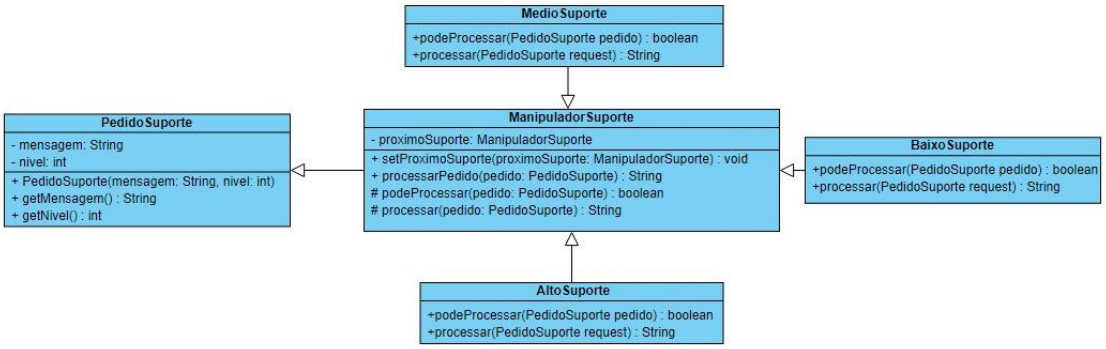
Padrão Chain of Responsibility - Sistema de Suporte



Este projeto usa o padrão de projeto Chain of Responsibility para processar pedidos de suporte em diferentes níveis (baixo, médio e alto). Cada nível de suporte trata pedidos conforme sua capacidade e, se não puder processar um pedido, o passa adiante na cadeia.

AltoSuporte.java

A classe `AltoSuporte` é um manipulador concreto que trata pedidos de suporte de nível alto (nível 3).

Métodos

- `@Override protected boolean podeProcessar(PedidoSuporte pedido)`: Verifica se o pedido é de nível 3.
- `@Override protected String processar(PedidoSuporte request)`: Processa o pedido de suporte de nível alto e retorna uma mensagem indicando que o pedido foi tratado.

BaixoSuporte.java

A classe `BaixoSuporte` é um manipulador concreto que trata pedidos de suporte de nível baixo (nível 1).

Métodos

- `@Override protected boolean podeProcessar(PedidoSuporte pedido)`: Verifica se o pedido é de nível 1.

- `@Override protected String processar(PedidoSuporte request)`: Processa o pedido de suporte de nível baixo e retorna uma mensagem indicando que o pedido foi tratado.

ManipuladorSuporte.java

A classe abstrata `ManipuladorSuporte` define a estrutura básica para um manipulador de suporte na cadeia.

Atributos

- `protected ManipuladorSuporte proximoSuporte`: Referência ao próximo manipulador na cadeia.

Métodos

- `public void setProximoSuporte(ManipuladorSuporte proximoSuporte)`: Define o próximo manipulador na cadeia.
- `public String processarPedido(PedidoSuporte pedido)`: Tenta processar o pedido e, se não puder, passa para o próximo manipulador na cadeia. Retorna uma mensagem de sucesso ou falha.
- `protected abstract boolean podeProcessar(PedidoSuporte request)`: Método abstrato para verificar se o manipulador pode processar o pedido.
- `protected abstract String processar(PedidoSuporte request)`: Método abstrato para processar o pedido.

MedioSuporte.java

A classe `MedioSuporte` é um manipulador concreto que trata pedidos de suporte de nível médio (nível 2).

Métodos

- `@Override protected boolean podeProcessar(PedidoSuporte pedido)`: Verifica se o pedido é de nível 2.
- `@Override protected String processar(PedidoSuporte request)`: Processa o pedido de suporte de nível médio e retorna uma mensagem indicando que o pedido foi tratado.

PedidoSuporte.java

A classe `PedidoSuporte` representa um pedido de suporte.

Atributos

- `private String mensagem`: A mensagem do pedido de suporte.
- `private int nivel`: O nível do pedido de suporte.

Construtor

- `public PedidoSuporte(String mensagem, int nivel)`: Inicializa o pedido de suporte com uma mensagem e um nível.

Métodos

- `public String getMensagem()`: Retorna a mensagem do pedido de suporte.
- `public int getNivel()`: Retorna o nível do pedido de suporte.

TestChainOfResponsability.java

O `TestChainOfResponsability.java` busca verificar o funcionamento do padrão Chain of Responsability nos três níveis de suporte que o projeto apresenta.

`public void setUp()`

- **Objetivo:** Configurar a cadeia de responsabilidade antes de cada teste.
- **Método:**
    - Criar instâncias de `BaixoSuporte`, `MedioSuporte`, e `AltoSuporte`.
    - Configurar a cadeia, onde `BaixoSuporte` aponta para `MedioSuporte` e `MedioSuporte` aponta para `AltoSuporte`.
- **Resultado Esperado:**
    - A cadeia de responsabilidade deve estar corretamente configurada com `suporteBaixo` apontando para `suporteMedio` e `suporteMedio` apontando para `suporteAlto`.

`public void testePedidoSuporteBaixo()`

- **Objetivo:** Verificar o processamento de um pedido de suporte de baixo nível.
- **Método:**
    - Criar um pedido de suporte com nível 1.
    - Chamar `processarPedido` no manipulador de baixo suporte (`suporteBaixo`).

    - Verificar se o resultado indica que o pedido foi tratado pelo baixo suporte.

- **Resultado Esperado:**
    - O resultado deve ser `"Baixo Suporte: Tratando pedido - Problema de baixo nível"`.

`public void testePedidoSuporteMedio()`

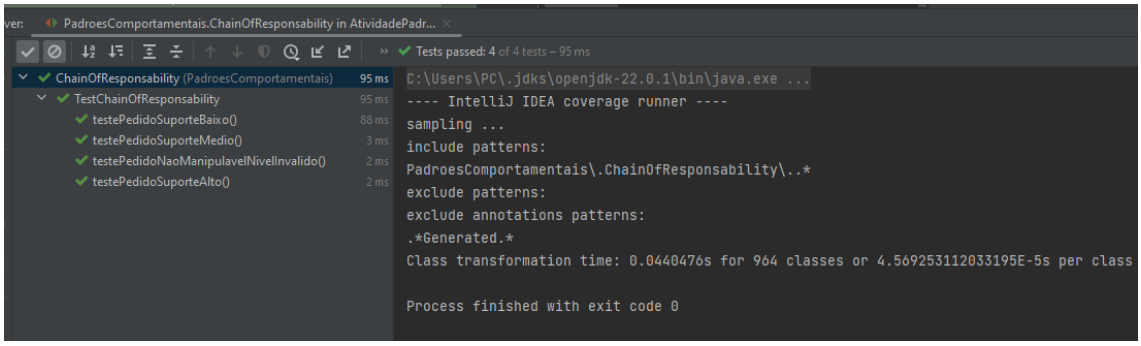
- **Objetivo:** Verificar o processamento de um pedido de suporte de nível médio.
- **Método:**
    - Criar um pedido de suporte com nível 2.
    - Chamar `processarPedido` no manipulador de baixo suporte (`suporteBaixo`).
    - Verificar se o resultado indica que o pedido foi tratado pelo médio suporte.
- **Resultado Esperado:**
    - O resultado deve ser `"Medio Suporte: Tratando pedido - Problema de nível médio"`.

`public void testePedidoSuporteAlto()`

- **Objetivo:** Verificar o processamento de um pedido de suporte de alto nível.
- **Método:**
    - Criar um pedido de suporte com nível 3.
    - Chamar `processarPedido` no manipulador de baixo suporte (`suporteBaixo`).
    - Verificar se o resultado indica que o pedido foi tratado pelo alto suporte.
- **Resultado Esperado:**
    - O resultado deve ser `"Alto Suporte: Tratando pedido - Problema de alto nível"`.

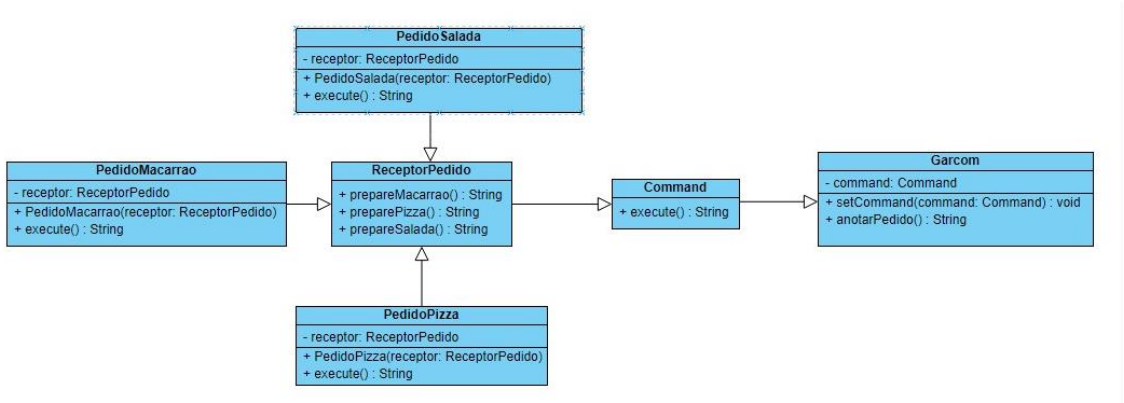
`public void testePedidoNaoManipulavelNivelInvalido()`

- **Objetivo:** Verificar o tratamento de um pedido de suporte com nível inválido.
- **Método:**
    - Criar um pedido de suporte com nível 4.
    - Chamar `processarPedido` no manipulador de baixo suporte (`suporteBaixo`).
    - Verificar se o resultado indica que o pedido não foi processado.
- **Resultado Esperado:**
    - O resultado deve ser `"Pedido não processado."`.



Element	Class, %	Method, %	Line, %
PadroesComportamentais.ChainOfRespons	100% (5/5)	100% (11/11)	100% (21/21)
AltoSuporte	100% (1/1)	100% (2/2)	100% (3/3)
BaixoSuporte	100% (1/1)	100% (2/2)	100% (3/3)
ManipuladorSuporte	100% (1/1)	100% (2/2)	100% (7/7)
MedioSuporte	100% (1/1)	100% (2/2)	100% (3/3)
PedidoSuporte	100% (1/1)	100% (3/3)	100% (5/5)

Padrão Command - Sistema de Pedidos em Restaurante



Este projeto usa o padrão de projeto Command para tratar os pedidos o como um objeto, permitindo que você trate essas solicitações de maneira flexível e reutilizáveis.

Command.java

A interface `Command` define um método para executar um comando.

Métodos

- `String execute()`: Executa a ação encapsulada pelo comando.

Garcom.java

A classe `Garcom` é o invocador que mantém uma referência a um comando e executa o comando quando solicitado.

Atributos

- `private Command command`: Referência ao comando a ser executado.

Métodos

- `public void setCommand(Command command)`: Define o comando a ser executado pelo garçom.
- `public String anotarPedido()`: Executa o comando definido e retorna a mensagem de execução.

PedidoMacarrao.java

A classe `PedidoMacarrao` implementa a interface `Command` e encapsula a solicitação de preparo de macarrão.

Atributos

- `private ReceptorPedido receptor`: Referência ao receptor que irá processar o pedido.

Construtor

- `public PedidoMacarrao(ReceptorPedido receptor)`: Inicializa o pedido com o receptor apropriado.

Métodos

- `@Override public String execute()`: Executa a ação de preparar macarrão através do receptor.

PedidoPizza.java

A classe `PedidoPizza` implementa a interface `Command` e encapsula a solicitação de preparo de pizza.

Atributos

- `private ReceptorPedido receptor`: Referência ao receptor que irá processar o pedido.

Construtor

- `public PedidoPizza(ReceptorPedido receptor)`: Inicializa o pedido com o receptor apropriado.

Métodos

- `@Override public String execute()`: Executa a ação de preparar pizza através do receptor.

PedidoSalada.java

A classe `PedidoSalada` implementa a interface `Command` e encapsula a solicitação de preparo de salada.

Atributos

- `private ReceptorPedido receptor`: Referência ao receptor que irá processar o pedido.

Construtor

- `public PedidoSalada(ReceptorPedido receptor)`: Inicializa o pedido com o receptor apropriado.

Métodos

- `@Override public String execute()`: Executa a ação de preparar salada através do receptor.

ReceptorPedido.java

A classe `ReceptorPedido` é responsável por processar os pedidos reais, fornecendo métodos para preparar cada tipo de prato.

Métodos

- `public String preparePizza()`: Prepara uma pizza e retorna uma mensagem.
- `public String prepareMacarrao()`: Prepara um macarrão e retorna uma mensagem.
- `public String prepareSalada()`: Prepara uma salada e retorna uma mensagem.

TestCommand.java

O `TestCommand.java` busca verificar o comportamento do padrão de design Command no contexto de um sistema de pedidos em um restaurante. Os testes verificam se os comandos para diferentes tipos de pedidos (Pizza, Macarrão, Salada) são corretamente processados.

