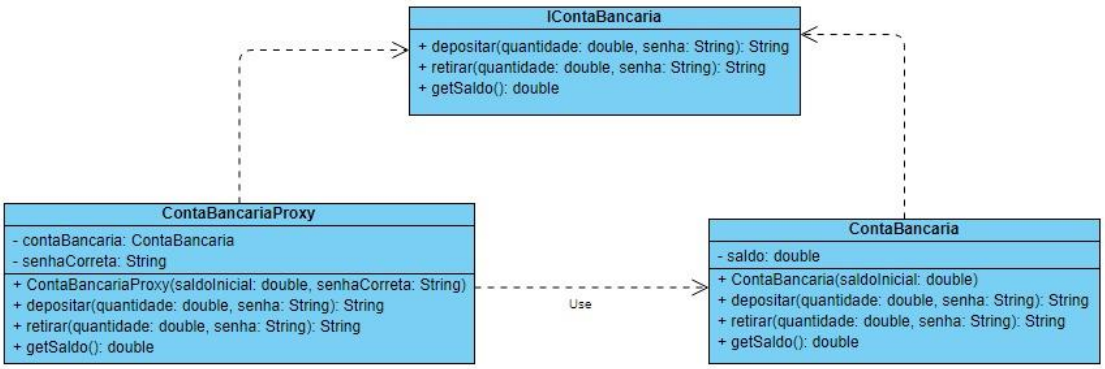
Padrão Proxy - Conta Bancária



O projeto implementa o padrão de projeto Proxy para fornecer uma camada de controle e segurança no sistema, o padrão Proxy é aplicado para gerenciar o acesso à conta bancária real, verificando a autenticação antes de permitir que operações como depósito, retirada e consulta de saldo sejam executadas.

ContaBancaria.java

A classe `ContaBancaria` representa uma conta bancária que permite operações de depósito, retirada e consulta de saldo.

Atributos:

- `private double saldo`: Armazena o saldo atual da conta.

Construtor:

- `public ContaBancaria(double saldoInicial)`: Inicializa a conta bancária com um saldo inicial.

Métodos:

- `public String depositar(double quantidade, String senha)`: Adiciona a quantidade especificada ao saldo e retorna uma mensagem de confirmação.
- `public String retirar(double quantidade, String senha)`: Deduz a quantidade especificada do saldo se houver fundos suficientes e retorna uma mensagem de confirmação ou de saldo insuficiente.

- `public double getSaldo()`: Retorna o saldo atual da conta.

ContaBancariaProxy.java

A classe `ContaBancariaProxy` é um proxy para a `ContaBancaria`, que adiciona uma camada de segurança verificando a senha antes de permitir operações na conta bancária.

Atributos:

- `private ContaBancaria contaBancaria`: Instância da conta bancária real.
- `private String senhaCorreta`: Senha correta para acessar a conta.

Construtor:

- `public ContaBancariaProxy(double saldoInicial, String senhaCorreta)`: Inicializa o proxy com o saldo inicial e a senha correta.

Métodos:

- `public String depositar(double quantidade, String senha)`: Verifica a senha e, se correta, delega a operação de depósito para a conta bancária real, retornando uma mensagem de confirmação ou de acesso negado.
- `public String retirar(double quantidade, String senha)`: Verifica a senha e, se correta, delega a operação de retirada para a conta bancária real, retornando uma mensagem de confirmação ou de acesso negado.
- `public double getSaldo()`: Retorna o saldo atual da conta bancária real.

IContaBancaria.java

A interface `IContaBancaria` define os métodos que devem ser implementados por qualquer classe que represente uma conta bancária.

Métodos:

- `String depositar(double quantidade, String senha)`: Define o método para depositar uma quantidade na conta.
- `String retirar(double quantidade, String senha)`: Define o método para retirar uma quantidade da conta.
- `double getSaldo()`: Define o método para obter o saldo atual da conta.

TestProxy.java

A classe `TestProxy` busca testas as funcionalidades do projeto, verificando se o uso do padrão Proxy está funcionando de acordo com o esperado.

`testeDepositarSenhaCorreta()`

- **Objetivo:** Verificar se é possível depositar na conta bancária utilizando a senha correta.
- **Método:** Cria uma instância de `ContaBancariaProxy` com um saldo inicial de 1000.0 e senha "123". Chama o método `depositar(200.0, "123")` e verifica se a mensagem de confirmação é retornada corretamente. Em seguida, verifica se o saldo foi atualizado corretamente para 1200.0.
- **Resultado Esperado:** O depósito deve ser aceito e refletir o saldo atualizado na conta bancária.

`testeDepositarSenhaIncorreta()`

- **Objetivo:** Verificar o comportamento ao tentar depositar com uma senha incorreta.
- **Método:** Cria uma instância de `ContaBancariaProxy` com um saldo inicial de 1000.0 e senha "123". Chama o método `depositar(200.0, "456")` e verifica se é retornada a mensagem "Acesso negado. Senha Incorreta.". Verifica também se o saldo permanece inalterado em 1000.0.
- **Resultado Esperado:** Deve ocorrer uma negação de acesso devido à senha incorreta e o saldo não deve ser alterado.

`testeRetirarSenhaCorreta()`

- **Objetivo:** Verificar se é possível realizar uma retirada na conta bancária utilizando a senha correta.
- **Método:** Cria uma instância de `ContaBancariaProxy` com um saldo inicial de 1000.0 e senha "123". Chama o método `retirar(200.0, "123")` e verifica se a mensagem de confirmação é retornada corretamente. Em seguida, verifica se o saldo foi atualizado corretamente para 800.0.
- **Resultado Esperado:** A retirada deve ser aceita e refletir o saldo atualizado na conta bancária.

`testeRetirarSenhaIncorreta()`

- **Objetivo:** Verificar o comportamento ao tentar realizar uma retirada com uma senha incorreta.
- **Método:** Cria uma instância de `ContaBancariaProxy` com um saldo inicial de 1000.0 e senha "123". Chama o método `retirar(200.0, "456")` e verifica se é retornada a mensagem "Acesso negado. Senha Incorreta.". Verifica também se o saldo permanece inalterado em 1000.0.

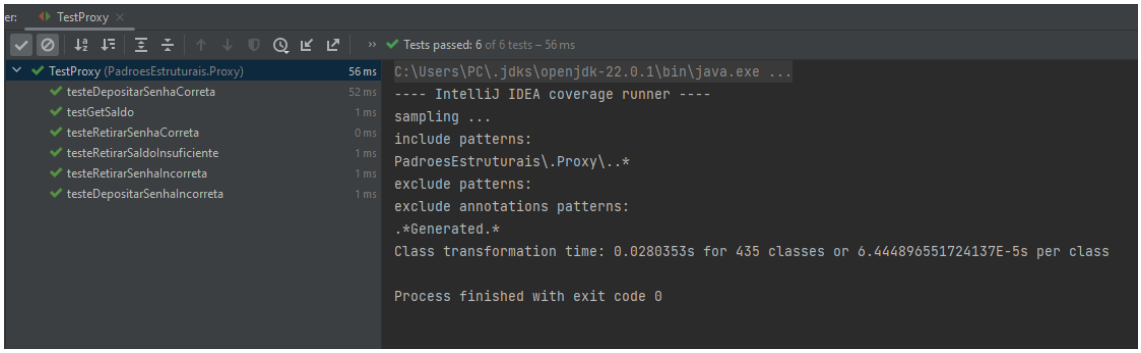
- **Resultado Esperado:** Deve ocorrer uma negação de acesso devido à senha incorreta e o saldo não deve ser alterado.

