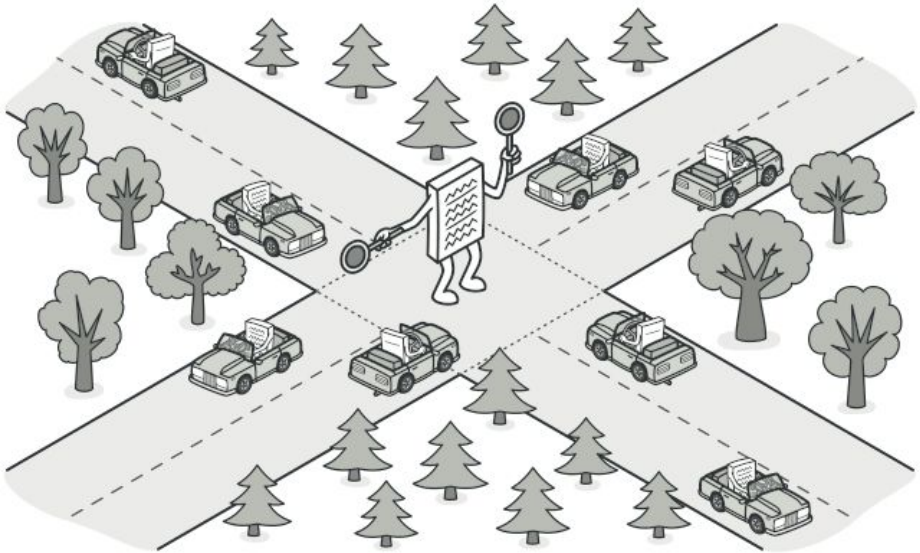
Padrões de Projeto

Mediator, Observer, Command, Bridge

Equipe: Antônio Nicassio, Cláudia Inês, Letícia Ribeiro

Padrão Mediator

- O Mediator é um padrão de projeto comportamental.
- O padrão restringe comunicações diretas entre objetos e os força a colaborar apenas através do objeto mediador.



Onde se aplica

- Programas onde há muita dependência entre os componentes.
- Exemplo: acionamento de botões. Um é acionado e habilita o outro.
- Classes interagindo umas com as outras de forma específica tendo o acoplamento muito forte, o que pode tornar a reutilização do objeto difícil.

Solução proposta

- O padrão Mediator sugere que você deveria cessar toda comunicação direta entre componentes que você quer tornar independentes um do outro.
- Componentes colaboram indiretamente através do mediador
- Como resultado, os componentes dependem apenas de uma única classe mediadora ao invés de serem acoplados a dúzias de outros colegas.

Diagrama - Sem uso do padrão

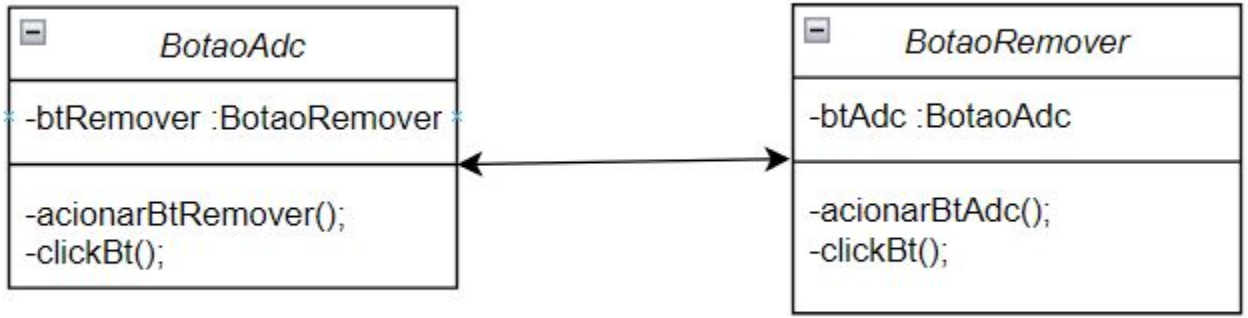
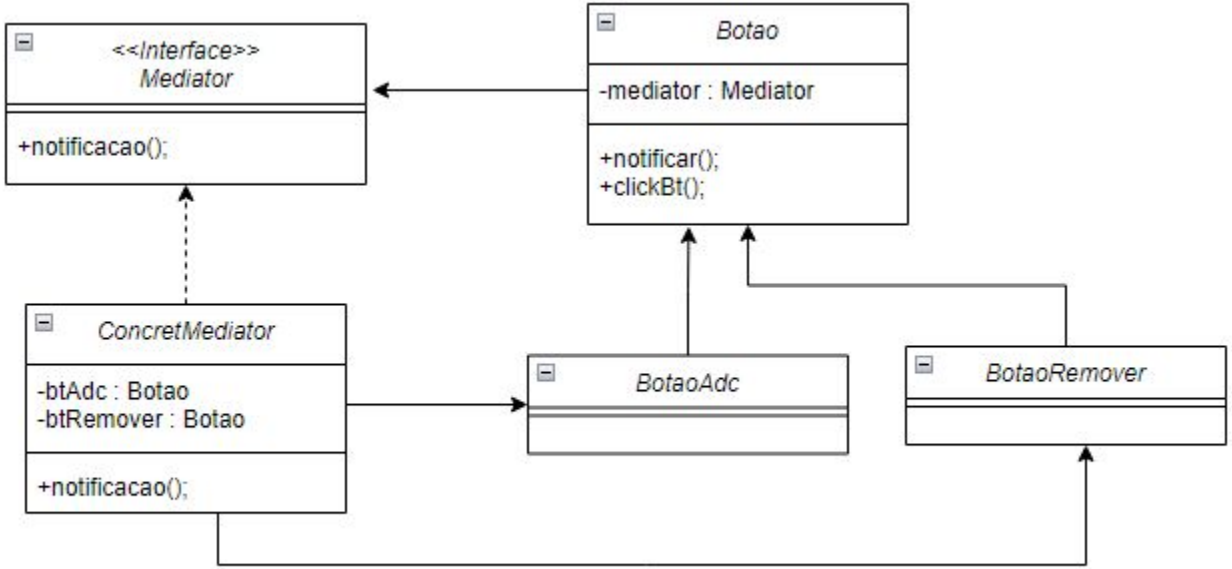


