**UNIVERSIDADE LUTERANA DO BRASIL**

DIRETORIA ACADÊMICA

CURSO DE ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

PROJETO DE DESENVOLVIMENTO DE SOFTWARE

**Padrões de projeto criacionais**

ACADÊMICOS:

S. Antônio Jr, Emanoel Leffa Mittmann de Oliveira,

Joao Manoel Dias Pereira,

Maria Cicera Ferreira dos Santos Ribeiro,

Tiago Boff do Nascimento, Lorenzo Da Cunha Cardoso

ORIENTADOR: Vinícius Silveira Magnus



Torres, 2023

# INTRODUÇÃO

Neste presente trabalho falaremos sobre Design Patterns (Padrões de projeto) que são soluções encontradas para facilitar o uso em templates abstratos.

O Design Patterns não é aplicado somente no desenvolvimento de software, ele pode ser encontrado em várias áreas da vida, da arquitetura até a engenharia. Ele foi idealizado em 1970 pelo arquiteto Christopher Alexander para construir um vocabulário comum para discussões sobre design.

A origem do design patterns que prevalece hoje na arquitetura de software nasceu das experiências e conhecimentos dos programadores utilizando a programação orientada a objetos.

Eles pegaram 23 design patterns e organizaram em 3 grupos.

Creational Patterns (Padrões de Criação): Tratam da construção do objeto e o de referência;

Structural Patterns (Padrões Estruturais): Tratam da relação entre objetos e como eles interagem entre si para formarem grandes objetos complexos;

Behavioral Patterns (Padrões Comportamentais): Tratam da comunicação entre os objetos, especialmente em termos de responsabilidade e de algoritmos.

O design patterns são soluções que foram utilizadas e testadas, o que nos dá mais confiança na sua eficácia, elas focam na reutilização de soluções .

Depois de décadas de programação orientada a objetos, a maioria dos problemas que você encontrará já terão sido resolvidas no passado e haverá um pattern disponível para ajudar na implementação da solução.

# Design Patterns Criacionais

Os Padrões Criacionais dentro dos design patterns são soluções generalizadas para problemas comuns na construção de software, com a intenção de tornar o código mais reutilizável, e fazer com que a criação de novos objetos seja menos complexa e mais flexível de modo geral os patterns criacionais se concentram em torna mais eficaz a criação de objetos.

## Factory Method

O método fábrica ou em inglês “**Factory method**” é um padrão de desenvolvimento de projetos que pertence a categoria dos criacionais, isso porque

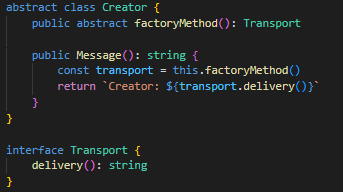
se trata de facilitar novas criações de objetos a partir de um contrato .

Esse método se torna adequado quando o programador não possui a informação de quais os tipos exatos e dependências dos objetos que se deve trabalhar. sendo assim se aplica esse padrão trabalhado o princípio de responsabilidade única, isso porque quando indicamos uma subclasse a ser responsável por apenas a criação de um tipo de produto, fazemos o desacoplamento entre a criação da classe, do uso dessa classe.

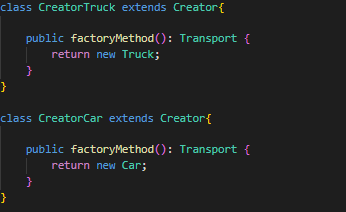
Bem como o princípio OCP sigla para “Open Close Principle” que na programação orientada a objetos entende por princípio aberto/fechado, no que diz respeito que entidades sejam elas classe, módulos ou funções devem ser abertas a extensão, mas fechadas para alteração.

No fim, o factory method define um contrato a ser seguido para criar um objeto, mas as subclasses definem qual objeto irá ser criado.

