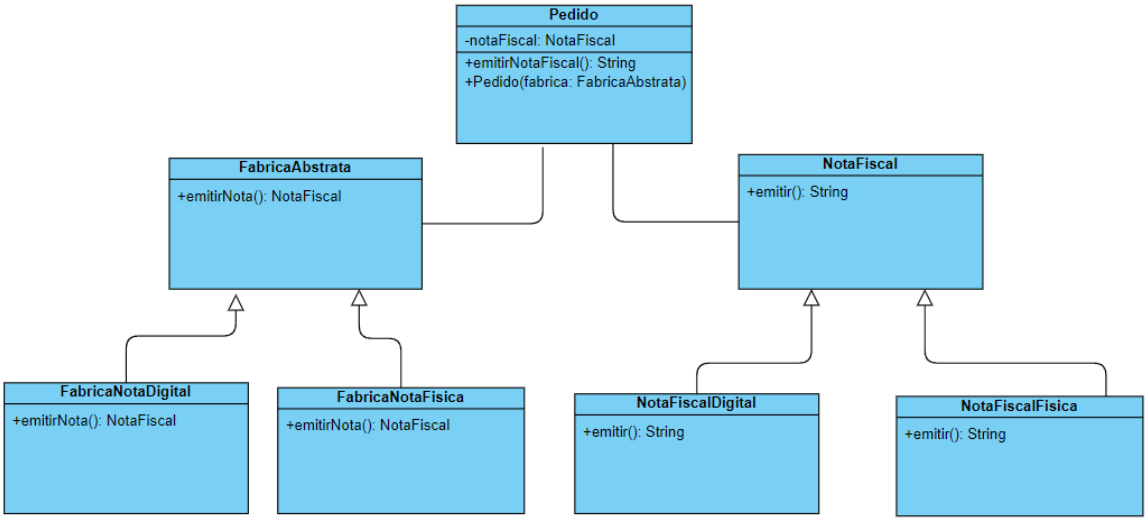
# Padrão Abstract Factory - Emissão de Notas Fiscais




## FabricaAbstrata.java

A interface `FabricaAbstrata` define o método `emitirNota`, que é responsável por emitir uma nota fiscal. Esse método retorna uma instância de `NotaFiscal`.

### Método:

- `NotaFiscal emitirNota()`: Método abstrato que deve ser implementado pelas subclasses para emitir uma nota fiscal.

## FabricaNotaDigital.java

A classe `FabricaNotaDigital` implementa a interface `FabricaAbstrata` e é responsável por emitir notas fiscais digitais. O método `emitirNota` retorna uma instância de `NotaFiscalDigital`.

### Método:

- `NotaFiscal emitirNota()`: Retorna uma instância de `NotaFiscalDigital`.

## FabricaNotaFisica.java

A classe `FabricaNotaFisica` também implementa a interface `FabricaAbstrata` e é responsável por emitir notas fiscais físicas. O método `emitirNota` retorna uma instância de `NotaFiscalFisica`.

### Método:

- `NotaFiscal emitirNota()`: Retorna uma instância de `NotaFiscalFisica`.

## NotaFiscal.java

A interface `NotaFiscal` define o método `emitir`, que retorna uma string representando a nota fiscal emitida.

### Método:

- `String emitir()`: Método abstrato que deve ser implementado pelas subclasses para emitir uma nota fiscal.

## NotaFiscalDigital.java

A classe `NotaFiscalDigital` implementa a interface `NotaFiscal` e representa uma nota fiscal digital. O método `emitir` retorna uma string que indica que uma nota fiscal digital foi emitida.

### Método:

- `String emitir()`: Retorna a string "Nota Fiscal Digital."

## NotaFiscalFisica.java

A classe `NotaFiscalFisica` implementa a interface `NotaFiscal` e representa uma nota fiscal física. O método `emitir` retorna uma string que indica que uma nota fiscal física foi emitida.

### Método:

- `String emitir()`: Retorna a string "Nota Fiscal Fisica."

## Pedido.java

A classe `Pedido` possui um atributo `notaFiscal` do tipo `NotaFiscal` e um construtor que recebe uma fábrica abstrata (`FabricaAbstrata`). O método `emitirNotaFiscal` utiliza a instância de `NotaFiscal` para emitir a nota fiscal correspondente.

### Variáveis:

- `private NotaFiscal notaFiscal`: A nota fiscal associada ao pedido.

### Construtor:

- `public Pedido(FabricaAbstrata fabrica)`: Recebe uma fábrica abstrata e emite uma nota fiscal usando essa fábrica.

### Métodos:

- `public String emitirNotaFiscal()`: Emite a nota fiscal correspondente e retorna a string representando a nota emitida.

## TestAbstractFactory

A classe `TestAbstractFactory` contém testes para validar a implementação do padrão Abstract Factory na emissão de notas fiscais. Os testes asseguram que as notas fiscais emitidas são do tipo correto (física ou digital) conforme a fábrica utilizada.

