Disciplina: Eng. Software II
Nome: Herick Andrei de Carvalho

1) COLETA DE INFORMAÇÕES

1. Analisar a documentação disponível do sistema legado (se houver):

(a) Diagramas UML, requisitos, manuais técnicos;

Não foram identificados diagramas UML, nem manuais técnicos. Os requisitos técnicos podem ser inferidos dentro de comentários no código ou estudando os atributos e métodos dos objetos.

(b) Scripts de banco de dados (MER, SQL);

Também não foram identificados. O estado do jogo é mantido em memória durante o jogo. A aplicação parece ter sido feita com um framework próprio, o NetGames (que deve ter sido baseado no .NET pelo nome). Procurando na internet, eu encontrei essa fonte que traz mais informações sobre esse framework:

https://www.inf.ufsc.br/~ricardo.silva/hp/netgames.html

Pra resumir, ele é fruto de um TCC de um aluno da UFSC em 2019 e tem o objetivo de permitir que desenvolvedores criem jogos distribuídos **sem precisar programar toda a parte de rede**. O desenvolvedor implementa apenas: a lógica do jogo e a parte **cliente** (o servidor já está implementado).

https://www.inf.ufsc.br/~ricardo.silva/netgames/download/Documentos%20-%20inclusive%20o%20TCC%20de%20Leonardo%20de%20Souza%20Brasil/NetGamesNRT_TCC.pdf

(c) Comentários e logs de execução.

O código possui comentários, mas eles não são claros, e onde importa, na classe Tabuleiro, que é onde se concentra a lógica de negócios, os comentários são ínfimos.

O log de execução se encontra na pasta de logs.

2. Mapear a estrutura geral do sistema:

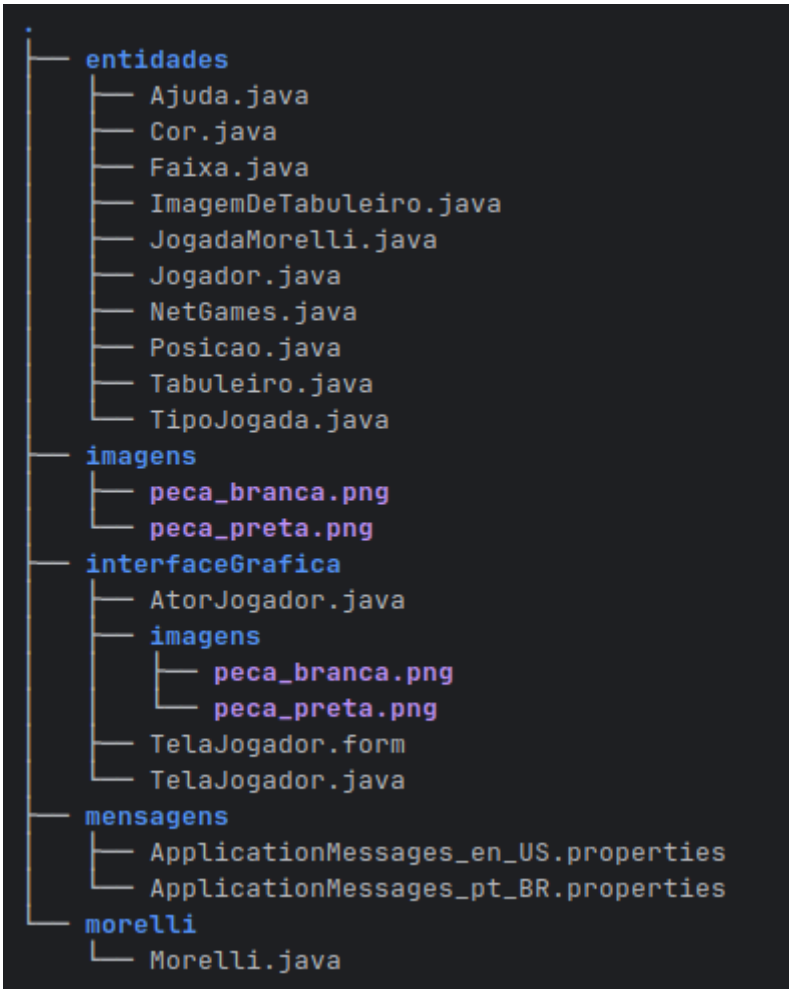
(a) Arquitetura utilizada (camadas, pacotes, módulos);