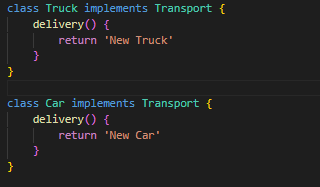
Definição de Classe base e Interface base

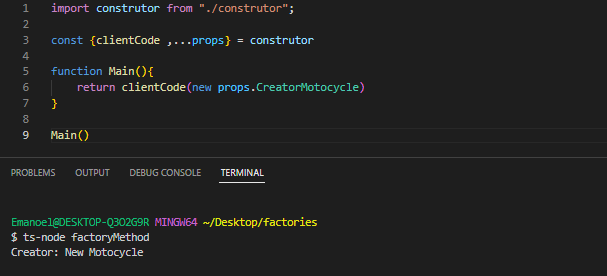


Classes herdeiras de Creator que Retornam o tipo de objeto que criam



Classes que Implementam os métodos de interface Base



Retorno de objeto criado

## Abstract Factory

O padrão de projeto Abstract Factory, é um padrão de projeto de software que fornece uma interface para criar famílias de objetos relacionados sem especificar suas classes concretas, ele permite que um cliente crie objetos sem precisar conhecer os detalhes de implementação desses objetos.

A ideia central do padrão abstract factory é fornecer uma abstração para criar família de objetos relacionados, onde cada fábrica concreta implementa essa abstração para criar objetos específicos de uma família, essa abstração é representada por uma interface ou classe abstrata conhecida como "fábrica abstrata”.

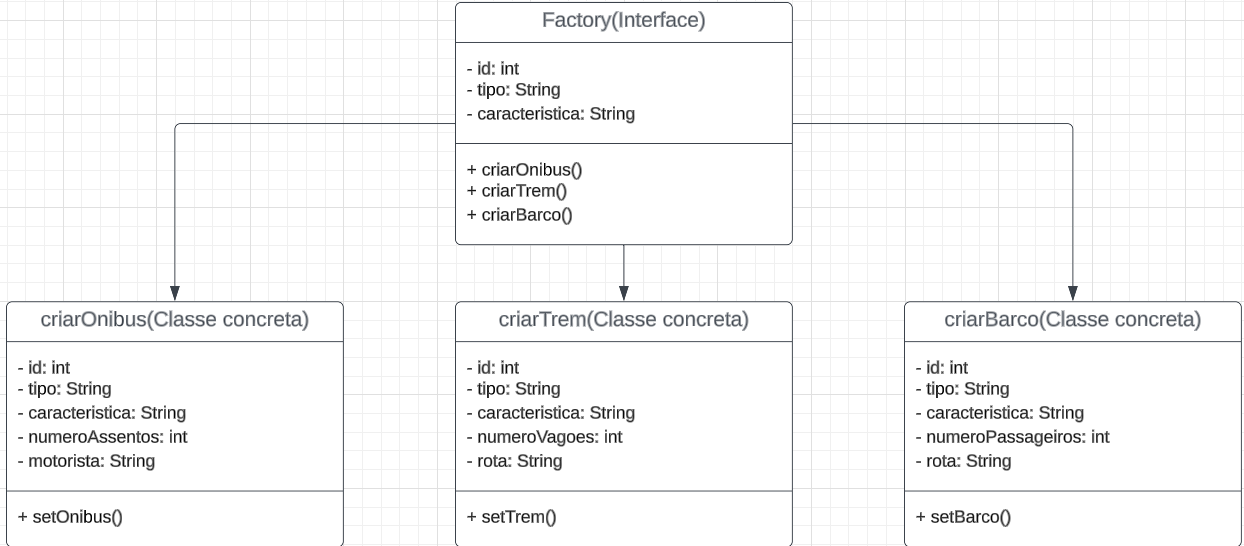
Imagine que você está projetando um sistema de gerenciamento de transporte público, o sistema precisa ser capaz de lidar com diferentes tipos de veículos, como ônibus, trens e barcos. Além disso, cada tipo de veículo pode ter características e comportamentos específicos.

