ACH 2028
Qualidade de Software

Aula 14 - Padrões de projeto (GoF)

Prof. Marcelo Medeiros Eler
marceloeler@usp.br

Escola de Artes, Ciências e Humanidades - Universidade de São Paulo

Projeto (Design) de Software

Tipicamente, esta etapa envolve a definição dos seguintes artefatos ou abstrações:

- Modelos da arquitetura do software
- Modelos de componentes, classes e algoritmos
- Modelos de dados
- Modelos e protótipos de interfaces

Projeto (Design) de Software

Tipicamente, esta etapa envolve a definição dos seguintes artefatos ou abstrações:

- Modelos da arquitetura do software
- **Modelos de componentes, classes e algoritmos**
- Modelos de dados [disciplina de BD]
- Modelos e protótipos de interfaces [disciplina de IHC]

Projeto de Componentes/Classes/Algoritmos

O projeto de componentes, classes e algoritmos especificam a estrutura e o comportamento do software em um nível mais detalhado do que a arquitetura do software

O projeto de software pode definir:

- Responsabilidades (quem faz o quê)
- Estruturas (elementos que constituem o software de acordo com a decomposição escolhida e como eles se relacionam)
- Comportamento

Padrões

Na Engenharia de Software, os padrões chamaram a atenção de desenvolvedores em 1987, quando Kent Beck e Ward Cunningham propuseram os primeiros padrões de projeto para a área.

Mas os padrões ficaram realmente populares na área quando o livro **“Design Patterns: Elements of Reusable Object-Oriented Software”** foi publicado em 1995, por Erich Gamma, Richard Helm, Ralph Johnson e John Vlissides. Esses quatro são conhecidos como a "Gangue dos Quatro" (Gang of Four) ou simplesmente "GoF".

Padrões

Padrões em ES permitem que desenvolvedores possam recorrer a soluções já existentes para solucionar problemas que normalmente ocorrem em desenvolvimento de software.

Padrões capturam experiência existente e comprovada em desenvolvimento de software, ajudando a promover boa prática de projeto.

Padrões

Vantagens:

- Padrões reduzem a complexidade da solução
- Padrões promovem o reuso
- Padrões facilitam a geração de alternativas
- Padrões facilitam a comunicação

Padrões

Categorias:

- Padrões Arquiteturais: expressam um esquema de organização estrutural fundamental para sistemas de software
- Padrões de Projeto: disponibilizam um esquema para refinamento de subsistemas ou componentes de um sistema de software (GAMMA et al., 1995)

Padrões GoF

Livro “Design Patterns: Elements of Reusable Object-Oriented Software”

Exploram soluções mais específicas de implementação

Categorias:

- Padrões de Criação
- Padrões Estruturais
- Padrões Comportamentais

Padrões GoF

Formato dos padrões GoF

- **Nome** (inclui número da página): um bom nome é essencial para que o padrão caia na boca do povo
- **Objetivo / Intenção / Motivação**: um cenário mostrando o problema e a necessidade da solução
- **Aplicabilidade**: como reconhecer as situações nas quais o padrão é aplicável
- **Estrutura**: uma representação gráfica da estrutura de classes do padrão
- **Participantes**: as classes e objetos que participam e quais são suas responsabilidades
- **Colaborações**: como os participantes colaboram para exercer as suas responsabilidades

Padrões GoF

Formato dos padrões GoF

- **Consequências**: vantagens e desvantagens, trade-offs
- **Implementação**: com quais detalhes devemos nos preocupar quando implementamos o padrão aspectos específicos de cada linguagem
- **Exemplo de Código**: no caso do GoF, em C++ (a maioria) ou Smalltalk
- **Usos Conhecidos**: exemplos de sistemas reais de domínios diferentes onde o padrão é utilizado
- **Padrões Relacionados**: quais outros padrões devem ser usados em conjunto com esse, quais padrões são similares a este e quais são as diferenças

Padrões de criação

Factory Method

Abstract Factory

Builder

Prototype

Singleton

Padrões estruturais

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

Padrões GoF

		Finalidade		
		De Criação	**Estrutural**	**Comportamental**
Escopo	**Classe**	Factory Methody	Adapter(class)	Interpreter Template Method
	Objeto	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Façade Flyweight Proxy	Chain of Responsibility Command Interator Mediator Memento Observer State Strategy Visitor

Padrões GoF

Existem outras propostas de classificação

Objetivo				
Interface	**Responsabilidade**	**Construção**	**Operações**	**Extensões**
Adapter	Singleton	Builder	Template Method	Decorator
Façade	Observer	Factory Method	State	Iterator
Composite	Mediator	Abstract Factory	Strategy	Visitor
Bridge	Proxy	Prototype	Command	
	Chain of Responsibility	Memento	Interpreter	
	Flyweight			

Padrões de criação

Factory Method

Abstract Factory

Builder

Prototype

Singleton

Padrões de criação

Factory Method

Abstract Factory

Builder

Prototype

Singleton

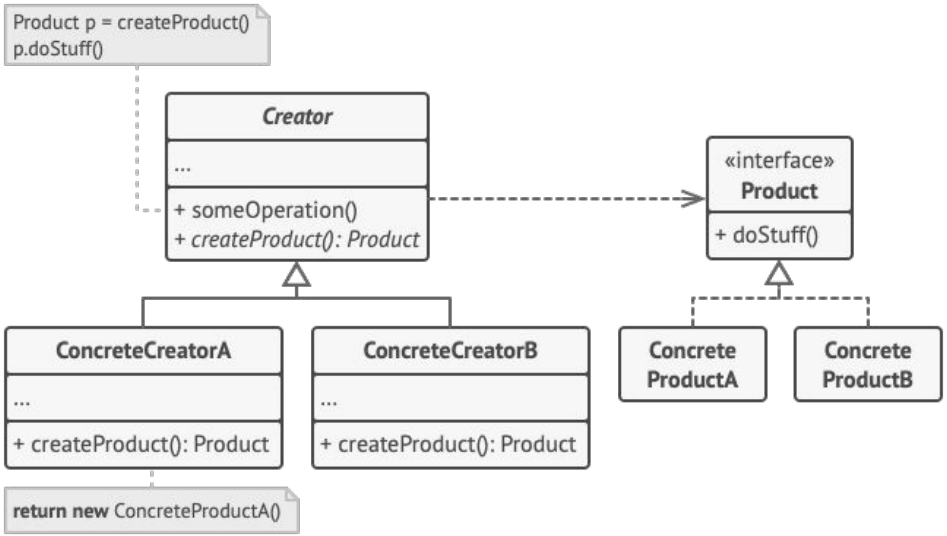
Factory Method

O Factory Method é um padrão de criação que fornece uma interface para criar objetos em uma superclasse, mas permite que as subclasses alterem o tipo de objetos que serão criados.

https://refactoring.guru/pt-br/design-patterns/factory-method

Factory Method

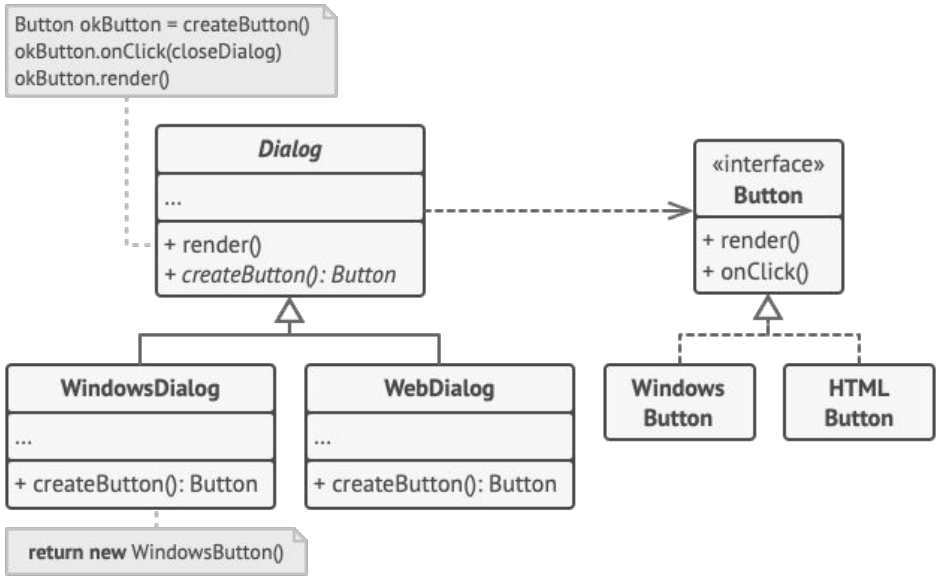
Estrutura geral



https://refactoring.guru/pt-br/design-patterns/factory-method

Factory Method

Exemplo com janelas de diálogos

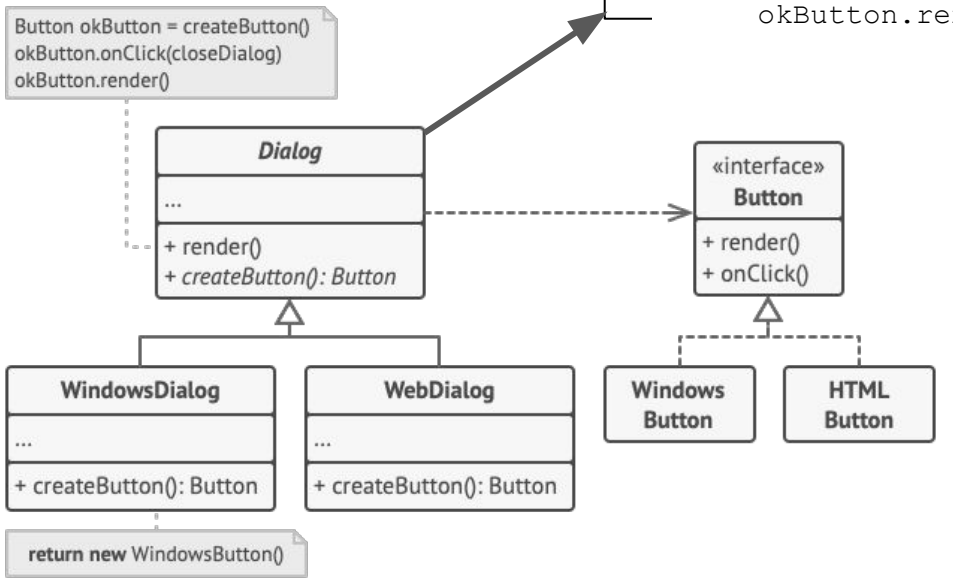


https://refactoring.guru/pt-br/design-patterns/factory-method

Factory Method

Exemplo com janelas de diálo

```
class Dialog is
    abstract method createButton():Button
    method render() is
        // Chame o método fábrica para criar um objeto
produto.
        Button okButton = createButton()
        // Agora use o produto.
        okButton.onClick(closeDialog)
        okButton.render()
```

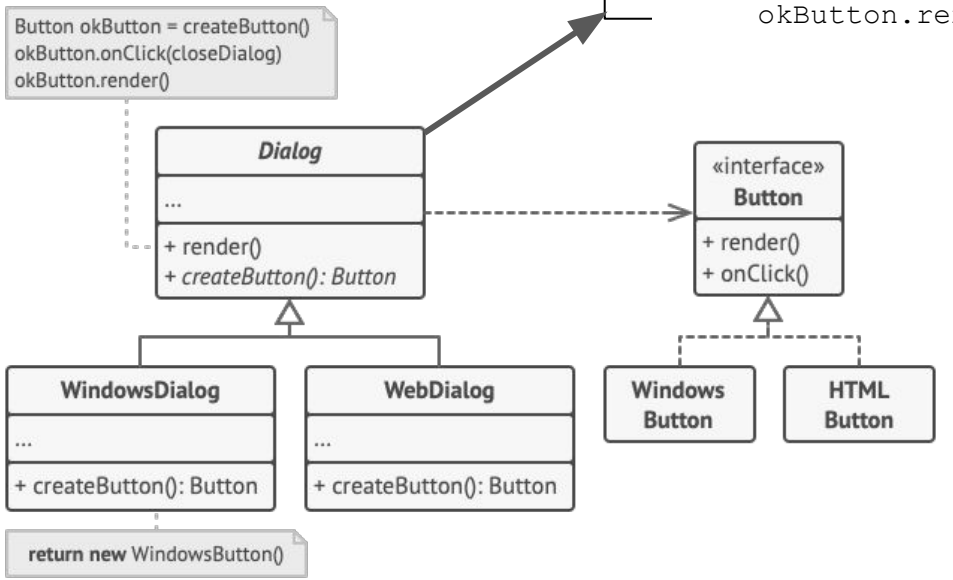


https://refactoring.guru/pt-br/design-patterns/factory-method

Factory Method

Exemplo com janelas de diálo

```
class Dialog is
    abstract method createButton():Button
    method render() is
        // Chame o método fábrica para criar um objeto
produto.
        Button okButton = createButton()
        // Agora use o produto.
        okButton.onClick(closeDialog)
        okButton.render()
```

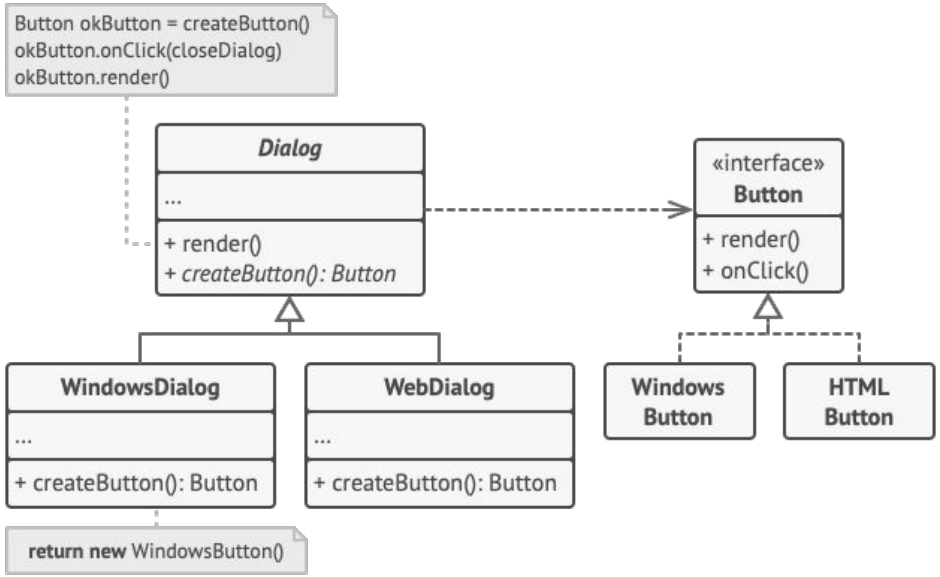



```
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()
```

https://refactoring.guru/pt-br/design-patterns/factory-method

```
class Dialog is
    abstract method createButton():Button
    method render() is
        // Chame o método fábrica para criar um objeto
produto.
        Button okButton = createButton()
        // Agora use o produto.
        okButton.onClick(closeDialog)
        okButton.render()
```

```
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()
```




```
interface Button is
    method render()
    method onClick(f)

class WindowsButton implements Button is
    method render(a, b) is
        // Renderiza um botão Windows.
    method onClick(f) is
        // Vincula um evento clique do SO

class HTMLButton implements Button is
    method render(a, b) is
        // Retorna uma representação HTML
    method onClick(f) is
        // Vincula um evento de clique web.
```

https://refactoring.guru/pt-br/design-patterns/factory-method

```
class Dialog is
    abstract method createButton():Button
    method render() is
        // Chame o método fábrica para criar um objeto
produto.
        Button okButton = createButton()
        // Agora use o produto.
        okButton.onClick(closeDialog)
        okButton.render()
```

```
class WindowsDialog extends Dialog is
    method createButton():Button is
        return new WindowsButton()

class WebDialog extends Dialog is
    method createButton():Button is
        return new HTMLButton()
```

```
class Application is
    field dialog: Dialog

    method initialize() is
        config = readApplicationConfigFile()
        if (config.OS == "Windows") then
            dialog = new WindowsDialog()
        else if (config.OS == "Web") then
            dialog = new WebDialog()
        else
            throw new Exception("Unknown OS!")

    method main() is
        this.initialize()
        dialog.render()
```

```
interface Button is
    method render()
    method onClick(f)

class WindowsButton implements Button is
    method render(a, b) is
        // Renderiza um botão Windows.
    method onClick(f) is
        // Vincula um evento clique do SO

class HTMLButton implements Button is
    method render(a, b) is
        // Retorna uma representação HTML
    method onClick(f) is
        // Vincula um evento de clique web.
```

https://refactoring.guru/pt-br/design-patterns/factory-method

Padrões de criação

Factory Method

Abstract Factory

Builder

Prototype

Singleton

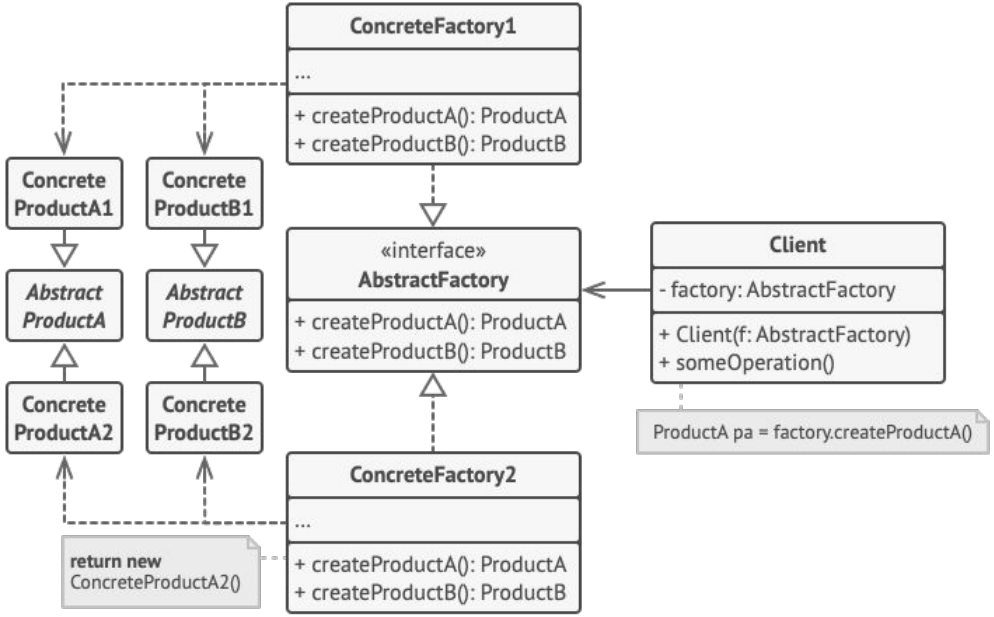
Abstract Factory (Fábrica abstrata)