A arquitetura usada é MVC adaptada para jogos usando o NetGames (temos com isso duas arquiteturas: MVC e Cliente-Servidor) .

O MVC está estruturado assim:

- O Controller parece ter sido implementado na classe AtorJogador.java;
- Os models encontram-se no pacote entidades;
- O View são as classes dentro da pasta InterfaceGrafica.

Abaixo está a estrutura do projeto:



Podemos ver acima que a pasta Imagens está sobrando no projeto.

Da parte da arquitetura Cliente-Servidor, temos que o Servidor é implementado pelo próprio NetGames, sendo que o desenvolvedor deve se preocupar em desenvolver o seguinte:

1) Interface Jogada:
    a) Não possui métodos.
    b) Deve ser implementada por classes que representam lances, no nosso caso, o JogadaMorelli, Faixa e Posicao.
    c) Permite enviar dados entre clientes.
2) Interface OuvindorProxy
    a) Define os métodos que o servidor invocará no cliente. (Ex.: iniciar nova partida, receber jogada, etc.)
3) Classe Proxy (Cliente)
    a) Usada pelo cliente para invocar métodos no servidor.
    b) Métodos principais: conectar(), desconectar(), iniciarPartida(), enviarJogada() e finalizarPartida()

Voltando ao nosso projeto, podemos ver que a interface OuvidorProxy é implementada pelo NetGames,

o Cliente é o AtorJogador.

(b) Tecnologias empregadas (frameworks, bibliotecas, banco de dados);

Linguagem: Java 8
GUI: Swing
IDE: NetBeans 8.2
Build: Apache Ant
Rede: NetGames Framework
Logging: Log4j
Internacionalização: ResourceBundle

(c) Principais classes, métodos e dependências.

Tabuleiro.java - Lógica central do jogo
Quase todos os métodos que trabalham a lógica do negócio;
AtorJogador.java - Controlador principal
Conectar();
Desconectar();
IniciarPartida();
EnviarPartida();
NetGames.java - Comunicação em rede (ouvidorProxy)
iniciarNovaPartida();
finalizarPartidaComErro();
receberMensagem();
receberJogada();
tratarConexaoPerdida();
tratarPartidaNaoIniciada();
JogadaMorelli.java;
TipoJogada.java;

3. Anotar dúvidas e pontos de atenção para análise posterior.

O Swing é considerado obsoleto, além disso, temos um alto acoplamento entre classes.

2) Análise estática

1. Ler e inspecionar o código-fonte:

(a) Identificar as variáveis e métodos não utilizados;

Anexo I.

(b) Identificar código morto e/ou código duplicado;

Anexo II

(c) Identificar uso incorreto de padrões de estilo e convenções;