Para resolver esse problema, você decide usar o padrão de design abstract factory. você cria uma interface chamada “Factory” que define métodos para criar diferentes tipos de veículos, como criarOnibus(), criaTrem() e criarBarco().

Em seguida, você implementa várias fábricas concretas que implementam essa interface. por exemplo, a fábrica de ônibus (concreta) tem um método criaOnibus() que retorna uma instância de classe Onibus. a fábrica de trem tem um metodos criarTrem() que retorna uma instância da classe trem, e assim por diante.

Dessa forma, você pode usar uma fábrica específica para criar objetos desse tipo. isso ajuda a abstrair a lógica de criação dos diferentes tipos de veículos e permite que você trabalhe com ele de maneira consistente.

Por exemplo, se você precisar adicionar um novo tipo de veículo, como bicicletas, basta criar uma nova fábrica de bicicletas que implemente a interface “Factory” e defina o método CriaBicicleta() para retornar uma instância da classe bicicleta.

****

## Builder

O padrão Builder, ou **Construtor,** é um padrão de projeto de software que oferece uma forma de construir objetos complexos passo a passo. Ele permite a criação de tipos e representações de construir um objeto usando o mesmo código de construção.

Uma implementação comum de padrão Builder envolve utilização de uma ou mais classes abstratas ou interfaces que define os métodos necessários para a construção do passo a passo do objeto. Essa abstração permite a criação de diferentes Builders que seguem a mesma estrutura, mas podem construir objetos de diferentes tipos.

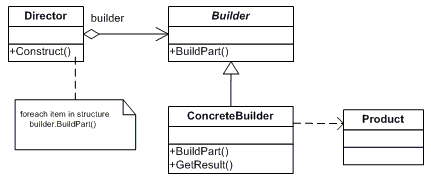
O principal objetivo do padrão Builder é separar a construção de um objeto de sua representação/estrutura, permitindo que o mesmo processo de construção possa criar diferentes representações do objeto.

Isso significa que você pode ter um objeto complexo, composto por várias partes e o Builder ajuda a construir essas partes de maneira flexível e independentes.

Cada Builder implementa a lógica para construir o objeto de acordo com a representação desejada. Dessa forma, você pode reutilizar o mesmo código de construção para criar objetos com diferentes características.

Uma vez que todas as partes do objeto tenham sido construídas pelos métodos do Builder, o objeto final pode ser obtido chamando um método específico, geralmente chamado de “getResult” ou similar. Esse método retorna o objeto completo e pronto para uso.

Modelo de **estrutura** de padrão Builder ilustrada abaixo:



(Estrutura do Padrão Builder. Fonte: [C# Builder Design Pattern - Dofactory](https://dofactory.com/net/builder-design-pattern#UML))

**Participantes**

As classes e objetos que participam desse padrão incluem:

**Construtor** (VehicleBuilder)

* especifica uma interface abstrata para criar partes de um objeto Product

**Construtor de Concreto** (MotorCycleBuilder, CarBuilder, ScooterBuilder)

* constrói e monta partes do produto implementando a interface do Builder
* define e mantém o controle da representação que ele cria
* fornece uma interface para recuperar o produto

**Diretor**  (Shop)

* constrói um objeto usando a interface do Builder

**Produto**  (Vehicle)

* representa o objeto complexo em construção. O ConcreteBuilder constrói o produto interno representação e define o processo pelo qual é montada
* inclui classes que definem as partes constituintes, incluindo interfaces para montagem das peças no resultado final

**Padrão Builder em java**

Classe Carro

public class Carro {

private String marca;

private String modelo;

private int ano;

private int potencia;

public Carro(String marca, String modelo,

int ano, int potencia) {

this.marca = marca;

this.modelo = modelo;

this.ano = ano;

this.potencia = potencia;

}

public String getMarca() {

return marca;

}

public String getModelo() {

return modelo;}

public int getAno() {

return ano;

}

public int getPotencia() {

return potencia;

}}

Interface CarroBuilder

public interface CarroBuilder {

CarroBuilder setMarca(String marca);

