Exemplo e Descrição de Padrões de Projetos

Padrões de Projetos ou **Design Patterns** são arquiteturas testadas para construir Softwares orientados a objetos flexíveis e sustentáveis. Os padrões ajudam a reduzir substancialmente a complexidade do processo de Design.

Surgiu graças a um Arquiteto chamado `Christopher Alexander`, onde ele construiu 23 padrões de projetos orientado a software.

Design Patterns são uma coleção de padrões de projetos de software que contém soluções para problemas conhecidos e recorrentes no desenvolvimento de software descrevendo uma solução comprovada para um problema de projeto recorrente. Auxilia no reuso de software também.

Singleton

- Permite criar objetos únicos para os quais há apenas uma instância.
- Oferece um ponto de acesso global.
- Singleton é necessário apenas dominar bem as variáveis e métodos de classe estáticos além dos modificadores de acesso.

Com o padrão Singleton uma Classe gerencia a própria instância dela além de evitar que qualquer outra Classe crie uma instância dela.

- Exemplo de implementação do Padrão Singleton:

```
import java.util.ArrayList;

public class CarrinhoCompras {
    private static CarrinhoCompras unicoCarrinho;
    private ArrayList<String> produtosGuardados = new ArrayList<>();

    private CarrinhoCompras(){}

    public static synchronized CarrinhoCompras getInstance(){
        if(unicoCarrinho == null){
            unicoCarrinho = new CarrinhoCompras();
        }
        return unicoCarrinho;
    }

    public void addProduto(String prod){
        produtosGuardados.add(prod);
    }

    public void getCarrinho(){
        System.out.println("\nCARRINHO:\n");
        for(String valor : produtosGuardados){
```

```
                System.out.println("Produto Adicionado: " + valor);
            }
        }
    }
```

- Utiliza-se o `synchronized` para que o método não possa ser usado por duas Threads ao mesmo tempo.
- O construtor é privado evitando que essa Classe seja instânciada fora dela, dessa forma só pode ser instânciado utilizando o Método público *getInstance()*. Como o *getInstance()* é estático ele pode ser chamado de outra classe sem precisar instanciar a classe Singleton. Caso a Classe já tenha sido instanciado o Atributo *uniqueInstance* não será nulo, assim vai retornar a única instância já criada.

1. **VANTAGENS**:
    1. Uma Classe pode ser instânciada e usada apenas quando necessário
    2. É definido um único ponto de acesso global sendo inclusive muito mais de gerenciar a criação e utilização da instância.

Observer

O padrão Observer é utilizado quando se precisa manter os objetos atualizados quando algo importante ocorre.

O Padrão Observer funciona como uma Assinatura de Jornal, onde tem uma Editora que publica as edições e as pessoas que assinam os jornais dessa editora recebem as novas edições assim que elas são Publicadas.

Nesse exemplo, a Editora no código é o **SUBJECT** e os Assinantes são os chamados **OBSERVER**

Os **OBSERVER**s registram-se no **SUBJECT** para receber atualizações quando os dados do **SUBJECT** são alterados. Os **OBSERVER**s também podem cancelar o seu registro e dessa forma não receber mais nenhuma atualização do **SUBJECT**.

Portanto existem MUITOS **OBSERVER**s para um único **SUBJECT**.

Em java, as APIs mais gerais possuem a interface Observer e a Classe Observable no pacote **java.util**.

A classe Observable possui os Seguintes Métodos:

Nome da Função	Uso
addObserver()	Adicina um Observer a um Subject
deleteObserver()	Deleta o Observer do Subject
notifyObservers()	Avisa os Observers de atualizações
setChanged()	Atualiza informações no Subject

Para usarmos essas Funções, devemos importar o **Observer** e o **Observable**:

```
import java.util.Observable;
import java.util.Observer;
```

A classe Observable nada mais faz do que monitorar todos os observadores e os notificar sobre alguma alteração no estado.