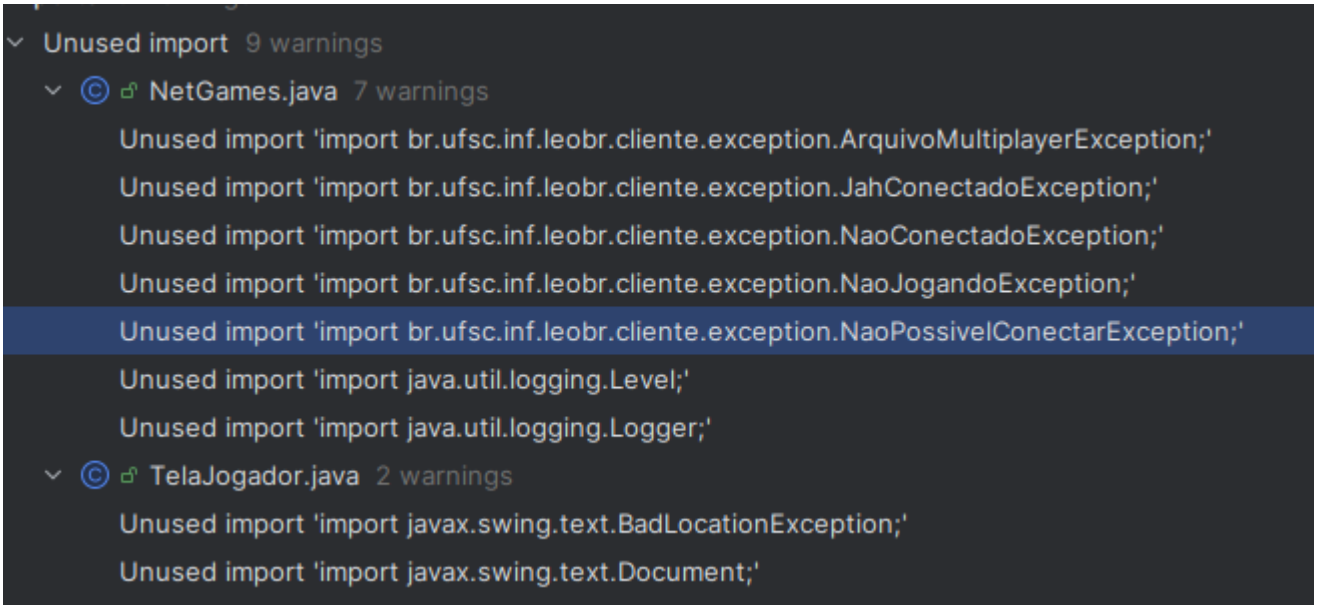
1) Nomenclatura: Mistura português/inglês

2) Visibilidade: Uso excessivo de `protected` em vez de `private`
3) Exception Handling: Catch genérico `Exception e`
4) Magic Numbers: Valores hardcoded (7 faixas, 48 peças)
5) Responsabilidade Única: Classes fazem múltiplas coisas

(d) Identificar código de alta complexidade (computacional ou cognitiva);