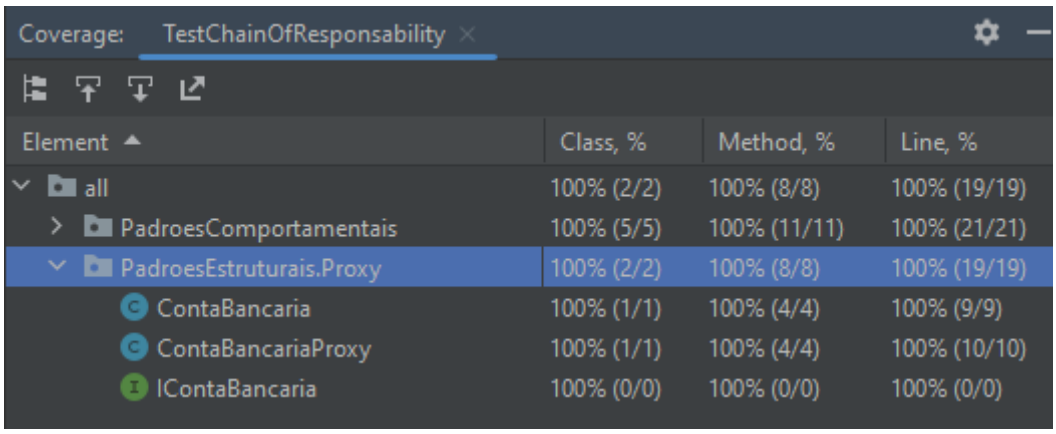
`testeRetirarSaldoInsuficiente()`

- **Objetivo:** Verificar o comportamento ao tentar realizar uma retirada com saldo insuficiente.
- **Método:** Cria uma instância de `ContaBancariaProxy` com um saldo inicial de 1000.0 e senha "123". Chama o método `retirar(1200.0, "123")` e verifica se é retornada a mensagem "Saldo Insuficiente.". Verifica também se o saldo permanece inalterado em 1000.0.
- **Resultado Esperado:** Deve ser indicado que o saldo é insuficiente para a retirada solicitada e o saldo não deve ser alterado.

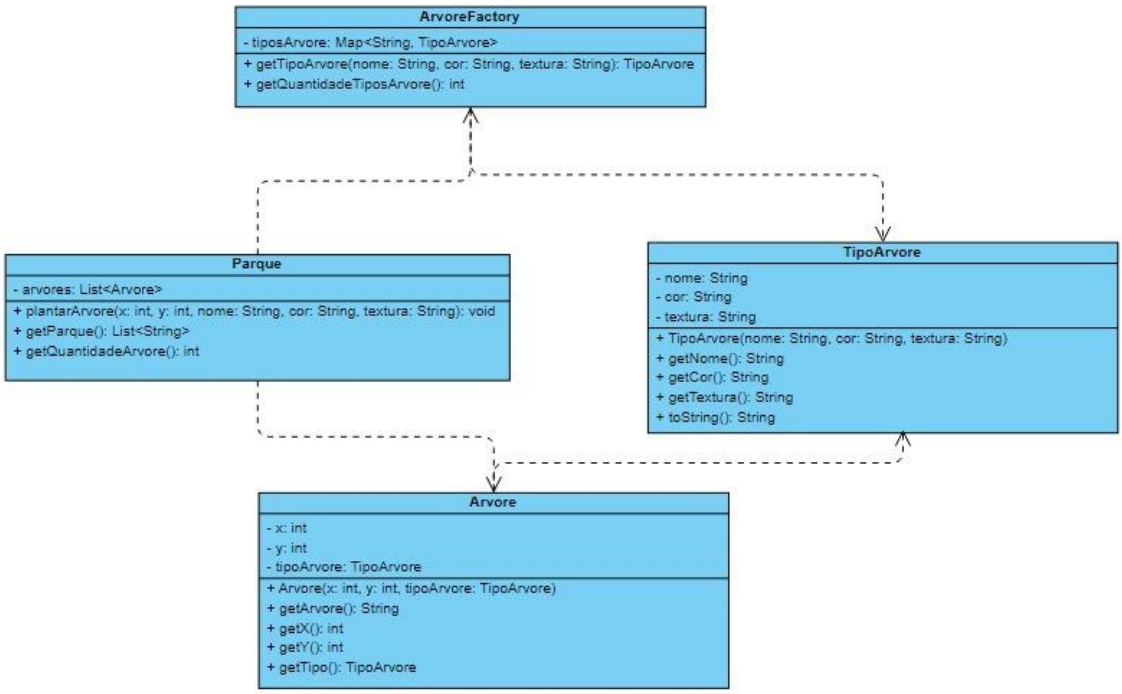
`testGetSaldo()`

- **Objetivo:** Verificar se o método `getSaldo()` retorna o saldo correto da conta bancária.
- **Método:** Cria uma instância de `ContaBancariaProxy` com um saldo inicial de 1000.0 e senha "123". Chama o método `getSaldo()` e verifica se o saldo retornado é igual a 1000.0.
- **Resultado Esperado:** O saldo retornado deve corresponder ao saldo atual da conta bancária.





Padrão Flyweight - Parque de Árvores



O projeto implementa o padrão de projeto Flyweight para otimizar o uso de memória ao lidar com múltiplos objetos semelhantes, neste caso, tipos diferentes de árvores em um parque.

Arvore.java

A classe `Arvore` representa uma árvore específica no parque.

Atributos:

- `private final int x`: Posição x da árvore no parque.
- `private final int y`: Posição y da árvore no parque.
- `private final TipoArvore tipoArvore`: Tipo específico de árvore.

Construtor:

- `public Arvore(int x, int y, TipoArvore tipoArvore)`: Inicializa uma árvore com uma posição específica (x, y) e um tipo de árvore.

Métodos:

- `public String getArvore()`: Retorna uma representação textual da árvore, incluindo seu tipo e posição.
- `public int getX()`: Retorna a coordenada x da árvore.
- `public int getY()`: Retorna a coordenada y da árvore.
- `public TipoArvore getTipo()`: Retorna o tipo de árvore desta instância.

ArvoreFactory.java

A classe `ArvoreFactory` implementa um padrão Flyweight para gerenciar tipos únicos de árvores no parque.

Atributo:

- `private static final Map<String, TipoArvore> tiposArvore`: Armazena os tipos de árvore conhecidos, utilizando uma combinação única de nome, cor e textura como chave.

Métodos:

- `public static TipoArvore getTipoArvore(String nome, String cor, String textura)`: Retorna um objeto `TipoArvore` existente ou cria um novo se não existir na fábrica.
- `public static int getQuantidadeTiposArvore()`: Retorna o número total de tipos de árvores diferentes gerenciados pela fábrica.

Parque.java

A classe `Parque` mantém uma coleção de todas as árvores plantadas no parque.

Atributo:

- `private final List<Arvore> arvores`: Lista de todas as árvores plantadas no parque.

Métodos:

- `public void plantarArvore(int x, int y, String nome, String cor, String textura)`: Planta uma nova árvore no parque com base nas características especificadas, utilizando a `ArvoreFactory` para obter o tipo de árvore correspondente.
- `public List<String> getParque()`: Retorna uma lista de representações textuais de todas as árvores no parque.

- `public int getQuantidadeArvore()`: Retorna o número total de árvores plantadas no parque.

TipoArvore.java

A classe `TipoArvore` define os atributos específicos que caracterizam um tipo de árvore.

Atributos:

- `private final String nome`: Nome do tipo de árvore.
- `private final String cor`: Cor predominante da árvore.
- `private final String textura`: Textura da casca da árvore.

Construtor:

- `public TipoArvore(String nome, String cor, String textura)`: Inicializa um tipo de árvore com nome, cor e textura específicos.

Métodos:

- `public String getNome()`: Retorna o nome do tipo de árvore.
- `public String getCor()`: Retorna a cor predominante da árvore.
- `public String getTextura()`: Retorna a textura da casca da árvore.
- `@Override public String toString()`: Retorna uma representação textual dos atributos do tipo de árvore.

TestFlyweight.java

Os testes a seguir verificam o funcionamento correto das classes implementadas utilizando o padrão Flyweight para gerenciamento eficiente de árvores em um parque.

`testTipoArvore()`

- **Objetivo:** Verificar a correta inicialização e obtenção dos atributos de um objeto `TipoArvore`.
- **Método:** Cria uma instância de `TipoArvore` com nome "Carvalho", cor "Verde" e textura "Aspero". Em seguida, compara se os métodos `getNome()`, `getCor()` e `getTextura()` retornam os valores esperados.
- **Resultado Esperado:** Os valores retornados pelos métodos devem corresponder aos valores passados durante a inicialização.

