padrão permite que subclasses customizem um algoritmo, mas sem mudar sua estrutura

```
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;
```

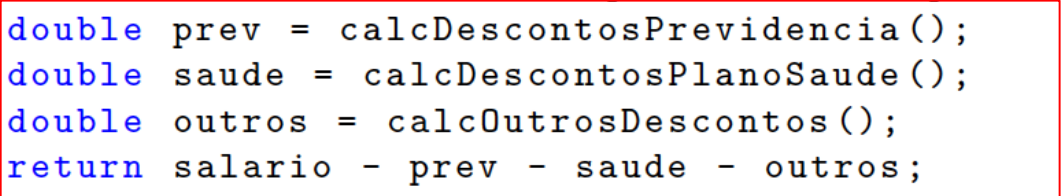
```
}
```

```
}
```

Serão  
implementados  
pelas  
subclasses

```
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;  
    }  
}
```

```
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;  
    }  
}
```



Passos principais (ou template, ou modelo)  
para cálculo de salário líquido



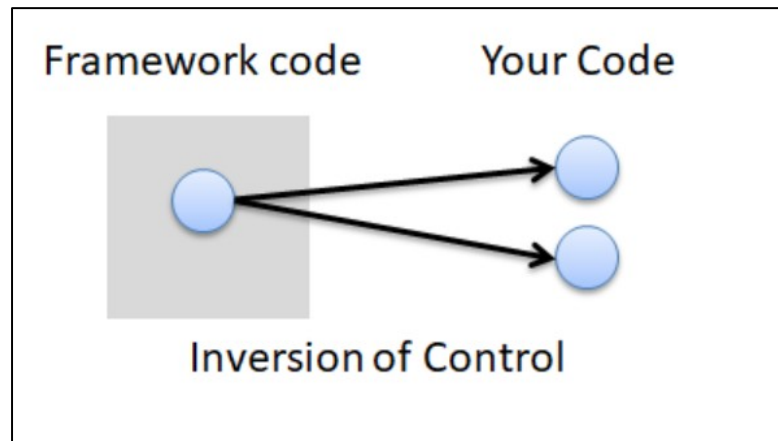
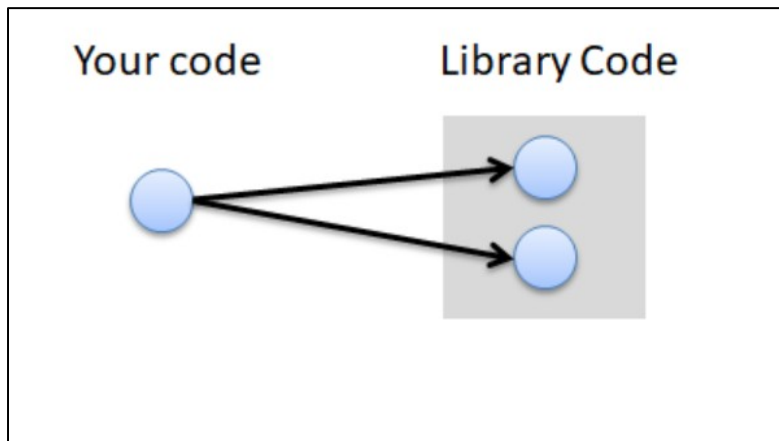
# Inversão de Controle

- Template Method é usado para implementar **Inversão de Controle**, principalmente em frameworks
- Framework: define o "modelo" de um algoritmo/sistema
  - Mas clientes podem parametrizar alguns passos
  - Framework que chama esse código dos clientes
  - Daí o termo inversão de controle

# Frameworks vs Bibliotecas

- Sistema X usa um sistema Y
- Y é uma biblioteca ou framework?
  - Se X tem total controle das chamadas, define a ordem delas, etc, então Y é uma **biblioteca**.
  - Mas se é Y que tem o controle principal (por exemplo, ele implementa o método main), então Y é um **framework**.