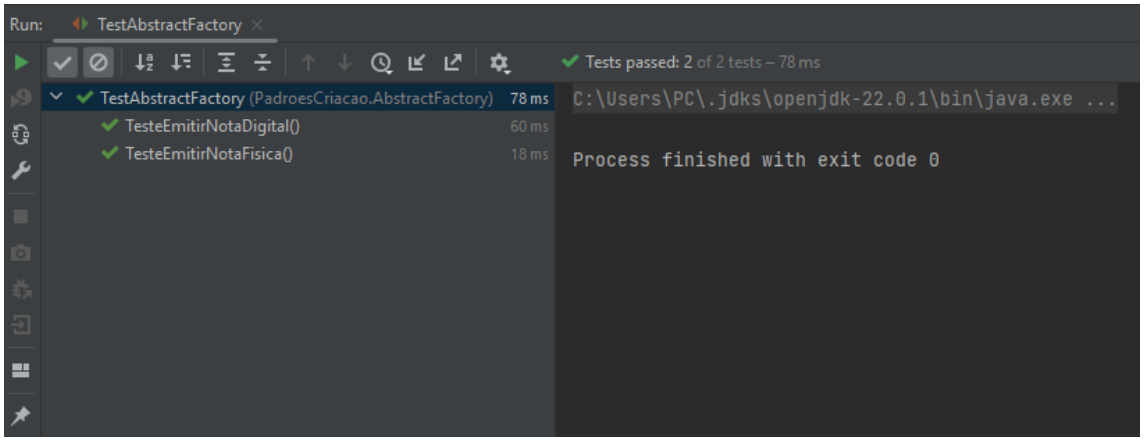
### TesteEmitirNotaFisica()

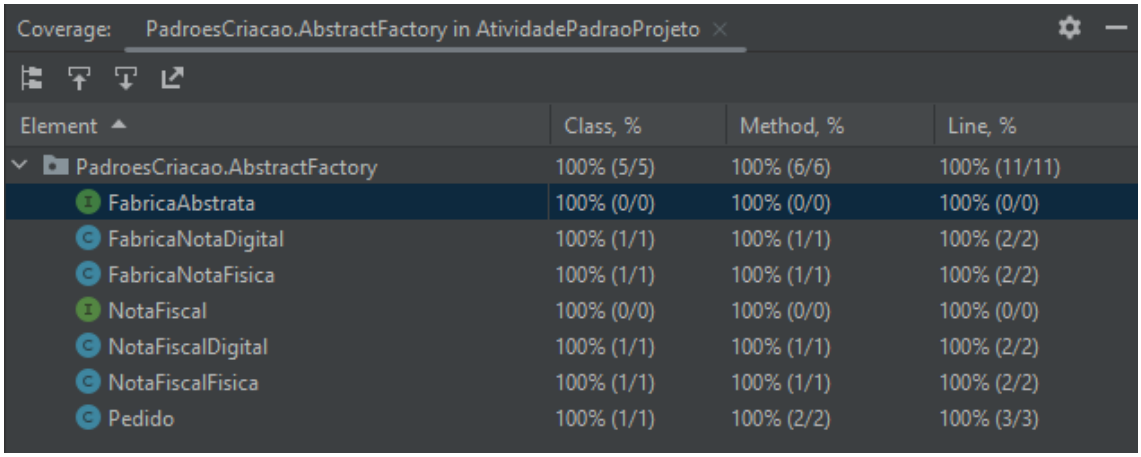
- **Objetivo**: Verificar se a emissão de uma nota fiscal física está funcionando corretamente.
- **Método**: Utiliza a fábrica `FabricaNotaFisica` para criar um pedido e emite a nota fiscal, verificando se o resultado é "Nota Fiscal Fisica.".
- **Resultado Esperado**: A string "Nota Fiscal Fisica." é retornada ao emitir a nota fiscal.

### TesteEmitirNotaDigital()

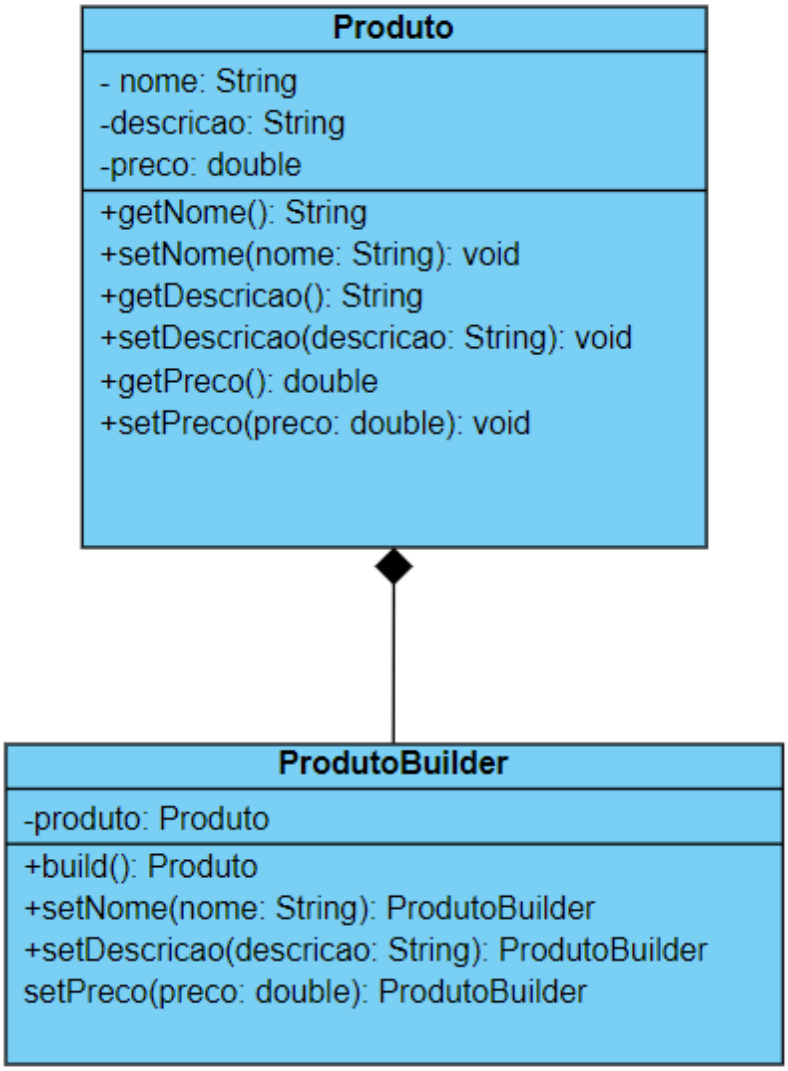
- **Objetivo**: Verificar se a emissão de uma nota fiscal digital está funcionando corretamente.
- **Método**: Utiliza a fábrica `FabricaNotaDigital` para criar um pedido e emite a nota fiscal, verificando se o resultado é "Nota Fiscal Digital.".