`testArvore()`

- **Objetivo:** Verificar a correta inicialização e obtenção dos atributos de um objeto `Arvore`.
- **Método:** Cria uma instância de `TipoArvore` com nome "Carvalho", cor "Verde" e textura "Aspero". Depois, cria uma instância de `Arvore` com posição (10, 20) e o tipo de árvore criado anteriormente. Verifica se os métodos `getX()`, `getY()` e `getTipo()` retornam os valores esperados.
- **Resultado Esperado:** Os valores retornados pelos métodos devem corresponder aos valores passados durante a inicialização.

`testGetArvore()`

- **Objetivo:** Verificar se o método `getArvore()` da classe `Arvore` retorna a representação correta da árvore.
- **Método:** Cria uma instância de `TipoArvore` com nome "Carvalho", cor "Verde" e textura "Aspero". Em seguida, cria uma instância de `Arvore` com posição (10, 20) e o tipo de árvore criado anteriormente. Verifica se o método `getArvore()` retorna a string esperada contendo o tipo de árvore e sua localização.
- **Resultado Esperado:** A string retornada pelo método `getArvore()` deve conter a representação correta da árvore.

`testArvoreFactory()`

- **Objetivo:** Verificar o funcionamento correto da `ArvoreFactory` ao gerenciar tipos de árvores utilizando o padrão Flyweight.
- **Método:** Utiliza a `ArvoreFactory` para obter tipos de árvores com as características "Carvalho Verde Aspero". Compara se duas chamadas para criar o mesmo tipo de árvore retornam a mesma instância (`assertSame`). Em seguida, obtém um novo tipo de árvore com características diferentes ("Pinheiro Verde Suave") e verifica se este é diferente dos anteriores (`assertNotSame`). Finalmente, verifica se o número total de tipos de árvores na fábrica é igual a 2.
- **Resultado Esperado:** A fábrica deve reutilizar tipos de árvores existentes sempre que possível e manter um registro correto do número de tipos de árvores diferentes.

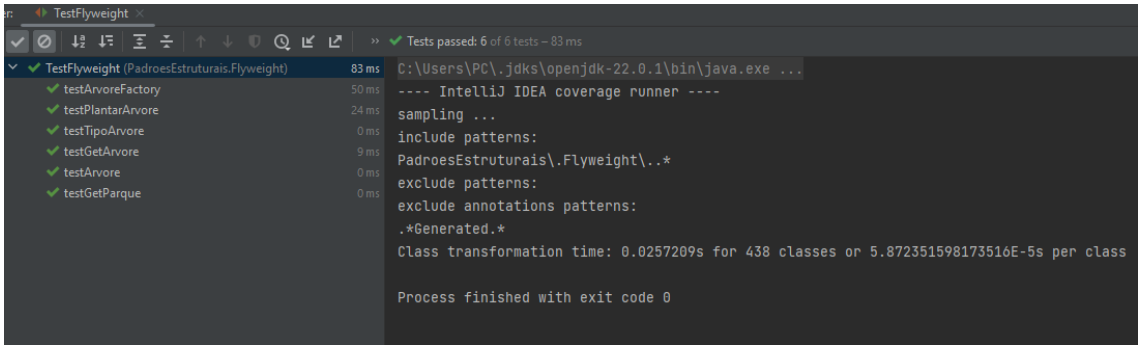
`testPlantarArvore()`

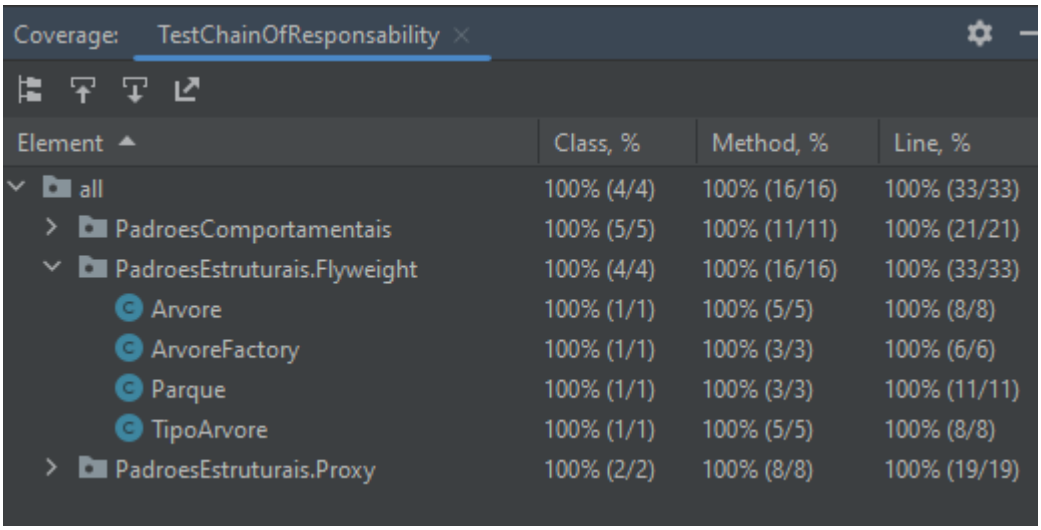
- **Objetivo:** Verificar se o método `plantarArvore()` da classe `Parque` adiciona corretamente as árvores à lista interna.
- **Método:** Cria uma instância de `Parque` e chama o método `plantarArvore()` duas vezes para adicionar árvores com o

mesmo tipo ("Carvalho Verde Aspero") em diferentes posições. Em seguida, verifica se o número total de árvores no parque é igual a 2.
- **Resultado Esperado:** O método `plantarArvore()` deve adicionar corretamente as árvores à lista do parque.

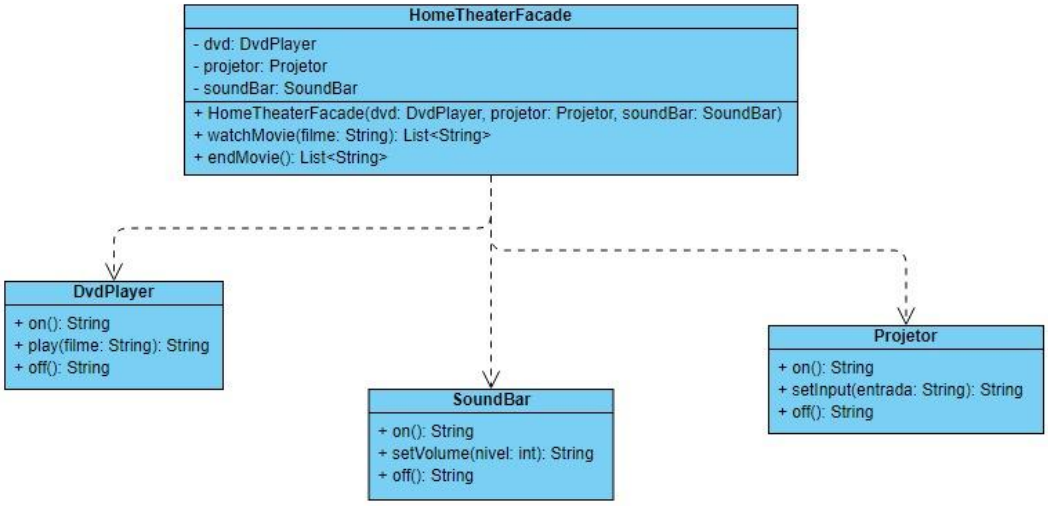
`testGetParque()`

- **Objetivo:** Verificar se o método `getParque()` da classe `Parque` retorna corretamente a lista de representações textuais das árvores no parque.
- **Método:** Cria uma instância de `Parque` e chama o método `plantarArvore()` duas vezes para adicionar árvores com tipos diferentes ("Carvalho Verde Aspero" e "Pinheiro Verde Suave"). Verifica se o método `getParque()` retorna uma lista contendo as representações textuais corretas das árvores adicionadas.
- **Resultado Esperado:** O método `getParque()` deve retornar corretamente a lista de representações textuais das árvores no parque.