Este padrão permite a criação de famílias de objetos relacionados ou dependentes por meio de uma única interface e sem que a classe concreta seja especificada.

https://refactoring.guru/pt-br/design-patterns/abstract-factory

Abstract Factory (Fábrica abstrata)

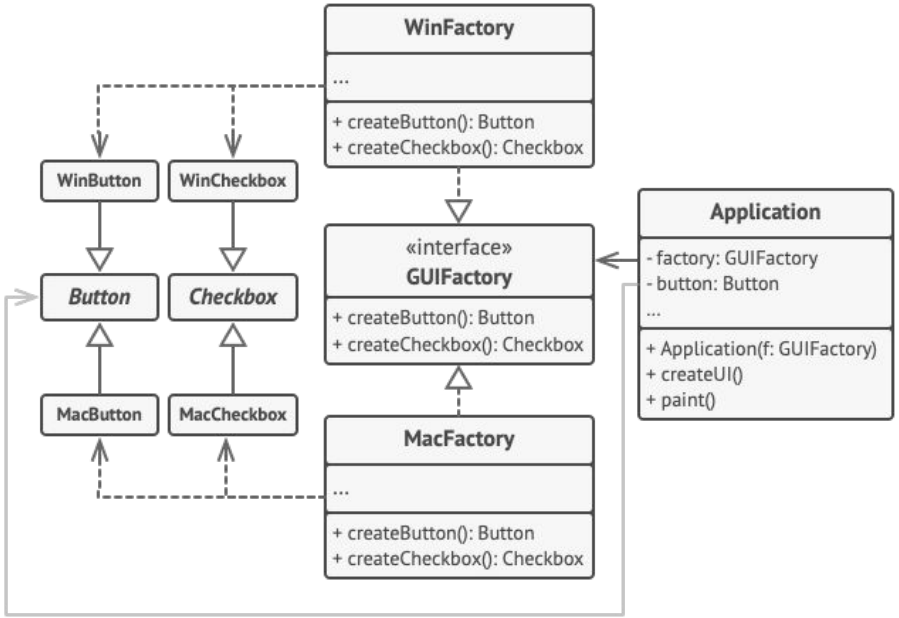
Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/abstract-factory

Abstract Factory (Fábrica abstrat

Exemplo:



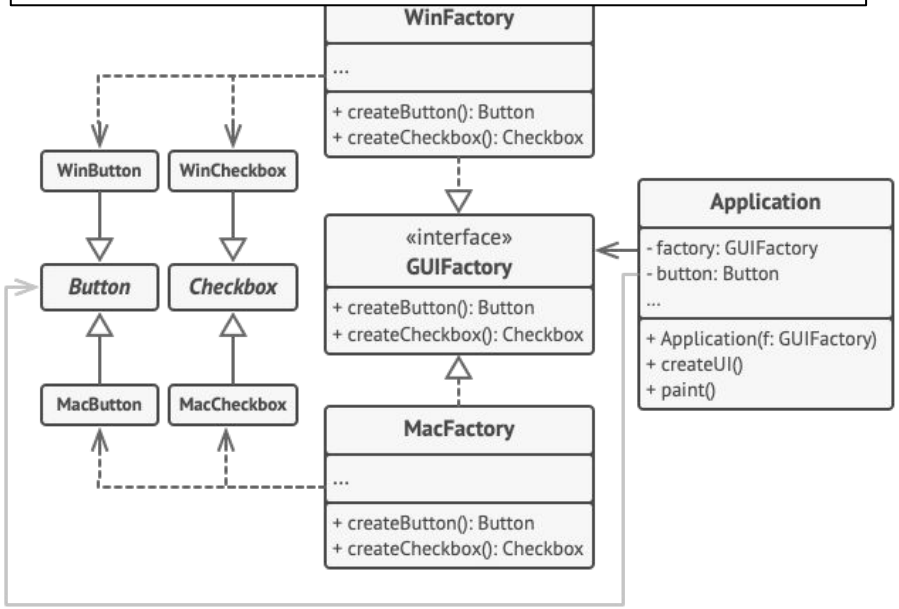
https://refactoring.guru/pt-br/design-patterns/abstract-factory

```
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox

class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()
```

```
interface Button is
    method paint()

class WinButton implements Button is
    method paint() is
        // Renderiza um botão no estilo Windows.

class MacButton implements Button is
    method paint() is
        // Renderiza um botão no estilo macOS.
```

strat

```
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI() is
        this.button = factory.createButton()
    method paint() is
        button.paint()
```



https://refactoring.guru/pt-br/design-patterns/abstract-factory

```
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox

class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()
```

```
interface Button is
    method paint()

class WinButton implements Button is
    method paint() is
        // Renderiza um botão no estilo Windows.

class MacButton implements Button is
    method paint() is
        // Renderiza um botão no estilo macOS.
```

```
class Application is
    private field factory: GUIFactory
    private field button: Button
    constructor Application(factory: GUIFactory) is
        this.factory = factory
    method createUI() is
        this.button = factory.createButton()
    method paint() is
        button.paint()
```


```
// A aplicação seleciona o tipo de fábrica dependendo
// da configuração do ambiente e cria o widget no tempo
// de execução (geralmente no estágio de inicialização).
class ApplicationConfigurator is
    method main() is
        config = readApplicationConfigFile()

        if (config.OS == "Windows") then
            factory = new WinFactory()
        else if (config.OS == "Mac") then
            factory = new MacFactory()
        else
            throw new Exception("Unknown OS!")

        Application app = new Application(factory)
```

https://refactoring.guru/pt-br/design-patterns/abstract-factory

```
interface GUIFactory is
    method createButton():Button
    method createCheckbox():Checkbox

class WinFactory implements GUIFactory is
    method createButton():Button is
        return new WinButton()
    method createCheckbox():Checkbox is
        return new WinCheckbox()

class MacFactory implements GUIFactory is
    method createButton():Button is
        return new MacButton()
    method createCheckbox():Checkbox is
        return new MacCheckbox()
```

```
interface Button is
    method paint()

class WinButton implements Button is
    method paint() is
        // Renderiza um botão no estilo Windows.

class MacButton implements Button is
    method paint() is
        // Renderiza um botão no estilo macOS.
```

Padrões de criação

Factory Method

Abstract Factory

Builder

Prototype

Singleton

Builder

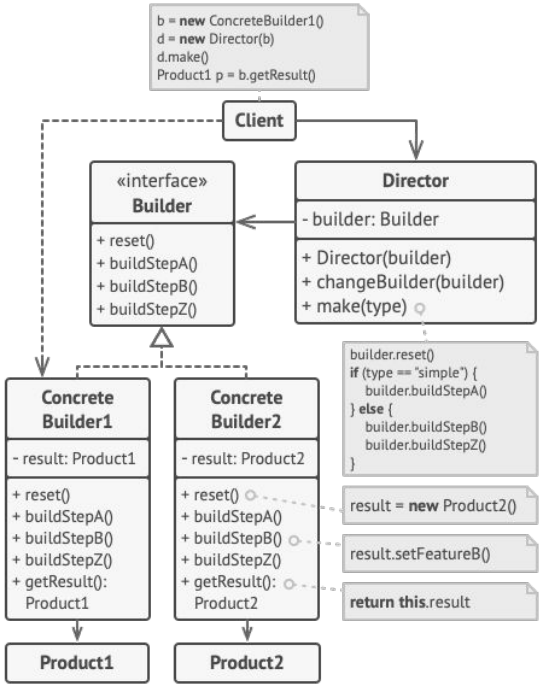
O Builder é um padrão de criação que permite a você construir objetos complexos passo a passo.

O padrão permite que se produza diferentes tipos e representações de um objeto usando o mesmo código de construção.

https://refactoring.guru/pt-br/design-patterns/builder

Builder

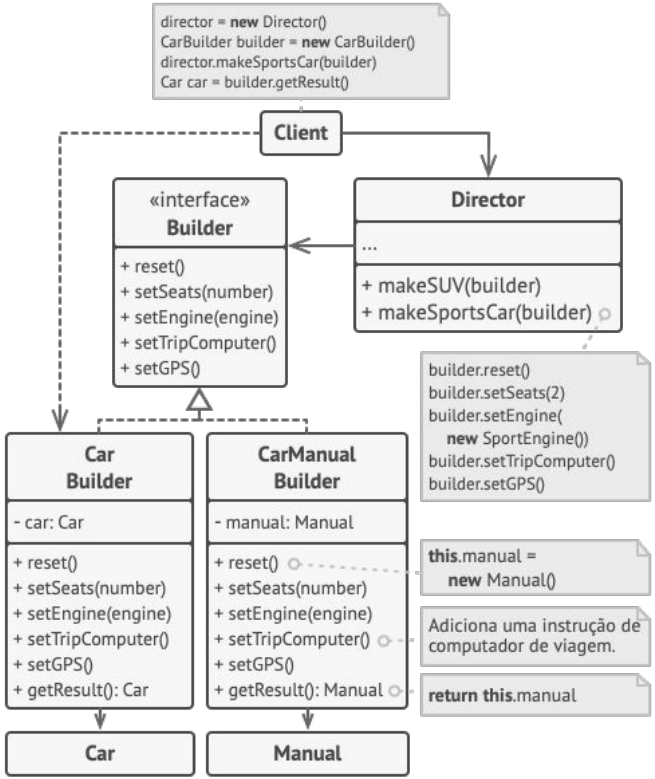
Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/builder

Builder

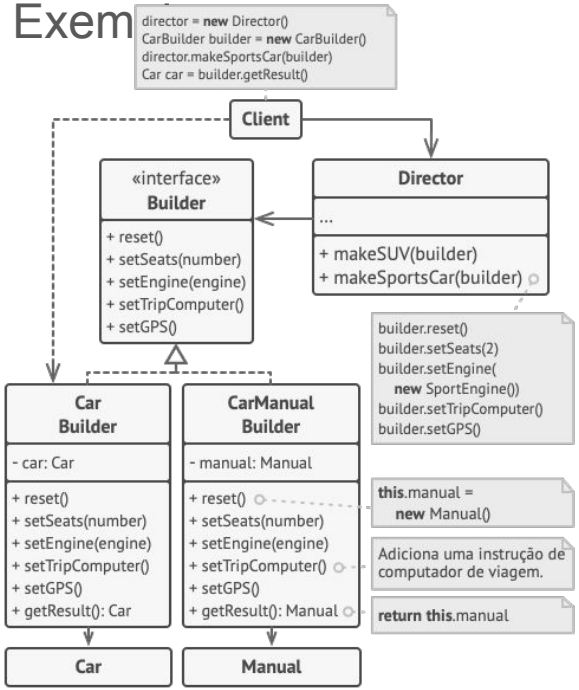
Exemplo:



https://refactoring.guru/pt-br/design-patterns/builder

Builder

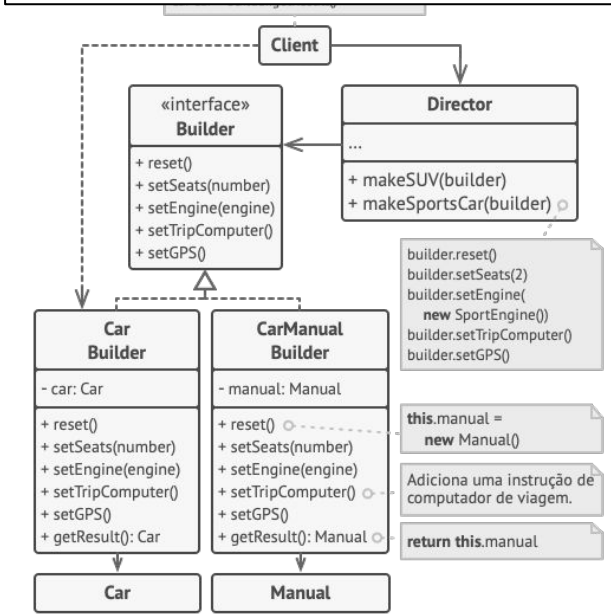
Exem



https://refactoring.guru/pt-br/design-patterns/builder

```
interface Builder is
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
    method setGPS(...)

class CarBuilder implements Builder is
    private field car:Car

    constructor CarBuilder() is
        this.reset()
    method reset() is
        this.car = new Car()
    method setSeats(...) is
        // Define núm. de assentos no carro.
    method setEngine(...) is
        // Instala um tipo de motor.
    method setTripComputer(...) is
        // Instala computador de bordo.
    method setGPS(...) is
          // Instala um GPS
    method getProduct():Car is
          product = this.car
          this.reset()
          return product
```

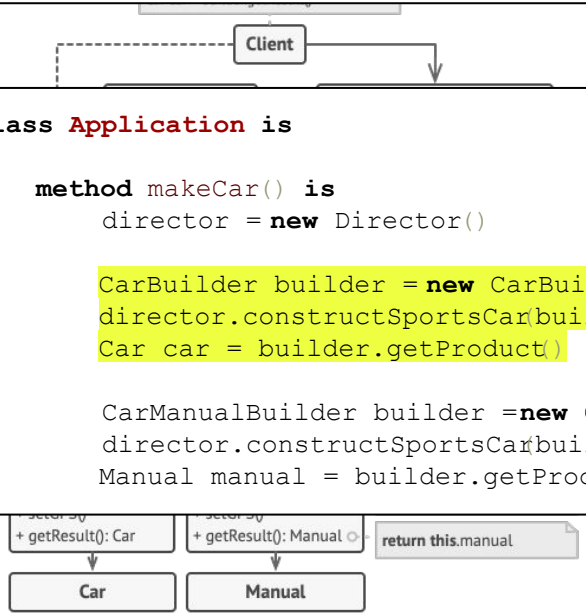
```
class Director is

    method constructSportsCar(builder: Builder) is
        builder.reset()
        builder.setSeats(2)
        builder.setEngine(new SportEngine())
        builder.setTripComputer(true)
        builder.setGPS(true)

    method constructSUV(builder: Builder) is
        // ...
```




```
interface Builder is
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
      method setGPS(...)

class CarBuilder implements Builder is
    private field car:Car

    constructor CarBuilder() is
        this.reset()
    method reset() is
        this.car = new Car()
    method setSeats(...) is
        // Define núm. de assentos no carro.
    method setEngine(...) is
        // Instala um tipo de motor.
    method setTripComputer(...) is
        // Instala computador de bordo.
    method setGPS(...) is
          // Instala um GPS
    method getProduct():Car is
          product = this.car
          this.reset()
          return product
```

```
class Director is

    method constructSportsCar(builder: Builder) is
        builder.reset()
        builder.setSeats(2)
        builder.setEngine(new SportEngine())
        builder.setTripComputer(true)
        builder.setGPS(true)

    method constructSUV(builder: Builder) is
        // ...
```

```
class Application is

    method makeCar() is
        director = new Director()

        CarBuilder builder = new CarBuilder()
        director.constructSportsCar(builder)
        Car car = builder.getProduct()

        CarManualBuilder builder =new CarManualBuilder()
        director.constructSportsCar(builder)
        Manual manual = builder.getProduct()
```




```
interface Builder is
    method reset()
    method setSeats(...)
    method setEngine(...)
    method setTripComputer(...)
      method setGPS(...)

class CarBuilder implements Builder is
    private field car:Car

    constructor CarBuilder() is
        this.reset()
    method reset() is
        this.car = new Car()
    method setSeats(...) is
        // Define núm. de assentos no carro.
    method setEngine(...) is
        // Instala um tipo de motor.
    method setTripComputer(...) is
        // Instala computador de bordo.
    method setGPS(...) is
          // Instala um GPS
    method getProduct():Car is
          product = this.car
          this.reset()
          return product
```

Padrões de criação

Factory Method

Abstract Factory

Builder

Prototype

Singleton

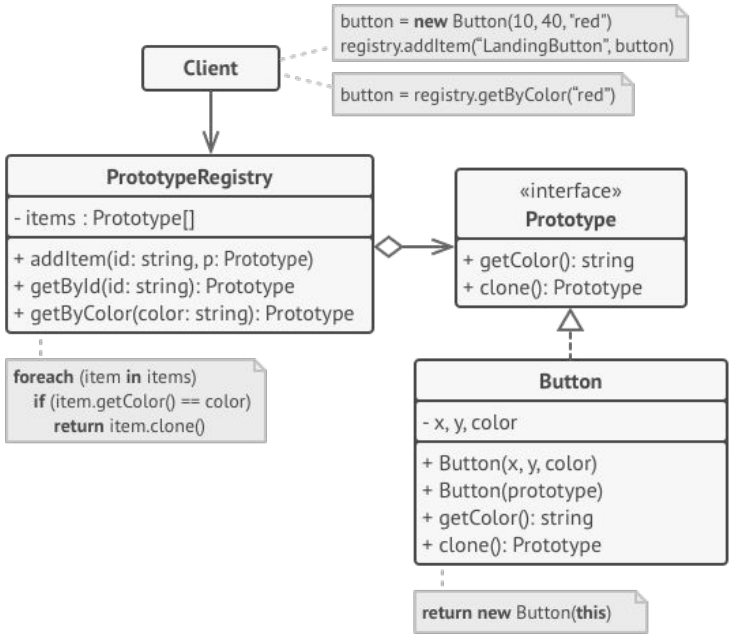
Prototype

O Prototype é um padrão de criação que permite copiar objetos existentes sem fazer seu código ficar dependente de suas classes.