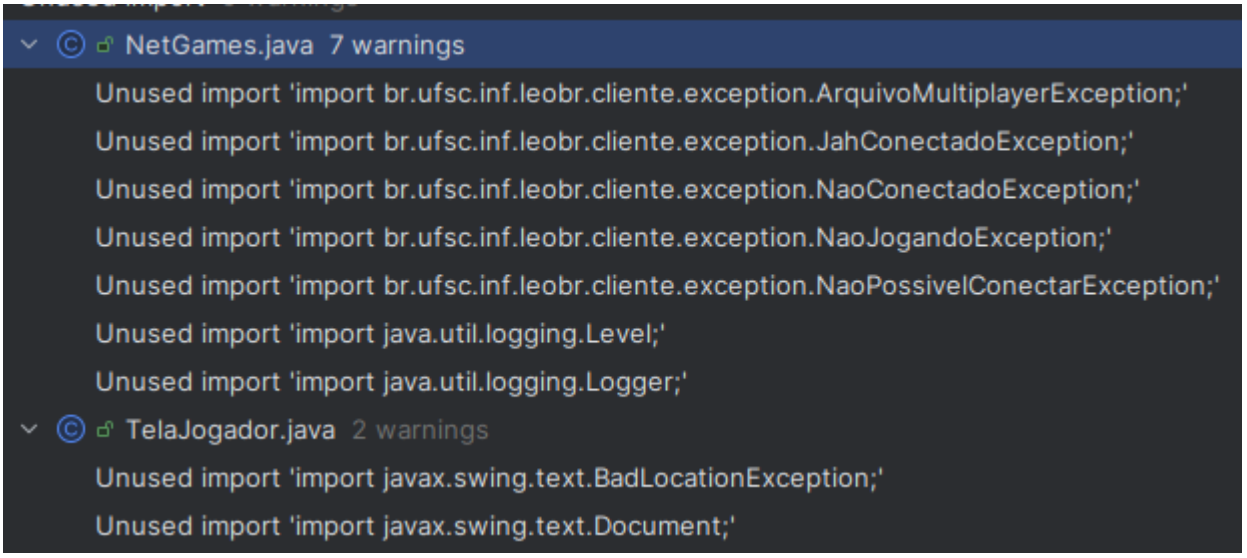
ANEXO III

(e) Identificar dependências desnecessárias.



2. Analisar dependências desnecessárias:

(a) Pacotes, frameworks e bibliotecas;



(b) Verificar acoplamento e modularidade.

As classes que possuem alto acoplamento são AtorJogador, que conhece todas as outras classes, e a classes Tabuleiro, que é acoplada diretamente à AtorJogador. Também temos a classe NetGames que é fortemente acoplado ao framework

3. Avaliar a complexidade (ex.: O(1), O(n), O(n²)) de todas as funções da classe Tabuleiro.

Métodos com tempo de execução constante:

setPartidaEmAndamento()
isPartidaEmAndamento()
atualizarTabuleiro()
getTabuleiro()
iniciarPartida()
limparTabuleiro()
distribuiPecas()
abandonarPartida()
realizarAcordo()
getAjuda()
atualizaJogadorDaVez()
getPosicaoOrigem()
setPosicaoOrigem()
getPosicaoDestino()
setPosicaoDestino()
movimentoAoCentro()
calcularMovimentoDiagonal()
calcularVerticeFaixa()
JogadaMorelli()
definePartidaFinalizada()
finalizaPartida()
recuperarFaixaDaPosicao()

Métodos com tempo de execução O(n):

calcularTomadaTrono()
atualizarPosicaoTabuleiro()
moverPeca(): O método faz uso de atualizaPosicaoTabuleiro() que tem tempo de execução O(n).
calcularMovimento()
verificarAdjacentes()
Complexidade O(n), já que possui um laço só.

Métodos com tempo de execução O(n²):

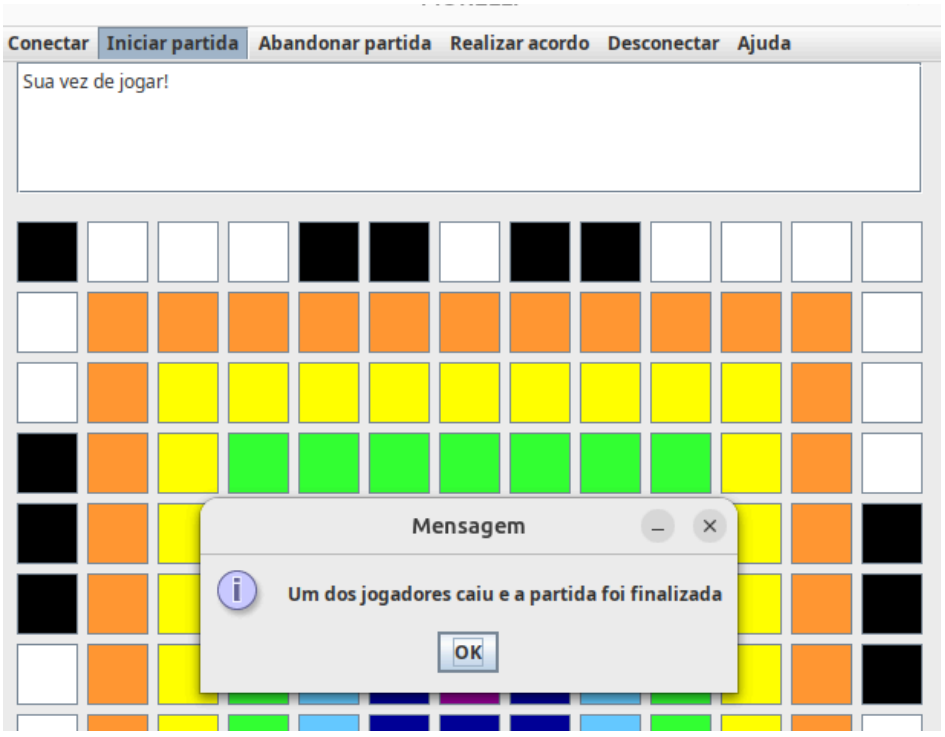
calcularMovimentoLinha()
Esse método depende do tamanho das entradas. Dentro deles temos um laço for que inicia na faixaAtual (igual a faixa de origem -1) e decrementa até chegar em 0, e depois temos um outro laço for percorrendo de 0 até o posicoesFaixa.length. Com isso temos que o tempo de execução é igual à O(faixaOrigem × posicoesFaixa.length) = O(n²). Temos que faixaOrigem<<posicoesFaixa,length, logo, o tamanho seria O(n). Também poderíamos