CarroBuilder setModelo(String modelo);

CarroBuilder setAno(int ano);

CarroBuilder setPotencia(int potencia);

Carro build();}

Classe CarroBuilderImplements

public class CarroBuilderImplents implements CarroBuilder {

private String marca;

private String modelo;

private int ano;

private int potencia;

public CarroBuilder setMarca(String marca) {

this.marca = marca;

return this;

}

public CarroBuilder setModelo(String modelo) {

this.modelo = modelo;

return this;

}

public CarroBuilder setAno(int ano) {

this.ano = ano;

return this;

}

public CarroBuilder setPotencia(int potencia) {

this.potencia = potencia;

return this;

}

public Carro build() {

return new Carro(marca, modelo, ano, potencia);

}}

Exemplo de uso padrão Builder

public class Main {

public static void main(String[] args) {

CarroBuilder builder = new CarroBuilderImplents();

Carro carro = builder

.setMarca("Ford")

.setModelo("Mustang")

.setAno(2022)

.setPotencia(450)

.build();

System.*out*.println("Marca: " + carro.getMarca());

System.*out*.println("Modelo: " + carro.getModelo());

System.*out*.println("Ano: " + carro.getAno());

System.*out*.println("Potência: " + carro.getPotencia());

}}

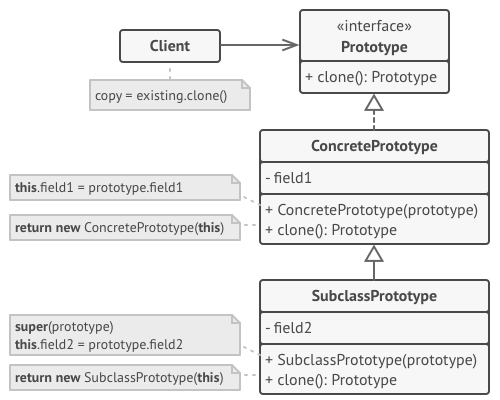
## Prototype

O padrão Prototype, ou **Protótipo**, é um padrão de projeto de software que permite criar cópias de objetos existentes de forma eficiente. Em vez de criar um novo objeto do zero, o padrão Prototype utiliza um objeto existente como modelo e cria novas instâncias a partir dele.Para utilizar o padrão Prototype, você precisa ter uma classe base que implemente a interface "Cloneable" ou forneça um método de clonagem personalizado. Esse objeto base é chamado de protótipo. Quando você precisa criar uma nova instância, em vez de criar um objeto do zero, você simplesmente clona o protótipo e faz as modificações necessárias na cópia.

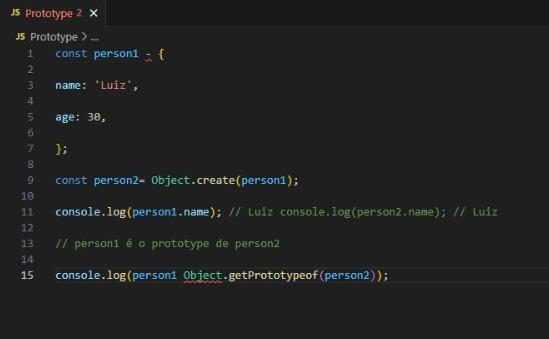
O padrão Prototype é útil quando a criação de um objeto é cara ou complexa, e você deseja evitar a duplicação de código e o processamento adicional necessário para criar um novo objeto. Ele permite criar objetos similares rapidamente, apenas clonando um protótipo existente.

Além disso, o padrão Prototype também ajuda a separar a criação de objetos de sua estrutura, permitindo que você crie novas instâncias com diferentes características a partir do mesmo protótipo.

Estrutura padrão exemplo:



Código



Aplicabilidade

Use o padrão prototype quando precisar que seu código não dependa de classes concretas para a criação de novos objetos

Use o padrão prototype quando quiser evitar explosão de subclasses para objetos muito similares

Use o padrão prototype para evitar a recriação de objetos “caros”.