Padrão Facade - Home Theater



O projeto implementa o padrão de projeto Facade para simplificar a interação com os componentes de um sistema de Home Theater, proporcionando uma interface unificada para operações complexas.

DvdPlayer.java

A classe `DvdPlayer` representa um player de DVD.

Métodos:

- `public String on()`: Liga o DVD Player.
- `public String play(String filme)`: Reproduz um filme específico.
- `public String off()`: Desliga o DVD Player.

Projetor.java

A classe `Projetor` representa um projetor utilizado no sistema de Home Theater.

Métodos:

- `public String on()`: Liga o projetor.
- `public String setInput(String entrada)`: Define a entrada do projetor (por exemplo, DVD, HDMI).
- `public String off()`: Desliga o projetor.

SoundBar.java

A classe `SoundBar` representa uma barra de som no sistema de Home Theater.

Métodos:

- `public String on()`: Liga a SoundBar.
- `public String setVolume(int nivel)`: Define o volume da SoundBar.
- `public String off()`: Desliga a SoundBar.

HomeTheaterFacade.java

A classe `HomeTheaterFacade` fornece uma interface simplificada para operações no sistema de Home Theater, encapsulando a complexidade dos componentes individuais.

Atributos:

- `private final DvdPlayer dvd`: Instância do DVD Player.
- `private final Projetor projetor`: Instância do Projetor.
- `private final SoundBar soundBar`: Instância da SoundBar.

Construtor:

- `public HomeTheaterFacade(DvdPlayer dvd, Projetor projetor, SoundBar soundBar)`: Inicializa a fachada com as instâncias dos componentes necessários.

Métodos:

- `public List<String> watchMovie(String filme)`: Inicia a reprodução de um filme, realizando uma série de ações sequenciais, como ligar o DVD Player, iniciar a reprodução, ligar o projetor, configurar a entrada e ajustar o volume da SoundBar.
- `public List<String> endMovie()`: Encerra a sessão de filme, desligando todos os componentes e finalizando o sistema de Home Theater.

TestFacade.java

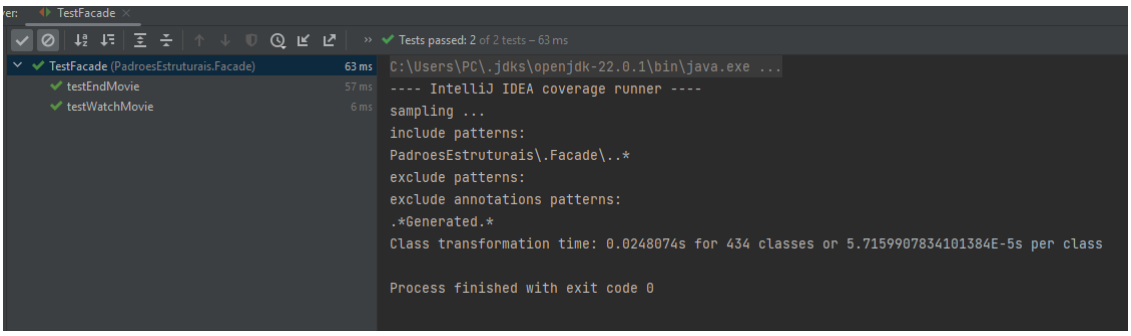
Os testes a seguir verificam o correto funcionamento da classe `HomeTheaterFacade`, que utiliza o padrão de projeto Facade para simplificar a interação com os dispositivos de um home theater.

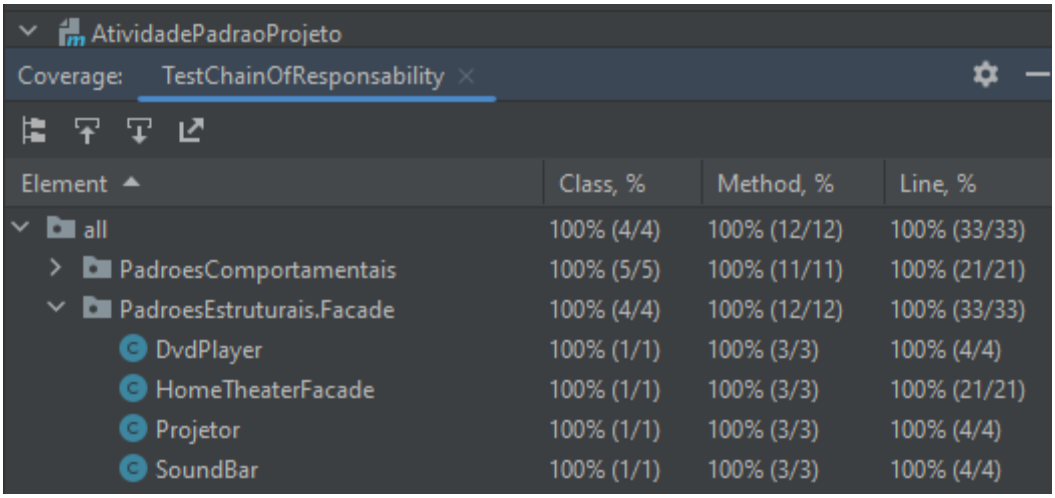
`testWatchMovie()`

- **Objetivo:** Verificar se o método `watchMovie()` da classe `HomeTheaterFacade` executa corretamente a sequência de ações para assistir a um filme.
- **Método:** Instancia objetos de `DvdPlayer`, `Projetor`, `SoundBar` e `HomeTheaterFacade`. Chama o método `watchMovie()` com o filme "The Matrix". Verifica se a lista de ações retorna possui o tamanho esperado e se cada ação ocorre na sequência correta.
- **Resultado Esperado:** O método `watchMovie()` deve ligar os dispositivos, iniciar a reprodução do filme no DVD, configurar o projetor e a SoundBar, e finalizar informando que o filme está pronto.

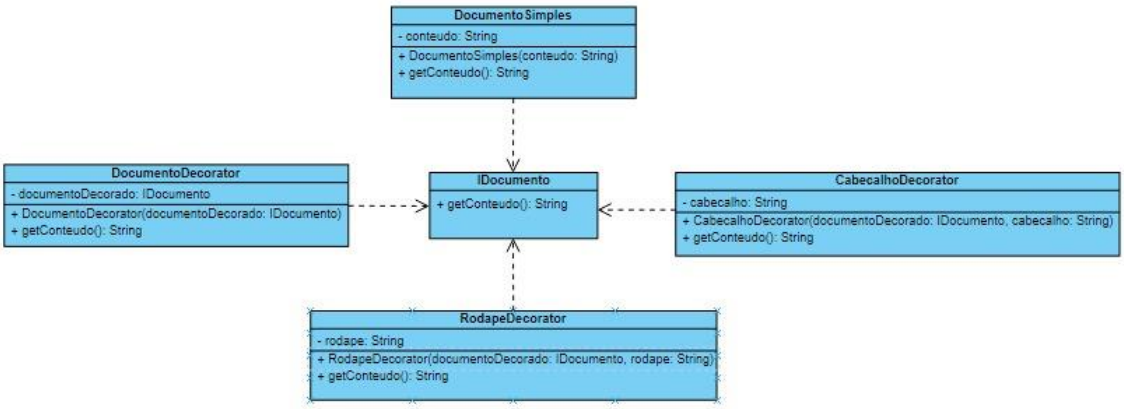
`testEndMovie()`

- **Objetivo:** Verificar se o método `endMovie()` da classe `HomeTheaterFacade` executa corretamente a sequência de ações para encerrar a exibição de um filme.
- **Método:** Instancia objetos de `DvdPlayer`, `Projetor`, `SoundBar` e `HomeTheaterFacade`. Chama o método `endMovie()`. Verifica se a lista de ações retorna possui o tamanho esperado e se cada ação ocorre na sequência correta.
- **Resultado Esperado:** O método `endMovie()` deve desligar os dispositivos e informar que o home theater foi desligado.





Padrão Decorator – Documentos



Este projeto implementa o padrão Decorator para adicionar funcionalidades a objetos de forma dinâmica, permitindo adicionar cabeçalhos e rodapés a documentos sem alterar a estrutura das classes de documento originais.

IDocumento.java

A interface `IDocumento` define o método que deve ser implementado pelas classes de documentos e seus decoradores.