Diagrama - Com uso do padrão



Codificação - Sem o uso do padrão

```java
public class BotaoAdc {

    private BotaoRemover btRemover;

    public BotaoAdc() {

    }

    public BotaoRemover getBtRemover() {
        return btRemover;
    }

    public void setBtRemover(BotaoRemover btRemover) {
        this.btRemover = btRemover;
    }

    private void acionarBtRemover() {
        btRemover.ativarbt();
    }

    public void clickBt() {
        System.out.println("Botão adicionar clicado!");
        acionarBtRemover();
    }

    public void ativarBt() {
        System.out.println("Botão Adicionar ativado");
    }

}
```

```java
public class BotaoRemover {

    private BotaoAdc btAdc;

    public BotaoRemover() {

    }

    public BotaoAdc getBtAdc() {
        return btAdc;
    }

    public void setBtAdc(BotaoAdc btAdc) {
        this.btAdc = btAdc;
    }

    private void acionarBtAdc() {
        btAdc.ativarBt();
    }

    public void clickBt() {
        System.out.println("Botão adicionar clicado!");
        acionarBtAdc();
    }

    public void ativarbt() {
        System.out.println("Botão remover ativado");

    }

}
```

Codificação - Com uso do padrão

```java
public class Botao {
    public Mediator mediator;

    public Botao(mediatorConcret mediator) {
        super();
        this.mediator = mediator;
    }

    public Mediator getMediator() {
        return mediator;
    }

    public void setMediator(mediatorConcret mediator) {
        this.mediator = mediator;
    }

}
```

```java
public interface Mediator {
    public void notificacao(Botao c);
}
```

```java
public class mediatorConcret implements Mediator{

    private ButtonAdc btAdc;
    private ButtonRemove btRemove;

    public mediatorConcret() {

    }

    public void notificacao(Botao c) {
        if(c==btAdc) {
            btRemove.ativarBt();
        }
        if(c==btRemove) {
            btAdc.ativarBt();
        }

    }

    public ButtonAdc getBtAdc() {
        return btAdc;
    }

    public void setBtAdc(ButtonAdc btAdc) {
        this.btAdc = btAdc;
    }

    public ButtonRemove getBtRemove() {
        return btRemove;
    }

    public void setBtRemove(ButtonRemove btRemove) {
        this.btRemove = btRemove;
    }

}
```