- **Resultado Esperado**: A string "Nota Fiscal Digital." é retornada ao emitir a nota fiscal.



# Padrão Builder – Produto




## Produto.java

A classe Produto representa um produto com atributos como nome, descrição e preço. Esta classe fornece métodos para acessar e modificar esses atributos.

### Variáveis:

- private String nome: Representa o nome do produto.
- private String descricao: Representa a descrição do produto.
- private double preco: Representa o preço do produto.

### Construtor:

Inicializa um novo produto com valores padrão (nome e descrição vazios, e preço igual a 0).

### Métodos get e set:

- public String getNome(): Retorna o nome do produto.
- public void setNome(String nome): Define o nome do produto.
- public String getDescricao(): Retorna a descrição do produto.
- public void setDescricao(String descricao): Define a descrição do produto.
- public double getPreco(): Retorna o preço do produto.
- public void setPreco(double preco): Define o preço do produto.

## ProdutoBuilder.java

A classe ProdutoBuilder busca aplicar o padrão “Builder” para o projeto. A partir dela, é possível construir uma instância nova da classe Produto de forma mais simples e passo-a-passo.

### Variáveis:

- private Produto produto: O objeto Produto a ser construído.

### Construtor:

Inicializa um novo objeto da classe Produto.

### Métodos:

- public Produto build(): Constrói e retorna a instância do Produto configurada. Antes de retornar o produto, valida se os campos obrigatórios estão corretamente preenchidos, retornando uma exceção caso o preço seja 0 ou o nome esteja vazio.
- public ProdutoBuilder setNome(String nome): Define o nome do produto.

- public ProdutoBuilder setDescricao(String descricao): Define a descrição do produto.
- public ProdutoBuilder setPreco(double preco): Define o preço do produto.

# TestBuilder.java

A classe TestBuilder contém testes para validar a implementação do padrão Builder na classe ProdutoBuilder. Os testes asseguram que um produto pode ser criado corretamente com o Builder e que as exceções apropriadas são lançadas quando faltam atributos obrigatórios.

## retornarProdutoSemNome()

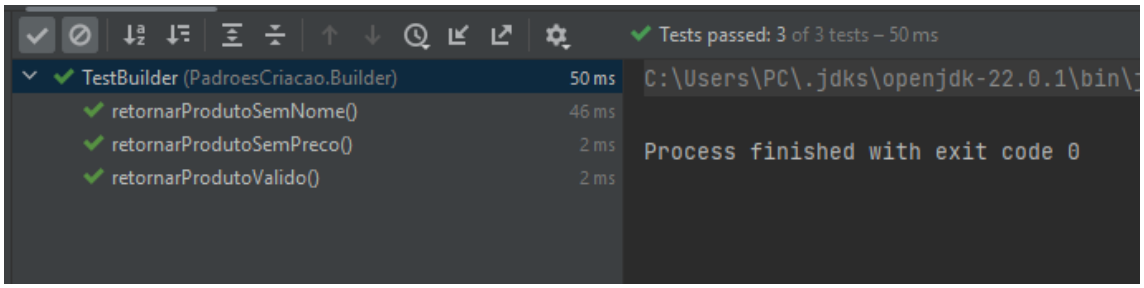
- Objetivo: Verificar se uma exceção é lançada ao tentar construir um produto sem definir o nome.
- Método: Tenta construir um produto sem definir o nome e espera uma exceção IllegalArgumentException com a mensagem "Nome inválido".
- Resultado Esperado: Uma exceção IllegalArgumentException é lançada com a mensagem "Nome inválido".

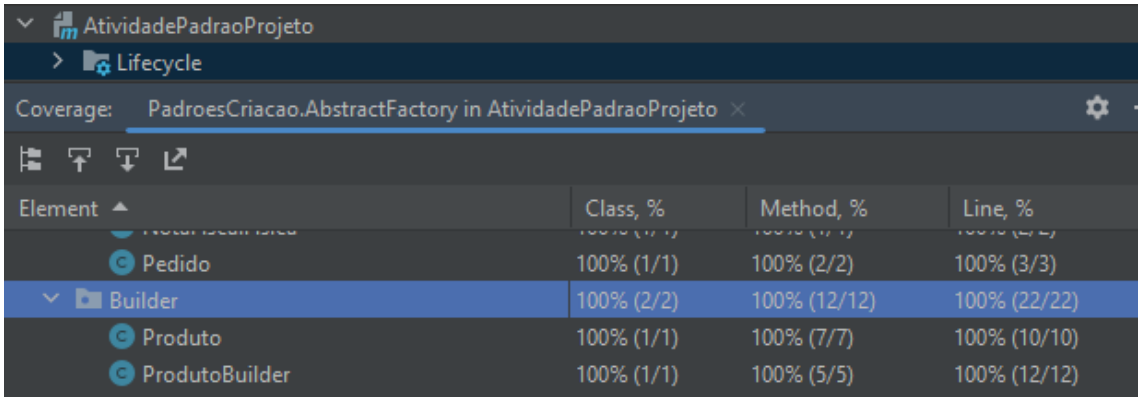
## retornarProdutoSemPreco()