argumentar que a função é de tempo constante, já que o tempo de execução dela é sempre limitado pelo tamanho do numero de faixas no tabuleiro.

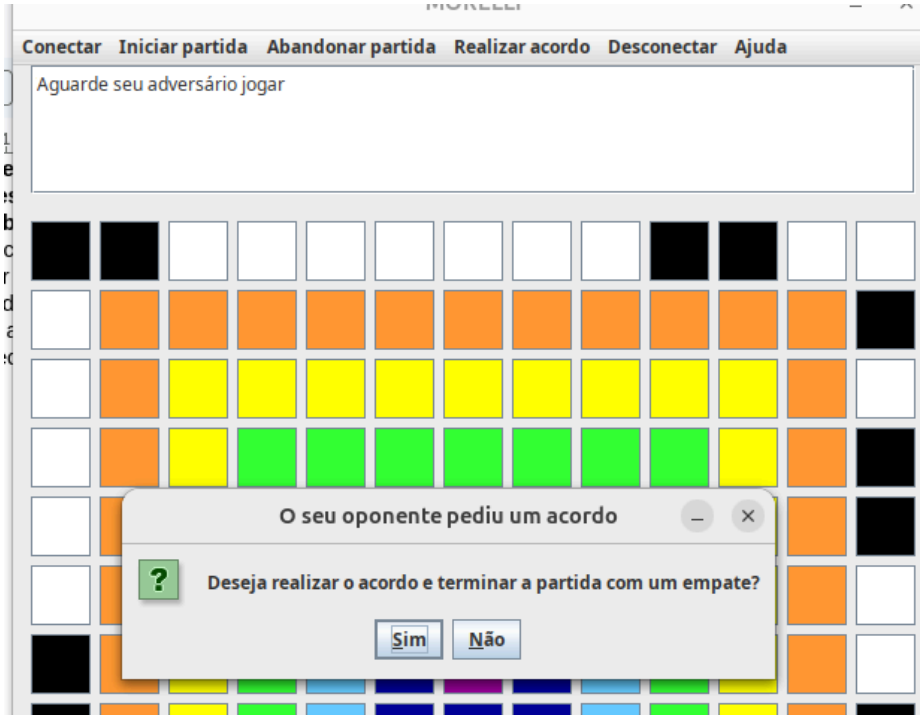
calcularMovimentoColuna()
calcularCaptura()
Mesmo raciocínio acima.