https://refactoring.guru/pt-br/design-patterns/prototype

Prototype

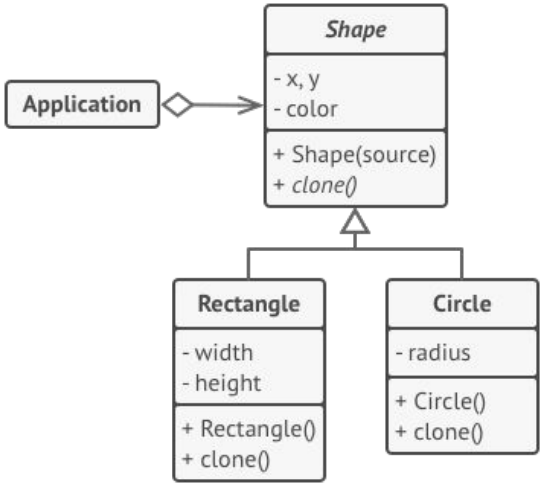
Exemplo 1:



https://refactoring.guru/pt-br/design-patterns/prototype

Prototype

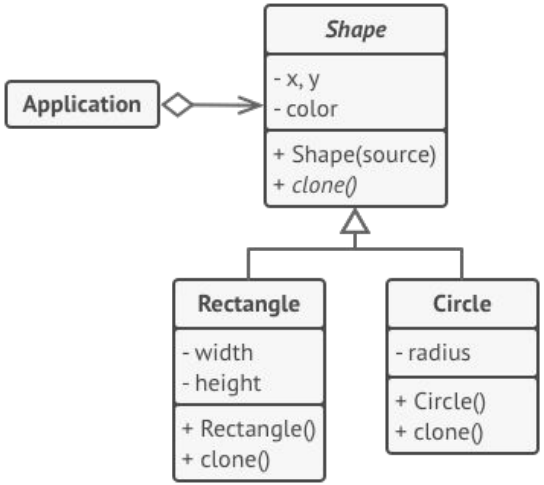
Exemplo 2:



https://refactoring.guru/pt-br/design-patterns/prototype

Prototype

Exemplo 2:

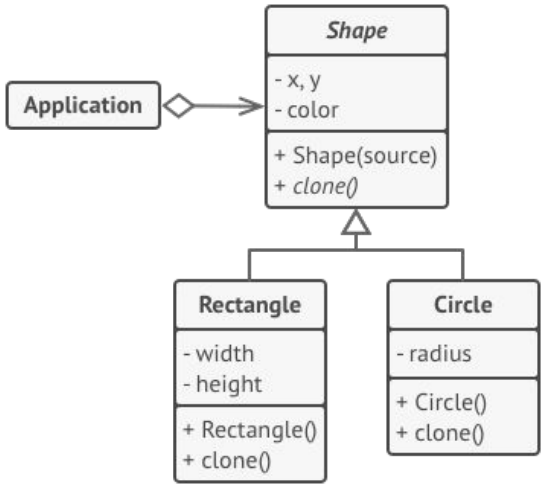



```
// Protótipo base.
abstract class Shape is
    field X: int
    field Y: int
    field color: string

    constructor Shape() is
        // ...

    constructor Shape(source: Shape) is
        this()
        this.X = source.X
        this.Y = source.Y
        this.color = source.color

    abstract method clone():Shape
```

https://refactoring.guru/pt-br/design-patterns/prototype

Prototype

Exemplo 2:



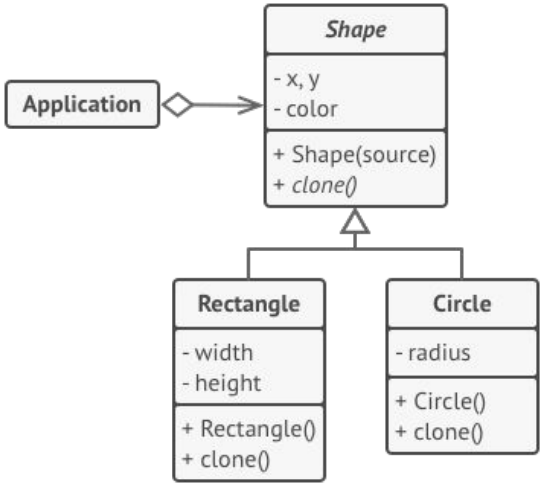

```
abstract class Shape is
    field X: int
    field Y: int
    field color: string

    constructor Shape() is
        // ...

    constructor Shape(source: Shape) is
        this()
        this.X = source.X
        this.Y = source.Y
        this.color = source.color

    abstract method clone():Shape
```

```
class Rectangle extends Shape is
    field width: int
    field height: int

    constructor Rectangle(source: Rectangle) is
        super(source)
        this.width = source.width
        this.height = source.height

    method clone():Shape is
          return new Rectangle(this)
```

https://refactoring.guru/pt-br/design-patterns/prototype

Prototype

Exemplo 2:




```
class Rectangle extends Shape is
    field width: int
    field height: int

    constructor Rectangle(source: Rectangle) is
        super(source)
        this.width = source.width
        this.height = source.height

    method clone():Shape is
        return new Rectangle(this)


class Circle extends Shape is
    field radius: int

    constructor Circle(source: Circle) is
        super(source)
        this.radius = source.radius

    method clone():Shape is
        return new Circle(this)
```

https://refactoring.guru/pt-br/design-patterns/prototype

```
// Em algum lugar dentro do código cliente.
class Application is
    field shapes: array of Shape

    constructor Application() is
        Circle circle = new Circle()
        circle.X = 10
        circle.Y = 10
        circle.radius = 20
        shapes.add(circle)

        Circle anotherCircle = circle.clone()
        shapes.add(anotherCircle)

        Rectangle rectangle = new Rectangle()
        rectangle.width = 10
        rectangle.height = 20
        shapes.add(rectangle)

    method businessLogic() is
        Array shapesCopy = new Array of Shapes.

        foreach (s in shapes) do
            shapesCopy.add(s.clone())
```

```
class Rectangle extends Shape is
    field width: int
    field height: int

    constructor Rectangle(source: Rectangle) is
        super(source)
        this.width = source.width
        this.height = source.height

    method clone():Shape is
        return new Rectangle(this)


class Circle extends Shape is
    field radius: int

    constructor Circle(source: Circle) is
        super(source)
        this.radius = source.radius

    method clone():Shape is
        return new Circle(this)
```

https://refactoring.guru/pt-br/design-patterns/prototype

Padrões de criação

Factory Method

Abstract Factory

Builder

Prototype

Singleton

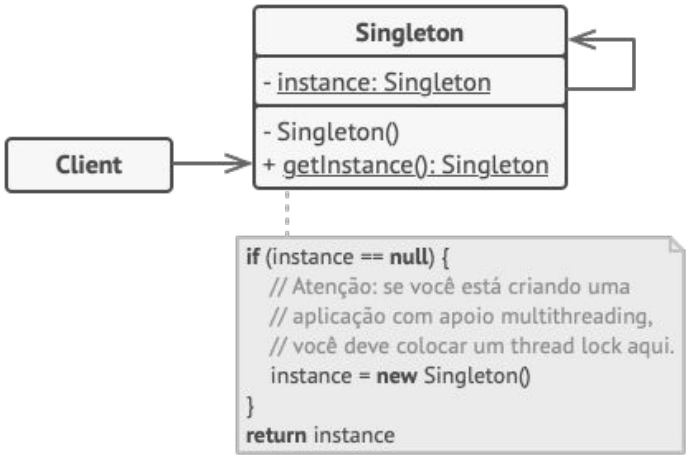
Singleton

O Singleton é um padrão de criação que permite a você garantir que uma classe tenha apenas uma instância, enquanto provê um ponto de acesso global para essa instância.

https://refactoring.guru/pt-br/design-patterns/singleton

Singleton

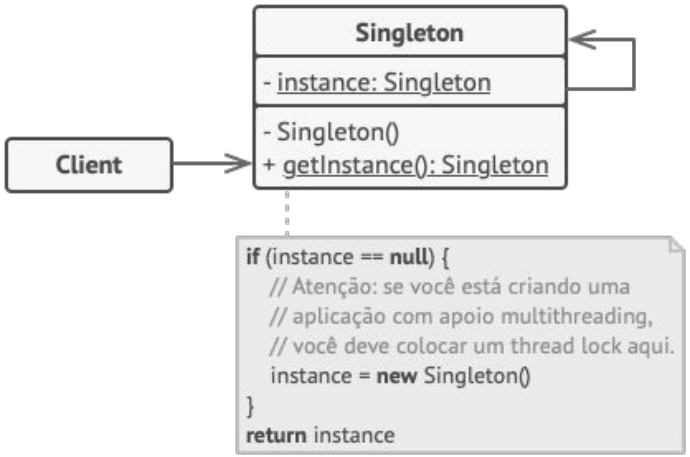
Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/singleton

Singleton

Exemplo:




```
class Database is
    private static field instance: Database

    private constructor Database() is
        // Algum código de inicialização, tal como uma conexão
        // com um servidor de base de dados.
        // ...

    public static method getInstance() is
        if (Database.instance == null) then
            Database.instance = new Database()
        return Database.instance

    public method query(sql) is
        // ...

class Application is
    method main() is
        Database foo = Database.getInstance()
        foo.query("SELECT ...")
        // ...
        Database bar = Database.getInstance()
        bar.query("SELECT ...")
        // A variável `bar` vai conter o mesmo objeto que a
        // variável `foo`.
```

https://refactoring.guru/pt-br/design-patterns/singleton

Padrões estruturais

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

Padrões estruturais

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

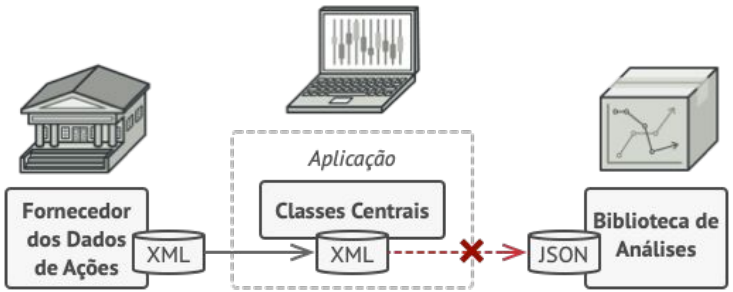
Adapter

O Adapter é um padrão de projeto estrutural que permite objetos com interfaces incompatíveis colaborarem entre si.

https://refactoring.guru/pt-br/design-patterns/adapter

Adapter

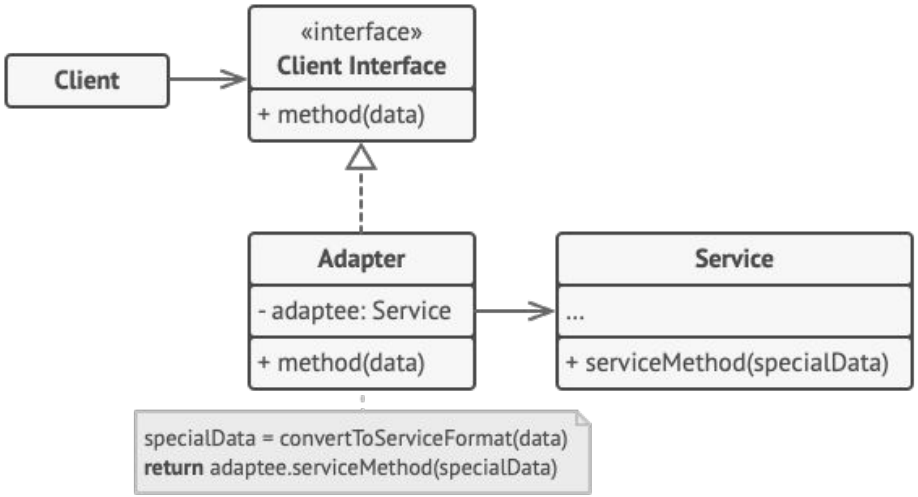
Problema (ilustração):



https://refactoring.guru/pt-br/design-patterns/adapter

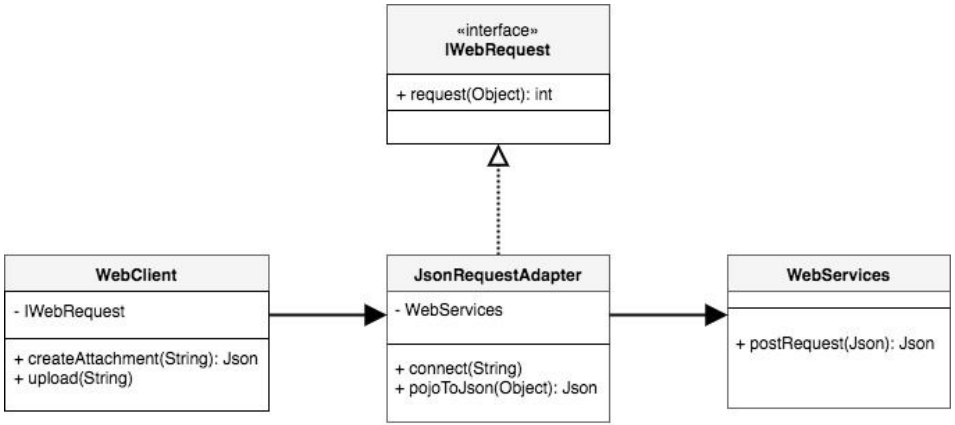
Adapter

Estrutura geral:



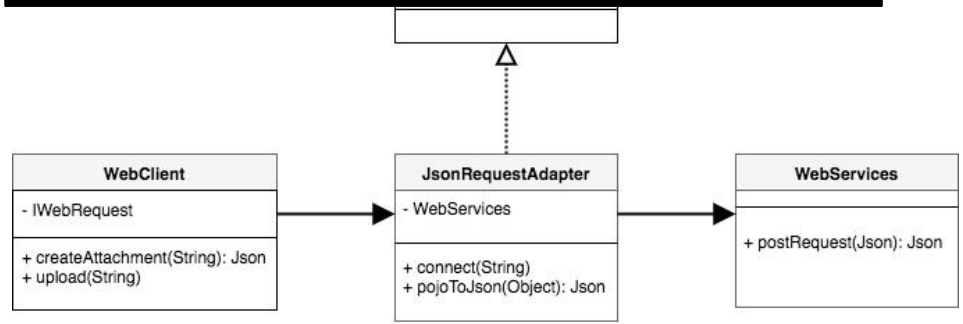
https://refactoring.guru/pt-br/design-patterns/adapter

Adapter



https://baraabytes.com/a-complete-guide-to-design-patterns-the-adapter-design-pattern/

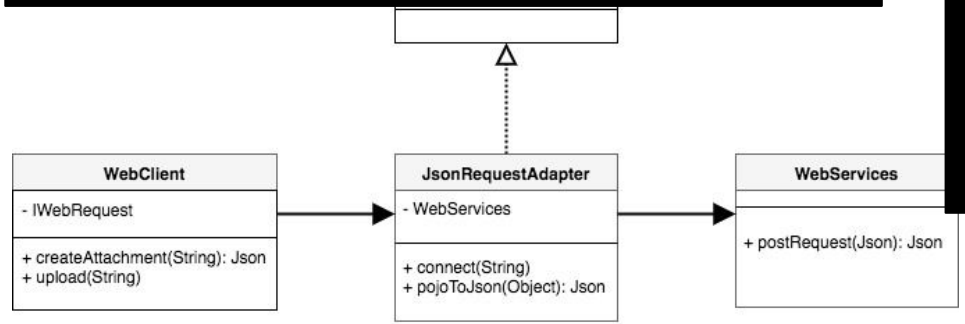
```java
public class WebClient  {
    IWebRequest webRequest;

    public WebClient(IWebRequest webRequest) {
        this.webRequest = webRequest;
    }

    private Object createAttachment(String content){
        // create plan old java object ...
        return null;
    }

    public void upload(String content){
        Object data = createAttachment(content);
        int resultCode = webRequest.request(data);
        if (resultCode == 200){
            System.out.println("Status: Ok!");
        }else {
            System.out.println("Status: Error!");
        }
    }
}
```



https://baraabytes.com/a-complete-guide-to-design-patterns-the-adapter-design-pattern/

```java
public class WebClient  {
    IWebRequest webRequest;

    public WebClient(IWebRequest webRequest) {
        this.webRequest = webRequest;
    }

    private Object createAttachment(String content){
        // create plan old java object ...
        return null;
    }

    public void upload(String content){
        Object data = createAttachment(content);
        int resultCode = webRequest.request(data);
        if (resultCode == 200){
            System.out.println("Status: Ok!");
        }else {
            System.out.println("Status: Error!");
        }
    }
}
```

```java
public interface IWebRequest {
    int request(Object Data)
}

public class JsonRequestAdapter implements IWebRequest {
    WebServices webServices;

    public JsonRequestAdapter(WebServices webServices) {
        this.webServices = webServices;
    }

    @Override
    public int request(Object data) {
        Json body = pojoToJson(data);
        Json response = webServices.postRequest(body);
        if(response != null){
            return 200; // success status code
        }
        return 500; //  Error status code
    }

    public void connect(String url){
       // connecting to the end point ...
    }

    public Json pojoToJson(Object data){
        // code for converting to json ...
        return null;
    }
}
```



https://baraabytes.com/a-complete-guide-to-design-patterns-the-adapter-design-pattern/

```java
public class WebClient  {
    IWebRequest webRequest;

    public WebClient(IWebRequest webRequest) {
        this.webRequest = webRequest;
    }

    private Object createAttachment(String content){
        // create plan old java object ...
        return null;
    }

    public void upload(String content){
        Object data = createAttachment(content);
        int resultCode = webRequest.request(data);
        if (resultCode == 200){
            System.out.println("Status: Ok!");
        }else {
            System.out.println("Status: Error!");
        }
    }
}
```

```java
public class Main {
    public static final void main(String[] args){

        WebServices webServices =  new WebServices();

        JsonRequestAdapter jsonRequestAdapter =  new
JsonRequestAdapter(webServices);
        jsonRequestAdapter.connect( "https://baraabytes.com" );

        WebClient webClient =  new
WebClient(jsonRequestAdapter);
        webClient.upload( "User data");

    }
}
```

```java
public interface IWebRequest {
    int request(Object Data)
}

public class JsonRequestAdapter implements IWebRequest {
    WebServices webServices;

    public JsonRequestAdapter(WebServices webServices) {
        this.webServices = webServices;
    }

    @Override
    public int request(Object data) {
        Json body = pojoToJson(data);
        Json response = webServices.postRequest(body);
        if(response != null){
            return 200; // success status code
        }
        return 500; //  Error status code
    }

    public void connect(String url){
       // connecting to the end point ...
    }

    public Json pojoToJson(Object data){
        // code for converting to json ...
        return null;
    }
}
```