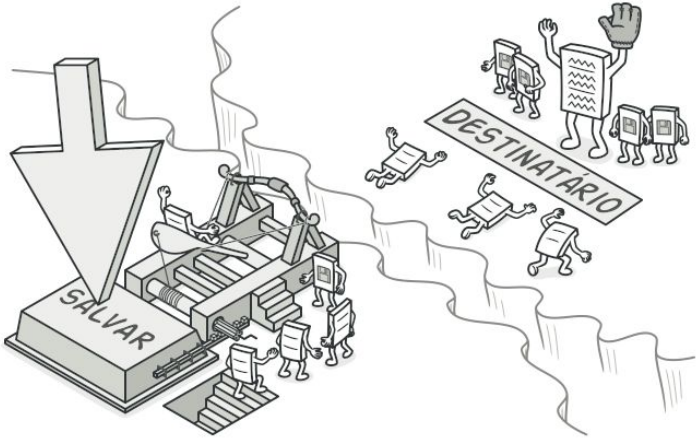
Codificação - Com uso do padrão

```java
public class ButtonRemove extends Botao {


    public ButtonRemove(mediatorConcret mediator) {
        super(mediator);

    }

    public void onBtRemove() {
        mediator.notificacao(this);
    }

    public void ativarBt() {
        System.out.println("Botão remove ativo");
    }

    public void clickBt() {
        System.out.println("Botão remove clicado!");
        onBtRemove();
    }

}
```

```java
public class ButtonAdc extends Botao {


    public ButtonAdc(mediatorConcret mediator) {
        super(mediator);
    }

    public void onBtAdc() {
        mediator.notificacao(this);
    }

    public void clickBt() {
        System.out.println("Botão adicionar clicado!");
        onBtAdc();

    }

    public void ativarBt() {
        System.out.println("Botão adicionar ativo!");
    }


}
```

Padrão Command

O Command é um padrão de projeto comportamental que tem como objetivo encapsular uma solicitação como um objeto, o que permite parametrizar outros objetos com diferentes solicitações, enfileirar ou registrar solicitações e implementar recursos de cancelamento de operações (desfazer). Ou seja, o objetivo do padrão é transformar um método de uma classe em um objeto, o qual pode executar a ação deste método.



Onde se aplica

- O Command é mais utilizado em programas em que você parametriza elementos da interface com ações, um exemplo disso são menus e controles em que que tem botões que realizam determinada ação
- Quando você quer desacoplar o objeto que envia a solicitação do objeto que a recebe
- Quando você quer tratar um comando como um objeto
- Quando você quer que solicitações sejam feitas e desfeitas

Solução proposta pelo padrão

- O padrão tem como solução transformar um método de uma classe em um objeto
- A partir disso este novo objeto pode realizar a ação ou método que o antigo objeto deve realizar

Diagrama - Sem uso do padrão

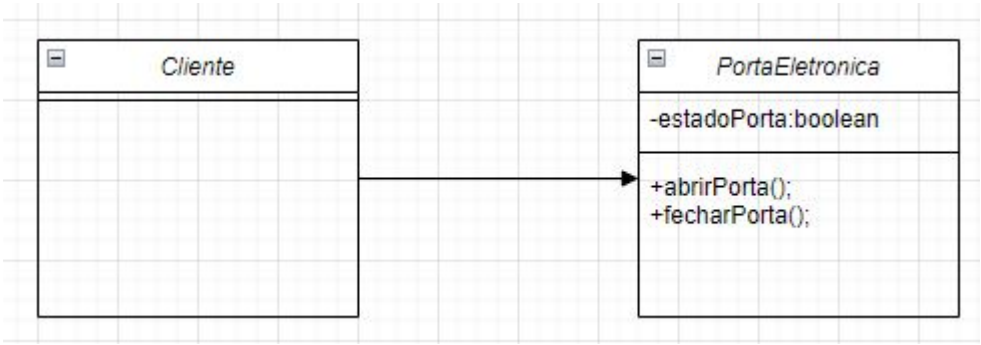
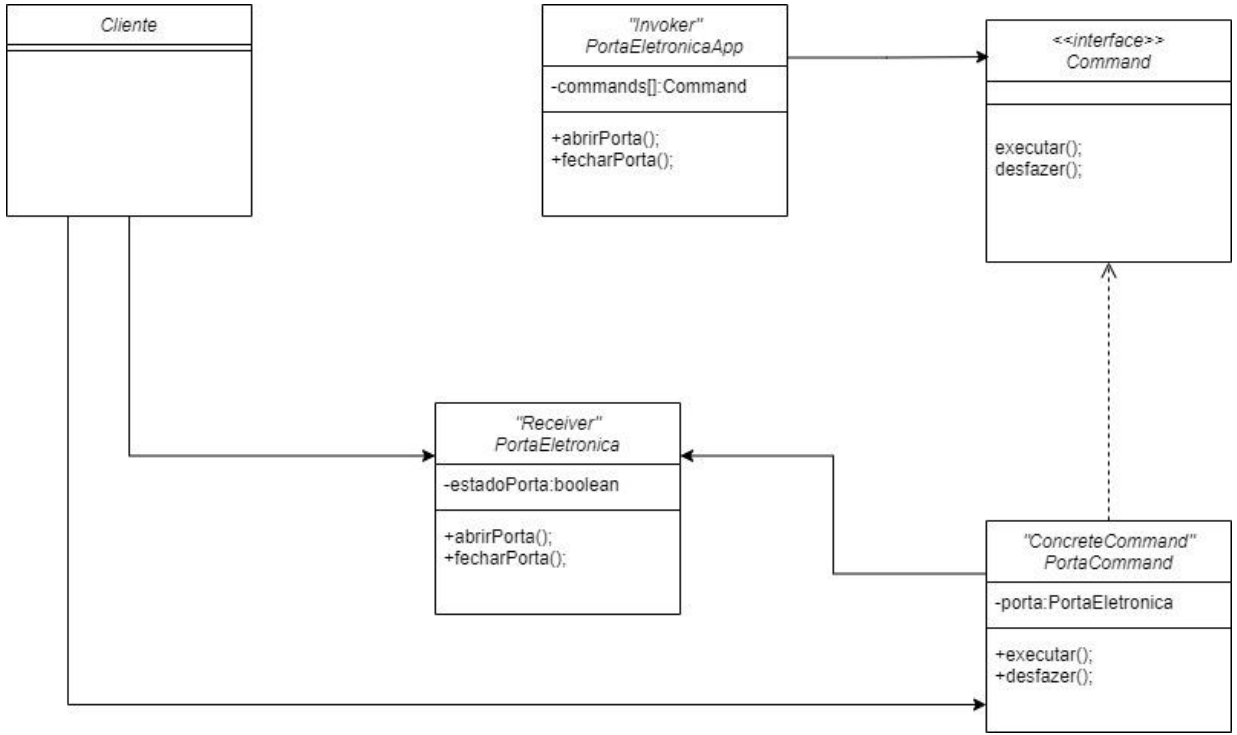