3) Análise dinâmica

1. Executar o sistema legado em ambiente controlado:
(a) Testar suas principais funcionalidades;
(b) Observar comportamento da interface e respostas do sistema.
Conectar -> funciona
Iniciar Partida -> funciona
Abandonar Partida -> não funciona como esperado. Aparece uma mensagem na tela informando que você abandonou o jogo, mas na verdade o jogo só trava. Se você clicar para Iniciar uma nova partida aparece na tela que um dos jogadores caiu e a partida foi finalizada:

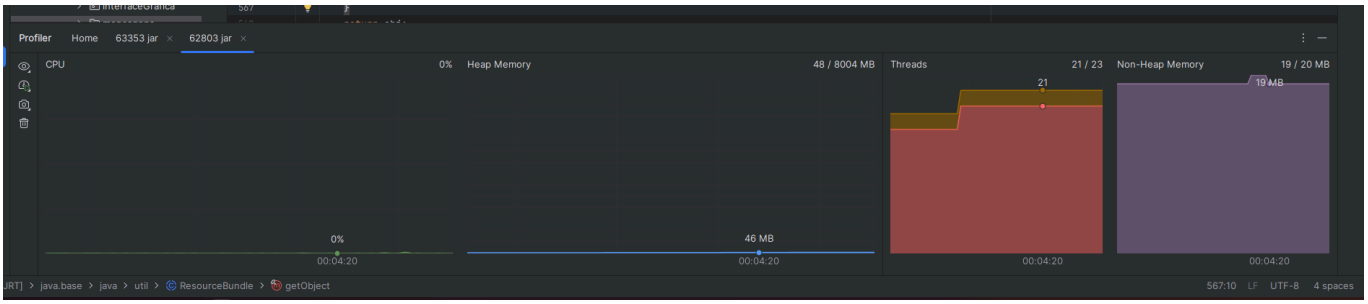


Realizar Acordo -> está bugada. A ideia é você oferecer um acordo, mas quando você clica na opção, aparece a mensagem que o adversário quer realizar o acordo e se você clicar sim o jogo acaba:

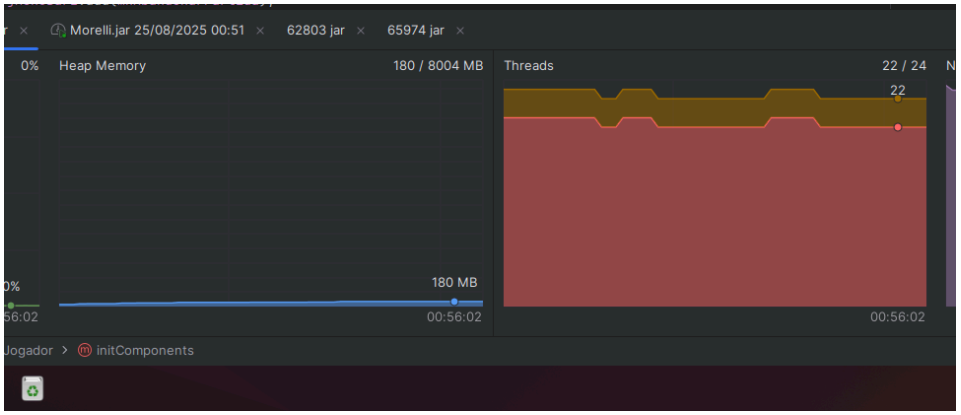


Desconectar -> funciona bem

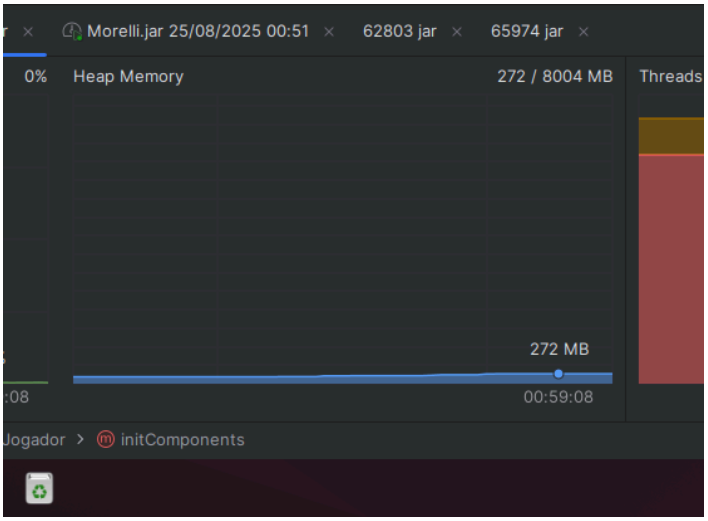
2. Registrar logs e fluxos de execução:
(a) Monitorar consumo de recursos (memória, rede, banco);
(b) Detectar gargalos de desempenho.