`public void testPedidoPizza()`

- **Objetivo:** Verificar o processamento de um pedido de pizza.
- **Método:**

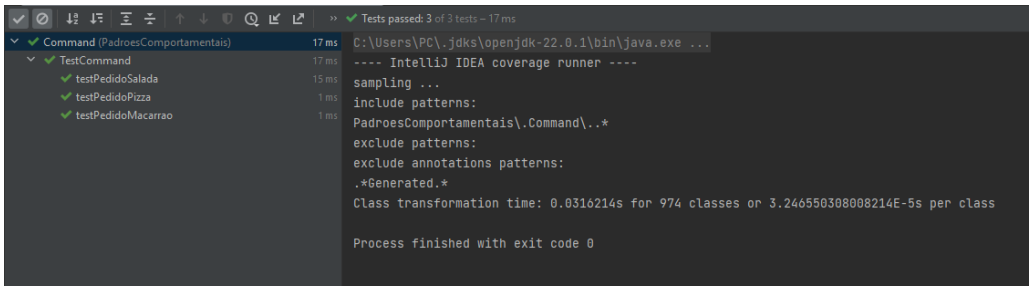
    - Criar uma instância de `ReceptorPedido`.
    - Criar uma instância de `PedidoPizza` passando o receptor criado.
    - Criar uma instância de `Garcom` e configurar o comando como `PedidoPizza`.
    - Chamar o método `anotarPedido` no garçom.
    - Verificar se o resultado indica que a pizza está sendo feita.
- **Resultado Esperado:**
    - O resultado deve ser `"A Pizza está sendo feita."`.

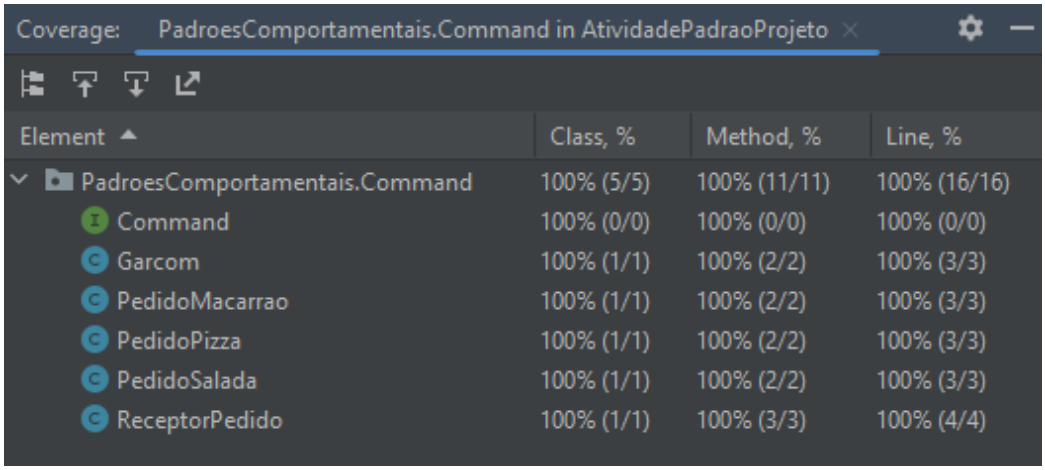
`public void testPedidoMacarrao()`

- **Objetivo:** Verificar o processamento de um pedido de macarrão.
- **Método:**
    - Criar uma instância de `ReceptorPedido`.
    - Criar uma instância de `PedidoMacarrao` passando o receptor criado.
    - Criar uma instância de `Garcom` e configurar o comando como `PedidoMacarrao`.
    - Chamar o método `anotarPedido` no garçom.
    - Verificar se o resultado indica que o macarrão está sendo feito.
- **Resultado Esperado:**
    - O resultado deve ser `"O Macarrao está sendo feito."`.

`public void testPedidoSalada()`

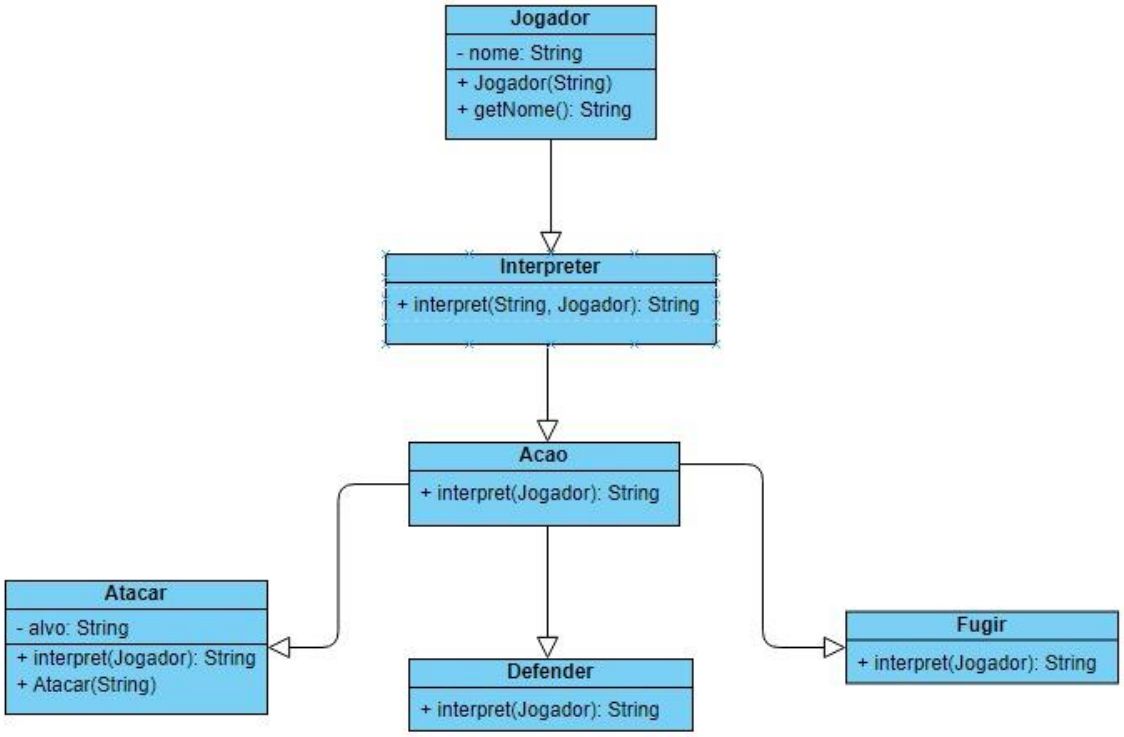
- **Objetivo:** Verificar o processamento de um pedido de salada.
- **Método:**
    - Criar uma instância de `ReceptorPedido`.
    - Criar uma instância de `PedidoSalada` passando o receptor criado.
    - Criar uma instância de `Garcom` e configurar o comando como `PedidoSalada`.
    - Chamar o método `anotarPedido` no garçom.
    - Verificar se o resultado indica que a salada está sendo feita.
- **Resultado Esperado:**
    - O resultado deve ser `"A Salada está sendo feita."`.





Element	Class, %	Method, %	Line, %
PadroesComportamentais.Command	100% (5/5)	100% (11/11)	100% (16/16)
Command	100% (0/0)	100% (0/0)	100% (0/0)
Garcom	100% (1/1)	100% (2/2)	100% (3/3)
PedidoMacarrao	100% (1/1)	100% (2/2)	100% (3/3)
PedidoPizza	100% (1/1)	100% (2/2)	100% (3/3)
PedidoSalada	100% (1/1)	100% (2/2)	100% (3/3)
ReceptorPedido	100% (1/1)	100% (3/3)	100% (4/4)

Padrão Interpreter - Ações de Jogo



Este projeto usa o padrão de projeto Interpreter em um sistema de jogo para interpretar e executar ações de um jogador com base em comandos textuais.

Acao.java

A interface `Acao` define um método para interpretar ações de um jogador.

Métodos

- `String interpret(Jogador jogador)`: Interpreta a ação e retorna uma descrição da ação realizada pelo jogador.
    - `jogador`: A instância do jogador que realiza a ação.

Atacar.java

A classe `Atacar` implementa a interface `Acao` e representa a ação de atacar um alvo.

Atributos

- `private String alvo`: O alvo a ser atacado.

Construtor

- `public Atacar(String alvo)`: Inicializa a ação com o alvo fornecido.
    - `alvo`: O nome do alvo a ser atacado.

Métodos

- `@Override public String interpret(Jogador jogador)`: Retorna uma descrição da ação de ataque realizada pelo jogador.
    - `jogador`: A instância do jogador que realiza a ação.

Defender.java

A classe `Defender` implementa a interface `Acao` e representa a ação de defender.

Métodos

- `@Override public String interpret(Jogador jogador)`: Retorna uma descrição da ação de defesa realizada pelo jogador.
    - `jogador`: A instância do jogador que realiza a ação.

Fugir.java

A classe `Fugir` implementa a interface `Acao` e representa a ação de fugir.

Métodos

- `@Override public String interpret(Jogador jogador)`: Retorna uma descrição da ação de fuga realizada pelo jogador.
    - `jogador`: A instância do jogador que realiza a ação.

Interpreter.java

A classe `Interpreter` é responsável por interpretar comandos textuais e executar as ações correspondentes.

Métodos

- `public String interpret(String contexto, Jogador jogador)`: Interpreta o contexto textual e executa a ação correspondente, retornando uma descrição da ação realizada.
    - `contexto`: O comando textual a ser interpretado.
    - `jogador`: A instância do jogador que realiza a ação.

Jogador.java

A classe `Jogador` representa um jogador no jogo.

Atributos

- `private String nome`: O nome do jogador.

Construtor

- `public Jogador(String nome)`: Inicializa o jogador com o nome fornecido.
    - `nome`: O nome do jogador.

Métodos

- `public String getNome()`: Retorna o nome do jogador.

TestInterpreter.java

O `TestInterpreter.java` contém testes unitários para validar o comportamento do padrão de design Interpreter no contexto de um jogo. Os testes verificam a funcionalidade do método `interpret` da classe `Interpreter` para diferentes comandos dados por um jogador.

`public void testeAtaque()`

- **Objetivo:** Verificar se o comando "atacar" é interpretado corretamente.
- **Método:**
    - Criar uma instância de `Interpreter`.
    - Criar uma instância de `Jogador` com o nome "player1".
    - Chamar o método `interpret` com o comando "atacar alvo1" e o jogador.
    - Verificar se a saída gerada corresponde ao valor esperado.
- **Resultado Esperado:**
    - A saída deve ser `"player1 ataca alvo1."`.

`public void testeDefender()`

- **Objetivo:** Verificar se o comando "defender" é interpretado corretamente.

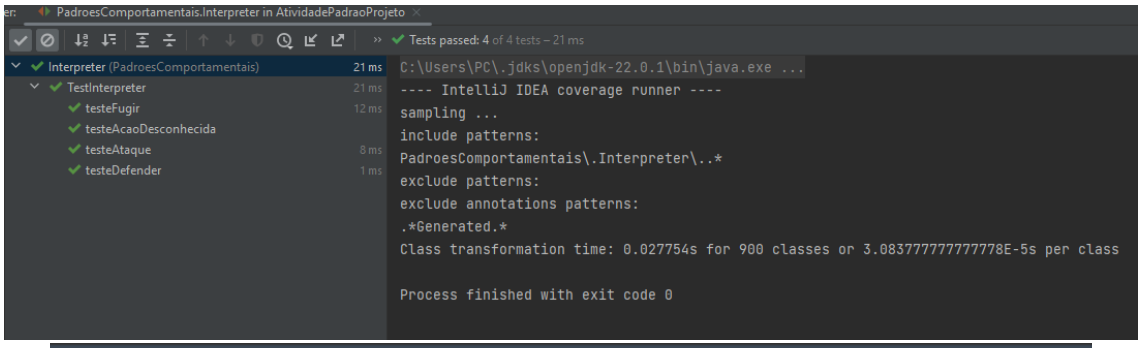
- **Método:**
    - Criar uma instância de `Interpreter`.
    - Criar uma instância de `Jogador` com o nome "player1".
    - Chamar o método `interpret` com o comando "defender" e o jogador.
    - Verificar se a saída gerada corresponde ao valor esperado.
- **Resultado Esperado:**
    - A saída deve ser `"player1 está defendendo."`.

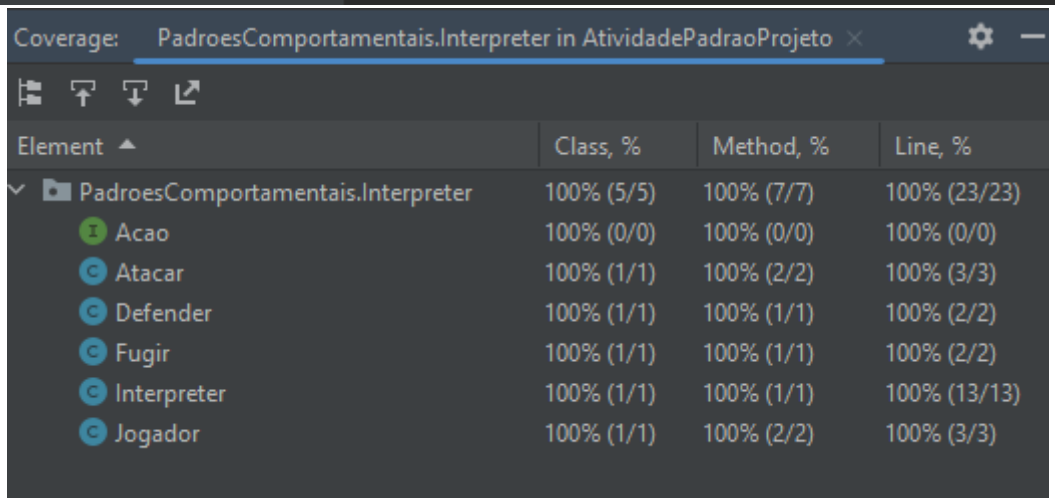
`public void testeFugir()`

- **Objetivo:** Verificar se o comando "fugir" é interpretado corretamente.
- **Método:**
    - Criar uma instância de `Interpreter`.
    - Criar uma instância de `Jogador` com o nome "player1".
    - Chamar o método `interpret` com o comando "fugir" e o jogador.
    - Verificar se a saída gerada corresponde ao valor esperado.
- **Resultado Esperado:**
    - A saída deve ser `"player1 foge da batalha."`.