- Objetivo: Verificar se uma exceção é lançada ao tentar construir um produto sem definir o preço.
- Método: Tenta construir um produto sem definir o preço e espera uma exceção IllegalArgumentException com a mensagem "Preco inválido".
- Resultado Esperado: Uma exceção IllegalArgumentException é lançada com a mensagem "Preco inválido".

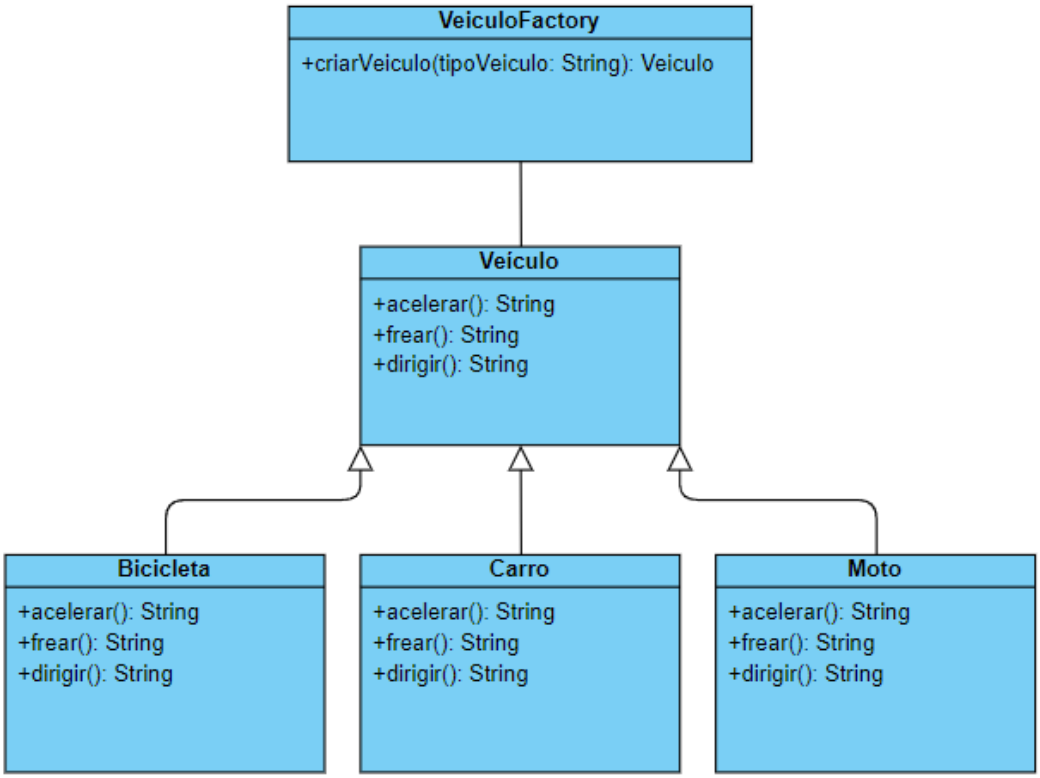
## retornarProdutoValido()

- Objetivo: Verificar se um produto válido pode ser construído usando o ProdutoBuilder.
- Método: Constrói um produto com nome, descrição e preço definidos e verifica se os atributos foram corretamente atribuídos.
- Resultado Esperado: O produto é criado com os atributos corretos e não é nulo.


Tests passed: 3 of 3 tests – 50 ms
TestBuilder (PadroesCriacao.Builder) 50 ms
retornarProdutoSemNome() 46 ms
retornarProdutoSemPreco() 2 ms
retornarProdutoValido() 2 ms
C:\Users\PC\.jdks\openjdk-22.0.1\bin\
Process finished with exit code 0


AtividadePadraoProjeto
Lifecycle
Coverage: PadroesCriacao.AbstractFactory in AtividadePadraoProjeto
Element
Class, %
Method, %
Line, %
Pedido 100% (1/1) 100% (2/2) 100% (3/3)
Builder 100% (2/2) 100% (12/12) 100% (22/22)
Produto 100% (1/1) 100% (7/7) 100% (10/10)
ProdutoBuilder 100% (1/1) 100% (5/5) 100% (12/12)

# Padrão Factory Method - Criação de Veículos




## VeiculoFactory.java

A classe `VeiculoFactory` é uma fábrica de veículos que cria instâncias de diferentes tipos de veículos com base em um tipo especificado utilizando o padrão de projeto Factory.

### Método:

- `public static Veiculo criarVeiculo(String tipoVeiculo)`: Cria um veículo com base no tipo fornecido como argumento. Aceita uma string que especifica o tipo de veículo a ser criado e retorna uma instância do veículo correspondente. Os valores aceitos são: “carro”, “moto” e “bicicleta”. Retorna uma exceção caso use algum valor não reconhecível como argumento.

## Veiculo.java

A interface `Veiculo` é responsável por definir os métodos padrões que todos os veículos devem ter.