Não sei como interpretar as informações acima. Rodando o programa, em um momento ele atingiu um pico de heap memory de 180MB.



Logo depois, ele passou pra 272MB



Ao que parece, há um aumento gradual de uso da memória, o que pode indicar a existência de um vazamento de memória.

3. Utilizar depuração ou logs adicionais para acompanhar o fluxo dos métodos principais.

Temos a pasta com o logs.

Sobre o fluxo, ele é basicamente o seguinte:
Cliente A → Servidor → Cliente B
Jogada enviada → Validação → Distribuição → Atualização UI

4) Modelagem e abstração

1. Produzir diagramas UML:
(a) Diagrama de classes com principais entidades;
Diagram na pasta UML
(b) Diagrama de sequência para fluxos relevantes;
Diagram na pasta UML
(c) Diagrama de componentes para arquitetura geral.
Diagram na pasta UML

2. Abstrair a arquitetura identificada:
(a) Camada de apresentação;
(b) Camada de negócio;
(c) Camada de persistência.
Diagrama na pasta UML (Diagrama de Camadas). Não existe propriamente uma camada de persistência da aplicação, já que o estado do jogo é salvo no servidor e nas classes que implementam a lógica de negócios.

ANEXO I

Variáveis e métodos não utilizados

Classe	Nome
Faixa	getOrdem()
	getCor()
	contemPosicao()
	getTamanho()
ImagemDeTabuleiro	ImagemDeTabuleiro (a própria classe nunca é usada), bem como todos os seus métodos e atributos. Parece ser uma classe legada.
JogadaMorelli	posicaoOrigem;
	posicaoDestino
Jogador	setNome()
	setDaVez()
	isDaVez()
	setVencedor()
	isVencedor()
	setAbandonarPartida()
	isAbandonarPartida()
NetGames	import br.ufsc.inf.leobr.cliente.exception.ArquivoMultiplayerException; import br.ufsc.inf.leobr.cliente.exception.JahConectadoException;

	import br.ufsc.inf.leobr.cliente.exception.NaoConectadoException; import br.ufsc.inf.leobr.cliente.exception.NaoJogandoException; import br.ufsc.inf.leobr.cliente.exception.NaoPossivelConectarException; import java.util.logging.Level; import java.util.logging.Logger;
Posicao	setJogador()
	getJogador()
	setAdjacentes()
	getAdjacentes()
	verificarAdjacentes()
	adjacente (Parâmetro não usado dentro de um método)
	possuiAdjacentes()
	posicao
	verficarCaminho()
	origem (Parâmetro não usado dentro de um método)
	destino (Parâmetro não usado dentro de um método)
Tabuleiro	jogador (Parâmetro não usado dentro de um método)
	abandonarPartida()
	realizarAcordo()
	atualizaJogadorDaVez()
	getPosicaoOrigem()

	setPosicaoOrigem()
	getPosicaoDestino()
	setPosicaoDestino()
	alcularVerticeFaixa()
	criaJogadaDeFinalizacaoPartida()
	definePartidaFinalizada()
	recuperarFaixaDaPosicao()
AtorJogador	getJogador()
	isDaVez()
	The value 'new JogadaMorelli(TipoJogada.encerramento)' assigned to 'jogada' is never used
	notificarFalhaDesconexao()
	notificarResultado()
	notificarNaoJogando()
	notificarJaConectado()
TelaJogador	linhaOrigem
	colunaOrigem
	linhaDestino
	colunaDestino

	linhaSelecionada
	colunaSelecionada
	enableEditar()