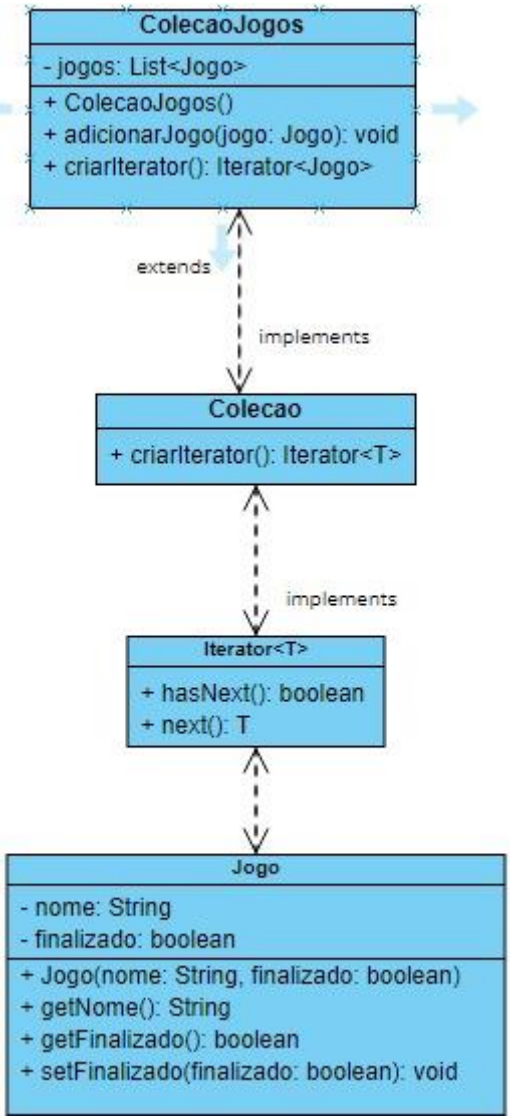
`public void testeAcaoDesconhecida()`

- **Objetivo:** Verificar o comportamento do sistema quando um comando não reconhecido é dado.
- **Método:**
    - Criar uma instância de `Interpreter`.
    - Criar uma instância de `Jogador` com o nome "player1".
    - Chamar o método `interpret` com o comando "usar item" e o jogador.
    - Verificar se a saída gerada corresponde ao valor esperado.
- **Resultado Esperado:**
    - A saída deve ser `"Ação não reconhecida."`.


PadroesComportamentais.Interpreter in AtividadePadraoProjeto
Tests passed: 4 of 4 tests – 21 ms
Interpreter (PadroesComportamentais) 21 ms
TestInterpreter 21 ms
testeFugir 12 ms
testeAcaoDesconhecida
testeAtaque 8 ms
testeDefender 1 ms
C:\Users\PC\.jdks\openjdk-22.0.1\bin\java.exe ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
PadroesComportamentais\.Interpreter\..*
exclude patterns:
exclude annotations patterns:
.*Generated.*
Class transformation time: 0.027754s for 900 classes or 3.083777777777778E-5s per class
Process finished with exit code 0


Coverage: PadroesComportamentais.Interpreter in AtividadePadraoProjeto
Element
Class, %
Method, %
Line, %
PadroesComportamentais.Interpreter 100% (5/5) 100% (7/7) 100% (23/23)
Acao 100% (0/0) 100% (0/0) 100% (0/0)
Atacar 100% (1/1) 100% (2/2) 100% (3/3)
Defender 100% (1/1) 100% (1/1) 100% (2/2)
Fugir 100% (1/1) 100% (1/1) 100% (2/2)
Interpreter 100% (1/1) 100% (1/1) 100% (13/13)
Jogador 100% (1/1) 100% (2/2) 100% (3/3)

Padrão Iterator - Coleção de Jogos



Este projeto implementa o padrão de projeto Iterator para permitir a navegação através de uma coleção de objetos (neste caso, jogos).

Colecao.java

A interface `Colecao` define um método para criar um iterator.

Métodos

- `Iterator<Jogo> criarIterator()`: Cria e retorna um iterator para a coleção.

ColecaoJogos.java

A classe `ColecaoJogos` implementa a interface `Colecao` e mantém uma coleção de objetos `Jogo`.

Atributos

- `private List<Jogo> jogos`: Lista que armazena os jogos.

Construtor

- `public ColecaoJogos()`: Inicializa a coleção de jogos com uma lista vazia.

Métodos

- `public void adicionarJogo(Jogo jogo)`: Adiciona um jogo à coleção.
- `@Override public Iterator<Jogo> criarIterator()`: Cria e retorna um iterator para a coleção de jogos.

Classe Interna `BibliotecaIterator`

A classe `BibliotecaIterator` é uma classe interna de `ColecaoJogos` que implementa a interface `Iterator` para iterar através da coleção de jogos.

Atributos

- `private int indiceAtual`: Índice atual do iterator na lista de jogos.

Métodos

- `@Override public boolean hasNext()`: Retorna `true` se houver mais elementos na coleção a serem iterados, `false` caso contrário.
- `@Override public Jogo next()`: Retorna o próximo elemento na coleção.

Iterator.java

A interface `Iterator` define os métodos necessários para iterar através de uma coleção de objetos.

Métodos

- `boolean hasNext()`: Verifica se há mais elementos na coleção.
- `T next()`: Retorna o próximo elemento na coleção.

Jogo.java

A classe `Jogo` representa um jogo com nome e status de finalização.

Atributos

- `private final String nome`: Nome do jogo.
- `private boolean finalizado`: Indica se o jogo foi finalizado.

Construtor

- `public Jogo(String nome, boolean finalizado)`: Inicializa um jogo com nome e status de finalização.

Métodos

- `public String getNome()`: Retorna o nome do jogo.
- `public boolean getFinalizado()`: Retorna o status de finalização do jogo.
- `public void setFinalizado(boolean finalizado)`: Define o status de finalização do jogo.

TestIterator.java

O `TestIterator.java` contém testes unitários para verificar o funcionamento do padrão de design Iterator no projeto de coleção de jogos.

`public void testIteratorQuantidade()`

- **Objetivo:** Verificar a contagem de jogos que foram finalizados na coleção.
- **Método:**
    - Criar uma instância de `ColecaoJogos`.
    - Adicionar quatro jogos à coleção, com diferentes valores para o atributo `finalizado`.
    - Criar um iterador para a coleção de jogos.
    - Iterar sobre a coleção usando o iterador, contando o número de jogos que foram finalizados (`finalizado` é `true`).
    - Verificar se a contagem corresponde ao valor esperado.
- **Resultado Esperado:**
    - A contagem de jogos finalizados deve ser 3.

`public void testSetFinalizado()`

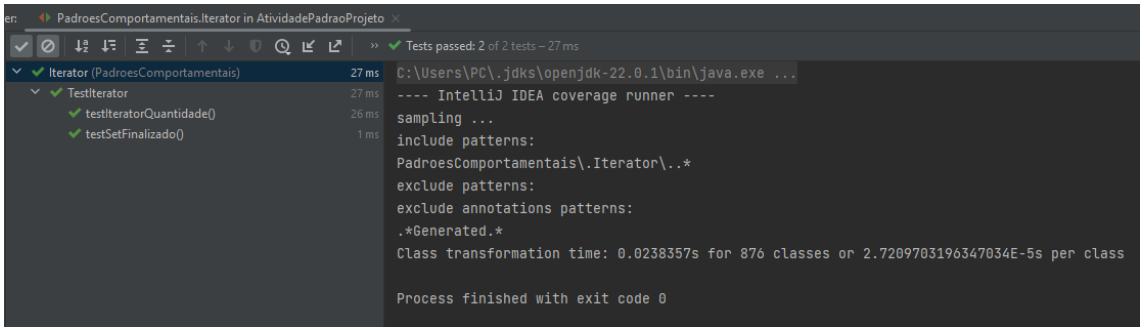
- **Objetivo:** Verificar se é possível modificar o atributo `finalizado` de um jogo durante a iteração e contar corretamente os jogos finalizados após a modificação.

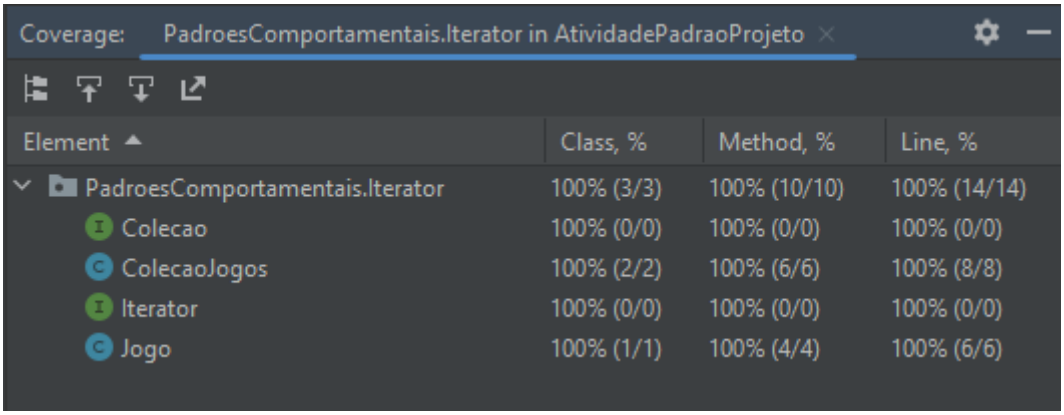
- **Método:**

    - Criar uma instância de `ColecaoJogos`.
    - Adicionar quatro jogos à coleção, com diferentes valores para o atributo `finalizado`.
    - Criar um iterador para a coleção de jogos.
    - Iterar sobre a coleção usando o iterador, modificando o atributo `finalizado` do jogo "Red Dead Redemption 2" para `true`.
    - Contar o número de jogos que foram finalizados após a modificação.
    - Verificar se a contagem corresponde ao valor esperado.

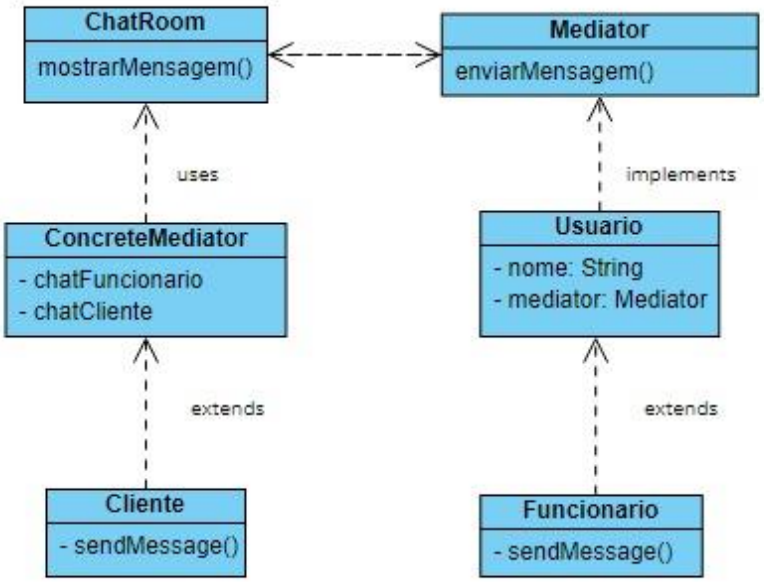
- **Resultado Esperado:**

    - A contagem de jogos finalizados deve ser 4.





Padrão Mediator - Sistema de Chat



Este projeto implementa o padrão de projeto Mediator para facilitar a comunicação entre diferentes tipos de usuários (Clientes e Funcionários) em um sistema de chat, sem que eles precisem referenciar diretamente uns aos outros.

ChatRoom.java

A classe `ChatRoom` representa uma sala de chat onde as mensagens são exibidas.

Métodos

- `public String mostrarMensagem(String message, Usuario usuario, String destino)`: Formata e retorna uma mensagem indicando quem enviou a mensagem e para onde foi enviada.

Cliente.java

A classe `Cliente` representa um usuário do tipo cliente que herda da classe `Usuario`.

Construtor

- `public Cliente(String nome, Mediator mediator)`: Inicializa um cliente com um nome e um mediador.

ConcreteMediator.java

A classe `ConcreteMediator` implementa a interface `Mediator` e coordena a comunicação entre `Funcionario` e `Cliente`.

Atributos

- `private ChatRoom chatFuncionario`: Sala de chat para funcionários.
- `private ChatRoom chatCliente`: Sala de chat para clientes.

Construtor

- `public ConcreteMediator(ChatRoom chatFuncionario, ChatRoom chatCliente)`: Inicializa o mediador com as salas de chat específicas para funcionários e clientes.

Métodos

- `@Override public String enviarMensagem(String message, Usuario usuario)`: Envia uma mensagem para a sala de chat apropriada com base no tipo de usuário. Retorna a mensagem formatada ou um aviso de tipo de usuário desconhecido.

Funcionario.java

A classe `Funcionario` representa um usuário do tipo funcionário que herda da classe `Usuario`.

Construtor

- `public Funcionario(String name, Mediator mediator)`: Inicializa um funcionário com um nome e um mediador.

Mediator.java

A interface `Mediator` define o método necessário para enviar mensagens entre usuários.

Métodos

- `String enviarMensagem(String mensagem, Usuario usuario)`: Envia uma mensagem de um usuário.