Diagrama - Com uso do padrão



Codificação - Sem o uso do padrão

```java
package PortaEletronica;

public class PortaEletronica {

    private boolean estadoPorta = false;

    public PortaEletronica() {
        this.estadoPorta = false;
    }


    public boolean abrirPorta() {
        this.estadoPorta = true;
        System.out.println("A porta está aberta!!");
        return this.estadoPorta;
    }

    public boolean fecharPorta() {
        this.estadoPorta = false;
        System.out.println("A porta está fechada!!");
        return this.estadoPorta;
    }
}
```

```java
package PortaEletronica;

public class Main {

    public static void main(String[] args) {


        PortaEletronica porta = new PortaEletronica();

        porta.abrirPorta();
        porta.fecharPorta();


    }
}
```

Codificação - Com o uso do padrão

```java
package PortaEletronica;

public class PortaEletronica {

    private boolean estadoPorta = false;

    public PortaEletronica() {
        this.estadoPorta = false;
    }


    public boolean abrirPorta() {
        this.estadoPorta = true;
        System.out.println("A porta está aberta!!");
        return this.estadoPorta;
    }

    public boolean fecharPorta() {
        this.estadoPorta = false;
        System.out.println("A porta está fechada!!");
        return this.estadoPorta;
    }
}
```

Codificação - Com o uso do padrão

```java
package PortaEletronica;

public interface Command {

    public void executar();

    public void desfazer();
}
```

```java
package PortaEletronica;


public class PortaCommand implements Command {

    private PortaEletronica porta;

    public PortaCommand(PortaEletronica porta) {
        this.porta = porta;
    }

    public void executar() {
        porta.abrirPorta();
    }

    public void desfazer() {
        porta.fecharPorta();
    }
}
```