Padrões estruturais

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

Bridge

O Bridge é um padrão de projeto estrutural que permite que você divida uma classe grande ou um conjunto de classes intimamente ligadas em duas hierarquias separadas—abstração e implementação—que podem ser desenvolvidas independentemente umas das outras.

https://refactoring.guru/pt-br/design-patterns/bridge

Padrões estruturais

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

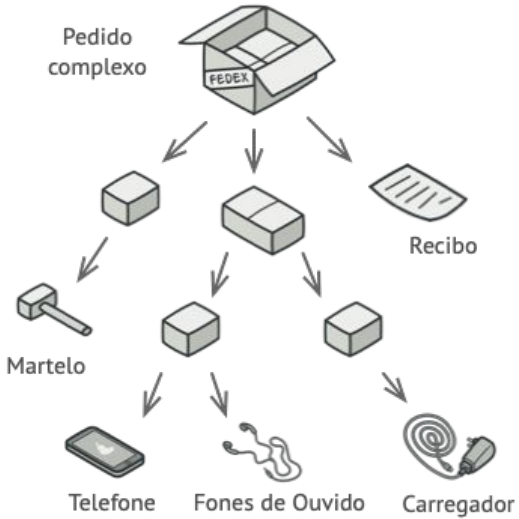
Composite

O Composite é um padrão de projeto estrutural que permite que você componha objetos em estruturas de árvores e então trabalhe com essas estruturas como se elas fossem objetos individuais.

https://refactoring.guru/pt-br/design-patterns/composite

Composite

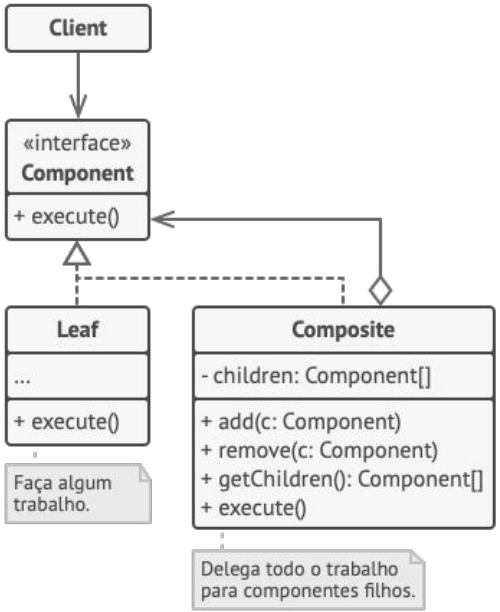
Problema (ilustração):



https://refactoring.guru/pt-br/design-patterns/composite

Composite

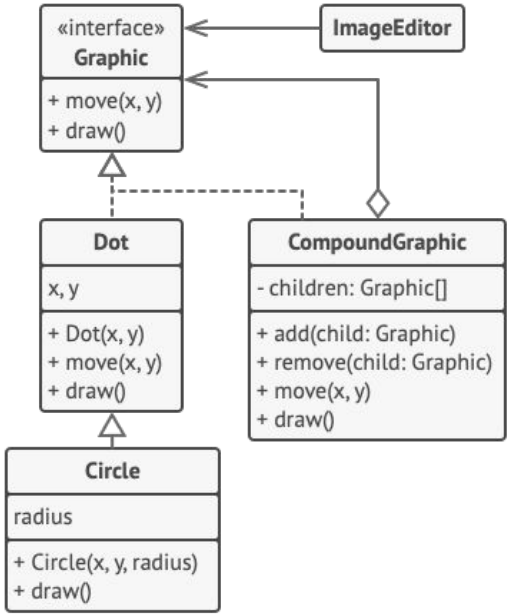
Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/composite

Composite

Exemplo:

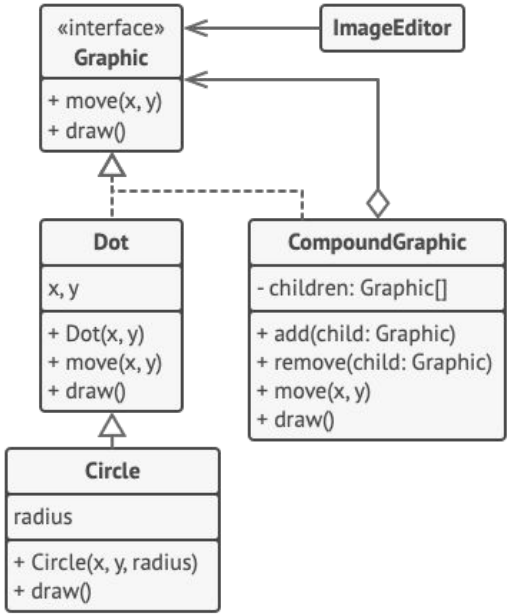



```
interface Graphic is
    method move(x, y)
    method draw()

class Dot implements Graphic is
    field x, y

    constructor Dot(x, y) { ... }

    method move(x, y) is
        this.x += x, this.y += y

    method draw() is
        // Desenhar um ponto em X e Y.

class Circle extends Dot is
    field radius

    constructor Circle(x, y, radius) { ... }

    method draw() is
        // Desenhar um círculo em X e Y com raio R.
```

https://refactoring.guru/pt-br/design-patterns/composite

Composite

Exemplo:

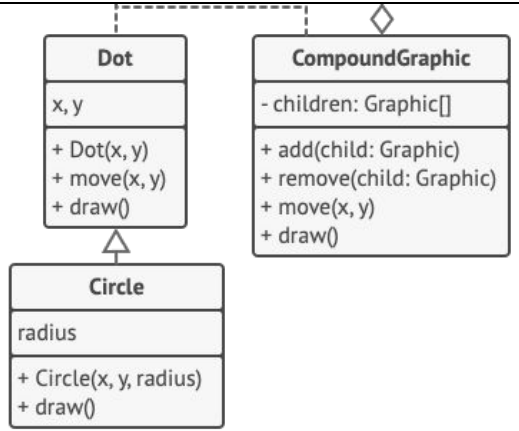


https://refactoring.guru/pt-br/design-patterns/composite

```
interface Graphic is
    method move(x, y)
    method draw()

class Dot implements Graphic is
    field x, y
    constructor Dot(x, y) { ... }

    method move(x, y) is
        this.x += x, this.y += y

    method draw() is
        // Desenhar um ponto em X e Y.

class Circle extends Dot is
    field radius
    constructor Circle(x, y, radius) { ... }

    method draw() is
        // Desenhar um círculo em X e Y com raio R.
```

```
class CompoundGraphic implements Graphic is
    field children: array of Graphic
    …
    method move(x, y) is
        foreach (child in children) do
            child.move(x, y)

    method draw() is
        // Desenhar todos os elementos
```

```
class ImageEditor is
    field all: CompoundGraphic

    method load() is
        all = new CompoundGraphic()
        all.add(new Dot(1, 2))
        all.add(new Circle(5, 3, 10))
        // ...

    method groupSelected(components: array of Graphic) is
        group = new CompoundGraphic()
        foreach (component in components) do
            group.add(component)
            all.remove(component)
        all.add(group)
        all.draw()
```



https://refactoring.guru/pt-br/design-patterns/composite

```
interface Graphic is
    method move(x, y)
    method draw()

class Dot implements Graphic is
    field x, y
    constructor Dot(x, y) { ... }

    method move(x, y) is
        this.x += x, this.y += y

    method draw() is
        // Desenhar um ponto em X e Y.

class Circle extends Dot is
    field radius
    constructor Circle(x, y, radius) { ... }

    method draw() is
        // Desenhar um círculo em X e Y com raio R.
```

```
class CompoundGraphic implements Graphic is
    field children: array of Graphic
    …
    method move(x, y) is
        foreach (child in children) do
            child.move(x, y)

    method draw() is
        // Desenhar todos os elementos
```

Padrões estruturais

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

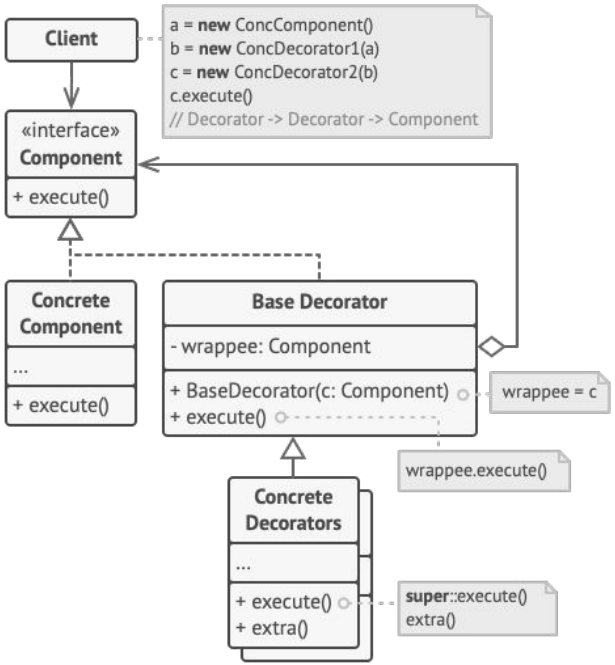
Decorator

O Decorator é um padrão de projeto estrutural que permite que você acople novos comportamentos para objetos ao colocá-los dentro de invólucros de objetos que contém os comportamentos.

https://refactoring.guru/pt-br/design-patterns/decorator

Decorator

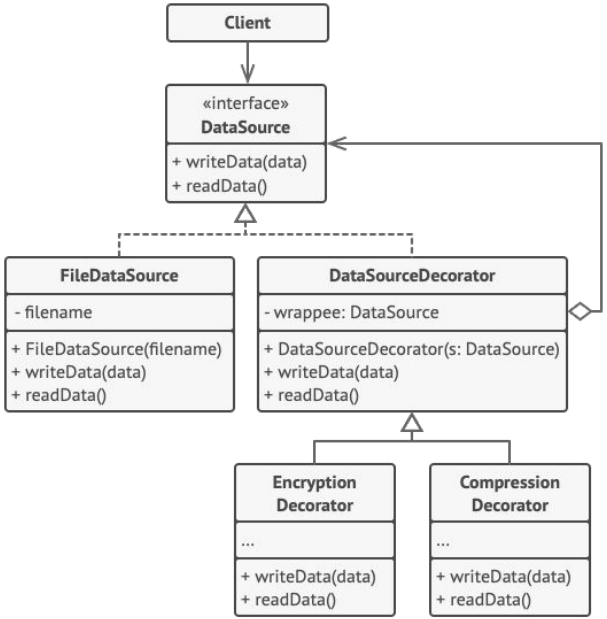
Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/decorator

Decorator

Exemplo:

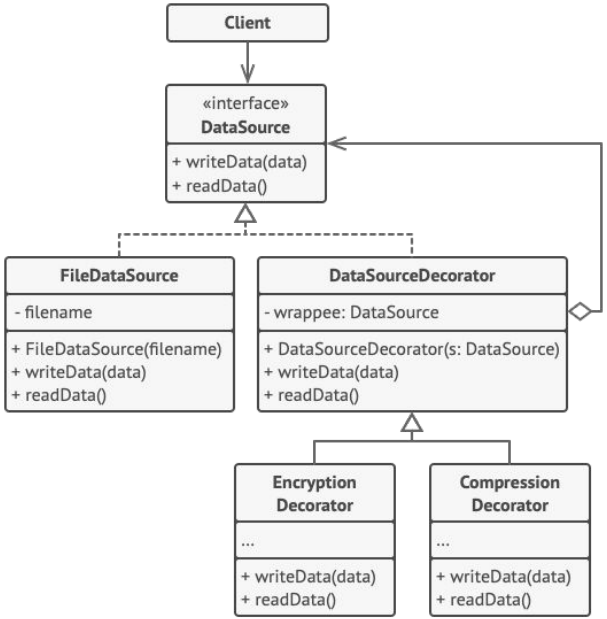



```
interface DataSource is
    method writeData(data)
    method readData():data

class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }

    method writeData(data) is
        // Escreve dados no arquivo.
    method readData():data is
        // Lê dados de um arquivo.
```

https://refactoring.guru/pt-br/design-patterns/decorator

Decorator

Exemplo:

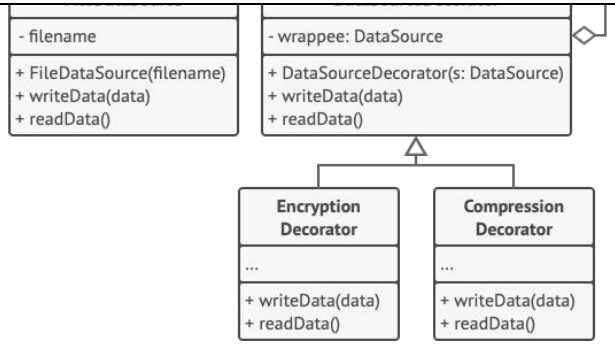


https://refactoring.guru/pt-br/design-patterns/decorator

```
interface DataSource is
    method writeData(data)
    method readData():data

class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }

    method writeData(data) is
        // Escreve dados no arquivo.
    method readData():data is
        // Lê dados de um arquivo.
```

```
class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

    constructor DataSourceDecorator(source: DataSource) is
        wrappee = source

    method writeData(data) is
        wrappee.writeData(data)

    method readData():data is
        return wrappee.readData()
```

```
class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Encriptar os dados passados.
        // 2. Passar dados encriptados p/ writeData

    method readData():data is
        // 1. Obter os dados do método readData
        // 2. Tentar decifrá-lo se for encriptado.
        // 3. Retornar o resultado.

class CompressionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Comprimir os dados passados.
        // 2. Passar os dados comprimidos p/ writeData

    method readData():data is
        // 1. Obter dados do método readData
        // 2. Tentar descomprimi-lo se for comprimido.
        // 3. Retornar o resultado.
```



https://refactoring.guru/pt-br/design-patterns/decorator

```
interface DataSource is
    method writeData(data)
    method readData():data

class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }

    method writeData(data) is
        // Escreve dados no arquivo.
    method readData():data is
        // Lê dados de um arquivo.
```

```
class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

    constructor DataSourceDecorator(source: DataSource) is
        wrappee = source

    method writeData(data) is
        wrappee.writeData(data)

    method readData():data is
        return wrappee.readData()
```

```
class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Encriptar os dados passados.
        // 2. Passar dados encriptados p/ writeData

    method readData():data is
        // 1. Obter os dados do método readData
        // 2. Tentar decifrá-lo se for encriptado.
        // 3. Retornar o resultado.

class CompressionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Comprimir os dados passados.
        // 2. Passar os dados comprimidos p/ writeData

    method readData():data is
        // 1. Obter dados do método readData
        // 2. Tentar descomprimi-lo se for comprimido.
        // 3. Retornar o resultado.
```


```
class Application is
    method dumbUsageExample() is
        source = new FileDataSource("somefile.dat")
        source.writeData(salaryRecords)

        source = new CompressionDecorator(source)
        source.writeData(salaryRecords)

        source = new EncryptionDecorator(source)
// Source agora contém isso:
//    * Encryption > Compression > FileDataSource
        source.writeData(salaryRecords)
```

```
interface DataSource is
    method writeData(data)
    method readData():data

class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }

    method writeData(data) is
        // Escreve dados no arquivo.
    method readData():data is
        // Lê dados de um arquivo.
```

```
class DataSourceDecorator implements DataSource is
    protected field wrappee: DataSource

    constructor DataSourceDecorator(source: DataSource) is
        wrappee = source

    method writeData(data) is
        wrappee.writeData(data)

    method readData():data is
        return wrappee.readData()
```

```
class EncryptionDecorator extends DataSourceDecorator is
    method writeData(data) is
        // 1. Encriptar os dados passados.
        // 2. Passar dados encriptados p/ writeData

    method readData():data is
        // 1. Obter os dados do método readData
        // 2. Tentar decifrá-lo se for encriptado.
        // 3. Retornar o resultado.

class CompressionDecorator
    method writeData(data)
        // 1. Comprimir os
        // 2. Passar os da

    method readData():data
        // 1. Obter dados
        // 2. Tentar desco
        // 3. Retornar o r
```

```
interface DataSource is
    method writeData(data)
    method readData():data

class FileDataSource implements DataSource is
    constructor FileDataSource(filename) { ... }

    method writeData(data) is
        // Escreve dados no arquivo.
        ():data is
        de um arquivo.
```

```
class ApplicationConfigurator is
    method configurationExample() is
        source = new FileDataSource("salary.dat")
        if (enabledEncryption)
            source = new EncryptionDecorator(source)
        if (enabledCompression)
            source = new CompressionDecorator(source)

        logger = new SalaryManager(source)
        salary = logger.load()
    // ...
```

```
 implements DataSource is
ee: DataSource

eDecorator(source: DataSource) is


 is
(data)

method readData():data is
    return wrappee.readData()
```

- filename
- wrappe

```
class Application is
    method dumbUsageExample() is
        source = new FileDataSource("somefile.dat")
        source.writeData(salaryRecords)

        source = new CompressionDecorator(source)
        source.writeData(salaryRecords)

        source = new EncryptionDecorator(source)
// Source agora contém isso:
//    * Encryption > Compression > FileDataSource
        source.writeData(salaryRecords)
```

Padrões estruturais

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

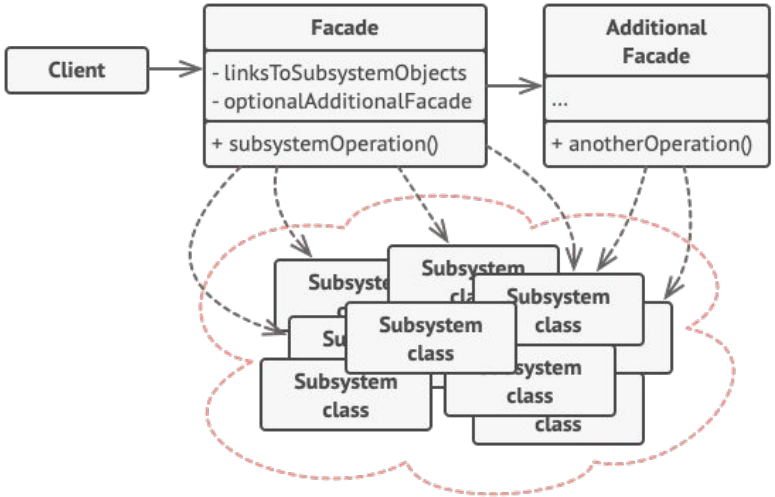
Façade

O Facade é um padrão de projeto estrutural que fornece uma interface simplificada para uma biblioteca, um framework, ou qualquer conjunto complexo de classes.