Usuario.java

A classe abstrata `Usuario` representa um usuário genérico do sistema de chat.

Atributos

- `protected String nome`: Nome do usuário.
- `protected Mediator mediator`: Mediador associado ao usuário.

Construtor

- `public Usuario(String nome, Mediator mediator)`: Inicializa um usuário com um nome e um mediador.

Métodos

- `public String sendMessage(String message)`: Envia uma mensagem através do mediador.
- `public String getNome()`: Retorna o nome do usuário.

UsuarioDesconhecido.java

A classe `UsuarioDesconhecido` representa um usuário de tipo desconhecido que herda da classe `Usuario`.

Construtor

- `public UsuarioDesconhecido(String nome, Mediator mediator)`: Inicializa um usuário desconhecido com um nome e um mediador.

TestMediator.java

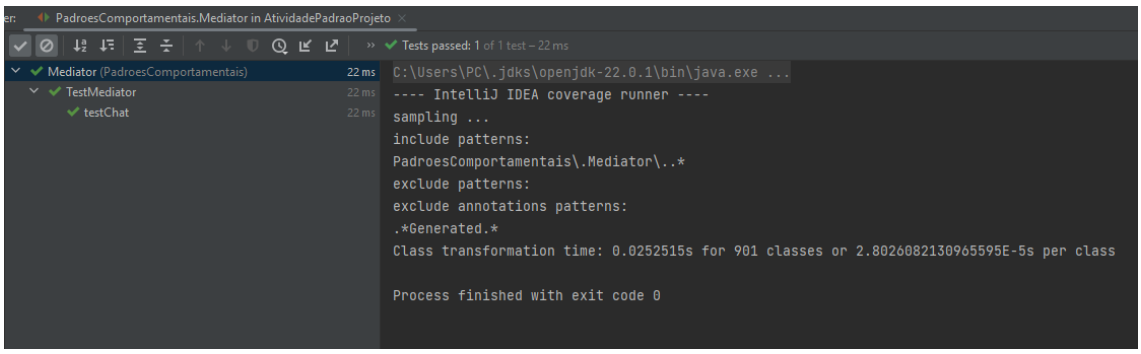
O `TestMediator.java` para verificar o funcionamento de uma sala de chat que gerencia a comunicação entre diferentes tipos de usuários.

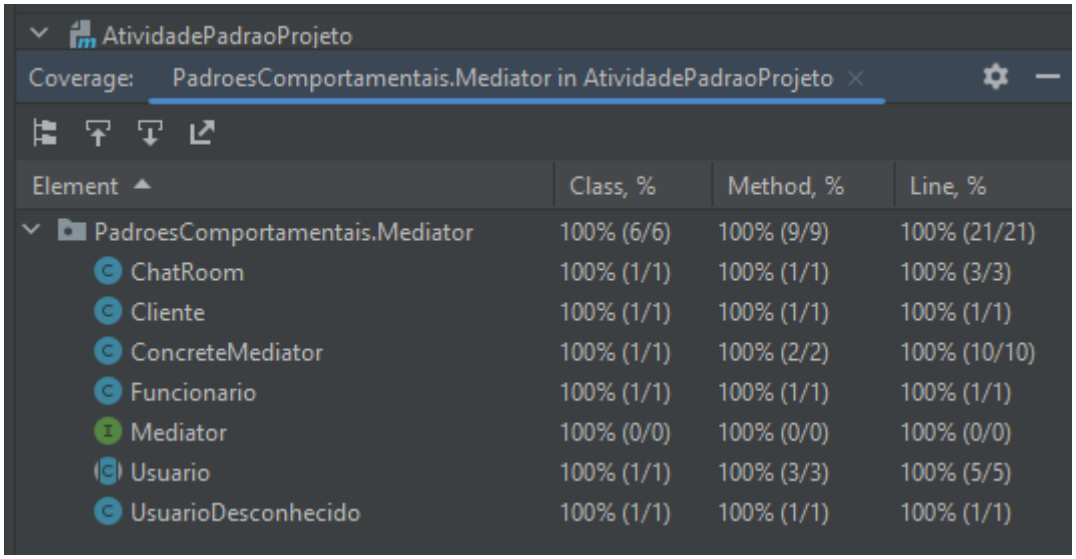
`public void testChat()`

- **Objetivo:** Verificar a comunicação entre diferentes tipos de usuários (Funcionário, Cliente e Usuário Desconhecido) através do mediador.
- **Método:**
    - Criar duas instâncias de `ChatRoom`, uma para funcionários e outra para clientes.
    - Criar uma instância de `ConcreteMediator`, passando as salas de chat criadas.
    - Criar instâncias de `Funcionario`, `Cliente`, e `UsuarioDesconhecido`, associando cada uma ao mediador criado.

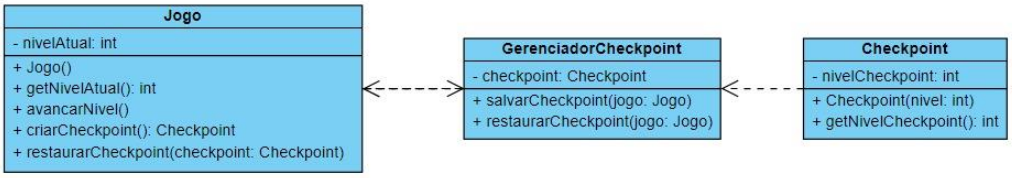
- Cada usuário envia uma mensagem usando o método `sendMessage`.
- Verificar se as mensagens enviadas retornam os resultados esperados.

- **Resultado Esperado:**
  - A mensagem enviada pelo funcionário deve ser `"Alice (Funcionario) disse: Olá, pessoal! (enviado para Chat Funcionarios)"`.
  - A mensagem enviada pelo cliente deve ser `"Bob (Cliente) disse: Olá, empresa! (enviado para Chat Clientes)"`.
  - A mensagem enviada pelo usuário desconhecido deve ser `"Tipo de Usuario Desconhecido!"`.





Padrão Memento - Sistema de Checkpoint de Jogo



Este projeto implementa o padrão de projeto Memento para permitir que um jogo salve e restaure seu estado em checkpoints.

Checkpoint.java

A classe `Checkpoint` representa um estado salvo do jogo em um determinado nível.

Atributos

- `private final int nivelCheckpoint`: O nível salvo no checkpoint.

Construtor

- `public Checkpoint(int nivel)`: Inicializa o checkpoint com o nível fornecido.
    - `nivel`: O nível do jogo a ser salvo no checkpoint.

Métodos

- `public int getNivelCheckpoint()`: Retorna o nível salvo no checkpoint.

GerenciadorCheckpoint.java

A classe `GerenciadorCheckpoint` gerencia a criação e restauração de checkpoints para o jogo.

Atributos

- `private Checkpoint checkpoint`: O checkpoint atual salvo.

Métodos

- `public void salvarCheckpoint(Jogo jogo)`: Cria e salva um checkpoint do estado atual do jogo.
    - `jogo`: A instância do jogo para a qual o checkpoint será criado.
- `public void restaurarCheckpoint(Jogo jogo)`: Restaura o estado do jogo para o checkpoint salvo.
    - `jogo`: A instância do jogo que será restaurada.
    - Lança `IllegalStateException` se nenhum checkpoint estiver salvo.

Jogo.java

A classe `Jogo` representa o estado do jogo, incluindo o nível atual e métodos para avançar, criar e restaurar checkpoints.

Atributos

- `private int nivelAtual`: O nível atual do jogo.

Construtor

- `public Jogo()`: Inicializa o jogo no nível 1.

Métodos

- `public int getNivelAtual()`: Retorna o nível atual do jogo.
- `public void avancarNivel()`: Avança o jogo para o próximo nível.
- `public Checkpoint criarCheckpoint()`: Cria um checkpoint no nível atual do jogo.
- `public void restaurarCheckpoint(Checkpoint checkpoint)`: Restaura o jogo para o nível salvo no checkpoint.
    - `checkpoint`: O checkpoint a partir do qual o jogo será restaurado.

TestMemento.java

O `TestMemento.java` contém testes para validar o comportamento do padrão Memento sendo usado em um jogo que permite salvar e restaurar checkpoints. Os testes verificam a funcionalidade de avançar níveis, salvar checkpoints, restaurar checkpoints e lidar com a ausência de checkpoints salvos.

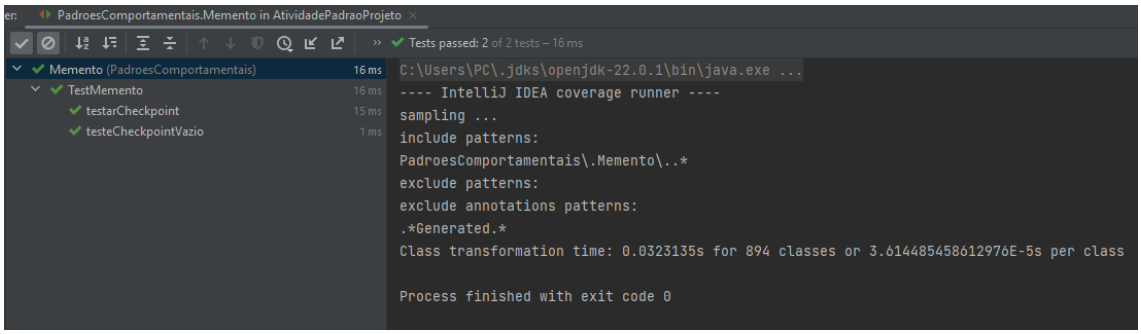
`public void testarCheckpoint()`

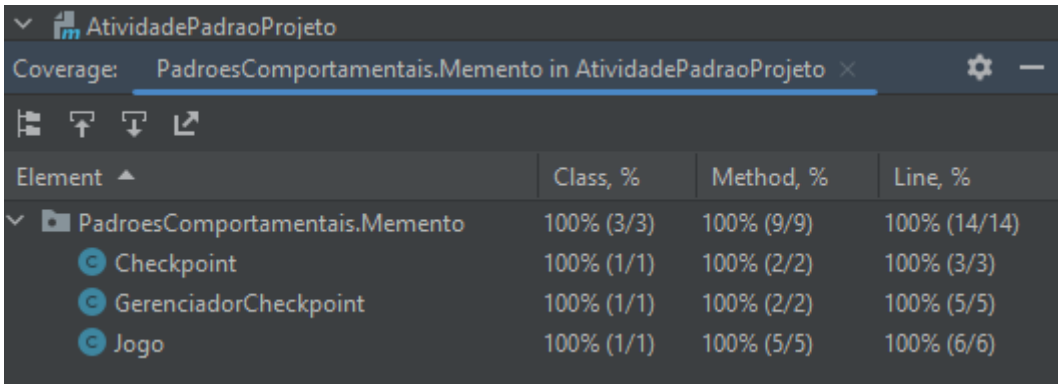
- **Objetivo:** Verificar a funcionalidade de salvar e restaurar checkpoints no jogo.

- **Método:**
    - Criar uma instância do jogo (`Jogo`).
    - Criar uma instância do gerenciador de checkpoints (`GerenciadorCheckpoint`).
    - Avançar o jogo para o próximo nível.
    - Salvar o estado atual do jogo no gerenciador de checkpoints.
    - Avançar o jogo por mais dois níveis.
    - Verificar se o nível atual do jogo é 4.
    - Restaurar o checkpoint salvo.
    - Verificar se o nível atual do jogo foi restaurado para 2.
- **Resultado Esperado:**
    - Antes de restaurar o checkpoint, o nível atual deve ser 4.
    - Após restaurar o checkpoint, o nível atual deve ser 2.

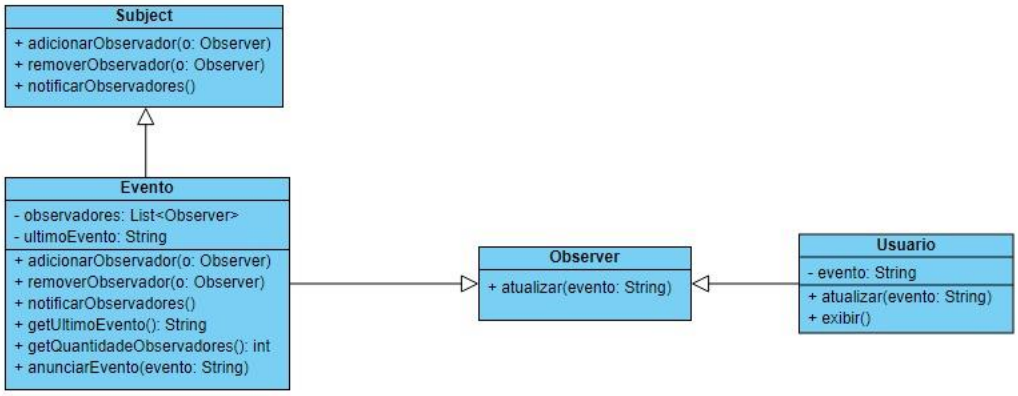
```
public void testeCheckpointVazio()
```

- **Objetivo:** Verificar o comportamento do sistema quando se tenta restaurar um checkpoint sem nenhum checkpoint salvo.
- **Método:**
    - Criar uma instância do jogo (`Jogo`).
    - Criar uma instância do gerenciador de checkpoints (`GerenciadorCheckpoint`).
    - Tentar restaurar um checkpoint sem nenhum checkpoint salvo e capturar a exceção `IllegalStateException`.
    - Verificar se a mensagem da exceção corresponde ao valor esperado.
- **Resultado Esperado:**
    - Deve ser lançada uma exceção `IllegalStateException` com a mensagem `"Nenhum checkpoint salvo."`.


er: PadroesComportamentais.Memento in AtividadePadraoProjeto
Tests passed: 2 of 2 tests – 16 ms
Memento (PadroesComportamentais) 16 ms
TestMemento 16 ms
testarCheckpoint 15 ms
testeCheckpointVazio 1 ms
C:\Users\PC\.jdks\openjdk-22.0.1\bin\java.exe ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
PadroesComportamentais\.Memento\..*
exclude patterns:
exclude annotations patterns:
.*Generated.*
Class transformation time: 0.0323135s for 894 classes or 3.614485458612976E-5s per class
Process finished with exit code 0


AtividadePadraoProjeto
Coverage: PadroesComportamentais.Memento in AtividadePadraoProjeto
Element
Class, %
Method, %
Line, %
PadroesComportamentais.Memento
100% (3/3)
100% (9/9)
100% (14/14)
Checkpoint
100% (1/1)
100% (2/2)
100% (3/3)
GerenciadorCheckpoint
100% (1/1)
100% (2/2)
100% (5/5)
Jogo
100% (1/1)
100% (5/5)
100% (6/6)

Padrão Observer - Evento e Observadores



Este projeto implementa o padrão Observer para permitir que objetos sejam notificados quando o estado de outro objeto muda.