# Frameworks vs Bibliotecas



Fonte: <https://github.com/prmr/SoftwareDesign/blob/master/modules/Module-06.md>

**(10) Visitor**

## Contexto: Veiculo e subclasses

- Suponha um sistema com uma classe `Veiculo`
- E três subclasses: `Carro`, `Onibus` e `Bicicleta`
- E uma lista polimórfica de objetos `Veiculo`:

```
List<Veiculo> list = new ArrayList<Veiculo>();  
list.add(new Carro(..));  
list.add(new Onibus(..));  
...
```

# Problema

- Frequentemente, temos que realizar uma operação com todos os veículos armazenados na lista do slide anterior
- Exemplo:
  - Imprimir dados de veículos (sejam `Carro`, `Onibus`,...)
  - Salvar dados de veículos em disco
  - Enviar uma msg para os donos dos veículos
  - etc

# Problema

- Queremos seguir o princípio Aberto/Fechado:
  - Manter `Veiculo` e suas subclasses fechados para mudanças, mas aberto para extensões
  - Extensões: diversas operações que queremos realizar com veículos

# Possível alternativa

```
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 moto"}  
}
```



## Possível alternativa (cont.)

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

## Possível alternativa (cont.)

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

- Em Java e linguagens similares, compilador não conhece o tipo dinâmico do parâmetro `veiculo`
- Logo, ele não sabe "identificar" qual dos três métodos `visit` de `PrintVisitor` deverá ser chamado

## Solução: **Padrão Visitor**

- Permite adicionar uma operação genérica em uma família de classes, sem mexer no código delas
- Permite simular *multiple dispatching*, em linguagens que oferecem apenas *single dispatching*

```
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
```

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

Chama accept do tipo dinâmico de `veiculo`. Suponha que seja `Carro`

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

Tipo de `this` é conhecido estaticamente (`Carro`)

Visitor: vantagens & desvantagens

# Visitor: Vantagens

- Visitors facilitam a adição de um método em uma hierarquia de classes
- Pode existir um segundo Visitor, com outras operações
  - Exemplo: calcular imposto de veículos de MG

# Visitor: Desvantagens

- A adição de uma nova classe na hierarquia (exemplo: `Caminhao`), implica que todos os Visitors (concretos) terão que ser atualizados, com um novo método:  
`visit(Caminhao)`



# Visitor: Desvantagens

- Visitors podem quebrar encapsulamento
  - Por exemplo, `Veiculo` pode ter que implementar métodos públicos expondo seu estado interno para que os Visitors tenham acesso a ele

**Quando não vale a pena usar  
padrões de projeto?**

# Paternite

- Termo usado por John Ousterhout: inflamação causada pelo uso precipitado de padrões de projeto
- Padrões de Projeto  $\Leftrightarrow$  Design for change
- Design for change: projeto acomoda mudanças futuras
- Se a chance de tais mudanças é zero, não precisamos de padrões de projeto

# Trade-off: Design for Change vs Complexidade

- Não existe almoço grátis! Tudo tem um um preço ...
- Para viabilizar "design for change", Design Patterns "complicam" um pouco o projeto
- Por exemplo, demandam a criação de classes extras

Exemplo: Strategy

# Solução sem padrão de projeto

```
class MyList {  
  
    ... // dados de uma lista  
    ... // métodos de uma lista: add, delete, search  
  
    public void sort() {  
        ... // ordena a lista usando Quicksort  
    }  
  
}
```

1 classe

## Solução sem padrão de projeto

```
class MyList {  
  
    ... // dados de uma lista  
    ... // métodos de uma lista: add, delete, search  
  
    public void sort() {  
        ... // ordena a lista usando Quicksort  
    }  
  
}
```

1 classe

## Solução usando Strategy

```
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);  
    }  
  
}
```

1 classe (com mais código)  
+1 classe abstrata  
+2 classes de estratégias

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

Paternite: um (provável) exemplo



# Paternite: um (provável) exemplo

org.springframework.aop.framework

**Class AbstractSingletonProxyFactoryBean**

[java.lang.Object](#)

└ [org.springframework.aop.framework.ProxyConfig](#)

└ [org.springframework.aop.framework.AbstractSingletonProxyFactoryBean](#)

All Implemented Interfaces:

[Serializable](#), [BeanClassLoaderAware](#), [FactoryBean](#), [InitializingBean](#)

Direct Known Subclasses:

[TransactionProxyFactoryBean](#)

```
public abstract class AbstractSingletonProxyFactoryBean
extends ProxyConfig
implements FactoryBean, BeanClassLoaderAware, InitializingBean
```

Convenient proxy factory bean superclass for proxy factory beans that create only singletons.

Manages pre- and post-interceptors (references, rather than interceptor names, as in [ProxyFactoryBean](#)) and provides consistent interface management.

**Fim**