https://refactoring.guru/pt-br/design-patterns/facade

Façade

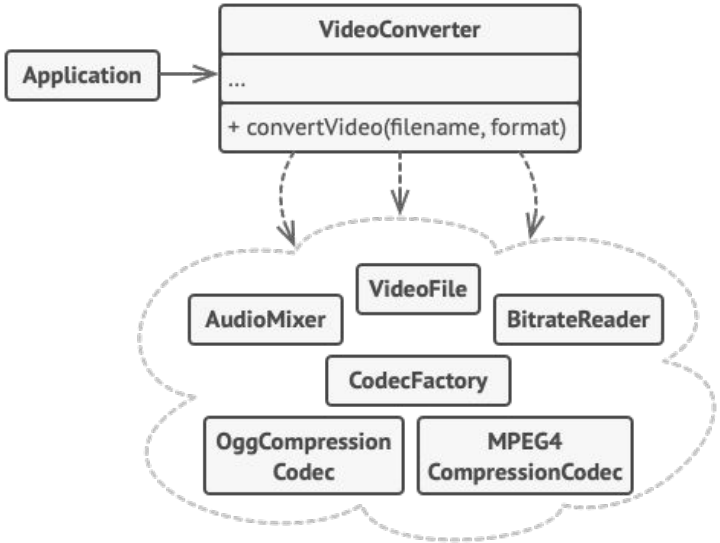
Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/facade

Façade

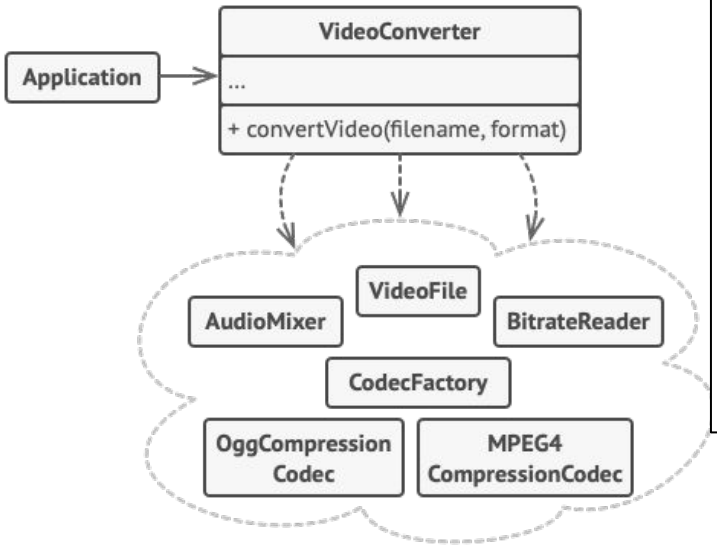
Exemplo:



https://refactoring.guru/pt-br/design-patterns/facade

Façade

Exemplo:




```
class VideoConverter is
    method convert(filename, format):File is
        file = new VideoFile(filename)
        sourceCodec = (new CodecFactory).extract(file)
        if (format == "mp4")
            destinationCodec = new MPEG4CompressionCodec()
        else
            destinationCodec = new OggCompressionCodec()
        buffer = BitrateReader.read(filename, sourceCodec)
        result = BitrateReader.convert(buffer, destinationCodec)
        result = (new AudioMixer()).fix(result)
        return new File(result)

// As classes da aplicação não dependem de um bilhão de classes
// fornecidas por um framework complexo. Também, se você decidir
// trocar de frameworks, você só precisa reescrever a classe
// fachada.
class Application is
    method main() is
        convertor = new VideoConverter()
        mp4 = convertor.convert("funny-video.ogg", "mp4")
        mp4.save()
```

https://refactoring.guru/pt-br/design-patterns/facade

Padrões estruturais

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

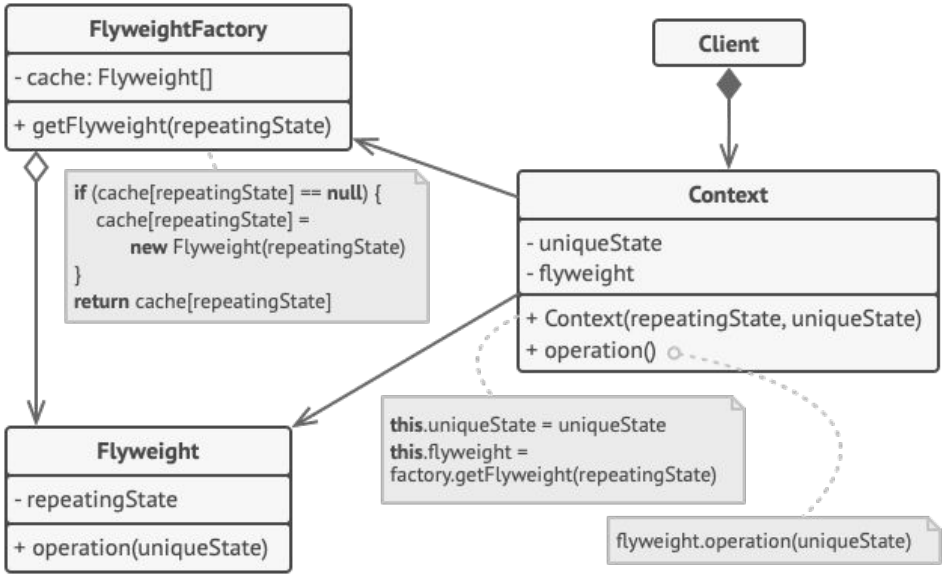
Flyweight

O Flyweight é um padrão de projeto estrutural que permite a você colocar mais objetos na quantidade de RAM disponível ao compartilhar partes comuns de estado entre os múltiplos objetos ao invés de manter todos os dados em cada objeto.

https://refactoring.guru/pt-br/design-patterns/flyweight

Flyweight

Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/flyweight

Padrões estruturais

Adapter

Bridge

Composite

Decorator

Façade

Flyweight

Proxy

Proxy

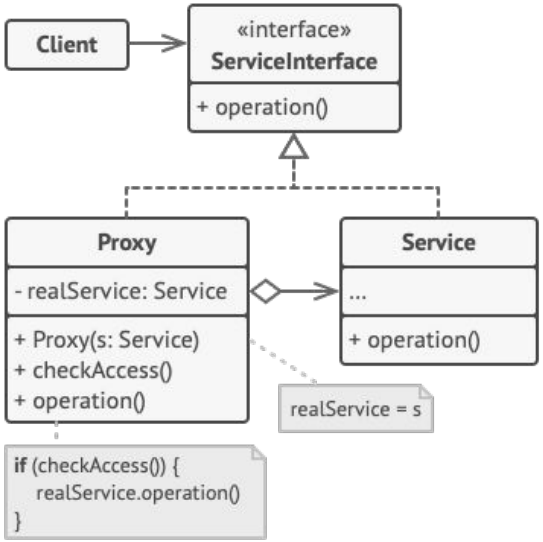
O Proxy é um padrão de projeto estrutural que permite que você forneça um substituto ou um espaço reservado para outro objeto.

Um proxy controla o acesso ao objeto original, permitindo que você faça algo ou antes ou depois do pedido chegar ao objeto original.

https://refactoring.guru/pt-br/design-patterns/proxy

Proxy

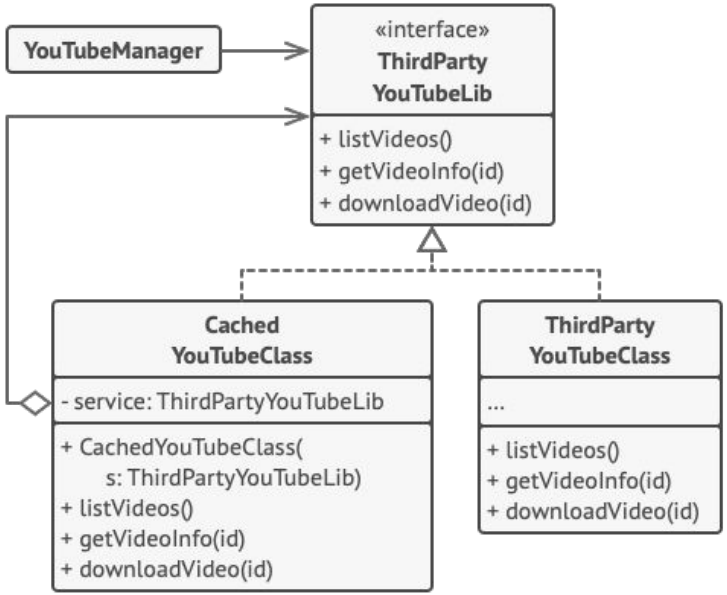
Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/proxy

Proxy

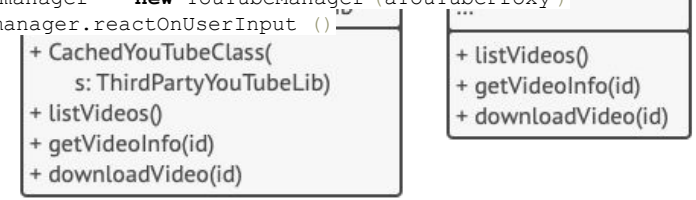
Exemplo:



https://refactoring.guru/pt-br/design-patterns/proxy

```
interface ThirdPartyYouTubeLib is
    method listVideos()
    method getVideoInfo(id)
    method downloadVideo(id)


class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
    method listVideos() is
        // Envia um pedido API para o YouTube.
    method getVideoInfo(id) is
        // Obtém metadados sobre algum vídeo.
    method downloadVideo(id) is
          // Baixa um arquivo de vídeo do YouTube.

class CachedYouTubeClass implements ThirdPartyYouTubeLib is
    private field service: ThirdPartyYouTubeLib
    private field listCache, videoCache
    field needReset

    constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib) is
        this.service = service

    method listVideos() is
        if (listCache == null || needReset)
            listCache = service.listVideos()
        return listCache

    method getVideoInfo(id) is
        if (videoCache == null || needReset)
            videoCache = service.getVideoInfo(id)
        return videoCache

    method downloadVideo(id) is
        if (!downloadExists(id) || needReset)
            service.downloadVideo(id)
```

```
class YouTubeManager is
    protected field service: ThirdPartyYouTubeLib

    constructor YouTubeManager(service: ThirdPartyYouTubeLib )
is
        this.service = service

    method renderVideoPage(id) is
        info = service.getVideoInfo (id)
        // Renderiza a página do vídeo.

    method renderListPanel() is
        list = service.listVideos ()
        // Renderiza a lista de miniaturas do vídeo.

    method reactOnUserInput() is
        renderVideoPage ()
        renderListPanel ()

// A aplicação pode configurar proxies de forma fácil e
rápida.
class Application is
    method init() is
        aYouTubeService = new ThirdPartyYouTubeClass ()
        aYouTubeProxy = new
CachedYouTubeClass (aYouTubeService )
        manager = new YouTubeManager (aYouTubeProxy )
        manager.reactOnUserInput ()
```



https://refactoring.guru/pt-br/design-patterns/proxy

```
interface ThirdPartyYouTubeLib is
    method listVideos()
    method getVideoInfo(id)
    method downloadVideo(id)


class ThirdPartyYouTubeClass implements ThirdPartyYouTubeLib is
    method listVideos() is
        // Envia um pedido API para o YouTube.
    method getVideoInfo(id) is
        // Obtém metadados sobre algum vídeo.
    method downloadVideo(id) is
          // Baixa um arquivo de vídeo do YouTube.

class CachedYouTubeClass implements ThirdPartyYouTubeLib is
    private field service: ThirdPartyYouTubeLib
    private field listCache, videoCache
    field needReset

    constructor CachedYouTubeClass(service: ThirdPartyYouTubeLib ) is
        this.service = service

    method listVideos() is
        if (listCache == null || needReset )
            listCache = service.listVideos ()
        return listCache

    method getVideoInfo(id) is
        if (videoCache == null || needReset )
            videoCache = service.getVideoInfo (id)
        return videoCache

    method downloadVideo(id) is
        if (!downloadExists(id) || needReset )
            service.downloadVideo (id)
```

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

Chain of Responsibility

O Chain of Responsibility é um padrão de projeto comportamental que permite que você passe pedidos por uma corrente de handlers. Ao receber um pedido, cada handler decide se processa o pedido ou o passa adiante para o próximo handler na corrente.

https://refactoring.guru/pt-br/design-patterns/chain-of-responsibility

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

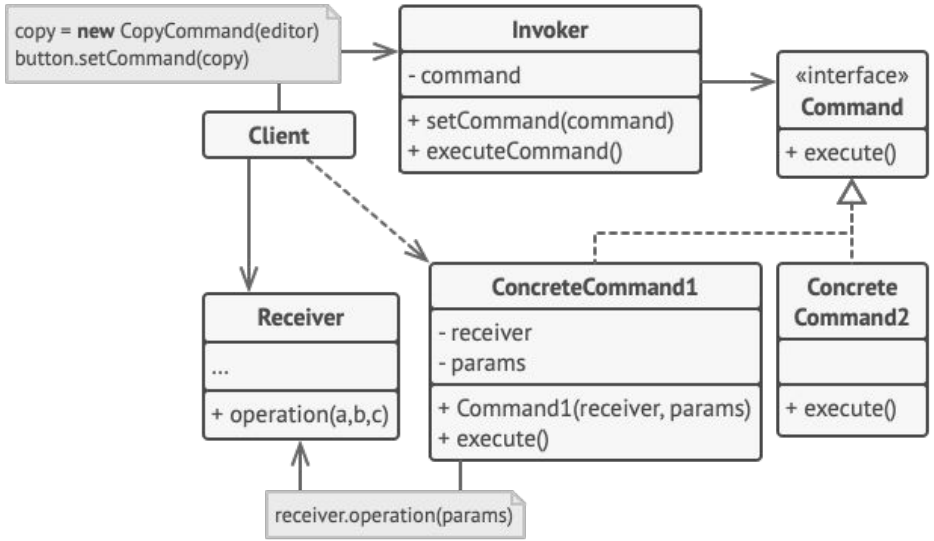
Command

O Command é um padrão de projeto comportamental que transforma um pedido em um objeto independente que contém toda a informação sobre o pedido. Essa transformação permite que você parametrize métodos com diferentes pedidos, atrase ou coloque a execução do pedido em uma fila, e suporte operações que não podem ser feitas.

https://refactoring.guru/pt-br/design-patterns/command

Command

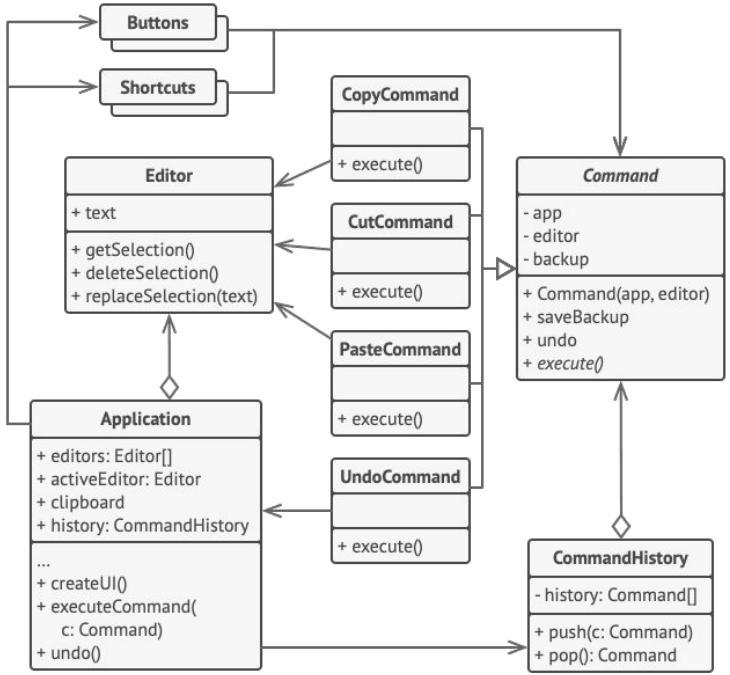
Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/command