Classes e Interfaces

Subject.java

A interface `Subject` define os métodos que devem ser implementados por qualquer classe que deseja atuar como sujeito no padrão Observer.

Métodos:

- **void adicionarObservador(Observer o)**: Adiciona um observador à lista de observadores.
- **void removerObservador(Observer o)**: Remove um observador da lista de observadores.
- **void notificarObservadores()**: Notifica todos os observadores sobre uma mudança de estado.

Observer.java

A interface `Observer` define o método que deve ser implementado por qualquer classe que deseja atuar como observador no padrão Observer.

Método:

- **void atualizar(String evento)**: Método chamado pelo sujeito para notificar o observador sobre um evento.

Evento.java

A classe `Evento` implementa a interface `Subject` e gerencia a lista de observadores. Ela notifica os observadores sobre novos eventos.

Variáveis:

- **private final List observadores**: Lista de observadores.
- **private String ultimoEvento**: Último evento anunciado.

Construtor:

- **public Evento()**: Inicializa a lista de observadores.

Métodos:

- **public void adicionarObservador(Observer o)**: Adiciona um observador à lista.
- **public void removerObservador(Observer o)**: Remove um observador da lista.
- **public void notificarObservadores()**: Notifica todos os observadores sobre o último evento.
- **public String getUltimoEvento()**: Retorna o último evento anunciado.
- **public int getQuantidadeObservadores()**: Retorna a quantidade de observadores registrados.
- **public void anunciarEvento(String evento)**: Define o último evento e notifica todos os observadores.

Usuario.java

A classe `Usuario` implementa a interface `Observer` e define a ação a ser tomada quando um evento é recebido.

Variáveis:

- **private String evento**: Evento recebido pelo observador.

Métodos:

- **public void atualizar(String evento)**: Atualiza o evento e chama o método `exibir`.
- **public void exibir()**: Exibe o evento no console.

TestObserver.java

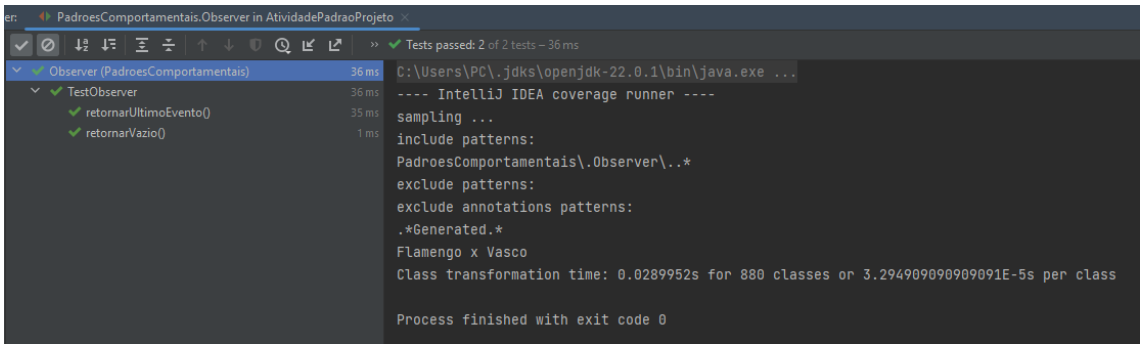
A classe `TestObserver` testa a funcionalidade das classes `Evento` e `Usuario` para garantir que as funcionalidades principais estão funcionando conforme esperado.

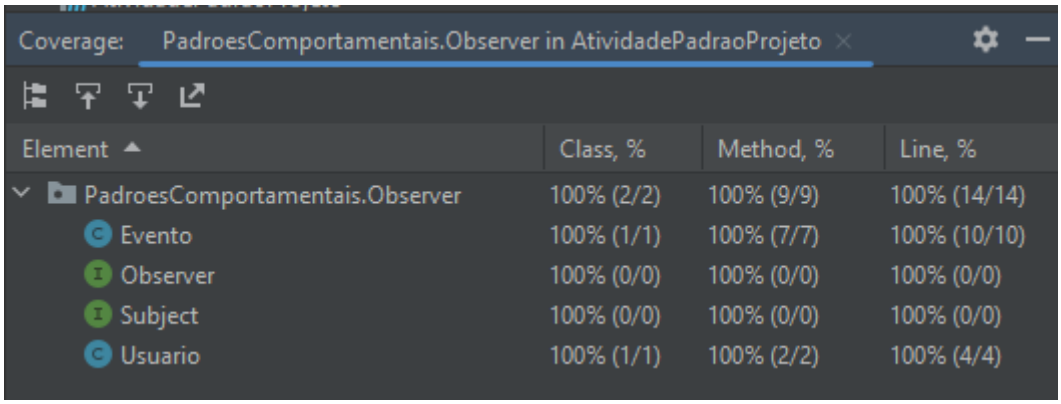
public void retornarUltimoEvento()

- **Objetivo**: Verificar se o último evento anunciado é corretamente retornado.
- **Método**:
    - Criar uma instância de `Evento`.
    - Criar uma instância de `Usuario` e adicioná-lo como observador do evento.
    - Anunciar um evento "Flamengo x Vasco".
    - Verificar se o último evento retornado é "Flamengo x Vasco".
- **Resultado Esperado**: O método `getUltimoEvento` deve retornar "Flamengo x Vasco".

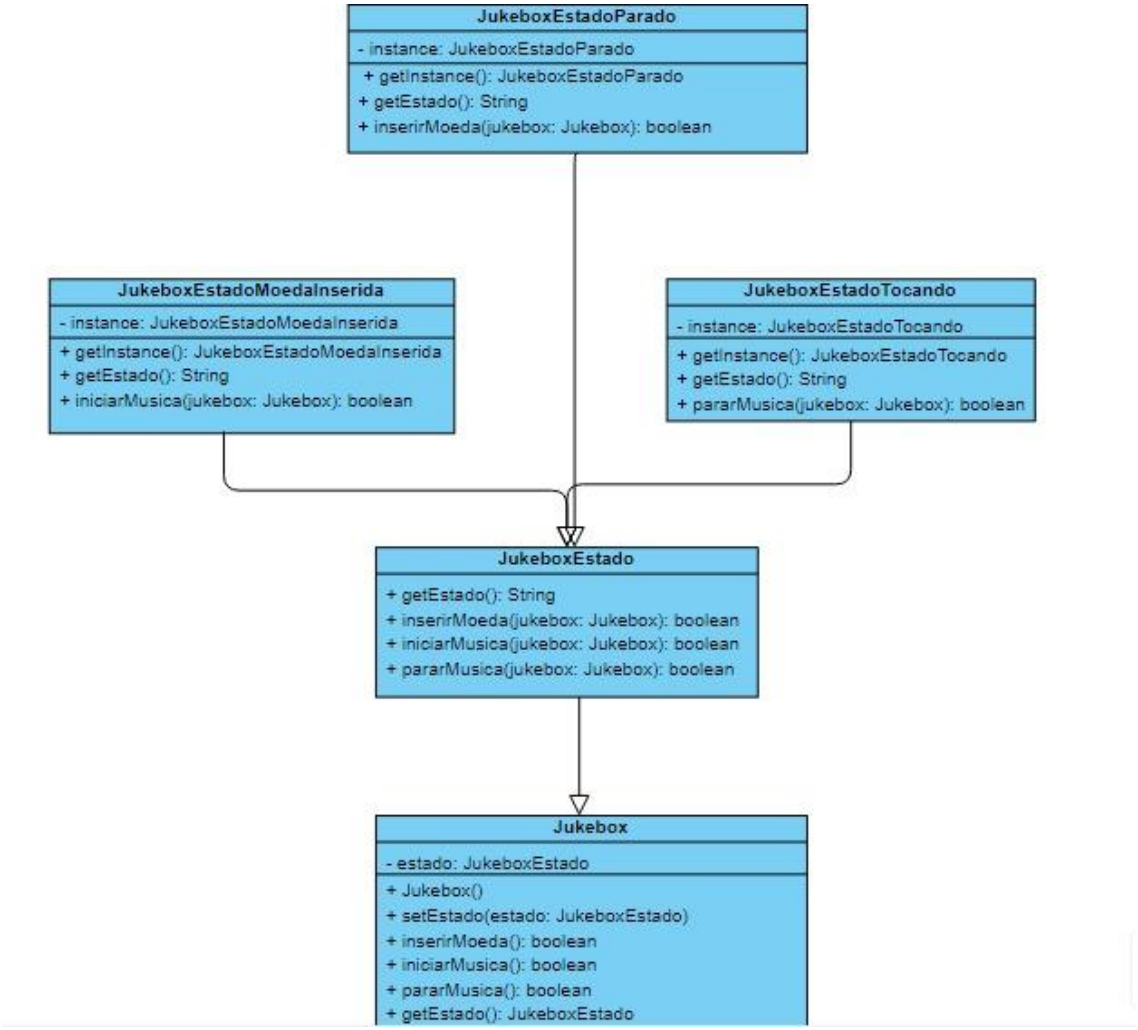
public void retornarVazio()

- **Objetivo**: Verificar se a lista de observadores está vazia após remover um observador.
- **Método**:
    - Criar uma instância de `Evento`.
    - Criar uma instância de `Usuario` e adicioná-lo como observador do evento.
    - Remover o observador.
    - Verificar se a quantidade de observadores é 0.
- **Resultado Esperado**: O método `getQuantidadeObservadores` deve retornar 0.


PadroesComportamentais.Observer in AtividadePadraoProjeto
Tests passed: 2 of 2 tests – 36 ms
Observer (PadroesComportamentais) 36 ms
TestObserver 36 ms
retornarUltimoEvento() 35 ms
retornarVazio() 1 ms
C:\Users\PC\.jdks\openjdk-22.0.1\bin\java.exe ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
PadroesComportamentais\.Observer\..*
exclude patterns:
exclude annotations patterns:
.*Generated.*
Flamengo x Vasco
Class transformation time: 0.0289952s for 880 classes or 3.2949090909090 91E-5s per class
Process finished with exit code 0


Coverage: PadroesComportamentais.Observer in AtividadePadraoProjeto
Element
Class, %
Method, %
Line, %
PadroesComportamentais.Observer 100% (2/2) 100% (9/9) 100% (14/14)
Evento 100% (1/1) 100% (7/7) 100% (10/10)
Observer 100% (0/0) 100% (0/0) 100% (0/0)
Subject 100% (0/0) 100% (0/0) 100% (0/0)
Usuario 100% (1/1) 100% (2/2) 100% (4/4)

Padrão State – Jukebox



O projeto implementa o padrão de projeto State para modelar o comportamento dinâmico de uma Jukebox, permitindo transições entre estados como Parado, Tocando e Moeda Inserida.

Jukebox.java

A classe `Jukebox` representa a máquina de Jukebox que pode estar em diferentes estados e executa ações com base nesses estados.

Atributos:

- `private JukeboxEstado estado`: Estado atual da Jukebox.

Construtor:

- `public Jukebox()`: Inicializa a Jukebox no estado Parado por padrão.

Métodos:

- `public void setEstado(JukeboxEstado estado)`: Define o estado da Jukebox.
- `public boolean inserirMoeda()`: Executa a ação de inserir moeda, delegando ao estado atual.
- `public boolean iniciarMusica()`: Executa a ação de iniciar música, delegando ao estado atual.
- `public boolean pararMusica()`: Executa a ação de parar música, delegando ao estado atual.
- `public JukeboxEstado getEstado()`: Retorna o estado atual da Jukebox.

JukeboxEstado.java

A classe abstrata `JukeboxEstado` define o contrato para todos os estados possíveis da Jukebox.

Métodos:

- `public abstract String getEstado()`: Retorna o nome do estado.
- `public boolean inserirMoeda(Jukebox jukebox)`: Método padrão para inserir moeda.
- `public boolean iniciarMusica(Jukebox jukebox)`: Método padrão para iniciar música.
- `public boolean pararMusica(Jukebox jukebox)`: Método padrão para parar música.

JukeboxEstadoParado.java

A classe `JukeboxEstadoParado` implementa o estado em que a Jukebox está parada.

Métodos:

- `public static JukeboxEstadoParado getInstance()`: Retorna a instância única do estado Parado.
- `public String getEstado()`: Retorna "Jukebox Parada".
- `public boolean inserirMoeda(Jukebox jukebox)`: Transição para o estado Moeda Inserida ao inserir moeda.