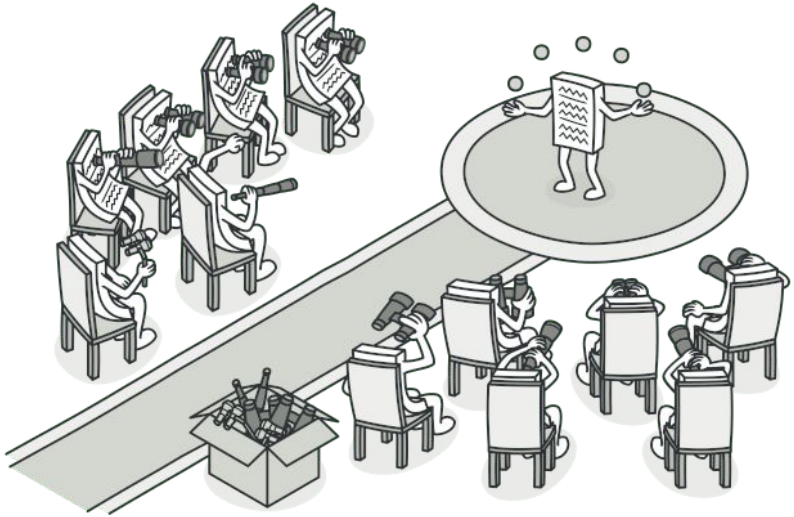
Codificação - Com o uso do padrão

```java
package PortaEletronica;

public class PortaEletronicaApp {

    private Command[] commands;


    public PortaEletronicaApp(Command Command) {
        this.commands = new Command[1];
        commands[0] = Command;
    }

    public void abrirPorta() {
        commands[0].executar();
    }

    public void fecharPorta() {
        commands[0].desfazer();
    }


}
```

```java
package PortaEletronica;

public class Main {

    public static void main(String[] args) {

        //Receiver
        PortaEletronica porta = new PortaEletronica();

        //Command
        PortaCommand portaComando = new PortaCommand(porta);

        //Controle - Invoker
        PortaEletronicaApp portaApp = new PortaEletronicaApp(portaComando);

        portaApp.abrirPorta();
        portaApp.fecharPorta();


    }
}
```

Padrão Observer

Padrão de projeto comportamental que permite definir um mecanismo de assinatura para notificar múltiplos objetos sobre quaisquer eventos que aconteçam com o objeto que eles estão observando.



Onde se aplica

- Mudança de estado de um objeto pode precisar alterar outros objetos.
- É possível aplicar o Observer sempre que a alteração de um objeto fará com que outros se alterem também.
- Exemplo: Loja e Cliente - Envio de notificação de chegada de produtos novos.

Solução proposta pelo padrão Observer

- Publicador e assinante.
- O padrão Observer sugere que você adicione um mecanismo de assinatura para a classe publicadora para que objetos individuais possam assinar ou desassinar uma corrente de eventos vindo daquela publicadora.

Diagrama - Sem uso do padrão

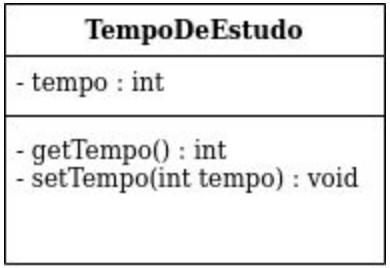
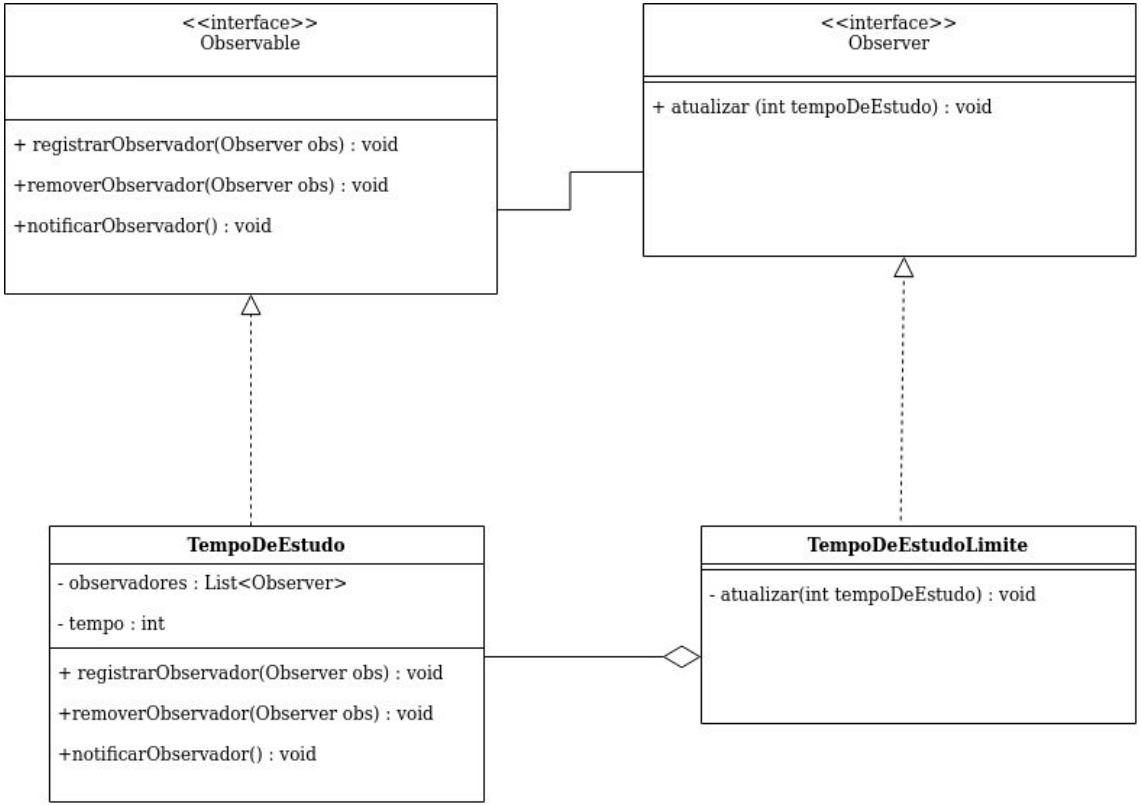