### Métodos:

- `String acelerar()`: Descreve a ação de acelerar, retornando uma string.
- `String frear()`: Descreve a ação de frear, retornando uma string.
- `String dirigir()`: Descreve a ação de dirigir, retornando uma string.

## Bicicleta.java

A classe `Bicicleta` implementa a interface `Veiculo` e representa uma bicicleta. Implementa os métodos para descrever as ações de acelerar, frear e dirigir uma bicicleta.

### Métodos:

- `String acelerar()`: Retorna a string "Pedalando a bicicleta...".
- `String frear()`: Retorna a string "Freando a bicicleta...".
- `String dirigir()`: Retorna a string "Andando de bicicleta...".

## Carro.java

A classe `Carro` implementa a interface `Veiculo` e representa um carro. Implementa os métodos para descrever as ações de acelerar, frear e dirigir um carro.

### Métodos:

- `String acelerar()`: Retorna a string "Acelerando o carro...".
- `String frear()`: Retorna a string "Freando o carro...".
- `String dirigir()`: Retorna a string "Dirigindo o carro...".

## Moto.java

A classe `Moto` implementa a interface `Veiculo` e representa uma moto. Implementa os métodos para descrever as ações de acelerar, frear e dirigir uma moto.

### Métodos:

- `String acelerar()`: Retorna a string "Acelerando a moto...".
- `String frear()`: Retorna a string "Freando a moto...".
- `String dirigir()`: Retorna a string "Dirigindo a moto...".

# TestFactoryMethod.java

O TestFactoryMethod contém testes para validar a implementação do padrão Factory Method na classe VeiculoFactory. Os testes asseguram que diferentes tipos de veículos possam ser criados corretamente e que seus comportamentos esperados sejam verificados adequadamente.

## testCriacaoCarro()

- **Objetivo:** Verificar se um carro pode ser criado corretamente usando a VeiculoFactory.
- **Método:** Invoca o método `criarVeiculo("carro")` da VeiculoFactory e verifica se o objeto retornado não é nulo.
- **Resultado Esperado:** O veículo do tipo carro não deve ser nulo após a criação.

## testCarroAcelerar()

- **Objetivo:** Verificar se um carro acelera corretamente.
- **Método:** Cria um veículo do tipo "carro" usando VeiculoFactory e verifica se a mensagem retornada ao acelerar é "Acelerando o carro...".
- **Resultado Esperado:** A mensagem de aceleração do carro deve ser corretamente retornada.

## testCarroFrear()

- **Objetivo:** Verificar se um carro freia corretamente.
- **Método:** Cria um veículo do tipo "carro" usando VeiculoFactory e verifica se a mensagem retornada ao frear é "Freando o carro...".
- **Resultado Esperado:** A mensagem de frenagem do carro deve ser corretamente retornada.

## testCarroDirigir()

- **Objetivo:** Verificar se um carro pode ser dirigido corretamente.
- **Método:** Cria um veículo do tipo "carro" usando VeiculoFactory e verifica se a mensagem retornada ao dirigir é "Dirigindo o carro...".

- **Resultado Esperado:** A mensagem de direção do carro deve ser corretamente retornada.

## testCriacaoMoto()

- **Objetivo:** Verificar se uma moto pode ser criada corretamente usando a VeiculoFactory.
- **Método:** Invoca o método `criarVeiculo("moto")` da VeiculoFactory e verifica se o objeto retornado não é nulo.
- **Resultado Esperado:** O veículo do tipo moto não deve ser nulo após a criação.

## testMotoAcelerar()

- **Objetivo:** Verificar se uma moto acelera corretamente.
- **Método:** Cria um veículo do tipo "moto" usando VeiculoFactory e verifica se a mensagem retornada ao acelerar é "Acelerando a moto...".
- **Resultado Esperado:** A mensagem de aceleração da moto deve ser corretamente retornada.

## testMotoFrear()

- **Objetivo:** Verificar se uma moto freia corretamente.
- **Método:** Cria um veículo do tipo "moto" usando VeiculoFactory e verifica se a mensagem retornada ao frear é "Freando a moto...".
- **Resultado Esperado:** A mensagem de frenagem da moto deve ser corretamente retornada.

## testMotoDirigir()

- **Objetivo:** Verificar se uma moto pode ser dirigida corretamente.
- **Método:** Cria um veículo do tipo "moto" usando VeiculoFactory e verifica se a mensagem retornada ao dirigir é "Dirigindo a moto...".
- **Resultado Esperado:** A mensagem de direção da moto deve ser corretamente retornada.

## testCriacaoBicicleta()

- **Objetivo:** Verificar se uma bicicleta pode ser criada corretamente usando a VeiculoFactory.

- **Método:** Invoca o método `criarVeiculo("bicicleta")` da VeiculoFactory e verifica se o objeto retornado não é nulo.
- **Resultado Esperado:** O veículo do tipo bicicleta não deve ser nulo após a criação.

## testBicicletaAcelerar()

- **Objetivo:** Verificar se uma bicicleta acelera corretamente.
- **Método:** Cria um veículo do tipo "bicicleta" usando VeiculoFactory e verifica se a mensagem retornada ao acelerar é "Pedalando a bicicleta...".
- **Resultado Esperado:** A mensagem de aceleração da bicicleta deve ser corretamente retornada.