Método:

- **String getConteudo()**: Retorna o conteúdo do documento.

DocumentoSimples.java

A classe `DocumentoSimples` representa um documento básico que implementa a interface `IDocumento`.

Variáveis:

- **private String conteudo**: Conteúdo do documento.

Construtor:

- **public DocumentoSimples(String conteudo)**: Inicializa uma instância de `DocumentoSimples` com o conteúdo fornecido.

Método:

- **public String getConteudo()**: Retorna o conteúdo do documento.

DocumentoDecorator.java

A classe `DocumentoDecorator` é um decorador abstrato que implementa a interface `IDocumento` e mantém uma referência a um documento que será decorado.

Variáveis:

- **protected IDocumento documentoDecorado**: Referência ao documento que será decorado.

Construtor:

- **public DocumentoDecorator(IDocumento documentoDecorado)**: Inicializa uma instância de `DocumentoDecorator` com o documento a ser decorado.

Método:

- **public String getConteudo()**: Retorna o conteúdo do documento decorado.

CabecalhoDecorator.java

A classe `CabecalhoDecorator` é um decorador concreto que adiciona um cabeçalho ao documento.

Variáveis:

- **private String cabecalho**: Cabeçalho a ser adicionado ao documento.

Construtor:

- **public CabecalhoDecorator(IDocumento documentoDecorado, String cabecalho)**: Inicializa uma instância de `CabecalhoDecorator` com o documento a ser decorado e o cabeçalho.

Método:

- **public String getConteudo()**: Retorna o conteúdo do documento com o cabeçalho adicionado no início.

RodapeDecorator.java

A classe `RodapeDecorator` é um decorador concreto que adiciona um rodapé ao documento.

Variáveis:

- **private String rodape**: Rodapé a ser adicionado ao documento.

Construtor:

- **public RodapeDecorator(IDocumento documentoDecorado, String rodape)**: Inicializa uma instância de `RodapeDecorator` com o documento a ser decorado e o rodapé.

Método:

- **public String getConteudo()**: Retorna o conteúdo do documento com o rodapé adicionado no final.

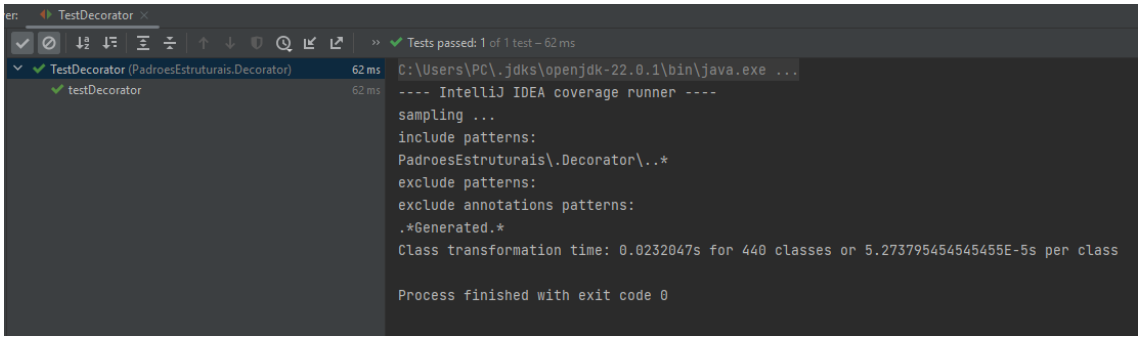
TestDecorator.java

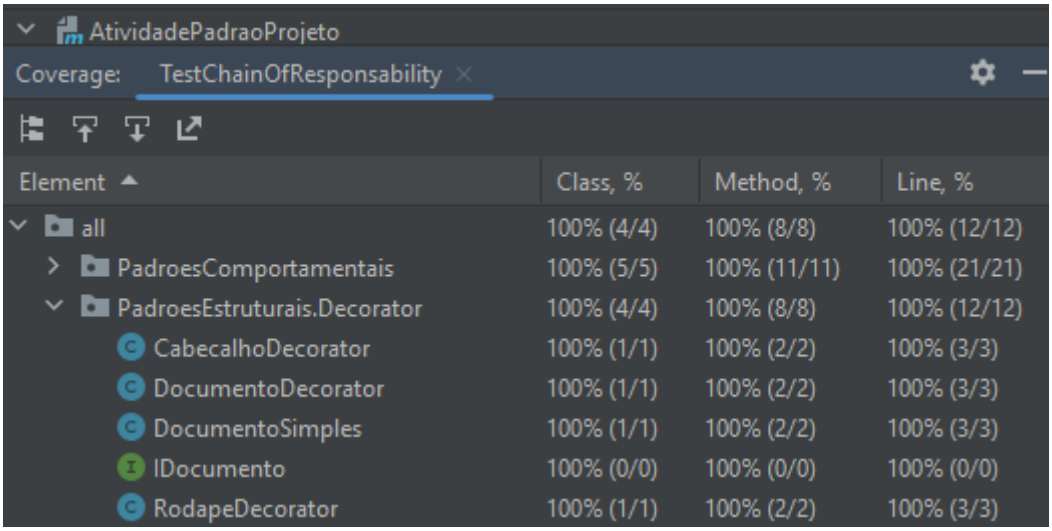
A classe `TestDecorator` testa a funcionalidade das classes de documento e seus decoradores para garantir que as funcionalidades principais estão funcionando conforme esperado.

Métodos de Teste:

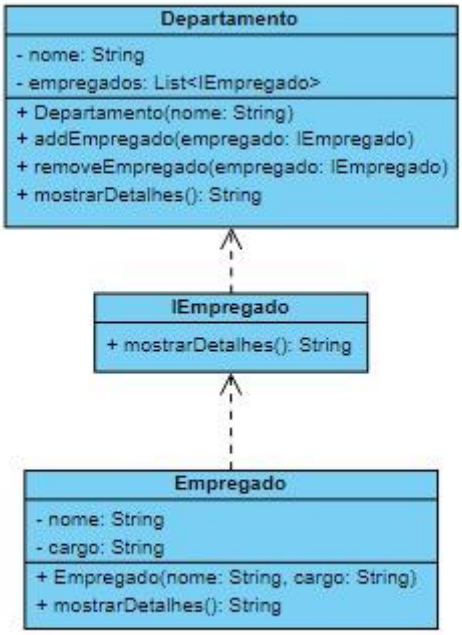
public void testDecorator()

- **Objetivo**: Verificar se a adição de um cabeçalho e um rodapé a um documento funciona corretamente.
- **Método**:
    - Criar uma instância de `DocumentoSimples` com o conteúdo "Conteudo".
    - Usar o `CabecalhoDecorator` para adicionar o cabeçalho "Cabeçalho" ao documento.
    - Usar o `RodapeDecorator` para adicionar o rodapé "Rodapé" ao documento.
    - Verificar se o conteúdo retornado é "Cabeçalho\nConteudo\nRodapé", sendo cada \n um paragrafo.
- **Resultado Esperado**: A string retornada deve ser "Cabeçalho\nConteudo\nRodapé".


er:
TestDecorator
Tests passed: 1 of 1 test – 62 ms
TestDecorator (PadroesEstruturais.Decorator) 62 ms
testDecorator 62 ms
C:\Users\PC\.jdks\openjdk-22.0.1\bin\java.exe ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
PadroesEstruturais\.Decorator\..*
exclude patterns:
exclude annotations patterns:
.*Generated.*
Class transformation time: 0.0232047s for 440 classes or 5.273795454545455E-5s per class
Process finished with exit code 0


AtividadePadraoProjeto
Coverage:
TestChainOfResponsability
Element
Class, %
Method, %
Line, %
all
100% (4/4)
100% (8/8)
100% (12/12)
PadroesComportamentais
100% (5/5)
100% (11/11)
100% (21/21)
PadroesEstruturais.Decorator
100% (4/4)
100% (8/8)
100% (12/12)
CabecalhoDecorator
100% (1/1)
100% (2/2)
100% (3/3)
DocumentoDecorator
100% (1/1)
100% (2/2)
100% (3/3)
DocumentoSimples
100% (1/1)
100% (2/2)
100% (3/3)
IDocumento
100% (0/0)
100% (0/0)
100% (0/0)
RodapeDecorator
100% (1/1)
100% (2/2)
100% (3/3)

Padrão Composite - Departamentos e Empregados



O projeto busca a implementação de um sistema que usa o padrão Composite para representar departamentos e empregados, onde um departamento pode conter vários empregados e/ou sub-departamentos em forma similar a estrutura de uma árvore.