JukeboxEstadoTocando.java

A classe `JukeboxEstadoTocando` implementa o estado em que a Jukebox está tocando música.

Métodos:

- `public static JukeboxEstadoTocando getInstance()`: Retorna a instância única do estado Tocando.
- `public String getEstado()`: Retorna "Jukebox Tocando".
- `public boolean pararMusica(Jukebox jukebox)`: Transição para o estado Parado ao parar a música.

JukeboxEstadoMoedaInserida.java

A classe `JukeboxEstadoMoedaInserida` implementa o estado em que há moeda inserida na Jukebox.

Métodos:

- `public static JukeboxEstadoMoedaInserida getInstance()`: Retorna a instância única do estado Moeda Inserida.
- `public String getEstado()`: Retorna "Jukebox Possui Moeda".
- `public boolean iniciarMusica(Jukebox jukebox)`: Transição para o estado Tocando ao iniciar a música.

TestState.java

O arquivo `TestState.java` contém testes unitários para verificar o comportamento da Jukebox em diferentes estados utilizando o padrão State: `JukeboxParada`, `JukeboxTocando` e `JukeboxMoedaInserida`.

`public void setUp()`
Antes de cada teste, as instâncias de Jukebox são configuradas com diferentes estados iniciais:

- `jukeboxParada`: Instância da Jukebox configurada com o estado inicial `JukeboxEstadoParado`.
- `jukeboxTocando`: Instância da Jukebox configurada com o estado inicial `JukeboxEstadoTocando`.
- `jukeboxMoedaInserida`: Instância da Jukebox configurada com o estado inicial `JukeboxEstadoMoedaInserida`.

Testes do Estado Jukebox Parada

`public void naoPodeIniciarJukeboxParada()`

- **Objetivo:** Verificar que a Jukebox não pode iniciar a música quando está no estado `JukeboxParada`.
- **Método:**
    - Chamar `iniciarMusica()` na instância de `jukeboxParada`.
    - Verificar se o método retorna `false`.
- **Resultado Esperado:**
    - O método `iniciarMusica()` deve retornar `false`, indicando que a música não pode ser iniciada quando a Jukebox está parada.

`public void naoPodePararJukeboxParada()`

- **Objetivo:** Verificar que a Jukebox não pode parar a música quando está no estado `JukeboxParada`.
- **Método:**
    - Chamar `pararMusica()` na instância de `jukeboxParada`.
    - Verificar se o método retorna `false`.
- **Resultado Esperado:**
    - O método `pararMusica()` deve retornar `false`, indicando que a música não pode ser parada quando a Jukebox está parada.

`public void deveAceitarMoedaJukeboxParada()`

- **Objetivo:** Verificar que a Jukebox aceita inserção de moeda quando está no estado `JukeboxParada`.
- **Método:**
    - Chamar `inserirMoeda()` na instância de `jukeboxParada`.
    - Verificar se o método retorna `true`.
    - Verificar se o estado da Jukebox após a inserção de moeda é `"Jukebox Possui Moeda"`.
- **Resultado Esperado:**
    - O método `inserirMoeda()` deve retornar `true`, indicando que a moeda foi inserida com sucesso.
    - O estado da Jukebox após a inserção de moeda deve ser `"Jukebox Possui Moeda"`.
    -

Testes do Estado Jukebox Tocando

`public void naoPodeIniciarJukeboxTocando()`

- **Objetivo:** Verificar que a Jukebox não pode iniciar a música quando está no estado `JukeboxTocando`.
- **Método:**
  - Chamar `iniciarMusica()` na instância de `jukeboxTocando`.
  - Verificar se o método retorna `false`.
- **Resultado Esperado:**
  - O método `iniciarMusica()` deve retornar `false`, indicando que a música não pode ser iniciada quando a Jukebox já está tocando.

`public void deveParararJukeboxTocando()`

- **Objetivo:** Verificar que a Jukebox pode parar a música quando está no estado `JukeboxTocando`.
- **Método:**
  - Chamar `pararMusica()` na instância de `jukeboxTocando`.
  - Verificar se o método retorna `true`.
  - Verificar se o estado da Jukebox após parar a música é `"Jukebox Parada"`.
- **Resultado Esperado:**
  - O método `pararMusica()` deve retornar `true`, indicando que a música foi parada com sucesso.
  - O estado da Jukebox após parar a música deve ser `"Jukebox Parada"`.

`public void naoPodeAceitarMoedaJukeboxTocando()`

- **Objetivo:** Verificar que a Jukebox não pode aceitar inserção de moeda quando está no estado `JukeboxTocando`.
- **Método:**
  - Chamar `inserirMoeda()` na instância de `jukeboxTocando`.
  - Verificar se o método retorna `false`.
- **Resultado Esperado:**
  - O método `inserirMoeda()` deve retornar `false`, indicando que não é possível inserir moeda enquanto a Jukebox está tocando música.

Testes do Estado JukeboxMoedaInserida

`public void deveIniciarJukeboxMoedaInserida()`

- **Objetivo:** Verificar que a Jukebox pode iniciar a música quando está no estado `JukeboxMoedaInserida`.
- **Método:**
  - Chamar `iniciarMusica()` na instância de `jukeboxMoedaInserida`.
  - Verificar se o método retorna `true`.
  - Verificar se o estado da Jukebox após iniciar a música é `"Jukebox Tocando"`.
- **Resultado Esperado:**
  - O método `iniciarMusica()` deve retornar `true`, indicando que a música foi iniciada com sucesso.
  - O estado da Jukebox após iniciar a música deve ser `"Jukebox Tocando"`.

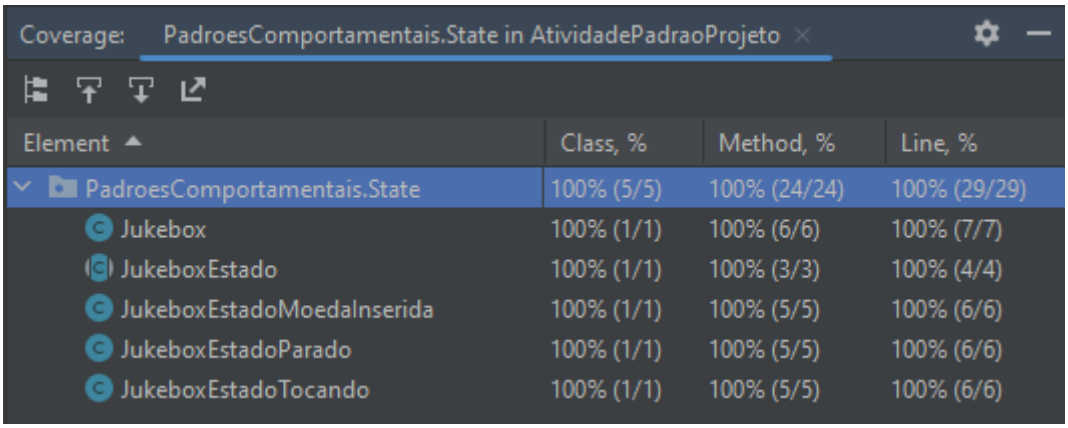
`public void naoPodePararJukeboxMoedaInserida()`

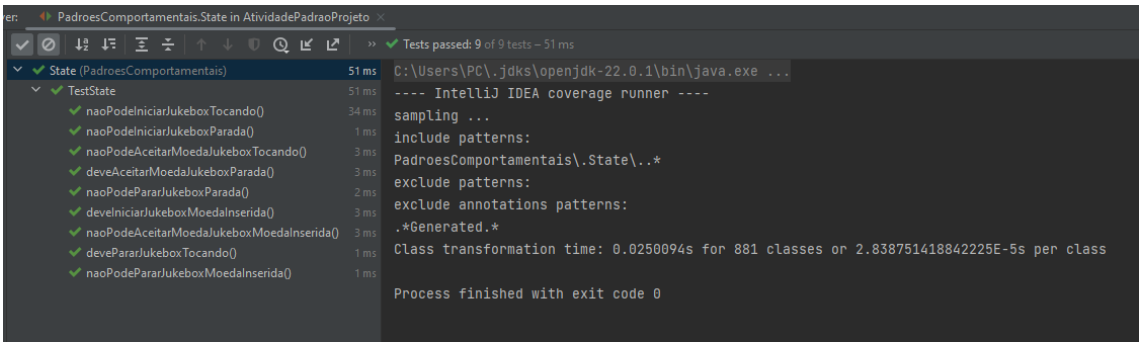
- **Objetivo:** Verificar que a Jukebox não pode parar a música imediatamente após inserção de moeda.
- **Método:**
  - Chamar `pararMusica()` na instância de `jukeboxMoedaInserida`.
  - Verificar se o método retorna `false`.
- **Resultado Esperado:**
  - O método `pararMusica()` deve retornar `false`, indicando que não é possível parar a música imediatamente após inserir moeda.

`public void naoPodeAceitarMoedaJukeboxMoedaInserida()`

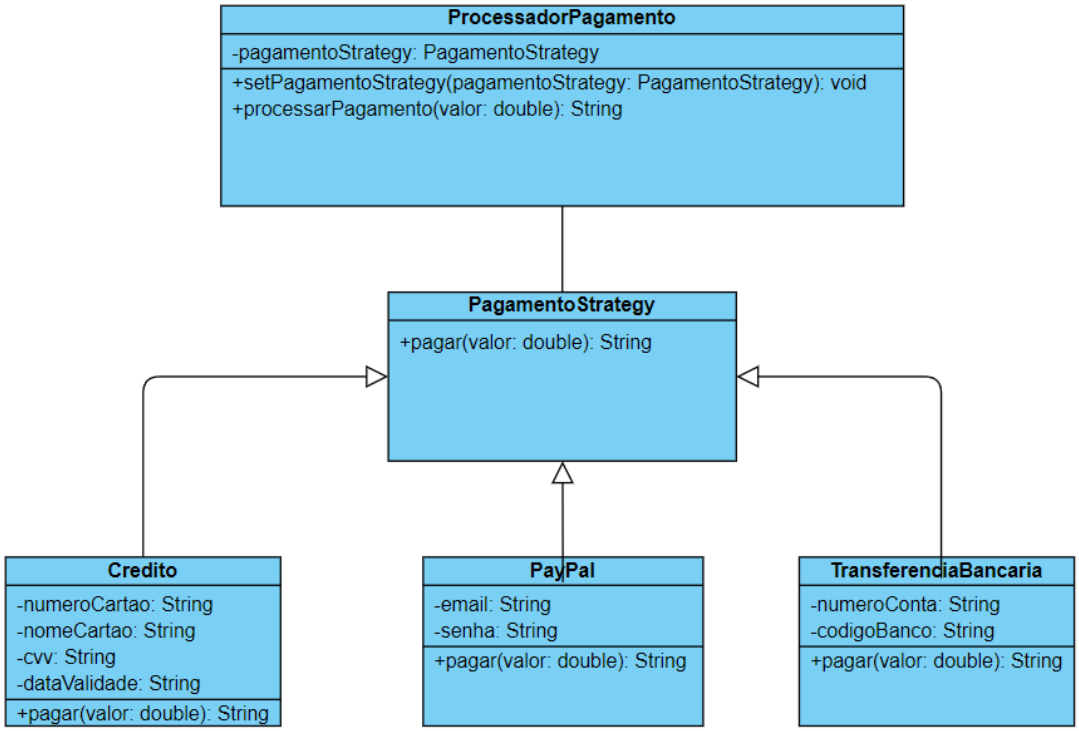
- **Objetivo:** Verificar que a Jukebox não pode aceitar inserção de moeda adicional quando já possui moeda inserida.
- **Método:**
  - Chamar `inserirMoeda()` na instância de `jukeboxMoedaInserida`.
  - Verificar se o método retorna `false`.
- **Resultado Esperado:**

- O método `inserirMoeda()` deve retornar `false`, indicando que não é possível inserir outra moeda enquanto já há uma moeda inserida na Jukebox.





Padrão Strategy - Processador de Pagamentos



Este projeto implementa o padrão de design Strategy para modelar diferentes métodos de pagamento. Neste caso, implementamos três estratégias de pagamento: cartão de crédito, PayPal e transferência bancária.

Classes

PagamentoStrategy.java

A interface `PagamentoStrategy` define o método que todas as estratégias de pagamento devem implementar.

Métodos:

- **String pagar(double valor)**: Método para realizar o pagamento, retorna uma string indicando o sucesso do pagamento.

Credito.java

A classe `Credito` implementa a estratégia de pagamento por cartão de crédito.

Variáveis:

- **private String numeroCartao**: Número do cartão de crédito.
- **private String nomeCartao**: Nome do titular do cartão.
- **private String cvv**: Código CVV do cartão.
- **private String dataValidade**: Data de validade do cartão.

Construtor:

- **public Credito(String numeroCartao, String nomeCartao, String cvv, String dataValidade)**: Inicializa os dados do cartão de crédito.

Métodos:

- **public String pagar(double valor)**: Realiza o pagamento e retorna uma string indicando o valor pago com cartão de crédito.

PayPal.java

A classe `PayPal` implementa a estratégia de pagamento pelo PayPal.

Variáveis:

- **private String email**: Email do usuário do PayPal.
- **private String senha**: Senha do usuário do PayPal.

Construtor:

- **public PayPal(String email, String senha)**: Inicializa os dados do PayPal.

Métodos:

- **public String pagar(double valor)**: Realiza o pagamento e retorna uma string indicando o valor pago com PayPal.

TransferenciaBancaria.java

A classe `TransferenciaBancaria` implementa a estratégia de pagamento por transferência bancária.