## testBicicletaFrear()

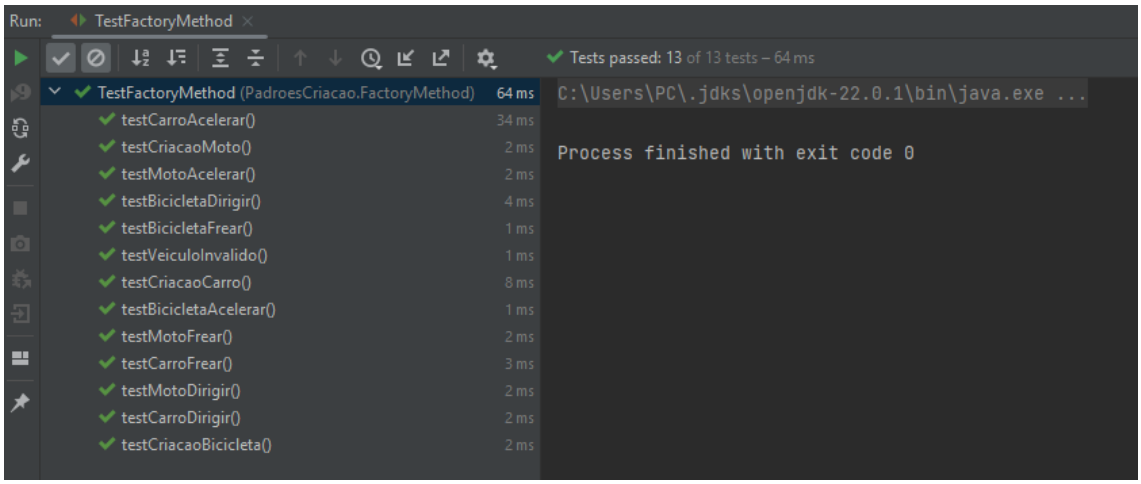
- **Objetivo:** Verificar se uma bicicleta freia corretamente.
- **Método:** Cria um veículo do tipo "bicicleta" usando VeiculoFactory e verifica se a mensagem retornada ao frear é "Freando a bicicleta...".
- **Resultado Esperado:** A mensagem de frenagem da bicicleta deve ser corretamente retornada.

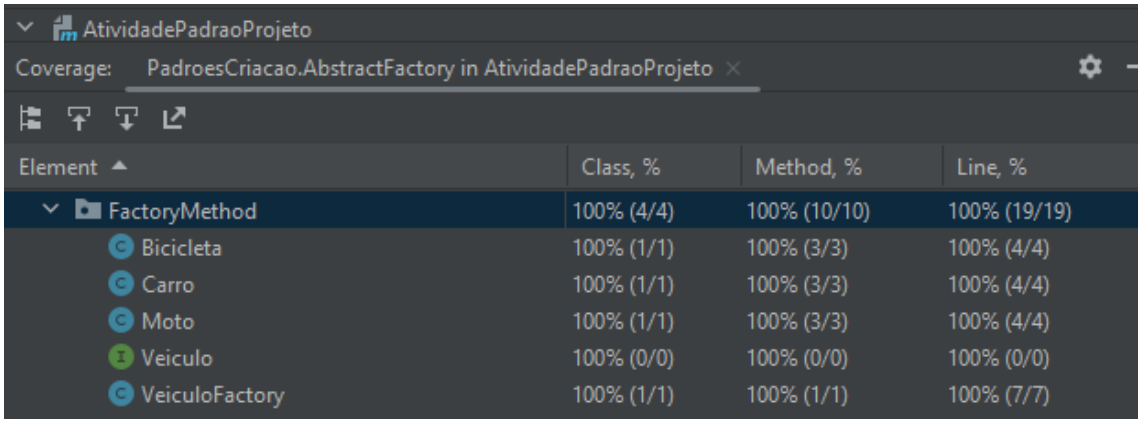
## testBicicletaDirigir()

- **Objetivo:** Verificar se uma bicicleta pode ser dirigida corretamente.
- **Método:** Cria um veículo do tipo "bicicleta" usando VeiculoFactory e verifica se a mensagem retornada ao dirigir é "Andando de bicicleta...".
- **Resultado Esperado:** A mensagem de direção da bicicleta deve ser corretamente retornada.

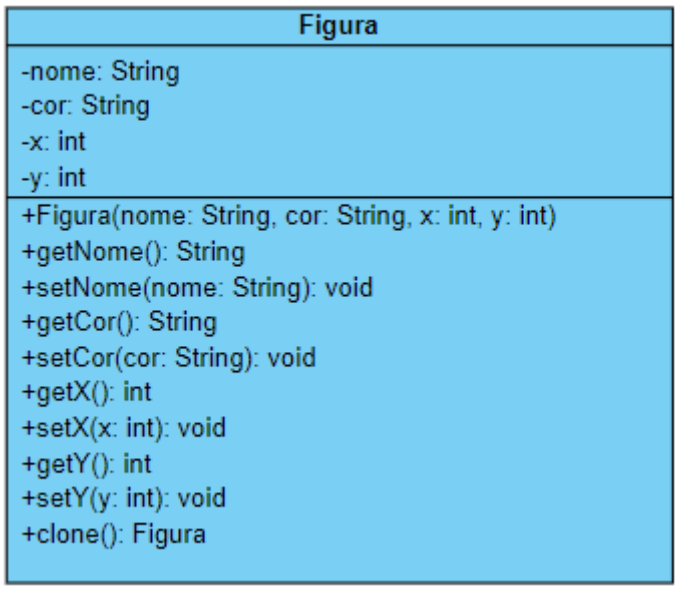
## testVeiculoInvalido()

- **Objetivo:** Verificar se uma exceção é lançada ao tentar criar um veículo com um tipo inválido.
- **Método:** Tenta criar um veículo usando `criarVeiculo("avião")` da VeiculoFactory e verifica se uma exceção do tipo IllegalArgumentException é lançada com a mensagem "Tipo de veículo inválido!".
- **Resultado Esperado:** Deve lançar uma exceção com a mensagem correta quando o tipo de veículo não é reconhecido pela fábrica.