Departamento.java

A classe `Departamento` representa um departamento dentro de uma organização, que pode conter múltiplos empregados e sub-departamentos.

Variáveis:

- **private final String nome**: Nome do departamento.
- **private final List empregados**: Lista de empregados e sub-departamentos dentro deste departamento.

Construtor:

- **public Departamento(String nome)**: Construtor que inicializa a instância `Departamento` com o nome fornecido.

Métodos:

- **public void addEmpregado(IEmpregado empregado)**: Método para adicionar um empregado ou sub-departamento ao departamento.
- **public void removeEmpregado(IEmpregado empregado)**: Método para remover um empregado ou sub-departamento do departamento.
- **public String mostrarDetalhes()**: Método que retorna uma string com os detalhes do departamento e seus empregados/sub-departamentos.
    - **detalhes**: StringBuilder que acumula os detalhes do departamento e de seus componentes.

Empregado.java

A classe `Empregado` representa um empregado individual dentro da organização.

Variáveis:

- **private final String nome**: Nome do empregado.
- **private final String cargo**: Cargo do empregado.

Construtor:

- **public Empregado(String nome, String cargo)**: Construtor que inicializa a instância `Empregado` com o nome e cargo fornecidos.

Métodos:

- **public String mostrarDetalhes()**: Método que retorna uma string com os detalhes do empregado.
    - **detalhes**: "Empregado: " + nome + ", Cargo: " + cargo + "\n"

IEmpregado.java

A interface `IEmpregado` define um método que deve ser implementado pelas classes `Empregado` e `Departamento`.

Métodos:

- **String mostrarDetalhes()**: Método que retorna uma string com os detalhes do empregado ou departamento.

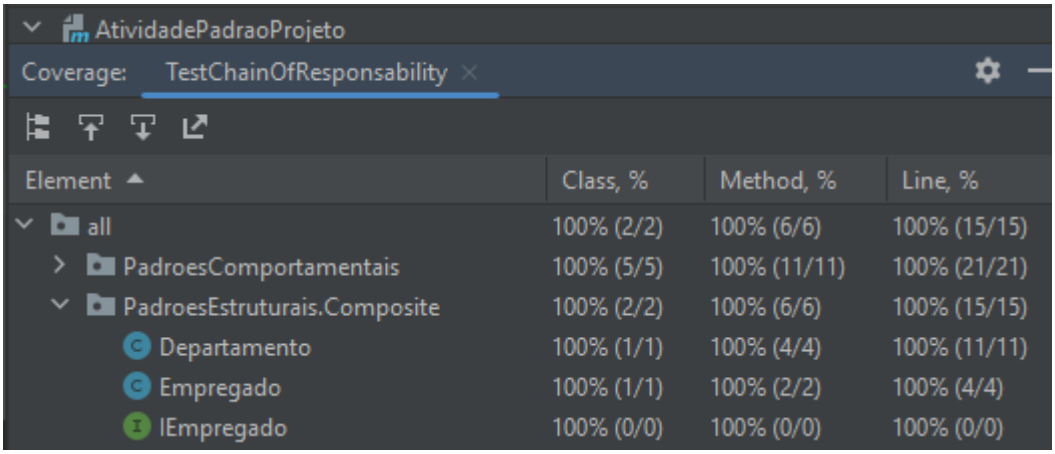
TestComposite.java

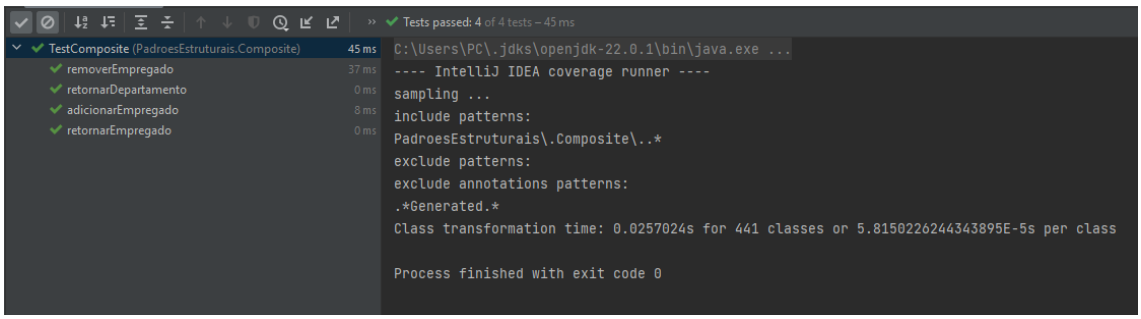
A classe `TestComposite` testa a implementação das classes `Departamento` e `Empregado` para garantir que as funcionalidades principais estão funcionando conforme esperado.

Métodos de Teste:

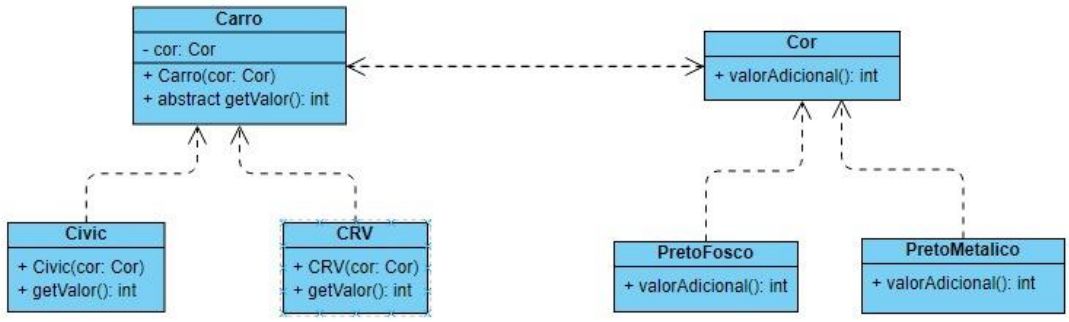
- **public void retornarDepartamento()**: Testa a criação de um departamento e a exibição de seus detalhes.
    - **Objetivo**: Verificar se a criação de um departamento e a exibição dos seus detalhes funcionam corretamente.
    - **Método**: Criar uma instância de `Departamento` com o nome "Gerencia" e chamar o método `mostrarDetalhes()`.
    - **Resultado Esperado**: A string retornada deve ser "Departamento: Gerencia\n".
- **public void retornarEmpregado()**: Testa a criação de um empregado e a exibição de seus detalhes.
    - **Objetivo**: Verificar se a criação de um empregado e a exibição dos seus detalhes funcionam corretamente.
    - **Método**: Criar uma instância de `Empregado` com o nome "Jorge" e o cargo "CEO" e chamar o método `mostrarDetalhes()`.
    - **Resultado Esperado**: A string retornada deve ser "Empregado: Jorge, Cargo: CEO".
- **public void adicionarEmpregado()**: Testa a adição de um empregado a um departamento e a exibição de seus detalhes.
    - **Objetivo**: Verificar se a adição de um empregado a um departamento funciona corretamente e se os detalhes são exibidos corretamente.
    - **Método**: Criar uma instância de `Departamento` com o nome "Gerencia" e uma instância de `Empregado` com o nome "Jorge" e o cargo "CEO". Adicionar o empregado ao departamento e chamar o método `mostrarDetalhes()`.
    - **Resultado Esperado**: A string retornada deve ser "Departamento: Gerencia \n Empregado: Jorge, Cargo: CEO", sendo cada \n um paragrafo demonstrando a estrutura de árvore do padrão.