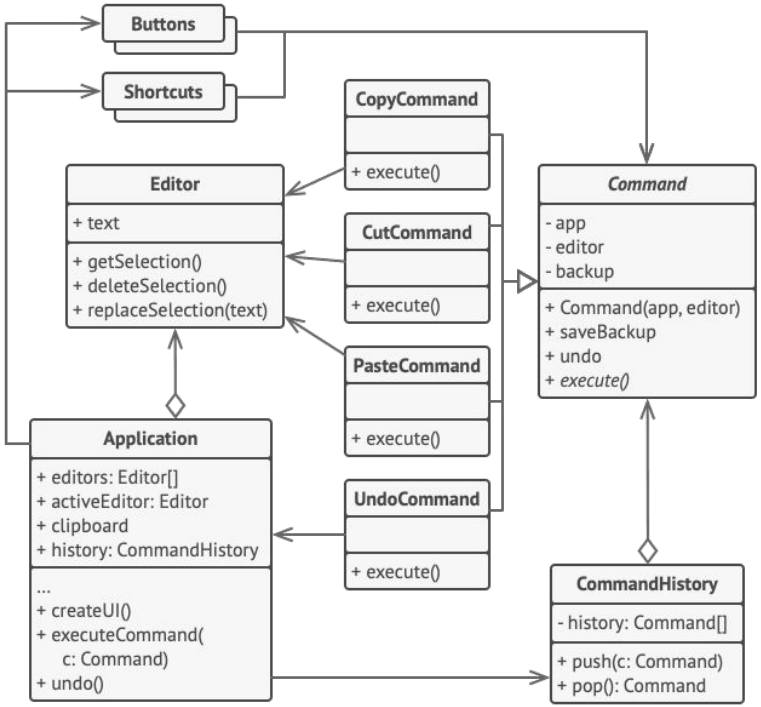
Command

Exemplo:



https://refactoring.guru/pt-br/design-patterns/command

Command




```
abstract class Command is
    protected field app: Application
    protected field editor: Editor
    protected field backup: text

    constructor Command(app: Application, editor: Editor) is
        this.app = app
        this.editor = editor

    method saveBackup() is
        backup = editor.text

    method undo() is
        editor.text = backup

      abstract method execute()
```

https://refactoring.guru/pt-br/design-patterns/command

```
class CopyCommand extends Command is
    method execute() is
        app.clipboard = editor.getSelection()
        return false

class CutCommand extends Command is
    method execute() is
        saveBackup()
        app.clipboard = editor.getSelection()
        editor.deleteSelection()
        return true

class PasteCommand extends Command is
    method execute() is
        saveBackup()
        editor.replaceSelection(app.clipboard)
        return true

class UndoCommand extends Command is
    method execute() is
        app.undo()
        return false

class CommandHistory is
    private field history: array of Command

    method push(c: Command) is
        // Empilha o comando

    method pop():Command is
          // Obtem o comando mais recente
```

```
abstract class Command is
    protected field app: Application
    protected field editor: Editor
    protected field backup: text

    constructor Command(app: Application, editor: Editor) is
        this.app = app
        this.editor = editor

    method saveBackup() is
        backup = editor.text

    method undo() is
        editor.text = backup

      abstract method execute()
```

```
class CopyCommand extends Command is
    method execute() is
        app.clipboard = editor.getSelection()
        return false

class CutCommand extends Command is
    method execute() is
        saveBackup()
        app.clipboard = editor.getSelection()
        editor.deleteSelection()
        return true

class PasteCommand extends Command is
    method execute() is
        saveBackup()
        editor.replaceSelection(app.clipboard)
        return true

class UndoCommand extends Command is
    method execute() is
        app.undo()
        return false

class CommandHistory is
    private field history: array of Command

    method push(c: Command) is
        // Empilha o comando

    method pop():Command is
          // Obtem o comando mais recente
```

```
class Application is
    field clipboard: string
    field editors: array of Editors
    field activeEditor: Editor
    field history: CommandHistory

    // O código que assinala comandos para objetos UI pode se
    // parecer como este.
    method createUI() is
        // ...
        copy = function() { executeCommand(
            new CopyCommand(this, activeEditor)) }
        copyButton.setCommand(copy)
        shortcuts.onKeyPress("Ctrl+C", copy)

        cut = function() { executeCommand(
            new CutCommand(this, activeEditor)) }
        cutButton.setCommand(cut)
        shortcuts.onKeyPress("Ctrl+X", cut)

        paste = function() { executeCommand(
            new PasteCommand(this, activeEditor)) }
        pasteButton.setCommand(paste)
        shortcuts.onKeyPress("Ctrl+V", paste)

        undo = function() { executeCommand(
            new UndoCommand(this, activeEditor)) }
        undoButton.setCommand(undo)
          shortcuts.onKeyPress("Ctrl+Z", undo)
```

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

Interpreter

Dada uma determinada linguagem, o padrão Interpreter define uma representação para sua gramática juntamente com um interpretador que usa a representação para interpretar sentenças na língua.

Ou mapear um domínio para uma língua, a língua para uma gramática e a gramática para um projeto de design hierárquico orientado a objetos.

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

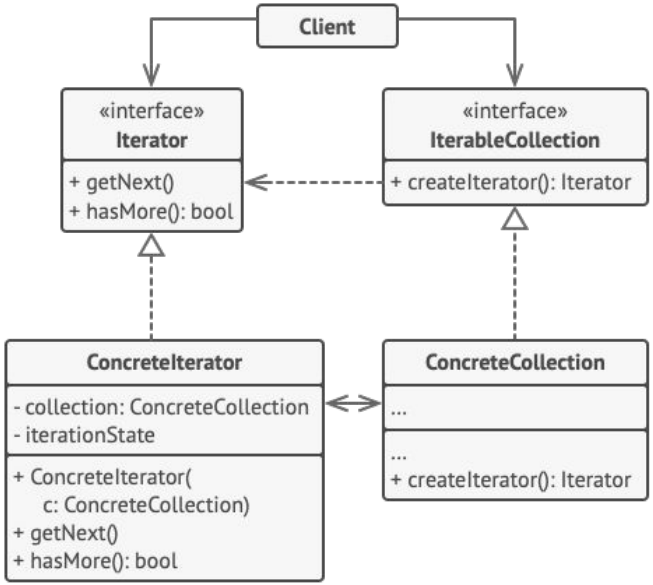
Iterator

O Iterator é um padrão de projeto comportamental que permite a você percorrer elementos de uma coleção sem expor as representações dele (lista, pilha, árvore, etc.)

https://refactoring.guru/pt-br/design-patterns/iterator

Iterator

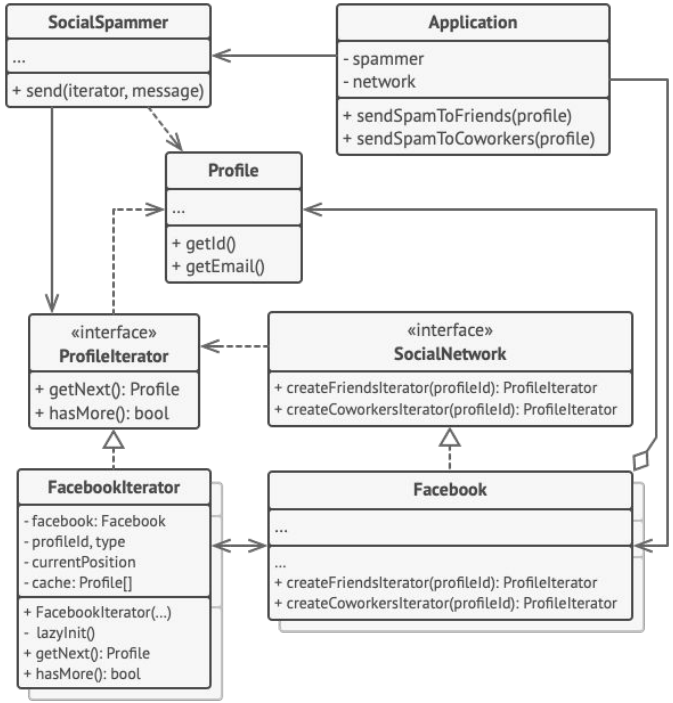
Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/iterator

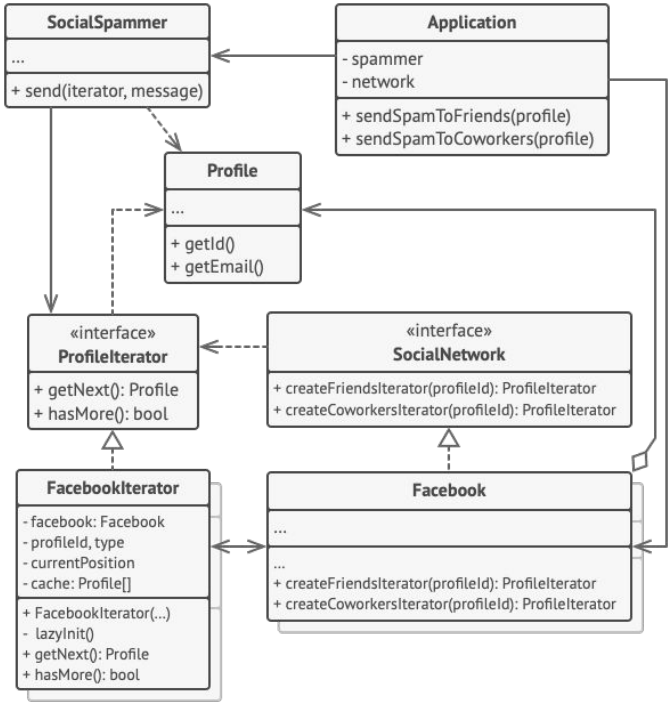
Iterator

Exemplo:



https://refactoring.guru/pt-br/design-patterns/iterator

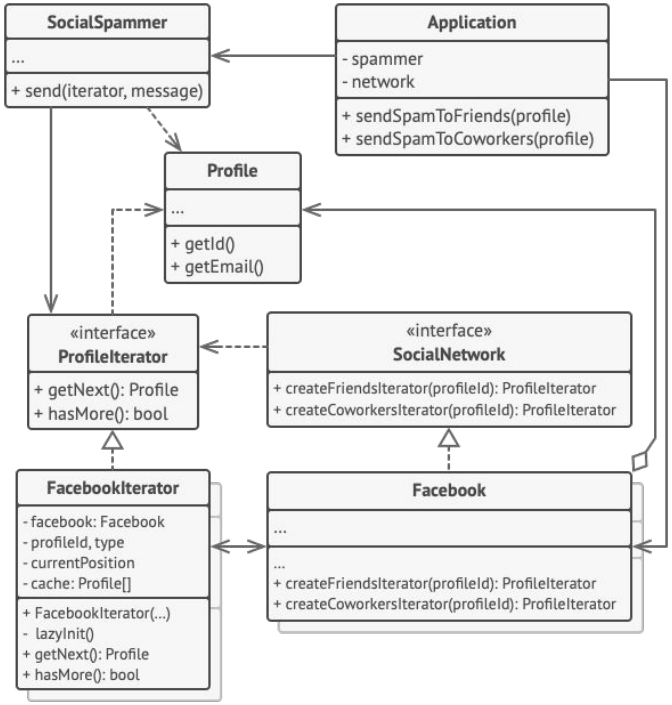
Iterator




```
interface SocialNetwork is
    method createFriendsIterator(profileId):ProfileIterator
    method createCoworkersIterator(profileId):ProfileIterator


class Facebook implements SocialNetwork is
    // Código de criação do iterador.
    method createFriendsIterator(profileId) is
        return new FacebookIterator(this, profileId, "friends")
    method createCoworkersIterator(profileId) is
          return new FacebookIterator(this, profileId, "coworkers")
```

https://refactoring.guru/pt-br/design-patterns/iterator

```
// A interface comum a todos os iteradores.
interface ProfileIterator is
    method getNext():Profile
      method hasMore():bool
```



https://refactoring.guru/pt-br/design-patterns/iterator

```
interface SocialNetwork is
    method createFriendsIterator(profileId):ProfileIterator
    method createCoworkersIterator(profileId):ProfileIterator


class Facebook implements SocialNetwork is
    // Código de criação do iterador.
    method createFriendsIterator(profileId) is
        return new FacebookIterator(this, profileId, "friends")
    method createCoworkersIterator(profileId) is
          return new FacebookIterator(this, profileId, "coworkers")
```

```
class FacebookIterator implements ProfileIterator is
    private field facebook: Facebook
    private field profileId, type: string
    private field currentPosition
    private field cache: array of Profile

    constructor FacebookIterator(facebook, profileId, type) is
        this.facebook = facebook
        this.profileId = profileId
        this.type = type

    private method lazyInit() is
        if (cache == null)
            cache = facebook.socialGraphRequest(profileId, type)

    method getNext() is
        if (hasMore())
            currentPosition++
            return cache[currentPosition]

    method hasMore() is
        lazyInit()
          return currentPosition < cache.length
```

```
// A interface comum a todos os iteradores.
interface ProfileIterator is
```

```
interface SocialNetwork is
    method createFriendsIterator(profileId):ProfileIterator
    method createCoworkersIterator(profileId):ProfileIterator
```

```
class SocialSpammer is
    method send(iterator: ProfileIterator, message: string) is
        while (iterator.hasMore())
            profile = iterator.getNext()
            System.sendEmail(profile.getEmail(), message)


// A classe da aplicação configura coleções e iteradores e então
// os passa ao código cliente.
class Application is
    field network: SocialNetwork
    field spammer: SocialSpammer

    method config() is
        if working with Facebook
            this.network = new Facebook()
        if working with LinkedIn
            this.network = new LinkedIn()
        this.spammer = new SocialSpammer()

    method sendSpamToFriends(profile) is
        iterator = network.createFriendsIterator(profile.getId())
        spammer.send(iterator, "Very important message")

    method sendSpamToCoworkers(profile) is
        iterator = network.createCoworkersIterator(profile.getId())
          spammer.send(iterator, "Very important message")
```

```
mplements SocialNetwork is
 criação do iterador.
eFriendsIterator(profileId) is
ew FacebookIterator(this, profileId, "friends")
eCoworkersIterator(profileId) is
 new FacebookIterator(this, profileId, "coworkers")
```

```
Iterator implements ProfileIterator is
eld facebook: Facebook
eld profileId, type: string
eld currentPosition
eld cache: array of Profile

r FacebookIterator(facebook, profileId, type) is
facebook = facebook
rofileId = profileId
ype = type

thod lazyInit() is
che == null)
che = facebook.socialGraphRequest(profileId, type)

Next() is
sMore())
rrentPosition++
turn cache[currentPosition]


method hasMore() is
    lazyInit()
      return currentPosition < cache.length
```

https://refactoring.guru/pt-br/design-patterns/iterator

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

Visitor

O Visitor é um padrão de projeto comportamental que permite que você separe algoritmos dos objetos nos quais eles operam

https://refactoring.guru/pt-br/design-patterns/visitor

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

Mediator

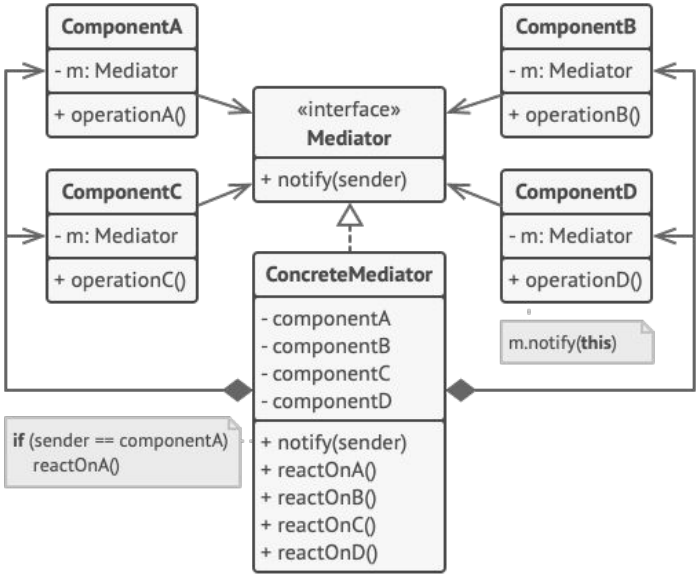
O Mediator é um padrão de projeto comportamental que permite que você reduza as dependências caóticas entre objetos.

O padrão restringe comunicações diretas entre objetos e os força a colaborar apenas através do objeto mediador.

https://refactoring.guru/pt-br/design-patterns/mediator

Mediator

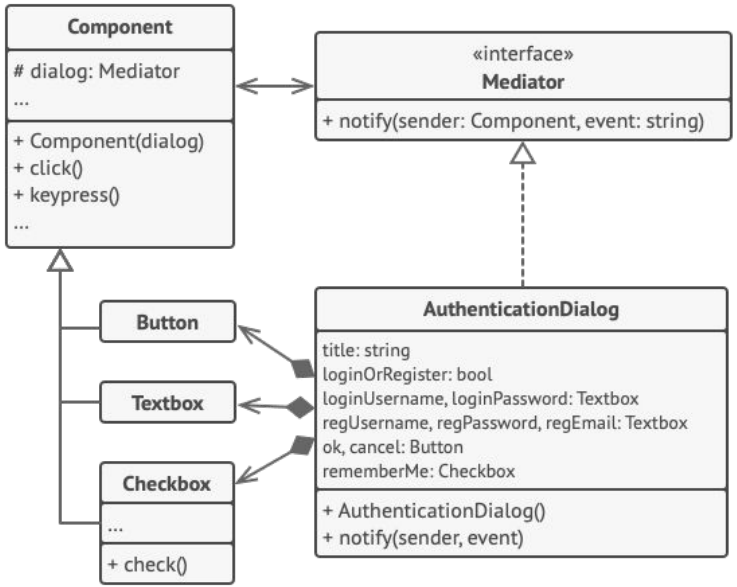
Estrutura geral:



https://refactoring.guru/pt-br/design-patterns/mediator

Mediator

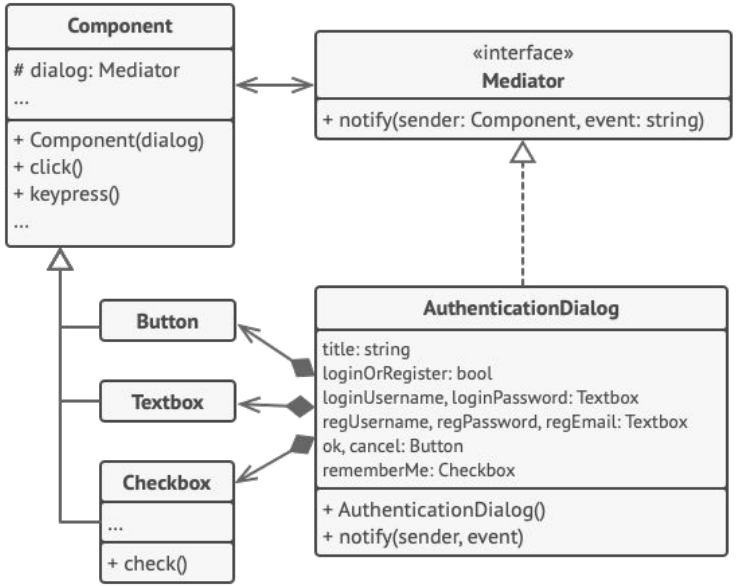
Exemplo:



https://refactoring.guru/pt-br/design-patterns/mediator

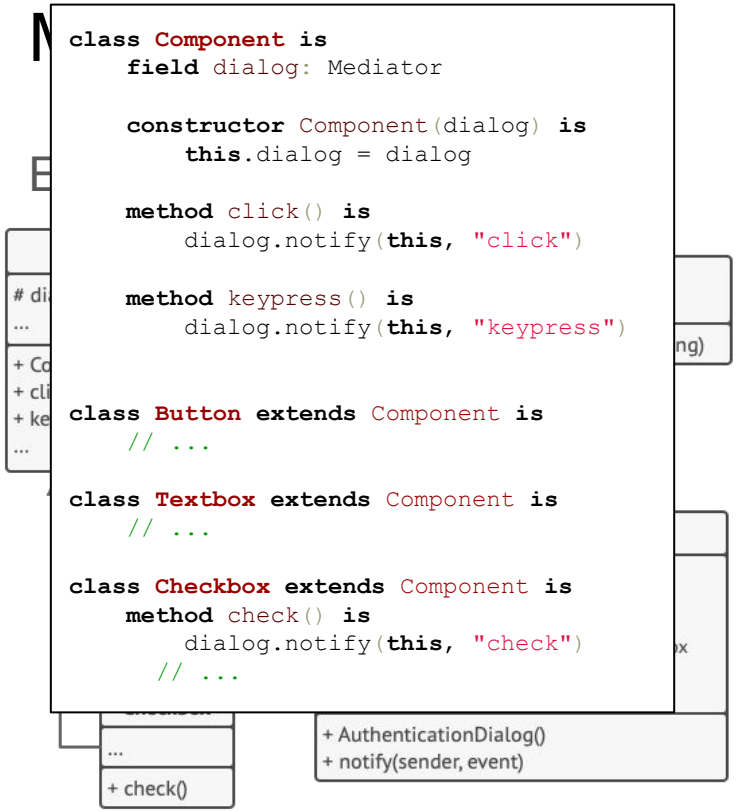
Mediator

Exemplo:



https://refactoring.guru/pt-br/design-patterns/mediator

```
interface Mediator is
    method notify(sender: Component, event: string)

class AuthenticationDialog implements Mediator is
    private field title: string
    private field loginOrRegisterChkBx: Checkbox
    private field loginUsername, loginPassword: Textbox
    private field registrationUsername, registrationPassword,
                  registrationEmail: Textbox
    private field okBtn, cancelBtn: Button

    constructor AuthenticationDialog() is
        // Cria todos os objetos componentes e passa o atual
        // mediador em seus construtores para estabelecer links.

    method notify(sender, event) is
        if (sender == loginOrRegisterChkBx and event == "check")
            if (loginOrRegisterChkBx.checked)
                title = "Log in"
                // 1. Mostra componentes de formulário de login.
                // 2. Esconde componentes de formulário de registro.
            else
                title = "Register"
                // 1. Mostra componentes de formulário de registro.
                // 2. Esconde componentes de formulário de login.
        if (sender == okBtn && event == "click")
            if (loginOrRegister.checked)
                // Tenta encontrar um usuário usando as infos de login.
                if (!found)
                    // Mostra uma mensagem de erro acima do campo login.
            else
                // 1. Cria uma conta de usuário usando dados do registro.
                // 2. Loga aquele usuário.
                // ...
```




```
class Component is
    field dialog: Mediator

    constructor Component(dialog) is
        this.dialog = dialog

    method click() is
        dialog.notify(this, "click")

    method keypress() is
        dialog.notify(this, "keypress")


class Button extends Component is
    // ...

class Textbox extends Component is
    // ...

class Checkbox extends Component is
    method check() is
        dialog.notify(this, "check")
      // ...
```

```
interface Mediator is
    method notify(sender: Component, event: string)

class AuthenticationDialog implements Mediator is
    private field title: string
    private field loginOrRegisterChkBx: Checkbox
    private field loginUsername, loginPassword: Textbox
    private field registrationUsername, registrationPassword,
                  registrationEmail: Textbox
    private field okBtn, cancelBtn: Button

    constructor AuthenticationDialog() is
        // Cria todos os objetos componentes e passa o atual
        // mediador em seus construtores para estabelecer links.

    method notify(sender, event) is
        if (sender == loginOrRegisterChkBx and event == "check")
            if (loginOrRegisterChkBx.checked)
                title = "Log in"
                // 1. Mostra componentes de formulário de login.
                // 2. Esconde componentes de formulário de registro.
            else
                title = "Register"
                // 1. Mostra componentes de formulário de registro.
                // 2. Esconde componentes de formulário de login.
        if (sender == okBtn && event == "click")
            if (loginOrRegister.checked)
                // Tenta encontrar um usuário usando as infos de login.
                if (!found)
                    // Mostra uma mensagem de erro acima do campo login.
            else
                // 1. Cria uma conta de usuário usando dados do registro.
                // 2. Loga aquele usuário.
                // ...
```

https://refactoring.guru/pt-br/design-patterns/mediator

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

Memento

O Memento é um padrão de projeto comportamental que permite que você salve e restaure o estado anterior de um objeto sem revelar os detalhes de sua implementação.

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

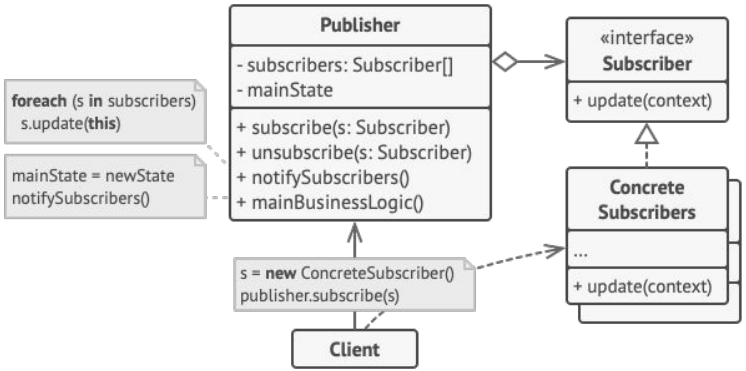
Observer

O Observer é um padrão de projeto comportamental que permite que você defina um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.

https://refactoring.guru/pt-br/design-patterns/observer

Observer

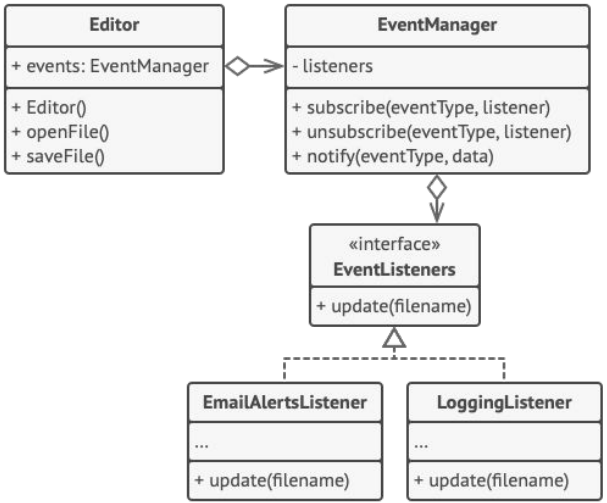
Estrutura padrão:



https://refactoring.guru/pt-br/design-patterns/observer

Observer

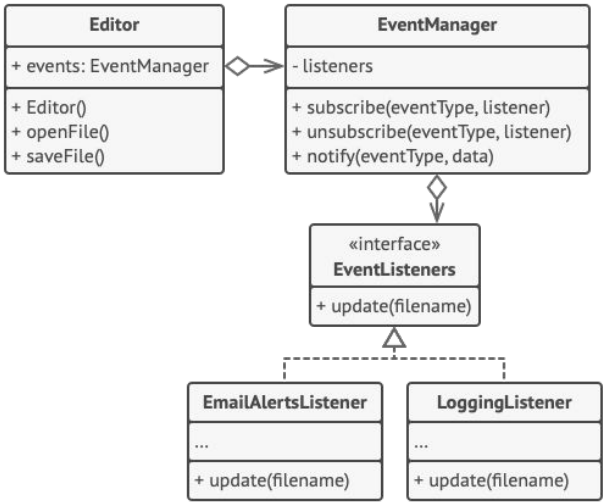
Exemplo:



https://refactoring.guru/pt-br/design-patterns/observer

Observer

Exemplo:

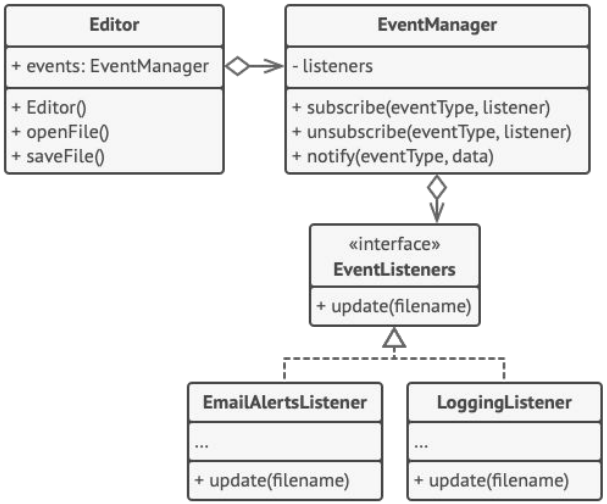



```
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)
```

https://refactoring.guru/pt-br/design-patterns/observer

Observer

Exemplo:




```
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)
```

```
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)
    // ...
```

https://refactoring.guru/pt-br/design-patterns/observer

Observer

```
interface EventListener is
    method update(filename)

class LoggingListener implements EventListener is
    private field log: File
    private field message: string

    constructor LoggingListener(log_filename, message) is
        this.log = new File(log_filename)
        this.message = message

    method update(filename) is
        log.write(replace('%s',filename,message))

class EmailAlertsListener implements EventListener is
    private field email: string
    private field message: string

    constructor EmailAlertsListener(email, message) is
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s',filename,message))
```

```
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        listeners.remove(eventType, listener)

    method notify(eventType, data) is
        foreach (listener in listeners.of(eventType)) do
            listener.update(data)
```

```
class Editor is
    public field events: EventManager
    private field file: File

    constructor Editor() is
        events = new EventManager()

    method openFile(path) is
        this.file = new File(path)
        events.notify("open", file.name)

    method saveFile() is
        file.write()
        events.notify("save", file.name)
    // ...
```

https://refactoring.guru/pt-br/design-patterns/observer

Observer

```
interface EventListener is
    method update(filename)

class LoggingListener implements EventListener is
    private field log: File
    private field message: string

    constructor LoggingListener(
        this.log = new File(log_f
        this.message = message

    method update(filename) is
        log.write(replace('%s',f

class EmailAlertsListener impleme
    private field email: string
    private field message: string

    constructor EmailAlertsListen
        this.email = email
        this.message = message

    method update(filename) is
        system.email(email, replace('%s',filename,message))
```

```
class Application is
    method config() is
        editor = new Editor()

        logger = new LoggingListener(
            "/path/to/log.txt",
            "Someone has opened the file: %s")
        editor.events.subscribe("open", logger)

        emailAlerts = new EmailAlertsListener(
            "admin@example.com",
            "Someone has changed the file: %s")
        editor.events.subscribe("save", emailAlerts)
```

```
class EventManager is
    private field listeners: hash map of event types and listeners

    method subscribe(eventType, listener) is
        listeners.add(eventType, listener)

    method unsubscribe(eventType, listener) is
        e, listener)

    ) is
    eners.of(eventType)) do
    a)
```

```
s: EventManager
: File

() is
ventManager()

th) is
    this.file = new File(path)
    events.notify("open", file.name)

method saveFile() is
    file.write()
    events.notify("save", file.name)
// ...
```

https://refactoring.guru/pt-br/design-patterns/observer

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

State

O State é um padrão de projeto comportamental que permite que um objeto altere seu comportamento quando seu estado interno muda. Parece como se o objeto mudasse de classe.

Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

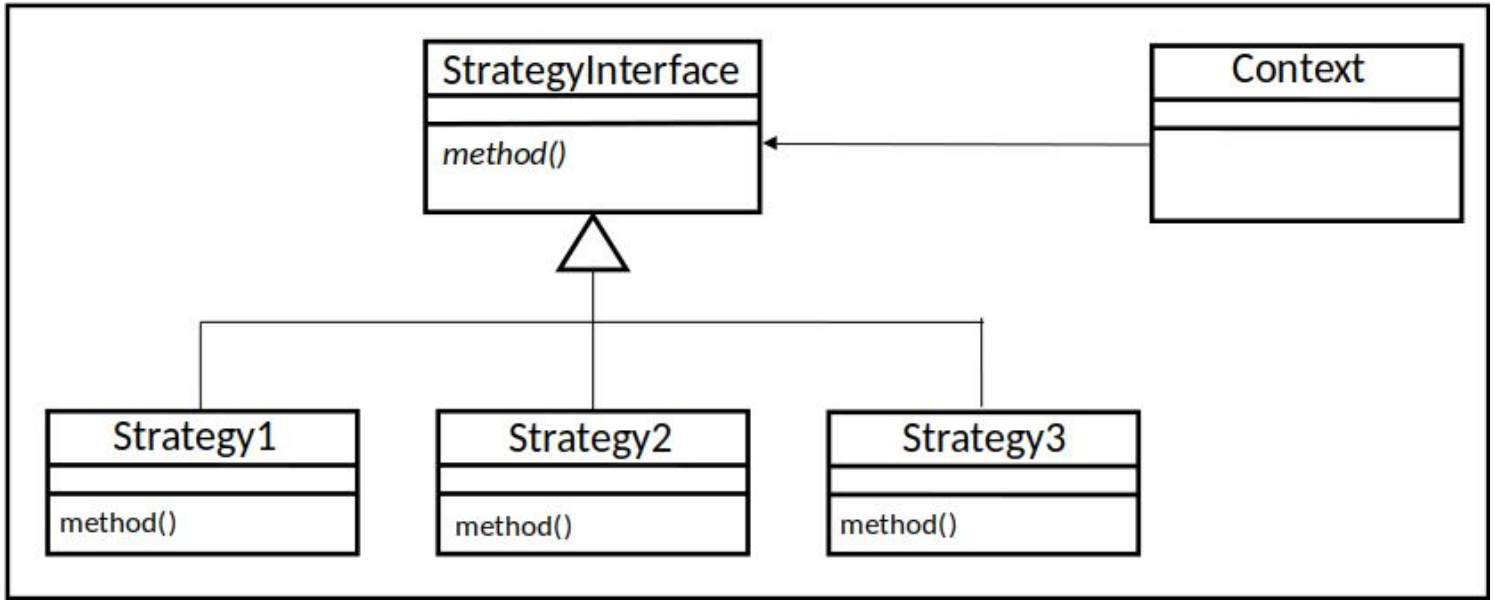
Strategy

Template Method

Strategy

O Strategy é um padrão de projeto comportamental que permite que você defina uma família de algoritmos, coloque-os em classes separadas, e faça os objetos deles intercambiáveis.

Strategy


StrategyInterface
method()
Context
Strategy1
method()
Strategy2
method()
Strategy3
method()

Strategy

Imagine um contexto em que um array deve ser ordenado de acordo com o algoritmo escolhido por um usuário

```
class Context {
    public void sortList(String type, int[] array){
        if (type.equals("quick"))
            quickSort(array);
        else
            if (type.equals("merge"))
                mergeSort(array);
            else
                if (type.equals("bubble"))
                    bubbleSort(array);
                else
                    System.out.println("invalid type");
    }
}
```

Strategy

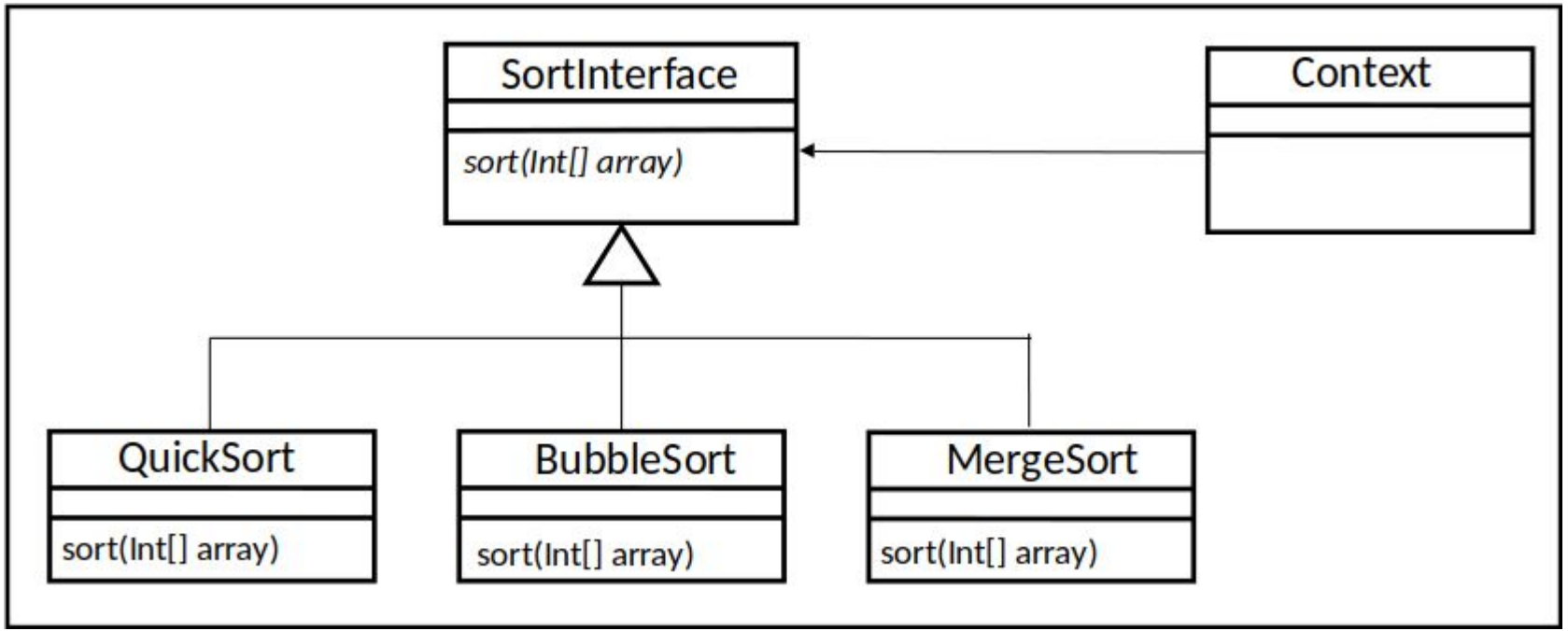
Imagine um contexto em que um array deve ser ordenado de acordo com o algoritmo escolhido por um usuário

```
class Context {
    public void sortList(String type, int[] array){
        if (type.equals("quick"))
            quickSort(array);
        else
            if (type.equals("merge"))
                mergeSort(array);
            else
                if (type.equals("bubble"))
                    bubbleSort(array);
                else
                    System.out.println("invalid type");
    }
}
```