Bom -Oculta classes concretas do código cliente, ajuda na criação de objetos caros ou complexos, evita a explosão de subclasses.

Ruim

Clonar objetos que tem referências para outros objetos pode ser super complexo

## Singleton

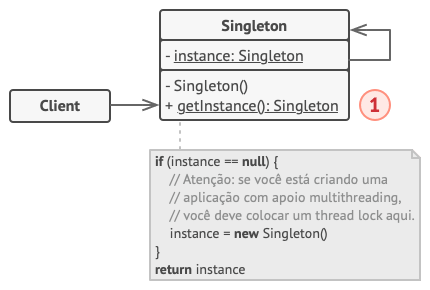
Esse padrão criacional tem como propósito garantir que uma classe de um projeto tenha apenas uma instância, impedindo a criação múltipla de um mesmo objeto.

O uso desse design combate dois problemas de uma vez só, o que gera uma contradição com os princípios [**SOLID**](https://medium.com/desenvolvendo-com-paixao/o-que-%C3%A9-solid-o-guia-completo-para-voc%C3%AA-entender-os-5-princ%C3%ADpios-da-poo-2b937b3fc530), sendo mais específico o princípio de responsabilidade única, que tem como objetivo fazer com que a classe mantenha apenas a responsabilidade de criar-se, nada mais.

Manter a instância única de uma classe é o primeiro problema que esse design possui foco em combater. Essa ideia visa cenários em que um projeto possui a necessidade de controlar o acesso a algum recurso compartilhado, como um banco de dados ou algum documento específico. Além disso, cria-se um ponto de acesso global à essa instância única, o que possibilita fácil acesso ao objeto desejado de qualquer lugar do programa, além de evitar que essa instância seja sobrescrita, devido à proteção que esse padrão entrega.

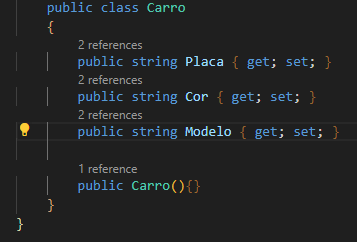
Como solução disso, existem alguns passos que o **Singleton** possui para ser implementado em um projeto, que são:

1. Criar um construtor padrão privado, para que o operador *new* não possa ser utilizado.
2. Criar um método estático de criação que age de forma semelhante a um construtor. E quando chamado, será verificado se a classe não possui nenhuma instância, se não houver, será feita a chamada ao construtor privado, que criará o objeto e irá mantê-lo salvo em um campo estático. Após a primeira chamada, as demais terão um retorno do objeto já criado anteriormente, como segue no exemplo a seguir:

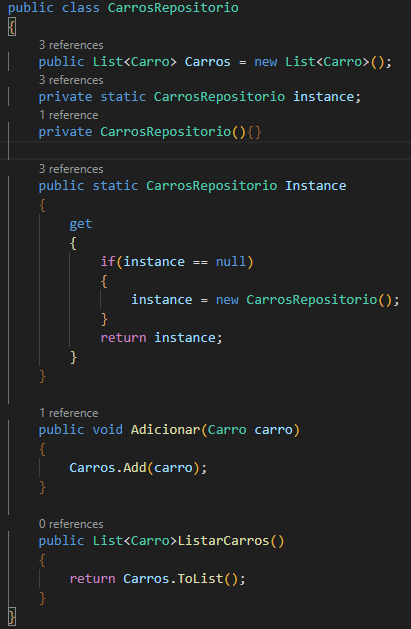
(Estrutura do padrão Singleton. Fonte: [Exemplo estrutura singleton](https://refactoring.guru/design-patterns/singleton))