- **public void removerEmpregado()**: Testa a remoção de um empregado de um departamento e a exibição de seus detalhes.
    - **Objetivo**: Verificar se a remoção de um empregado de um departamento funciona corretamente e se os detalhes são exibidos corretamente.
    - **Método**: Criar uma instância de `Departamento` com o nome "Gerencia" e uma instância de `Empregado` com o nome "Jorge" e o cargo "CEO". Adicionar o empregado ao departamento, removê-lo e chamar o método `mostrarDetalhes()`.
    - **Resultado Esperado**: A string retornada deve ser "Departamento: Gerencia".





Padrão Bridge – Carros



Esse projeto busca usar o padrão Bridge para a implementação de um sistema de carros que utiliza o padrão Bridge para separar a cor e marca dos carro para que depois a cor do carro possa escolhida por meio de uma referência á classe `Cor`, permitindo a fácil implementação de mais cores sem modificar os modelos e vice-versa.

CRV.java

A classe `CRV` estende a classe `Carro` e implementa a lógica específica para o carro do modelo CRV.

Construtor:

- **public CRV(Cor cor)**: Construtor que inicializa a instância `CRV` com a cor fornecida.

Métodos:

- **public int getValor()**: Método que retorna o valor do carro CRV, somando o valor base do carro com o valor adicional da cor.
  - **valorBase**: 350000
  - **valorCor**: `cor.valorAdicional()`

Carro.java

A classe abstrata `Carro` define a estrutura básica para os carros e associa uma cor a cada carro.

Variáveis:

- **protected Cor cor**: Instância da interface `Cor` que representa a cor do carro.

Construtor:

- **public Carro(Cor cor)**: Construtor que inicializa a instância `Carro` com a cor fornecida.

Métodos:

- **public abstract int getValor()**: Método abstrato que deve ser implementado pelas subclasses para retornar o valor do carro.

Civic.java

A classe `Civic` estende a classe `Carro` e implementa a lógica específica para o carro do modelo Civic.

Construtor:

- **public Civic(Cor cor)**: Construtor que inicializa a instância `Civic` com a cor fornecida.

Métodos:

- **public int getValor()**: Método que retorna o valor do carro Civic, somando o valor base do carro com o valor adicional da cor.
  - **valorBase**: 260000
  - **valorCor**: `cor.valorAdicional()`

Cor.java

A interface `Cor` define um método que deve ser implementado pelas classes que representam cores diferentes.

Métodos:

- **int valorAdicional()**: Método que retorna o valor adicional da cor.

PretoFosco.java

A classe `PretoFosco` implementa a interface `Cor` e representa a cor preta fosca.

Métodos:

- **public int valorAdicional()**: Método que retorna o valor adicional para a cor preta fosca.
    - **Valor Adicional**: 0

PretoMetalico.java

A classe `PretoMetalico` implementa a interface `Cor` e representa a cor preta metálica.

Métodos:

- **public int valorAdicional()**: Método que retorna o valor adicional para a cor preta metálica.
    - **Valor Adicional**: 5000

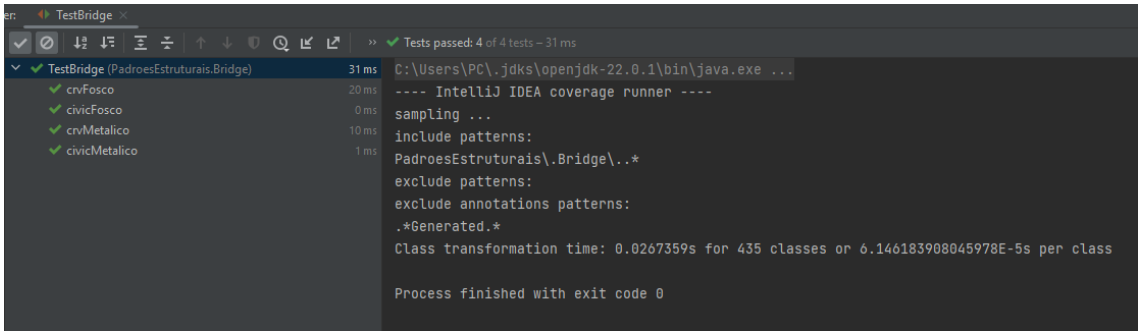
TestBridge.java

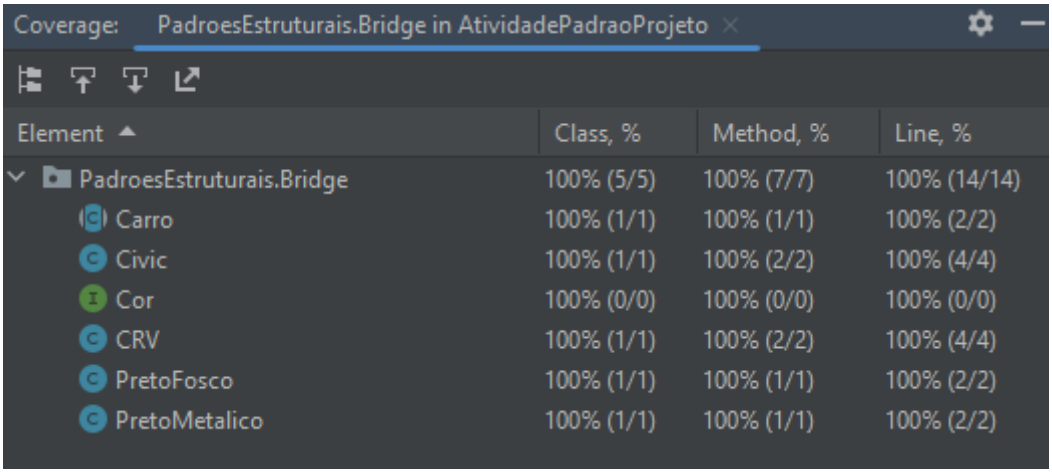
A classe `TestBridge` testa a implementação das classes `CRV`, `Civic`, `PretoFosco` e `PretoMetalico` para garantir que as funcionalidades de cálculo de valor dos carros estão funcionando conforme esperado.

Métodos de Teste:

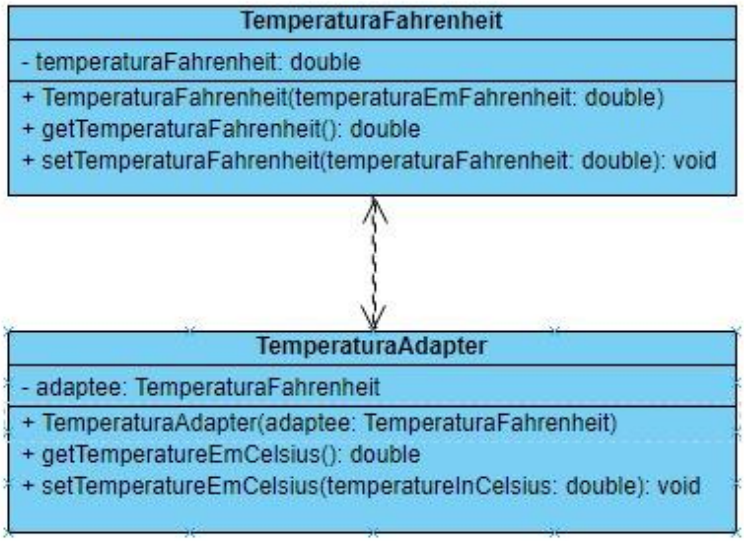
- **public void civicMetalico()**: Testa o valor do carro `Civic` com a cor `PretoMetalico`.
    - **Objetivo**: Verificar se o valor do carro Civic é calculado corretamente com a cor preta metálica.
    - **Método**: Criar uma instância de `Civic` com a cor `PretoMetalico` e chamar o método `getValor()`.
    - **Resultado Esperado**: O valor do carro deve ser 265000.
- **public void civicFosco()**: Testa o valor do carro `Civic` com a cor `PretoFosco`.
    - **Objetivo**: Verificar se o valor do carro Civic é calculado corretamente com a cor preta fosca.