- **Exemplo de Implementação:**

Criando o Subject:

```
@Deprecated
class Subject extends Observable{
    CarrinhoCompras carrinho;
    String produto;

    public Subject(CarrinhoCompras car){
        carrinho = car;
    }

    public void setNovoProduto(String prod){
        produto = prod;
        carrinho.addProduto(produto);
        setChanged();
        notifyObservers();
    }

    public String getProduto(){
        return produto;
    }

    public void getProdutos(){
        carrinho.getCarrinho();
    }
}
```

Criando o Observer:

```
@Deprecated
class ObserverClient implements Observer{
    Observable subject;
    String produtoNovo;

    public ObserverClient(Observable sub){
        super();
        subject = sub;
        subject.addObserver(this);
    }

    @Override
```

```
    public void update(Observable sub, Object arg1){
        if(sub instanceof Subject){
            Subject subject = (Subject) sub;
            produtoNovo = subject.getProduto();
            System.out.println("Produto " + produtoNovo + " adicionado com
sucesso!");
        }
    }
}
```

Agora criamos a Classe principal, onde vão ser chamados o Subject e o Observer:

```
public class AtualizaCarrinho {
    public static void main(String args[]){
        String novoProduto = "PS5";
        Object none = null;
        CarrinhoCompras carrinho = CarrinhoCompras.getInstance();
        Subject subject = new Subject(carrinho);
        ObserverClient observer = new ObserverClient(subject);
        subject.setNovoProduto(novoProduto);
        observer.update(subject,none);
    }
}
```

Como é mostrado acima, o Subject é quem lida com o Singleton criado, onde é o carrinho do E-commerce, então o Object lida com o recebimento do Update do Subject

---

Strategy e Factory

Strategy serve para que você pegue uma classe que faz algo específico em diversas maneiras diferentes e extraia todos esses algoritmos para classes separadas chamadas *strategies*.

A classe original, chamada *conext* deve ter um campo para armazenar uma referência para uma dessas estratégias. O contexto delega o trabalho para um objeto estratégia ao invés de executá-lo por conta própria.

O padrão Strategy é muito comum no código Java. É frequentemente usado em várias estruturas para fornecer aos usuários uma maneira de alterar o comportamento de uma classe sem estendê-la.

- Implementação do Strategy:

Primeira coisa é definir uma Interface comum para todas as formas de pagamento

```
interface PagamentoEstrategia{
    boolean pago(int valorTotal);
    void detalhesPagamentos();
}
```

Cada Classe de Pagamento que são comuns vão ter que implementar os Métodos colocados na Interface, como por exemplo o Pagamento por paypal:

```
class PagamentoPaypal implements PagamentoEstrategia{
    private static final Map<String,String> database = new
HashMap<String,String>();
    private String username;
    private String password;

    public PagamentoPaypal(String nome,String senha){
        username = nome;
        password = senha;
    }

    public String getUsername(){
        return username;
    }

    public String getPassword(){
        return password;
    }

    @Override
    public void detalhesPagamentos(){
        database.put("Gabriel Fanto","1234");
        if(password.equals(database.get(username))){
            System.out.println("Verificação feita com sucesso!");
        }else{
            System.out.println("Usuário não encontrado");
        }
    }

    @Override
    public boolean pago(int valor){
        if(valor > 0){
            System.out.println("R$ " + valor + " pago em Paypal");
            return true;
        }else{
            return false;
        }
    }
}
```

E outra bem parecida que é para pagamento com Cartão de Crédito:

```
class PagamentoCartaoCredito implements PagamentoEstrategia{
    private static final Map<String,String> database = new
HashMap<String,String>();
```

```
    private String username;
    private String password;

    public PagamentoCartaoCredito(String nome,String senha){
        username = nome;
        password = senha;
    }

    public String getUsername(){
        return username;
    }

    public String getPassword(){
        return password;
    }

    @Override
    public void detalhesPagamentos(){
        database.put("Gabriel Fanto","1234");
        if(password.equals(database.get(username))){
            System.out.println("Verificação feita com sucesso!");
        }else{
            System.out.println("Usuário não encontrado");
        }
    }

    @Override
    public boolean pago(int valor){
        if(valor > 0){
            System.out.println("R$ " + valor + " pago no Cartão de
Crédito");
            return true;
        }else{
            return false;
        }
    }
}
```