Diagrama - Com uso do padrão



Codificação - Sem o uso do padrão

```java
public class TempoDeEstudo{


    private int tempo;


    public int getTempo() {
        return tempo;
    }

    public void setTempo(int tempo) {
        this.tempo = tempo;
    }
```

```java
import java.util.Scanner;

public class main {

    public static Scanner entrada = new Scanner(System.in);

    public static void main(String args[]) {

        TempoDeEstudo tempo = new TempoDeEstudo();
        int minutos;


        System.out.println("Digite a quanto tempo você está estudando em minutos:");
        minutos = entrada.nextInt();
        entrada.nextLine();

        tempo.setTempo(minutos);

        if (tempo.getTempo() >= 60) {
            System.out.println("Você está a mais de 1 hora estudando, "
                    + "\n recomendo que dê uma pausa");
        }

    }


}
```

Codificação - Com o uso do padrão

```java
public interface Observable {

    public void registrarObservador(Observer observador);
    public void remomverObservador(Observer observador);
    public void notificarObservadores();

}
```

```java
public class TempoDeEstudo implements Observable{

    private List<Observer>observadores = new ArrayList<>();
    private int tempo;


    public int getTempo() {
        return tempo;
    }

    public void setTempo(int tempo) {
        this.tempo = tempo;
        this.notificarObservadores();
    }

    @Override
    public void registrarObservador(Observer observador) {
        observadores.add(observador);

    }

    @Override
    public void remomverObservador(Observer observador) {
        observadores.remove(observador);

    }

    @Override
    public void notificarObservadores() {

        for (Observer ob : observadores) {
            System.out.println("Notificando observadores");
            ob.atualizar(this.tempo);
        }
```

Codificação - Com o uso do padrão

```java
public interface Observer {

    public void atualizar(int tempoDeEstudo);

}
```

```java
public class TempoDeEstudoLimite implements Observer{

    @Override
    public void atualizar(int tempoDeEstudo) {

        if (tempoDeEstudo >= 60) {
            System.out.println("Você está a mais de 1 hora estudando, "
                    + "\n recomendo que dê uma pausa");
        }

    }
```