**Padrão Singleton em C# - Exemplo**

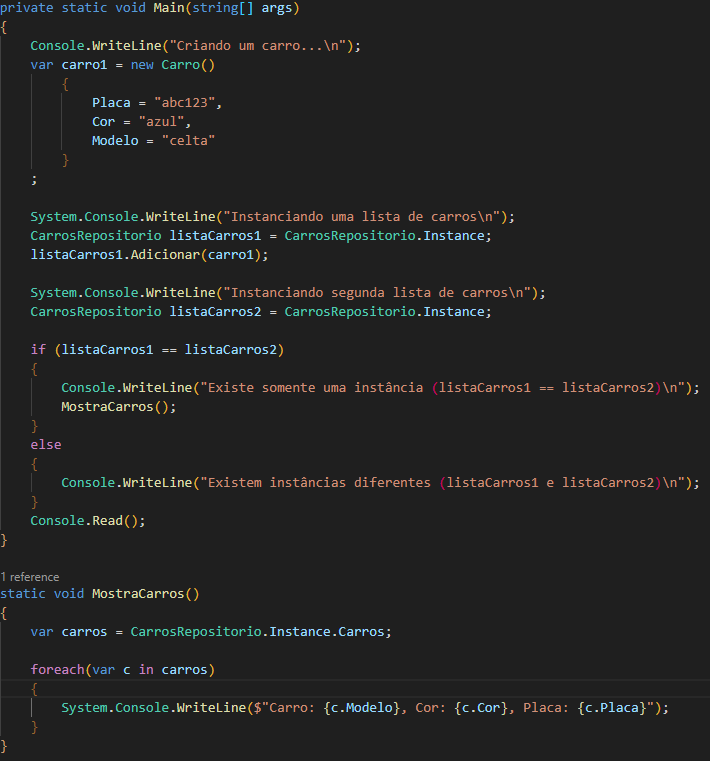
Classe de domínio Carro

****

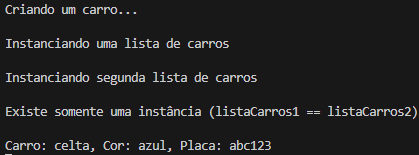
Classe de repositório do Carro

****

Classe Program com apresentação e criação do objeto carro e listas



Resultado console



O padrão **Singleton** pode ser aplicado quando um sistema possui uma classe que deve ter apenas uma instância, que é compartilhada por todas as partes do projeto através de um acesso global. Um exemplo disso é um sistema que compartilha o acesso a uma base de dados em diversos locais.

Outra alternativa de utilização desse padrão é quando surge a necessidade de obter um controle maior sobre as variáveis globais de um projeto, visto que a mudança sobre uma instância de objeto do padrão **Singleton** só pode ocorrer dentro do método estático (getInstance), evitando que o código seja alterado de fora da classe.

# CONCLUSÃO

Conclui-se, portanto, que os padrões de projeto criacionais são ferramentas valiosas para os profissionais de desenvolvimento de software, oferecendo abordagens eficientes e testadas para a criação de objetos. Ao empregar esses padrões de maneira adequada, é possível melhorar a modularidade, a flexibilidade e a manutenção dos sistemas, contribuindo para o desenvolvimento de software de qualidade.

Ao final deste estudo, nosso objetivo é que a turma adquira uma visão abrangente sobre design patterns, compreendendo suas implicações, desafios e oportunidades. Essa compreensão aprofundada servirá como base para futuras pesquisas e tomadas de decisão informadas, impulsionando avanços no campo do desenvolvimento de software e promovendo a implementação bem-sucedida de projetos. Dessa forma, podemos promover o progresso e alcançar projetos de alta qualidade.

# 

# REFERÊNCIAS BIBLIOGRÁFICAS CITADAS NO TEXTO.

Padrões de Projeto Criacionais. (Fonte:[Padrões de projeto criacionais](https://refactoring.guru/pt-br/design-patterns/creational-patterns))

Aula Ulbra. Fone:([Aula 17 - Design Patterns](https://docs.google.com/presentation/d/19XQ_l-qe24Pl_GFvhrWlktNB6FFSpmX7vmGQpNfv5S0/edit#slide=id.p7))

Refactoring Guru https://refactoring.guru/design-patterns/prototype

# 