    - **Método**: Criar uma instância de `Civic` com a cor `PretoFosco` e chamar o método `getValor()`.
    - **Resultado Esperado**: O valor do carro deve ser 260000.
- **public void crvMetalico()**: Testa o valor do carro `CRV` com a cor `PretoMetalico`.
    - **Objetivo**: Verificar se o valor do carro CRV é calculado corretamente com a cor preta metálica.
    - **Método**: Criar uma instância de `CRV` com a cor `PretoMetalico` e chamar o método `getValor()`.
    - **Resultado Esperado**: O valor do carro deve ser 355000.
- **public void crvFosco()**: Testa o valor do carro `CRV` com a cor `PretoFosco`.
    - **Objetivo**: Verificar se o valor do carro CRV é calculado corretamente com a cor preta fosca.
    - **Método**: Criar uma instância de `CRV` com a cor `PretoFosco` e chamar o método `getValor()`.
    - **Resultado Esperado**: O valor do carro deve ser 350000.





Padrão Adapter - Conversão de Temperatura



TemperaturaAdapter.java

A classe `TemperaturaAdapter` é uma implementação do padrão Adapter, que busca adaptar a `TemperaturaFahrenheit` para que possa ser convertida e usada com a medida Celsius.

Variáveis:

- **private final TemperaturaFahrenheit adaptee**: Instância de `TemperaturaFahrenheit` que será adaptada.

Construtor:

- **public TemperaturaAdapter(TemperaturaFahrenheit adaptee)**: Construtor que inicializa a instância `adaptee` com um objeto de `TemperaturaFahrenheit`.

Métodos:

- **public double getTemperatureEmCelsius()**: Método que retorna a temperatura em graus Celsius, convertendo o valor de Fahrenheit armazenado na instância `adaptee`.
    - Fórmula de conversão: `(adaptee.getTemperaturaFahrenheit() - 32) * 5.0 / 9.0`
- **public void setTemperatureEmCelsius(double temperatureInCelsius)**: Método que define a temperatura em graus Celsius, convertendo o valor para Fahrenheit e armazenando na instância `adaptee`.
    - Fórmula de conversão: `(temperatureInCelsius * 9.0 / 5.0) + 32`

TemperaturaFahrenheit.java

A classe `TemperaturaFahrenheit` gerencia a temperatura em graus Fahrenheit.

Variáveis:

- **private double temperaturaFahrenheit**: A temperatura armazenada em graus Fahrenheit.

Construtor:

- **public TemperaturaFahrenheit(double temperaturaEmFahrenheit)**: Construtor que inicializa a temperatura com o valor fornecido em graus Fahrenheit.

Métodos:

- **public double getTemperaturaFahrenheit()**: Método que retorna a temperatura armazenada em graus Fahrenheit.
- **public void setTemperaturaFahrenheit(double temperaturaFahrenheit)**: Método que define a temperatura armazenada com o valor fornecido em graus Fahrenheit.

TestAdapter.java

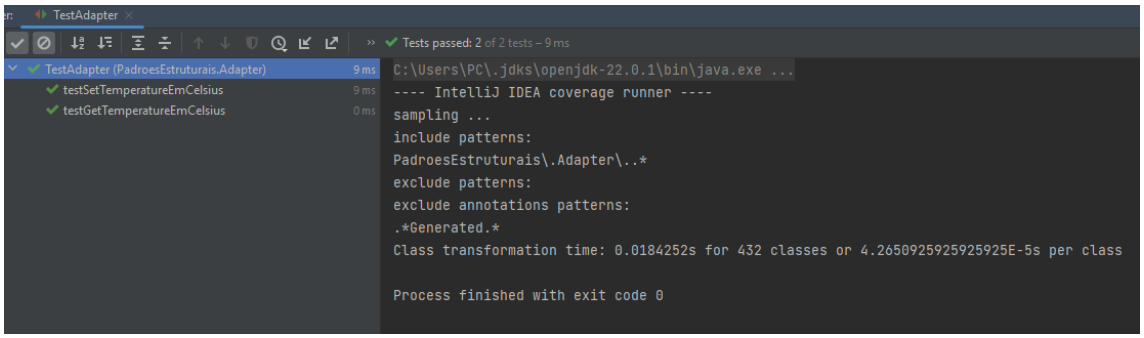
A classe `TestAdapter` busca garantir que as funcionalidades do padrão Adapter para a conversão de temperatura estejam funcionando conforme esperado.

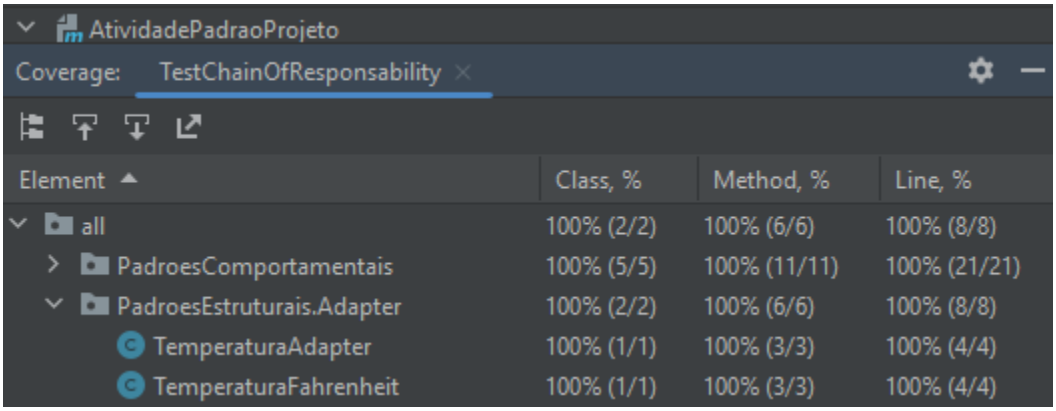
Variáveis:

- **private TemperaturaAdapter adapter**: Instância da classe `TemperaturaAdapter` que será testada.
- **private TemperaturaFahrenheit temperaturaFahrenheit**: Instância da classe `TemperaturaFahrenheit` que será usada pela classe `TemperaturaAdapter`.

Métodos:

- **public void setUp()**: Método que inicializa as instâncias de `TemperaturaFahrenheit` e `TemperaturaAdapter` antes de cada teste.
- **public void testGetTemperatureEmCelsius()**: Método que testa se a conversão de Fahrenheit para Celsius está correta.
    - **Objetivo**: Verificar se a temperatura em Celsius é calculada corretamente.
    - **Método**: Chamar `adapter.getTemperatureEmCelsius()` e comparar o resultado com o valor esperado (0.0).
    - **Resultado Esperado**: A temperatura em Celsius deve ser 0.0 quando a temperatura em Fahrenheit é 32.0.
- **public void testSetTemperatureEmCelsius()**: Método que testa se a conversão de Celsius para Fahrenheit está correta.
    - **Objetivo**: Verificar se a temperatura em Fahrenheit é configurada corretamente ao definir a temperatura em Celsius.
    - **Método**: Chamar `adapter.setTemperatureEmCelsius(100.0)` e verificar o valor de `temperaturaFahrenheit.getTemperaturaFahrenheit()`.
    - **Resultado Esperado**: A temperatura em Fahrenheit deve ser 212.0 quando a temperatura em Celsius é definida como 100.0.


TestAdapter
Tests passed: 2 of 2 tests – 9 ms
TestAdapter (PadroesEstruturais.Adapter) 9 ms
testSetTemperatureEmCelsius 9 ms
testGetTemperatureEmCelsius 0 ms
C:\Users\PC\.jdks\openjdk-22.0.1\bin\java.exe ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
PadroesEstruturais\.Adapter\..*
exclude patterns:
exclude annotations patterns:
.*Generated.*
Class transformation time: 0.0184252s for 432 classes or 4.2650925925925925E-5s per class
Process finished with exit code 0


AtividadePadraoProjeto
Coverage: TestChainOfResponsability
Element
Class, %
Method, %
Line, %
all
100% (2/2)
100% (6/6)
100% (8/8)
PadroesComportamentais
100% (5/5)
100% (11/11)
100% (21/21)
PadroesEstruturais.Adapter
100% (2/2)
100% (6/6)
100% (8/8)
TemperaturaAdapter
100% (1/1)
100% (3/3)
100% (4/4)
TemperaturaFahrenheit
100% (1/1)
100% (3/3)
100% (4/4)