Codificação - Com o uso do padrão

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

        TempoDeEstudoLimite observador = new TempoDeEstudoLimite();
        TempoDeEstudo observado = new TempoDeEstudo();
        observado.registrarObservador(observador);
        observado.setTempo(60);
    }
```

Padrão Bridge

Padrão de projeto estrutural que permite dividir 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.

Onde se aplica

- Renderização gráfica em diferentes plataformas
- Conexão a banco de dados JDBC (Java™ EE Database Connectivity)
- Aplicações multi-plataforma
- Quando você quer dividir e organizar uma classe monolítica que tem diversas variantes da mesma funcionalidade (por exemplo, se a classe pode trabalhar com diversos servidores de base de dados).

Solução proposta

Propõe utilizar composição de objetos. Isso significa que você extrai uma das dimensões em uma hierarquia de classe separada, para que as classes originais referenciem um objeto da nova hierarquia, ao invés de ter todos os seus estados e comportamentos dentro de uma classe.

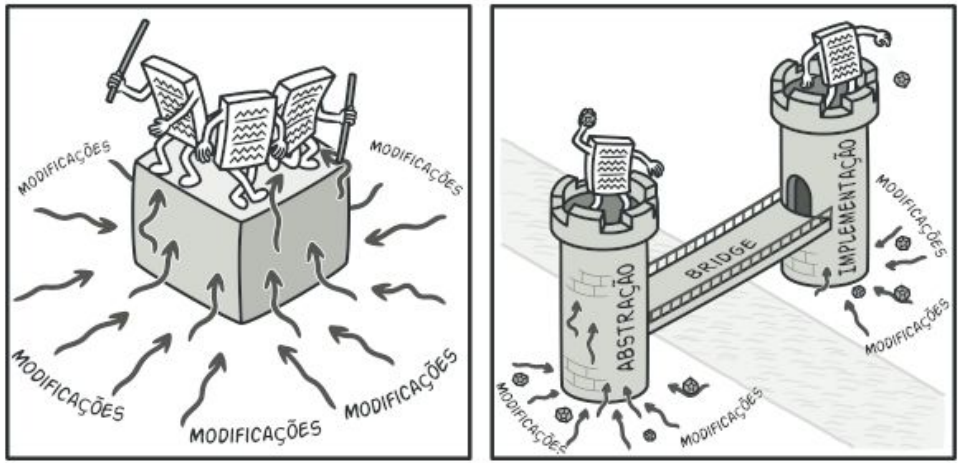


Diagrama - Sem uso do padrão

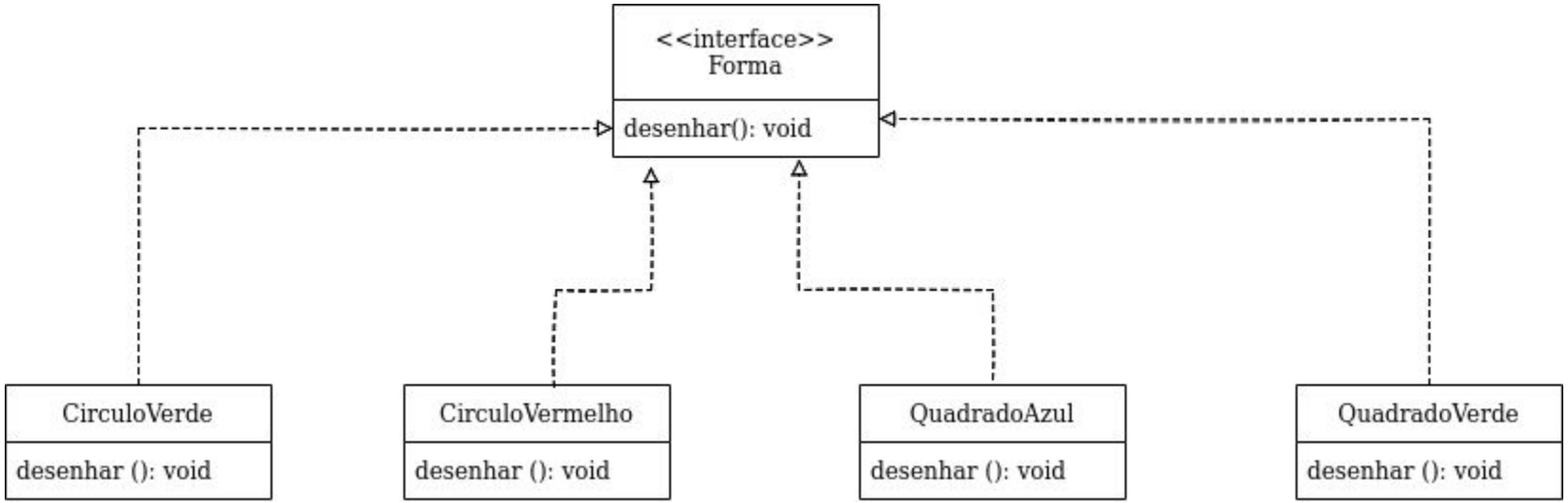
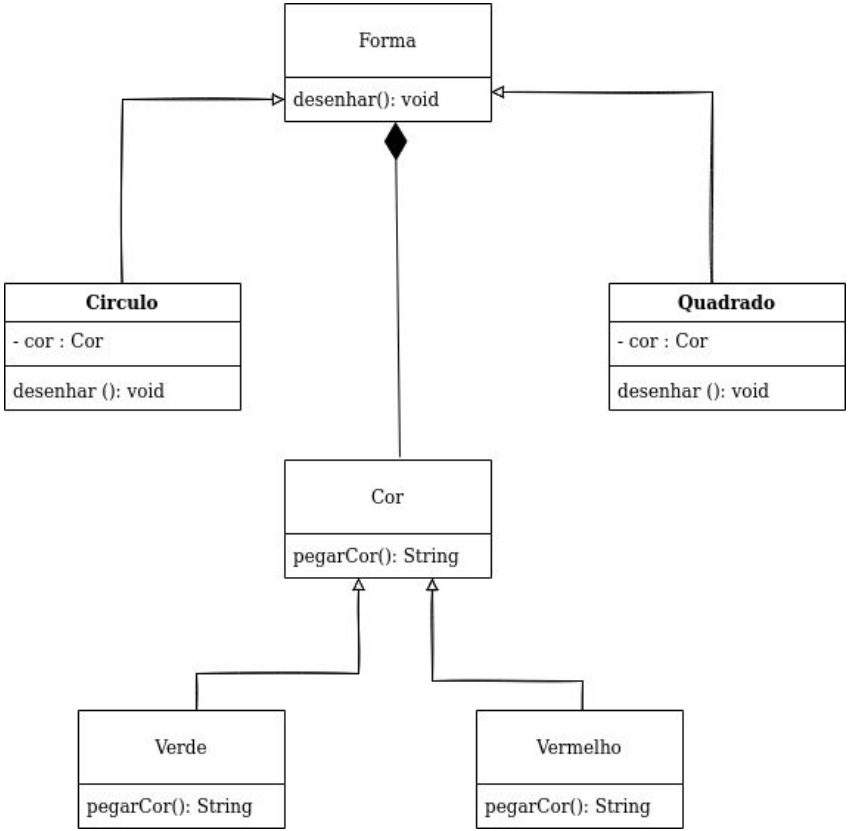


Diagrama - Com uso do padrão



Codificação - Sem uso do padrão