Run:
TestFactoryMethod
Tests passed: 13 of 13 tests – 64 ms
TestFactoryMethod (PadroesCriacao.FactoryMethod) 64 ms
testCarroAcelerar() 34 ms
testCriacaoMoto() 2 ms
testMotoAcelerar() 2 ms
testBicicletaDirigir() 4 ms
testBicicletaFrear() 1 ms
testVeiculoInvalido() 1 ms
testCriacaoCarro() 8 ms
testBicicletaAcelerar() 1 ms
testMotoFrear() 2 ms
testCarroFrear() 3 ms
testMotoDirigir() 2 ms
testCarroDirigir() 2 ms
testCriacaoBicicleta() 2 ms
C:\Users\PC\.jdks\openjdk-22.0.1\bin\java.exe ...
Process finished with exit code 0


AtividadePadraoProjeto
Coverage: PadroesCriacao.AbstractFactory in AtividadePadraoProjeto
Element
Class, %
Method, %
Line, %
FactoryMethod 100% (4/4) 100% (10/10) 100% (19/19)
Bicicleta 100% (1/1) 100% (3/3) 100% (4/4)
Carro 100% (1/1) 100% (3/3) 100% (4/4)
Moto 100% (1/1) 100% (3/3) 100% (4/4)
Veiculo 100% (0/0) 100% (0/0) 100% (0/0)
VeiculoFactory 100% (1/1) 100% (1/1) 100% (7/7)

# Padrão Prototype – Figuras




## Figura.java

Na classe Figura, o objetivo é fazer um padrão Prototype para a criação de protótipos de figuras geométricas, isso permite com que sejam criados objetos pré-montados que, caso seja necessário uma instância de mesma configuração, seja possível apenas clonar o protótipo original.

### Variáveis:

- private String nome: Representa o nome da figura.
- private String cor: Representa a cor da figura.
- private int x: Representa a coordenada x da figura.
- private int y: Representa a coordenada y da figura.

### Construtor:

Inicializa um novo protótipo. Para criar um novo protótipo, o construtor pede, na ordem, o nome da figura em formato de string, a cor da figura em formato de string, e as coordenadas x e y, ambas em formato de int.

### Métodos get e set:

- public String getNome(): Retorna o nome da figura.

- public void setNome(String nome): Define o nome da figura.
- public String getCor(): Retorna a cor da figura.
- public void setCor(String cor): Define a cor da figura.
- public int getX(): Retorna a coordenada x da figura.
- public void setX(int x): Define a coordenada x da figura.
- public int getY(): Retorna a coordenada y da figura.
- public void setY(int y): Define a coordenada y da figura.

### Método Clone:

O método "clone ()" é o que permite com que o protótipo seja clonado caso seja necessário a criação de uma instância nova com a mesma configuração do objeto protótipo. O método retorna as configurações do protótipo, permitindo que sejam clonados na nova instância, além disso, ele irá lançar uma exceção "CloneNotSupportedException" caso a clonagem resulte em erro.

## TestPrototype

A classe TestPrototype verifica a implementação do padrão Prototype na classe Figura. Este teste garante que a clonagem de objetos funcione corretamente e que as alterações no clone não afetem o objeto original.

### criarProtótipo():

Configura um protótipo de Figura com valores iniciais e cria um clone desse protótipo antes de cada teste.

### nomesIguais():

- Objetivo: Verificar se o clone tem o mesmo nome que o objeto original.
- Método: Compara os nomes das figuras original e clone usando assertEquals.
- Resultado Esperado: Os nomes das figuras original e clone devem ser iguais.
-

### coresIguais()

- Objetivo: Verificar se o clone tem a mesma cor que o objeto original.

- Método: Compara as cores das figuras original e clone usando assertEquals.
- Resultado Esperado: As cores das figuras original e clone devem ser iguais.

## coordenadasXIguais()

- Objetivo: Verificar se o clone tem a mesma coordenada X que o objeto original.
- Método: Compara as coordenadas X das figuras original e clone usando assertEquals.
- Resultado Esperado: As coordenadas X das figuras original e clone devem ser iguais.

## coordenadasYIguais()

- Objetivo: Verificar se o clone tem a mesma coordenada Y que o objeto original.
- Método: Compara as coordenadas Y das figuras original e clone usando assertEquals.
- Resultado Esperado: As coordenadas Y das figuras original e clone devem ser iguais.

## nomesDiferentes()

- Objetivo: Verificar se a modificação do nome no clone não afeta o objeto original.
- Método: Altera o nome do clone e compara com o nome do original usando assertNotEquals.
- Resultado Esperado: Os nomes das figuras original e clone devem ser diferentes após a modificação.