Então dai iremos criar uma Classe Main para que o usuário selecione qual forma de pagamento e com isso sabermos qual Estratégia de Pagamento vai ser feito:

```
public class MetodosPagamento {
    private static PagamentoEstrategia estrategia;
    public static void main(String[] args){
        Scanner entrada = new Scanner(System.in);
        System.out.println("Selecione uma Forma de Pagamento: ");
        System.out.println("1 - Paypal");
        System.out.println("2 - Cartão de Crédito");
        int choice = entrada.nextInt();

        if(choice == 1){
```

```
            estrategia = new PagamentoPaypal("Gabriel Fanto", "1234");
        }else{
            if(choice == 2){
                estrategia = new PagamentoCartaoCredito("Gabriel
Fanto","1234");
            }else{
                System.out.println("Escolha não permitida!");
            }
        }
        entrada.close();

        //Saída
        estrategia.detalhesPagamentos();
        estrategia.pago(200);
    }
}
```

Facade

Facade oculta toda a complexidade de uma ou mais classes através de uma Fachada.

- O Facade fornece uma interface simplificada para uma Biblioteca, um Framework ou qualquer outro conjunto complexo de Classes.
- Todas as Classes e bibliotecas usadas serão chamadas e utilizadas dentro de um Facade, onde depois o Método criado nele vai ser chamado na Classe Main, como no exemplo abaixo:

Criando as Classes que serão utilizadas:

```
class Ecommerce{

    String nome;

    public Ecommerce(String nome){
        this.nome = nome;
    }

    public String getNomeEcommerce(){
        return nome;
    }
}

class Produto{
    String nome;

    public Produto(String nome){
        this.nome = nome;
    }

    public String getNomeProduto(){
```

```
        return nome;
    }
}

class Pedido{
    int quantidade;

    public Pedido(int quant){
        quantidade = quant;
    }

    public int getQuantidade(){
        return quantidade;
    }
}
```

Criamos um Facade que vai pegar todas as informações necessárias dessas classes:

```
class Facade{
    public void infos(String ecommerce,String produto, int quantidade){
        Ecommerce novoEcommerce = new Ecommerce(ecommerce);
        Produto novoProduto = new Produto(produto);
        Pedido novoPedido = new Pedido(quantidade);

        System.out.println("Ecommerce se chama " +
novoEcommerce.getNomeEcommerce());
        System.out.println("Produto se chama " +
novoProduto.getNomeProduto());
        System.out.println("Numero de Itens no Pedido são " +
novoPedido.getQuantidade());
    }
}
```

Depois, iremos chamar o Facade e a função que construimos dela na nossa Main:

```
public class FacadeEx {
    public static void main(String[] args){
        Facade testeFacade = new Facade();
        testeFacade.infos("Playstation","PS5",1);
    }
}
```

Exemplos

Todos os Exemplos completos dos mostrados acima se encontram no Repositório **https://github.com/F4NT0/Design_Patterns_Java**

Como testar

Acesse o Diretório *src/Exercício/* para usar os comando Abaixos

Design Pattern	**Comando de Terminal**
Singleton	`javac CarrinhoCompras.java Testes.java` e depois `java Testes`
Observer	`javac CarrinhoCompras.java AtualizaCarrinho.java` e depois `java AtualizaCarrinho`
Strategy	`javac MetodosPagamento.java` e depois `java MetodosPagamento`
Facade	`javac FacadeEx.java` e depois `java FacadeEx`