A cada nova estratégia é preciso acrescentar mais opções para escolher o tipo de ordenação, e alterar classes afetadas por isso

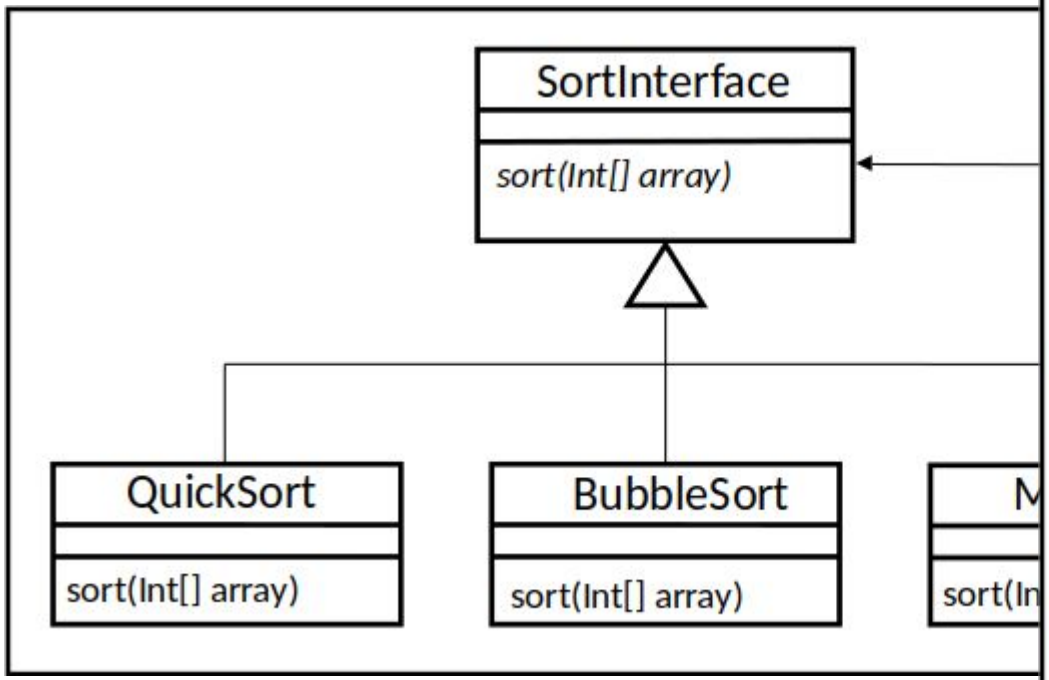
Strategy

Solução do padrão Strategy



Strategy

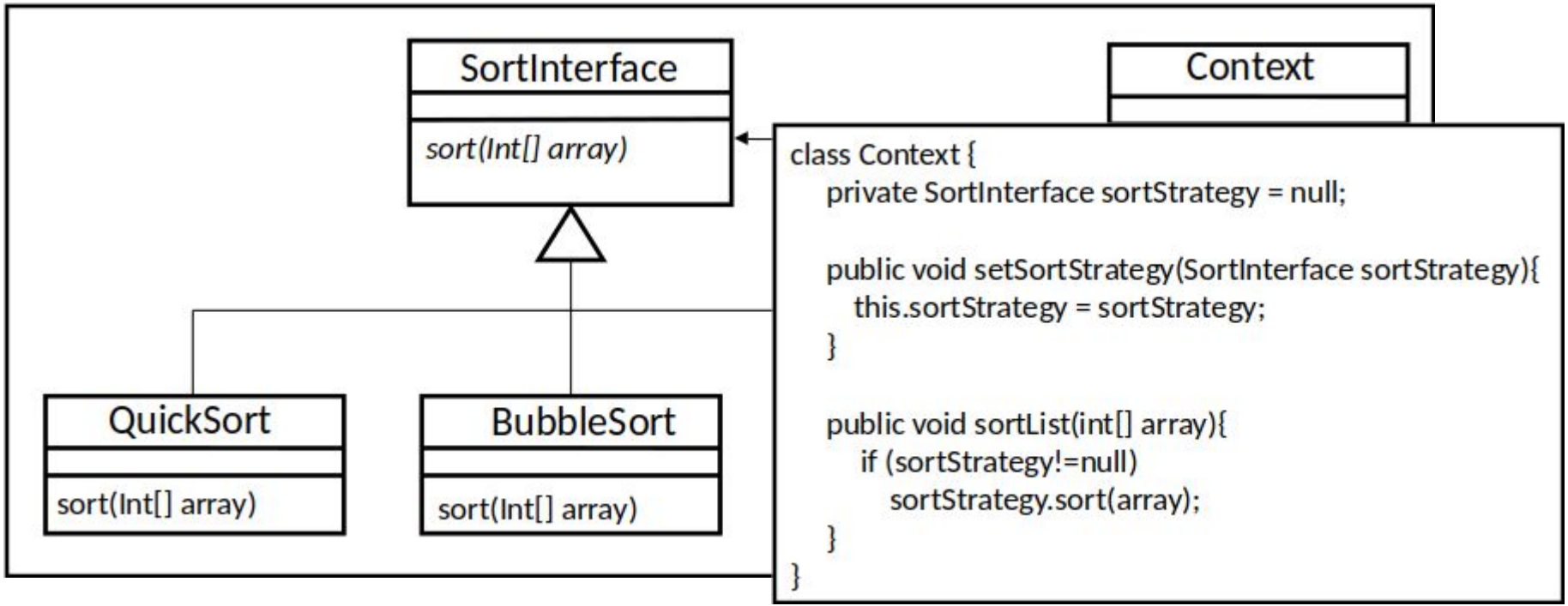
Solução do padrão Strategy




```
interface SortInterface {
   public void sort(int[] array);
}

class QuickSort implements SortInterface {
     public void sort(int[] array){
      ...
     }
}

class BubbleSort implements SortInterface {
     public void sort(int[] array){
      ...
     }
}

class MergeSort implements SortInterface {
     public void sort(int[] array){
      ...
     }
}
```

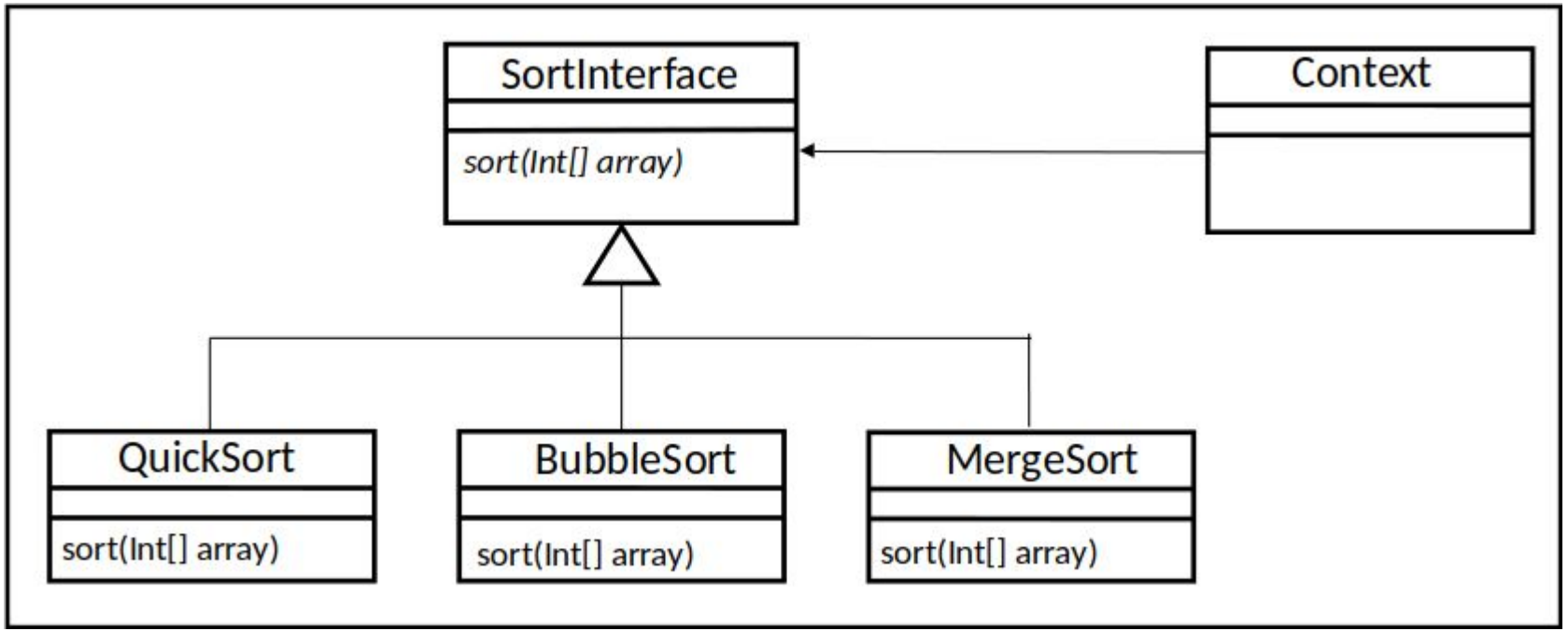
Strategy

Solução do padrão Strategy



Strategy

Solução do padrão Strategy



Padrões Comportamentais

Chain of Responsibility

Command

Interpreter

Iterator

Visitor

Mediator

Memento

Observer

State

Strategy

Template Method

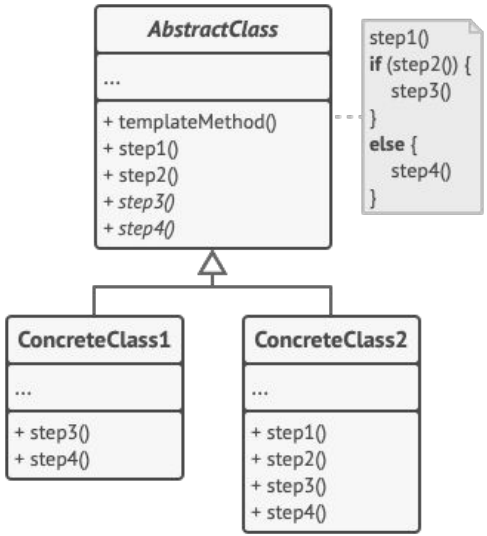
Template Method

O Template Method é um padrão de projeto comportamental que define o esqueleto de um algoritmo na superclasse, mas deixa as subclasses sobrescreverem etapas específicas do algoritmo sem modificar sua estrutura.

https://refactoring.guru/pt-br/design-patterns/template-method

Template Method

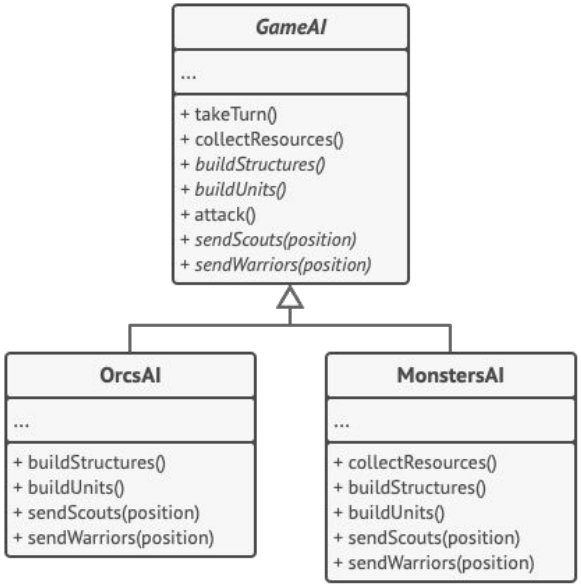
Estrutura padrão:



https://refactoring.guru/pt-br/design-patterns/template-method

Template Method

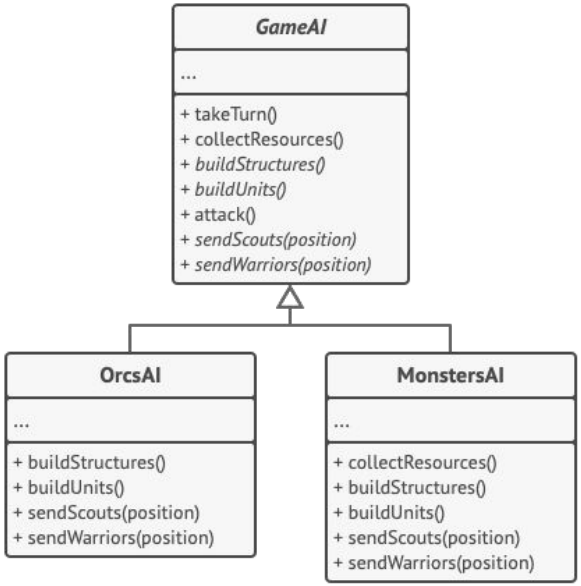
Exemplo:



https://refactoring.guru/pt-br/design-patterns/template-method

Template Method

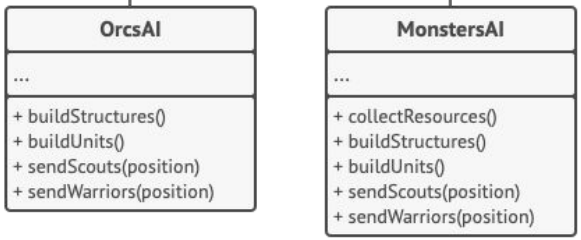
Exemplo:




```
class GameAI is
    method turn() is
        collectResources()
        buildStructures()
        buildUnits()
        attack()

    method collectResources() is
        foreach (s in this.builtStructures) do
            s.collect()

    abstract method buildStructures()
    abstract method buildUnits()

    method attack() is
        enemy = closestEnemy()
        if (enemy == null)
            sendScouts(map.center)
        else
            sendWarriors(enemy.position)

    abstract method sendScouts(position)
    abstract method sendWarriors(position)
```

https://refactoring.guru/pt-br/design-patterns/template-method

```
class OrcsAI extends GameAI is
    method buildStructures() is
        if (there are some resources) then
            // Construir fazendas, quartéis, etc

    method buildUnits() is
        if (there are plenty of resources) then
            if (there are no scouts)
                // Construir peão, adicionar ao grupo
            else
                // Construir um bruto, adicionar ao grupo

    method sendScouts(position) is
        if (scouts.length > 0) then
            // Enviar batedores para posição.

    method sendWarriors(position) is
        if (warriors.length > 5) then
            // Enviar guerreiros para posição.
```




```
class GameAI is
    method turn() is
        collectResources()
        buildStructures()
        buildUnits()
        attack()

    method collectResources() is
        foreach (s in this.builtStructures) do
            s.collect()

    abstract method buildStructures()
    abstract method buildUnits()

    method attack() is
        enemy = closestEnemy()
        if (enemy == null)
            sendScouts(map.center)
        else
            sendWarriors(enemy.position)

    abstract method sendScouts(position)
    abstract method sendWarriors(position)
```

https://refactoring.guru/pt-br/design-patterns/template-method

Atividade

Pesquisar e relatar exemplos de aplicação de três dos seis padrões de projeto que não foram apresentados em detalhes nesta aula:

- Bridge
- Chain of Responsibility
- Interpreter
- Visitor
- Memento
- State

Prazo para entrega: 21/10

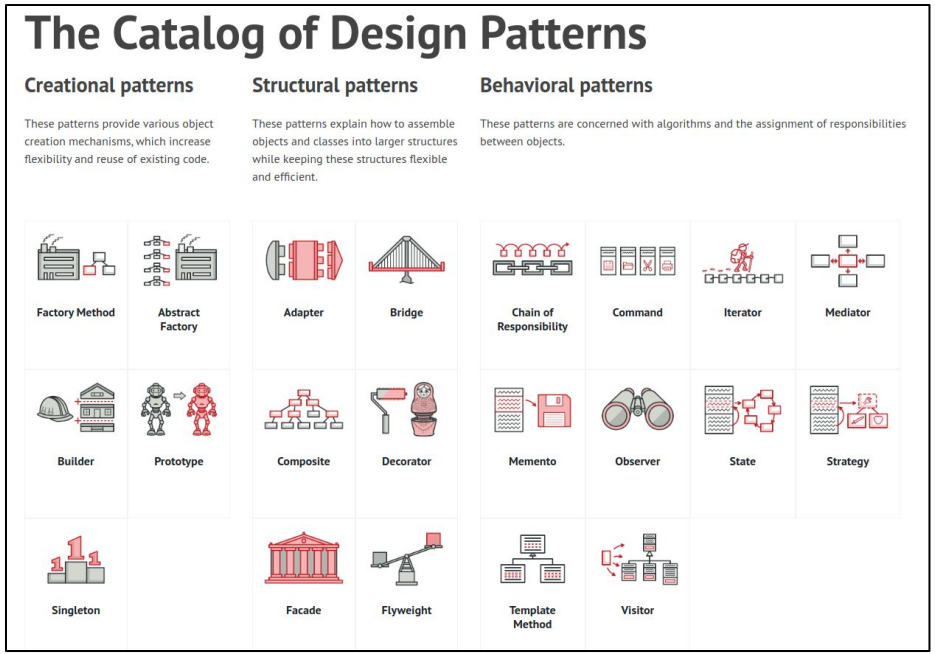
Trabalho individual

Leitura recomendada

- **Livro**: Engenharia de Software Moderna - Princípios e Práticas para Desenvolvimento de Software com Produtividade
- **Autor**: Marco Tulio Valente
- Capítulo 6 - Padrões de projeto (https://engsoftmoderna.info/cap6.html)

Leitura recomendada

- https://refactoring.guru/design-patterns/catalog



Referências

Arquitetura Limpa: O Guia do Artesão para Estrutura e Design de Software.Robert Martin. Alta Books, 2018.

Engineering Software as a Service: An Agile Approach Using Cloud Computing Second Edition. 2021. Armando Fox and David Patterson. Download gratuito: http://www.saasbook.info/

Engenharia de Software Moderna - Princípios e Práticas para Desenvolvimento de Software com Produtividade. Marco Tulio Valente. Livro online.

Utilizando UML e padrões. Craig Larman.

Padrões de Projetos: Soluções Reutilizáveis de Software Orientados a Objetos. Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides.

ACH 2028
Qualidade de Software

Aula 14 - Padrões de projeto (GoF)

Prof. Marcelo Medeiros Eler
marceloeler@usp.br

Escola de Artes, Ciências e Humanidades - Universidade de São Paulo