## coresDiferentes()

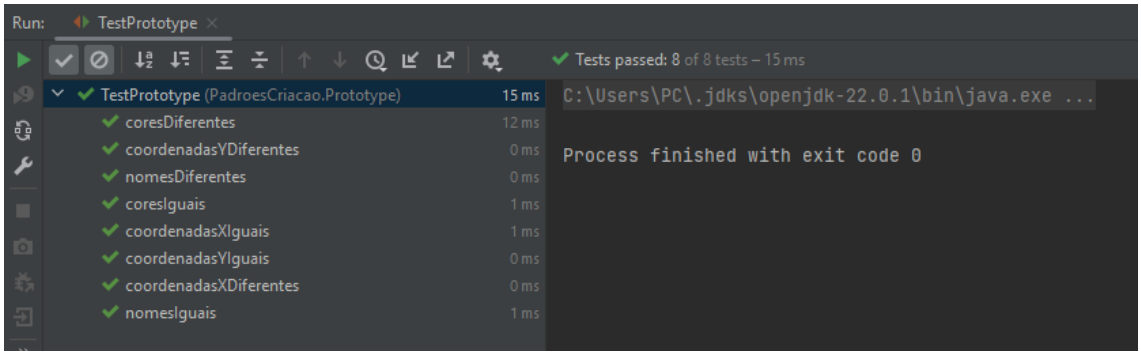
- Objetivo: Verificar se a modificação da cor no clone não afeta o objeto original.
- Método: Altera a cor do clone e compara com a cor do original usando assertNotEquals.
- Resultado Esperado: As cores das figuras original e clone devem ser diferentes após a modificação.

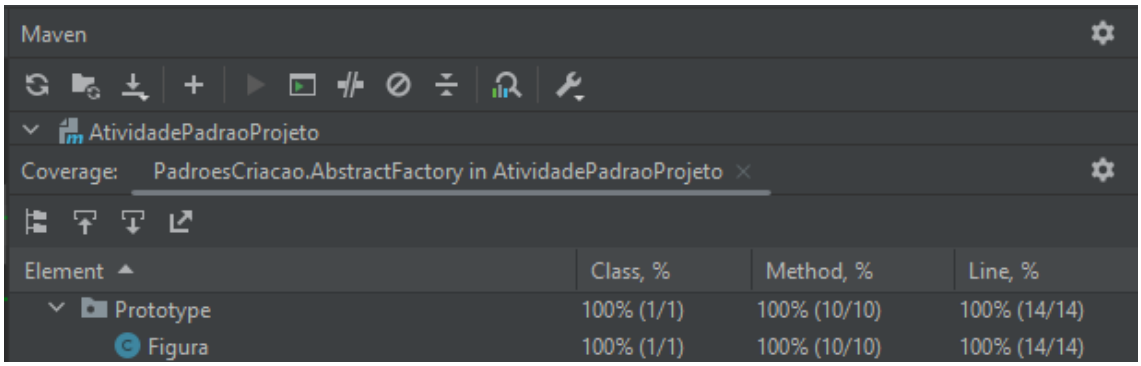
## coordenadasXDiferentes()

- Objetivo: Verificar se a modificação da coordenada X no clone não afeta o objeto original.
- Método: Altera a coordenada X do clone e compara com a coordenada X do original usando assertNotEquals.
- Resultado Esperado: As coordenadas X das figuras original e clone devem ser diferentes após a modificação.

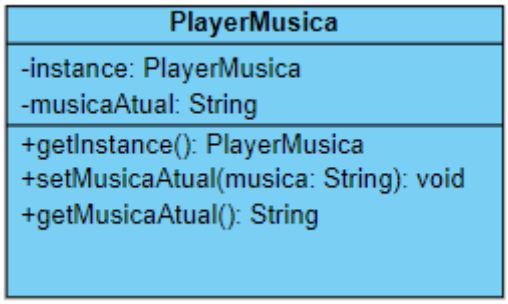
## coordenadasYDiferentes()

- Objetivo: Verificar se a modificação da coordenada Y no clone não afeta o objeto original.
- Método: Altera a coordenada Y do clone e compara com a coordenada Y do original usando assertNotEquals.
- Resultado Esperado: As coordenadas Y das figuras original e clone devem ser diferentes após a modificação.



# Padrão Singleton - Player de Musica




## PlayerMusica.java

A classe PlayerMusica busca gerenciar qual música está tocando no programa. É utilizado o padrão Singleton para que seja garantido que apenas uma instância dessa classe seja criada enquanto o programa esteja sendo executado.

### Variáveis:

- private static PlayerMusica instance: A única instância da classe PlayerMusica.
- private String musicaAtual: A música atualmente sendo reproduzida.

### Métodos:

- public static PlayerMusica getInstance(): Método estático que retorna a única instância da classe PlayerMusica. Se a instância ainda não foi criada, cria uma nova.
- public void setMusicaAtual(String musica): Método que define a música atual a ser reproduzida, recebe uma string “musica” e passa o valor para a variável “musicaAtual”
- public String getMusicaAtual(): Método que retorna a música atualmente sendo reproduzida, buscando seu nome na variável “musicaAtual” e retornando em formato de string.

## TestSingleton.java

A classe TestSingleton testa a implementação da classe PlayerMusica para garantir que siga o padrão Singleton e que as funcionalidades principais estão funcionando conforme esperado.

## deveObterInstanciaUnicaDoPlayer():

Objetivo: Verificar se o método getInstance da classe PlayerMusica retorna sempre a mesma instância. Método: Chamar PlayerMusica.getInstance() duas vezes e comparar as instâncias retornadas. Resultado Esperado: As duas instâncias obtidas devem ser a mesma (singleton).

## deveTocarMusicaAtual():

- Objetivo: Verificar se é possível configurar e recuperar a música atual no player.
- Método: Chamar PlayerMusica.getInstance(), definir uma música usando setMusicaAtual, e verificar se a música atual é a mesma usando getMusicaAtual.
- Resultado Esperado: A música configurada deve ser recuperada corretamente.