```java
public interface Forma {
    public void desenhar();
}
```

```java
public class CirculoVermelho implements Forma {
    public void desenhar() {
        System.out.println("Desenhando circulo vermelho");
    }
}
```

```java
public class QuadradoAzul implements Forma {
    public void desenhar() {
        System.out.println("Desenhando quadrado azul");
    }
}
```

```java
    public static void main(String args[]) {
        QuadradoAzul qa = new QuadradoAzul();
        qa.desenhar();

        QuadradoVerde qv = new QuadradoVerde();
        qv.desenhar();

        CirculoVerde ca = new CirculoVerde();
        ca.desenhar();

        CirculoVermelho cv = new CirculoVermelho();
        cv.desenhar();
    }
```

Codificação - Com uso do padrão

```java
public abstract class Forma {
    public abstract void desenhar();
}

public abstract class Cor {
    public abstract String pegarCor();
}
```

```java
public class Quadrado extends Forma {
    private Cor cor;

    public Quadrado(Cor cor) {
        this.cor = cor;
    }

    public void desenhar() {
        System.out.println("Desenhando um quadrado " + this.cor.pegarCor());
    }
}
```

```java
public class Vermelho extends Cor {
    public String pegarCor() {
        return "vermelho";
    }
}
```

Codificação - Com uso do padrão

```
public static void main(String args[]) {
    Circulo circuloVerde = new Circulo(new Verde());
    circuloVerde.desenhar();

    Circulo circuloVerm = new Circulo(new Vermelho());
    circuloVerm.desenhar();

    Quadrado quadradoVerde = new Quadrado(new Verde());
    quadradoVerde.desenhar();

    Quadrado quadradoVerm = new Quadrado(new Vermelho());
    quadradoVerm.desenhar();
}
```

Referências

GURU, Refactoring. **Observer**. Disponível em: https://refactoring.guru/pt-br/design-patterns/observer. Acesso em: 20 jun. 2022.

GURU, Refactoring. **Bridge**. Disponível em: https://refactoring.guru/pt-br/design-patterns/bridge. Acesso em: 04 jul. 2022.

GURU, Refactoring. **Mediator**. Disponível em: https://refactoring.guru/pt-br/design-patterns/mediator#:~:text=O%20padr%C3%A3o%20Mediator%20sugere%20que,chamadas%20para%20os%20componentes%20apropriados.. Acesso em: 20 jun. 2022.

DEVMEDIA. **Padrão de Projeto Observer em Java**. Disponível em: https://www.devmedia.com.br/padrao-de-projeto-observer-em-java/26163#:~:text=O%20padr%C3%A3o%20Observer%20%C3%A9%20utilizado,atualizados%20quando%20algo%20importante%20ocorre.. Acesso em: 20 jun. 2022.

DEVMEDIA .**Padrão de Projeto Command em Java**. Disponível em:

<https://www.devmedia.com.br/padrao-de-projeto-command-em-java/26456>. Acesso em: 27 jun. 2022.

CSIUNEB. **Command - Padrões de Projeto - Design Patterns**. Disponível em: <http://www.csi.uneb.br/padroes_de_projetos/command_2.html>. Acesso em: 27 jun. 2022.

Padrões em Java: Observer. Disponível em: <http://www.linhadecodigo.com.br/artigo/3643/padroes-em-java-observer.aspx>. Acesso em: 4 jul. 2022.