Variáveis:

- **private String numeroConta**: Número da conta bancária.
- **private String codigoBanco**: Código do banco.

Construtor:

- **public TransferenciaBancaria(String numeroConta, String codigoBanco)**: Inicializa os dados da conta bancária.

Métodos:

- **public String pagar(double valor)**: Realiza o pagamento e retorna uma string indicando o valor pago por transferência bancária.

ProcessadorPagamento.java

A classe `ProcessadorPagamento` utiliza uma instância de `PagamentoStrategy` para processar o pagamento.

Variáveis:

- **private PagamentoStrategy pagamentoStrategy**: A estratégia de pagamento atual.

Métodos:

- **public void setPagamentoStrategy(PagamentoStrategy pagamentoStrategy)**: Define a estratégia de pagamento.
- **public String processarPagamento(double valor)**: Processa o pagamento utilizando a estratégia definida. Lança uma exceção se a estratégia de pagamento não estiver definida.

TestStrategy.java

O `TestStrategy.java` contém testes unitários para validar o comportamento do processamento de pagamentos utilizando diferentes estratégias (`Credito`, `PayPal`, `TransferenciaBancaria`) implementadas com o padrão Strategy.

```
public void metodoCartaoCredito()
```

- **Objetivo:** Verificar o processamento de pagamento usando Cartão de Crédito.
- **Método:**
    - Criar uma instância de `ProcessadorPagamento`.
    - Configurar a estratégia de pagamento como Cartão de Crédito com os detalhes específicos (número do cartão, nome, código de segurança, data de validade).

    - Chamar `processarPagamento` com o valor de 100.0.
    - Verificar se o resultado indica pagamento bem-sucedido usando Cartão de Crédito.

- **Resultado Esperado:**

    - O resultado deve ser `"Pago 100.0 usando Cartão de Crédito."`.

`public void metodoPayPal()`

- **Objetivo:** Verificar o processamento de pagamento usando PayPal.
- **Método:**

    - Criar uma instância de `ProcessadorPagamento`.
    - Configurar a estratégia de pagamento como PayPal com os detalhes específicos (e-mail e senha).
    - Chamar `processarPagamento` com o valor de 200.0.
    - Verificar se o resultado indica pagamento bem-sucedido usando PayPal.

- **Resultado Esperado:**

    - O resultado deve ser `"Pago 200.0 usando PayPal."`.

`public void metodoTransferenciaBancaria()`

- **Objetivo:** Verificar o processamento de pagamento usando Transferência Bancária.
- **Método:**

    - Criar uma instância de `ProcessadorPagamento`.
    - Configurar a estratégia de pagamento como Transferência Bancária com os detalhes específicos (número da conta e código do banco).
    - Chamar `processarPagamento` com o valor de 300.0.
    - Verificar se o resultado indica pagamento bem-sucedido por Transferência Bancária.

- **Resultado Esperado:**

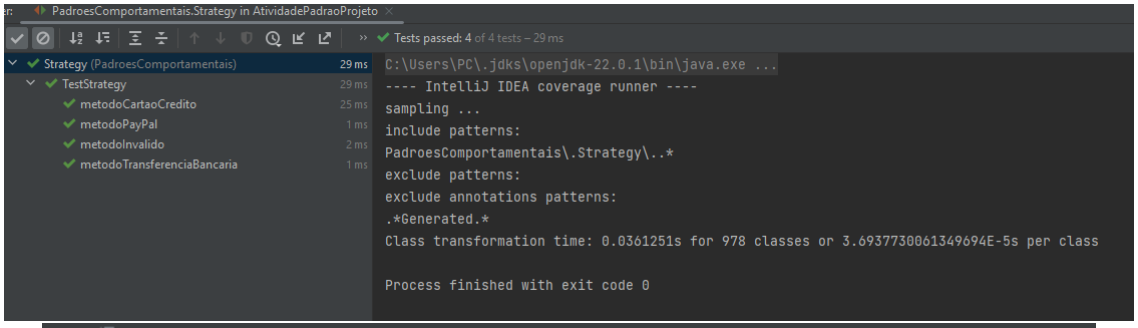
    - O resultado deve ser `"Pago 300.0 por Transferência Bancária."`.

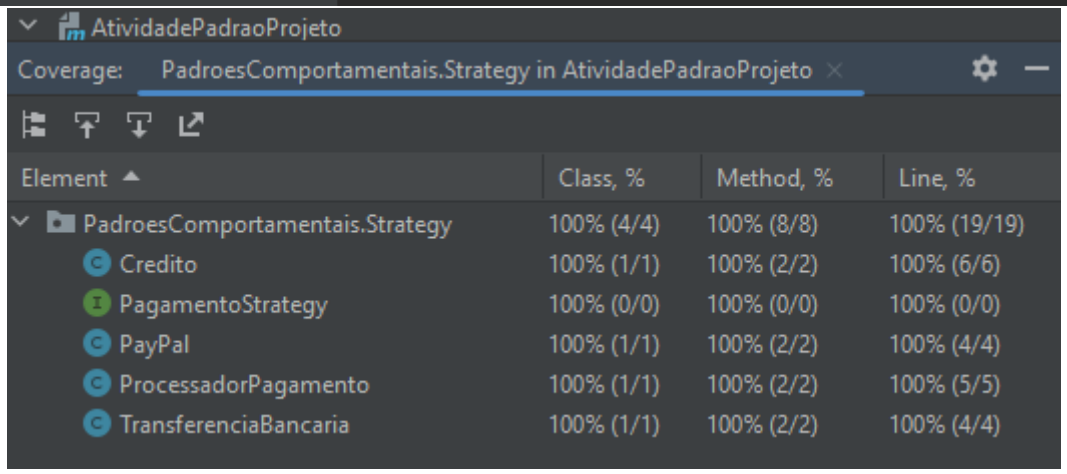
`public void metodoInvalido()`

- **Objetivo:** Verificar o tratamento de método de pagamento inválido.
- **Método:**

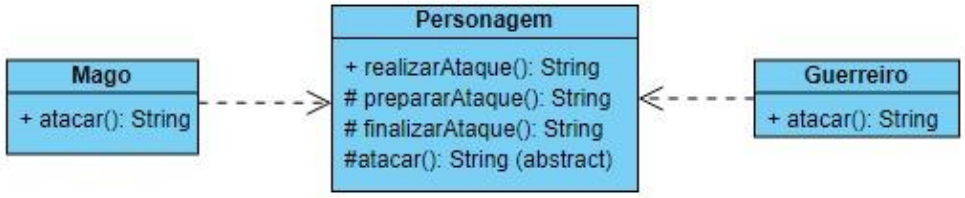
- Criar uma instância de `ProcessadorPagamento`.
- Chamar `processarPagamento` sem definir uma estratégia de pagamento válida.
- Verificar se uma exceção do tipo `IllegalStateException` é lançada.
- Verificar se a mensagem de exceção é `"Metodo de pagamento invalido."`.

- **Resultado Esperado:**
  - Deve ser lançada uma exceção `IllegalStateException` com a mensagem `"Metodo de pagamento invalido."`.





Padrão Template Method - Sistema de Ataque de Personagens



Este projeto implementa o padrão de projeto Template Method para definir o esqueleto de um algoritmo de ataque de personagens, permitindo que subclasses definam etapas específicas do ataque sem alterar a estrutura geral.

Personagem.java

A classe abstrata `Personagem` define o método template `realizarAtaque` e fornece implementações padrão para algumas etapas do ataque.

Métodos

- `public final String realizarAtaque()`: Define o esqueleto do algoritmo de ataque, chamando métodos que são ou podem ser sobrescritos por subclasses.
    - `StringBuilder resultado = new StringBuilder()`: Cria um `StringBuilder` para compilar os resultados das etapas do ataque.
    - `resultado.append(prepararAtaque()).append("\n")`: Adiciona o resultado do método `prepararAtaque`.
    - `resultado.append(atacar()).append("\n")`: Adiciona o resultado do método abstrato `atacar`.
    - `resultado.append(finalizarAtaque()).append("\n")`: Adiciona o resultado do método `finalizarAtaque`.
    - `return resultado.toString()`: Retorna a string compilada com os resultados das etapas do ataque.
- `protected String prepararAtaque()`: Fornece uma implementação padrão para a preparação do ataque.
    - Retorna: "Preparando para atacar..."
- `protected String finalizarAtaque()`: Fornece uma implementação padrão para a finalização do ataque.
    - Retorna: "Ataque finalizado."

- `protected abstract String atacar()`: Método abstrato que deve ser implementado pelas subclasses para definir o ataque específico do personagem.

Mago.java

A classe `Mago` é uma subclasse de `Personagem` que implementa o método `atacar` para definir o ataque específico de um mago.

Métodos

- `@Override protected String atacar()`: Define o ataque específico do mago.
    - Retorna: "Mago lançando uma bola de fogo!"

Guerreiro.java

A classe `Guerreiro` é uma subclasse de `Personagem` que implementa o método `atacar` para definir o ataque específico de um guerreiro.

Métodos

- `@Override protected String atacar()`: Define o ataque específico do guerreiro.
    - Retorna: "Guerreiro atacando com espada!"

TestTemplateMethod.java

O `TestTemplateMethod.java` contém testes unitários para verificar o comportamento do padrão de design Template Method no contexto de personagens de um jogo realizando ataques.

`public void testGuerreiroRealizarAtaque()`

- **Objetivo:** Verificar o comportamento do método `realizarAtaque` para um personagem do tipo `Guerreiro`.
- **Método:**
    - Criar uma instância de `Guerreiro`.
    - Chamar o método `realizarAtaque` do `Guerreiro`.
    - Verificar se a saída gerada corresponde ao valor esperado.
- **Resultado Esperado:**
    - A saída deve ser:

Copiar código

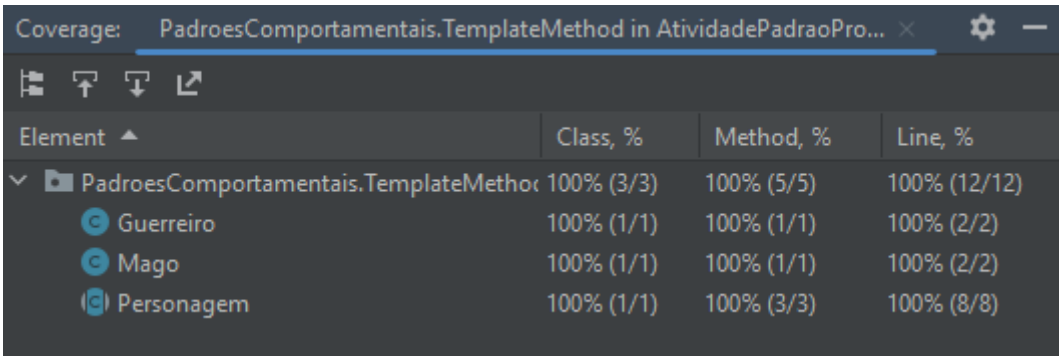
```
Preparando para atacar... Guerreiro atacando com espada!
Ataque finalizado.
```

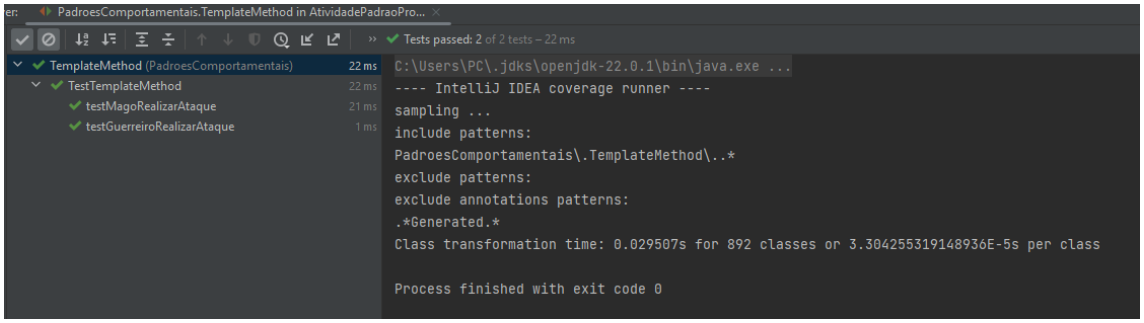
```
public void testMagoRealizarAtaque()
```

- **Objetivo:** Verificar o comportamento do método `realizarAtaque` para um personagem do tipo `Mago`.
- **Método:**
  - Criar uma instância de `Mago`.
  - Chamar o método `realizarAtaque` do `Mago`.
  - Verificar se a saída gerada corresponde ao valor esperado.
- **Resultado Esperado:**
  - A saída deve ser:

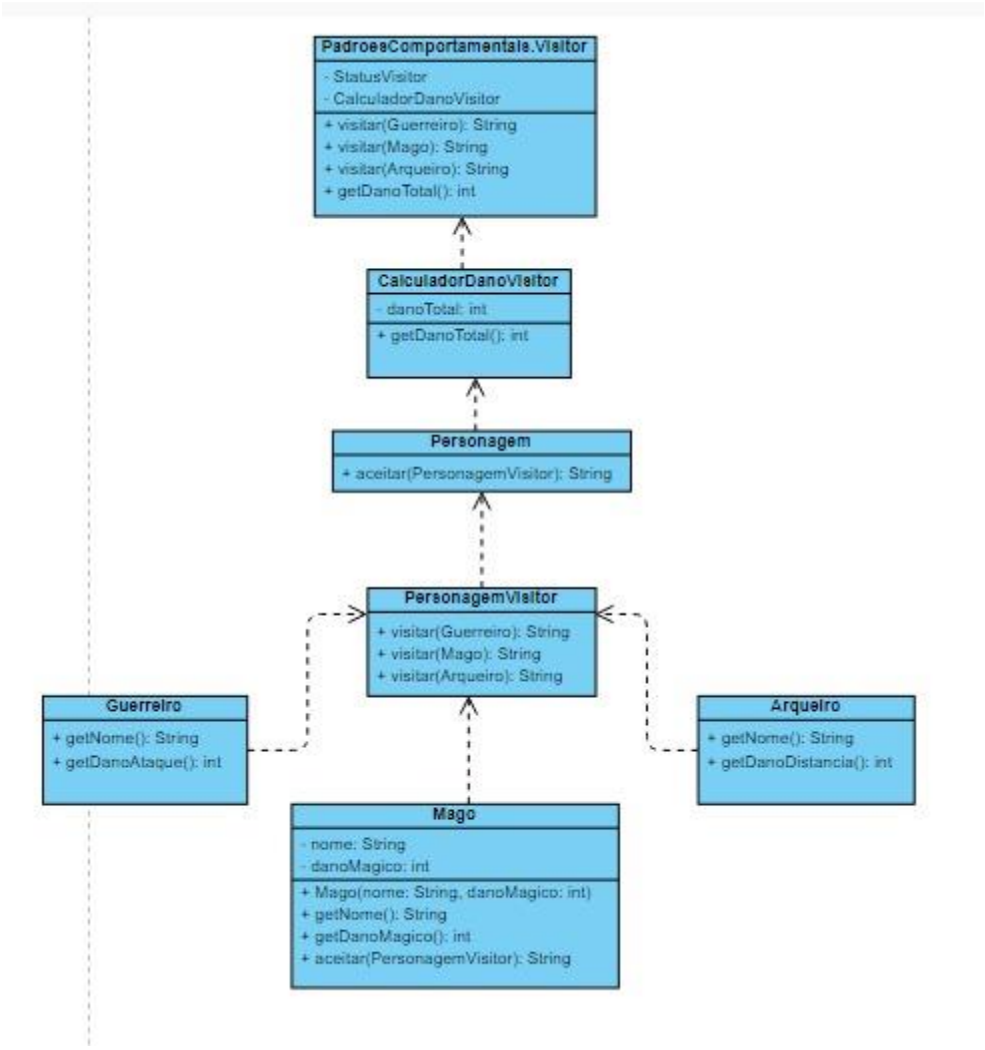
Copiar código

```
Preparando para atacar... Mago lançando uma bola de fogo!
Ataque finalizado.
```





Padrão Visitor - Sistema de Personagens



Este projeto usa o padrão de projeto Visitor num sistema de personagens de jogo com o objetivo é permitir operações diversas, como cálculo de dano e geração de relatórios, sobre os personagens sem modificar suas classes.

Arqueiro.java

A classe `Arqueiro` representa um personagem do tipo Arqueiro no jogo.

Atributos

- `private String nome`: O nome do arqueiro.
- `private int danoDistancia`: O dano causado pelo arqueiro à distância.

Construtor

- `public Arqueiro(String nome, int danoDistancia)`: Inicializa o arqueiro com o nome e dano à distância fornecidos.
    - `nome`: O nome do arqueiro.
    - `danoDistancia`: O dano à distância do arqueiro.

Métodos

- `public String getNome()`: Retorna o nome do arqueiro.
- `public int getDanoDistancia()`: Retorna o dano à distância do arqueiro.
- `@Override public String aceitar(PersonagemVisitor visitor)`: Aceita um visitante e chama seu método específico para o arqueiro.

CalculadorDanoVisitor.java

A classe `CalculadorDanoVisitor` implementa um visitante para calcular o dano total de diferentes tipos de personagens.

Atributos

- `private int danoTotal`: O dano total acumulado de todos os personagens visitados.

Métodos

- `@Override public String visitar(Guerreiro guerreiro)`: Calcula o dano do guerreiro e o adiciona ao dano total.
    - `guerreiro`: A instância do guerreiro a ser visitada.
- `@Override public String visitar(Mago mago)`: Calcula o dano do mago e o adiciona ao dano total.
    - `mago`: A instância do mago a ser visitada.
- `@Override public String visitar(Arqueiro arqueiro)`: Calcula o dano do arqueiro e o adiciona ao dano total.
    - `arqueiro`: A instância do arqueiro a ser visitada.
- `public int getDanoTotal()`: Retorna o dano total acumulado.

Guerreiro.java

A classe `Guerreiro` representa um personagem do tipo Guerreiro no jogo.

Atributos

- `private String nome`: O nome do guerreiro.

- `private int danoAtaque`: O dano causado pelo guerreiro em combate corpo a corpo.

Construtor

- `public Guerreiro(String nome, int danoAtaque)`: Inicializa o guerreiro com o nome e dano de ataque fornecidos.
    - `nome`: O nome do guerreiro.
    - `danoAtaque`: O dano de ataque do guerreiro.

Métodos

- `public String getNome()`: Retorna o nome do guerreiro.
- `public int getDanoAtaque()`: Retorna o dano de ataque do guerreiro.
- `@Override public String aceitar(PersonagemVisitor visitor)`: Aceita um visitante e chama seu método específico para o guerreiro.

Mago.java

A classe `Mago` representa um personagem do tipo Mago no jogo.

Atributos

- `private String nome`: O nome do mago.
- `private int danoMagico`: O dano causado pelo mago usando magia.

Construtor

- `public Mago(String nome, int danoMagico)`: Inicializa o mago com o nome e dano mágico fornecidos.
    - `nome`: O nome do mago.
    - `danoMagico`: O dano mágico do mago.

Métodos

- `public String getNome()`: Retorna o nome do mago.
- `public int getDanoMagico()`: Retorna o dano mágico do mago.
- `@Override public String aceitar(PersonagemVisitor visitor)`: Aceita um visitante e chama seu método específico para o mago.

Personagem.java

A interface `Personagem` define um método para aceitar visitantes.

Métodos

- `String aceitar(PersonagemVisitor visitor)`: Aceita um visitante.

PersonagemVisitor.java

A interface `PersonagemVisitor` define métodos de visita para diferentes tipos de personagens.

Métodos

- `String visitar(Guerreiro guerreiro)`: Visita um guerreiro.
- `String visitar(Mago mago)`: Visita um mago.
- `String visitar(Arqueiro arqueiro)`: Visita um arqueiro.

StatusVisitor.java

A classe `StatusVisitor` implementa um visitante para gerar um relatório do status dos personagens.

Atributos

- `private StringBuilder relatorio`: Acumula o relatório dos personagens visitados.

Métodos

- `@Override public String visitar(Guerreiro guerreiro)`: Adiciona informações sobre o guerreiro ao relatório.
    - `guerreiro`: A instância do guerreiro a ser visitada.
- `@Override public String visitar(Mago mago)`: Adiciona informações sobre o mago ao relatório.
    - `mago`: A instância do mago a ser visitada.
- `@Override public String visitar(Arqueiro arqueiro)`: Adiciona informações sobre o arqueiro ao relatório.
    - `arqueiro`: A instância do arqueiro a ser visitada.
- `public String getRelatorio()`: Retorna o relatório acumulado.

TestVisitor.java

O `TestVisitor.java` contém testes unitários para validar o comportamento do padrão de design Visitor no contexto de personagens de um jogo. Os testes verificam a funcionalidade de cálculo de dano total e geração de relatórios de status para diferentes tipos de personagens (`Guerreiro`, `Mago`, `Arqueiro`) utilizando visitantes (`CalculadorDanoVisitor` e `StatusVisitor`).

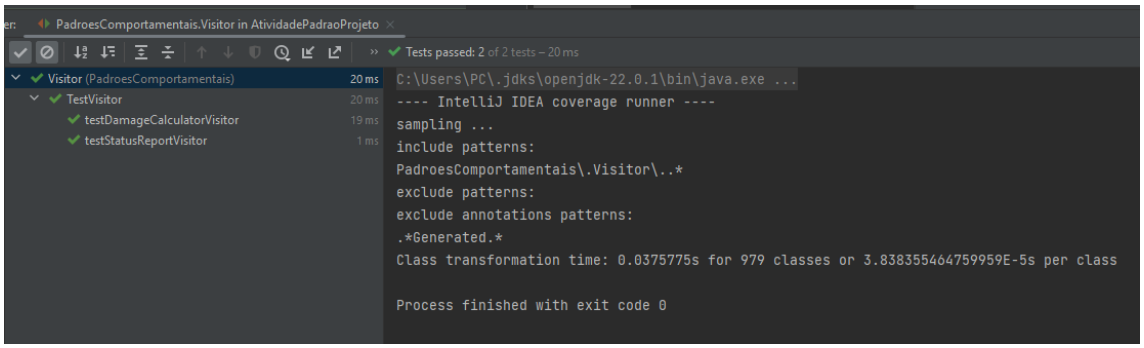
`public void testDamageCalculatorVisitor()`

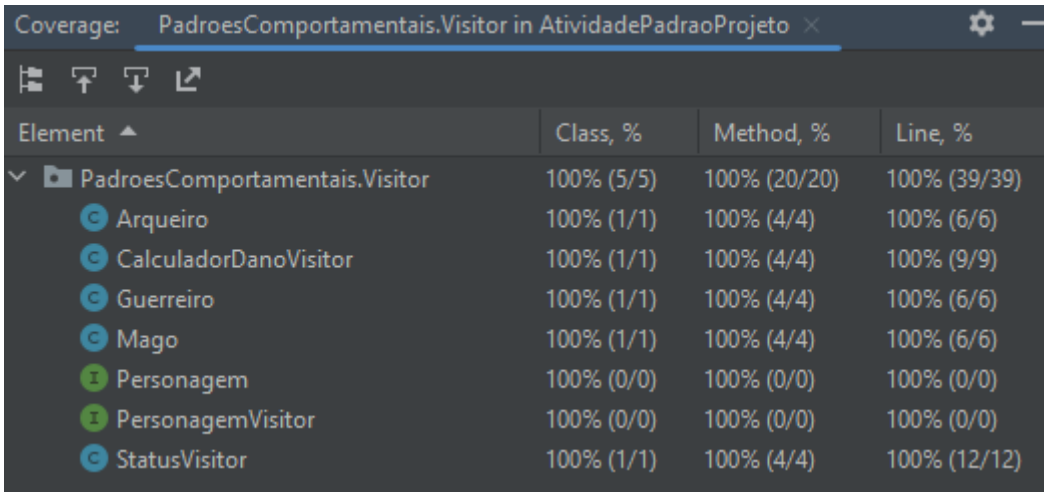
- **Objetivo:** Verificar o cálculo do dano total causado por diferentes personagens utilizando o visitante `CalculadorDanoVisitor`.
- **Método:**
    - Criar um array de `Personagem` contendo instâncias de `Guerreiro`, `Mago` e `Arqueiro` com valores específicos de poder de ataque ou mágico.
    - Criar uma instância de `CalculadorDanoVisitor`.
    - Fazer cada personagem aceitar o visitante `CalculadorDanoVisitor`, permitindo que o visitante calcule o dano total.
    - Verificar se o dano total calculado corresponde ao valor esperado.
- **Resultado Esperado:**
    - O dano total deve ser 160.

`public void testStatusReportVisitor()`

- **Objetivo:** Verificar a geração de um relatório de status para diferentes personagens utilizando o visitante `StatusVisitor`.
- **Método:**
    - Criar um array de `Personagem` contendo instâncias de `Guerreiro`, `Mago` e `Arqueiro` com valores específicos de poder de ataque ou mágico.
    - Criar uma instância de `StatusVisitor`.
    - Fazer cada personagem aceitar o visitante `StatusVisitor`, permitindo que o visitante gere um relatório de status.
    - Verificar se o relatório de status gerado corresponde ao valor esperado.
- **Resultado Esperado:**
    - O relatório de status deve ser:

```
Guerreiro: O Guerreiro, Poder de Ataque: 50
Mago: O Mago, Poder Mágico: 70
Arqueiro: O Arqueiro, Poder de Ataque à Distância: 40
```


PadroesComportamentais.Visitor in AtividadePadraoProjeto
Tests passed: 2 of 2 tests – 20 ms
Visitor (PadroesComportamentais) 20 ms
TestVisitor 20 ms
testDamageCalculatorVisitor 19 ms
testStatusReportVisitor 1 ms
C:\Users\PC\.jdks\openjdk-22.0.1\bin\java.exe ...
---- IntelliJ IDEA coverage runner ----
sampling ...
include patterns:
PadroesComportamentais\.Visitor\..*
exclude patterns:
exclude annotations patterns:
.*Generated.*
Class transformation time: 0.0375775s for 979 classes or 3.838355464759959E-5s per class
Process finished with exit code 0


Coverage: PadroesComportamentais.Visitor in AtividadePadraoProjeto
Element
Class, %
Method, %
Line, %
PadroesComportamentais.Visitor 100% (5/5) 100% (20/20) 100% (39/39)
Arqueiro 100% (1/1) 100% (4/4) 100% (6/6)
CalculadorDanoVisitor 100% (1/1) 100% (4/4) 100% (9/9)
Guerreiro 100% (1/1) 100% (4/4) 100% (6/6)
Mago 100% (1/1) 100% (4/4) 100% (6/6)
Personagem 100% (0/0) 100% (0/0) 100% (0/0)
PersonagemVisitor 100% (0/0) 100% (0/0) 100% (0/0)
StatusVisitor 100% (1/1) 100% (4/4) 100% (12/12)